System Support for Large-scale Geospatial Data Analytics

by

Jia Yu

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved May 2020 by the
Graduate Supervisory Committee:

Mohamed Sarwat, Chair
Kasim Selcuk Candan
Wenwen Li
Ming Zhao

ARIZONA STATE UNIVERSITY

August 2020

ABSTRACT

The volume of available spatial data has increased tremendously. Such data includes but is not limited to: weather maps, socioeconomic data, vegetation indices, geo-tagged social media, and more. These applications need a powerful data management platform to support scalable and interactive analytics on big spatial data. Even though existing single-node spatial database systems (DBMSs) provide support for spatial data, they suffer from performance issues when dealing with big spatial data. Challenges to building large-scale spatial data systems are as follows: (1) System Scalability: The massive-scale of available spatial data hinders making sense of it using traditional spatial database management systems. Moreover, large-scale spatial data, besides its tremendous storage footprint, may be extremely difficult to manage and maintain due to the heterogeneous shapes, skewed data distribution and complex spatial relationship. (2) Fast analytics: When the user runs spatial data analytics applications using graphical analytics tools, she does not tolerate delays introduced by the underlying spatial database system. Instead, the user needs to see useful information quickly.

In this dissertation, I focus on designing efficient data systems and data indexing mechanisms to bolster scalable and interactive analytics on large-scale geospatial data. I first propose a cluster computing system GeoSpark which extends the core engine of Apache Spark and Spark SQL to support spatial data types, indexes, and geometrical operations at scale. In order to reduce the indexing overhead, I propose Hippo, a fast, yet scalable, sparse database indexing approach. In contrast to existing tree index structures, Hippo stores disk page ranges (each works as a pointer of one or many pages) instead of tuple pointers in the indexed table to reduce the storage space occupied by the index. Moreover, I present Tabula, a middleware framework that

sits between a SQL data system and a spatial visualization dashboard to make the user experience with the dashboard more seamless and interactive. Tabula adopts a materialized sampling cube approach, which pre-materializes samples, not for the entire table as in the SampleFirst approach, but for the results of potentially unforeseen queries (represented by an OLAP cube cell).

DEDICATION

To my beloved parents for their unconditional love and support

ACKNOWLEDGMENTS

I would like to thank my advisor, Mohamed Sarwat, for his guidance and support during my dissertation research. He is an outstanding mentor and lifelong friend. The five-year experience with him completely changed my life and reshaped my understanding of scientific research. This also prompted me to decide to become a professor. There were lots of frustration and obstacles on the path to this dissertation. He is the lamp to light up my path. I would like to thank my dissertation committee members, Kasim Selcuk Candan, Wenwen Li and Ming Zhao, for their valuable interactions and feedback. I really enjoyed the thought-provoking discussions with them.

I have been working closely with the members of our Data Systems Lab over the years. I want to thank Yuhan Sun, Venkata Vamsikrishna Meduri, Kanchan Chowdhury, Ankita Sharma, Rishabh Wadhawan, Jinxuan Wu, Zongsi Zhang, Zishan Fu, Anique Tahir for their valuable help.

During my Ph.D., I interned at Apple, IBM Almaden Research Center and Microsoft Research and met many great software engineers and researchers. They have provided me comments, support and encouragement. I would like to thank Huang-Hsiang Cheng, Alex Radeski, Vijayshankar Raman, Yuanyuan Tian, Fatma Ozcan, Yingjun Wu, Umar Farooq Minhas and David Lomet.

Without the help and support from my friends, I cannot have such a wonderful Ph.D. life at ASU. I want to thank Yonghui Fan, Fei Gao, Ruocheng Guo, Rongyu Lin, Sicong Liu, Xu Liu, Kai Shu, Yaozhong Song, Yuan Wang, Tianyi Xing, Daqi Yin, Ruozhou Yu, Ziming Zhao, Dawei Zhou, Yao Zhou.

Last but not least, I would like to express my gratitude to my parents Zhongjie Yu and Guijuan Wang for their love and support all these years. I also want to thank

my girlfriend Lilian Luo for her love and patience. Whenever I meet with difficulties and setbacks, they always comfort me and encourage me to continue to pursue my dream. I cannot have arrived here without them.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Chapter 1

INTRODUCTION

The volume of available spatial data has increased tremendously. Such data includes but is not limited to: weather maps, socioeconomic data, vegetation indices, geo-tagged social media, and more. Furthermore, several cities are beginning to leverage the power of sensors to monitor the urban environment. For instance, the city of Chicago started installing sensors across its road intersections to monitor the environment, air quality, and traffic. Making sense of such spatial data will be beneficial for several applications that may transform science and society – For example: (1) Socio-Economic Analysis: that includes climate change analysis [19], study of deforestation [109], population migration [18], and variation in sea levels [95], (2) Urban Planning: assisting government in city/regional planning, road network design, and transportation / traffic engineering, (3) Commerce and Advertisement [23]: e.g., point-of-interest (POI) recommendation services. These applications need a powerful data management platform to support scalable and interactive analytics on big spatial data. Existing spatial database systems (DBMSs) [74] extend relational DBMSs with new data types, operators, and index structures to handle spatial operations based on the Open Geospatial Consortium standards [92]. Even though such systems provide support for spatial data, they suffer from performance issues when dealing with big spatial data. Challenges to building large-scale spatial data systems are as follows:

- **Challenge I: System Scalability.** The massive-scale of available spatial data hinders making sense of it using traditional spatial database management systems.

Moreover, large-scale spatial data, besides its tremendous storage footprint, may be extremely difficult to manage and maintain due to the heterogeneous shapes, skewed data distribution and complex spatial relationship. The underlying database system must be able to digest Terabytes of spatial data and effectively analyze it.

- **Challenge II: Fast Analytics.** In the past decade, interactive graphical analytics tools, such as Tableau, Apache Zeppelin [8], and Jupyter Notebook, become very popular. When the user runs spatial data analytics applications using these tools, he or she will not tolerate delays introduced by the underlying spatial database system. Instead, the user needs to see useful information quickly. Hence, the underlying spatial data processing system must figure out effective ways to execute spatial analytics in a short time window.

Recent works (e.g., [5, 29]) extend cluster computing systems, such as the Hadoop MapReduce [42], to perform spatial analytics at scale. Although the Hadoop-based approach achieves high scalability, it still exhibits slow run time performance and the user will not tolerate such delays. On the other hand, Apache Spark is now the defacto general-purpose cluster computing framework due to its speed and ease of use. It provides a novel data abstraction called Resilient Distributed Datasets (RDDs) [106] that are collections of objects partitioned across a cluster of machines. Each RDD is built using parallelized transformations (filter, join or groupBy) that could be traced back to recover the RDD data. All RDD transformations compose a Directed Acyclic Graph that is used by Spark's job scheduler to further optimize the execution. In-memory RDDs and DAG scheduler allow Spark to outperform existing models (MapReduce). Unfortunately, the native Spark ecosystem does not provide support for spatial data and operations. Hence, Spark users need to perform the tedious task of programming their own spatial data processing jobs on top of Spark.

In addition, since Spark puts intermediate data in-memory during the computation, it requires a great deal of memory. In fact, most analytics applications usually impose some filters (for both regular data and geospatial data) and hence only use a part of the raw data which is usually stored on disk-based database systems. To reduce the memory consumption, Spark and other in-memory computation engines such as Apache Flink [7] try to push some filters down to the underlying database systems, ask the underlying systems to perform the filtering tasks, and only load necessary data to the memory of a cluster. Disk-based database systems such as PostgresSQL [75] and PostGIS [74] often employ index structures, e.g., B+Tree [20] for regular data and R-Tree [39] for spatial data, to speed up filtering queries on the indexed table. Even though classic database indexes reduce the query response time, they face the following challenges: (1) A non-clustered database index usually yields 5% to 15% additional storage overhead (e.g., Solid State Drives, Non-Volatile Memory and Hard Disk Drive). Although the overhead may not seem too high in small databases, it results in non-ignorable dollar cost in big data scenarios (explained in Section 4). (2) Maintaining a database index incurs high latency because the DBMS has to locate and update those index entries affected by the underlying table changes (explained in Section 4).

Moreover, data scientists tend to explore a spatial dataset using graphic interfaces such as visualization dashboards (i.e., Tableau and ArcGIS) or web-based interactive notebooks (i.e., Apache Zeppelin [8]). Data exploration on the visualization dashboards often involves several interactions between the dashboard and underlying databases. In each interaction, the dashboard application first issues a query to extract the data of interest from the underlying data system, and then runs the visual analysis task (e.g., heat maps and statistical analysis) on the selected data. The same user may iteratively go through such steps several times to draw useful insights from a dataset. Every iteration may take a significant amount of time to run when dealing with large-scale data. An important observation is that data scientists may tolerate slight accuracy loss for this kind of visualization applications as long as the dashboard can maintain interactive performance. Therefore, sampling techniques are widely used in practice. There are two kinds of approaches used by practitioners: (1) prebuilt samples or stratified samples [24, 2, 76]: draw a small sample of the entire data table and run the dashboard on this subset (2) draw samples on-the-fly [38, 71]: run SQL queries over the entire table for every interaction, draw a sample of the extracted population and send it back to the dashboard. Unfortunately, the former cannot provide deterministic accuracy loss for geospatial visualization while the latter suffers from non-negligible query latency and sampling overhead.

Therefore, in this dissertation, I study the problem of designing efficient data systems and data indexing mechanisms to bolster scalable and interactive analytics on large-scale geospatial data. Specifically, I build data systems to answer the following questions:

- How to efficiently and precisely analyze big geospatial data?
- How to reduce the indexing overhead (in terms of storage and maintenance overhead) in big spatial data systems?
- How to interactively analyze big geospatial data (response time within a few seconds)?

By answering the above questions, the main contribution of the dissertation can be summarized as follows. I first propose a cluster computing system GeoSpark [1] which extends the core engine of Apache Spark and Spark SQL to support spatial data types, indexes, and geometrical operations at scale. In order to reduce the indexing overhead, I propose Hippo [2], a fast, yet scalable, sparse database indexing approach. In contrast to existing tree index structures, Hippo stores disk page ranges (each works as a pointer of one or many pages) instead of tuple pointers in the indexed table to reduce the storage space occupied by the index. Moreover, I present Tabula [3], a middleware framework that sits between a SQL data system and a spatial visualization dashboard to make the user experience with the dashboard more seamless and interactive. Tabula adopts a materialized sampling cube approach, which pre-materializes samples, not for the entire table as in the SampleFirst approach, but for the results of potentially unforeseen queries (represented by an OLAP cube cell).

The remainder of this dissertation is structured as follows. In Chapter 2, I review the related work. In Chapter 3, I discuss the proposed cluster computing system, GeoSpark. In Chapter 4, I explain the lightweight indexing mechanism, Hippo index. Chapter 5 shows the sampling middleware system Tabula. Chapter 6 concludes the dissertation and list future work.

[1] GeoSpark website: https://datasystemslab.github.io/GeoSpark/

[2] Hippo source code: https://github.com/DataSystemsLab/hippo-postgresql

[3] Tabula source code: https://github.com/DataSystemsLab/Tabula

Chapter 2

BACKGROUND AND RELATED WORK

In this section, I will briefly introduce the background and related work in scalable and interactive spatial data systems.

2.1 Spatial Database Operations

Spatial database operations are deemed vital for spatial analysis and spatial data mining. Users can combine query operations to assemble a complex spatial data mining application.

Spatial Range query: A spatial range query [69] returns all spatial objects that lie within a geographical region. For example, a range query may find all parks in the Phoenix metropolitan area or return all restaurants within one mile of the user's current location. In terms of the format, a spatial range query takes a set of points or polygons and a query window as input and returns all the points / polygons which lie in the query area.

Spatial Join: Spatial join queries [72] are queries that combine two datasets or more with a spatial predicate, such as distance relations. There are also some real scenarios in life: tell me all the parks which have rivers in Phoenix and tell me all of the gas stations which have grocery stores within 500 feet. Spatial join query needs one set of points, rectangles or polygons (Set A) and one set of query windows (Set B) as inputs and returns all points and polygons that lie in each one of the query window set.

Spatial K Nearest Neighbors (KNN) query: Spatial KNN query takes a query center point, a spatial object set as inputs and finds the K nearest neighbors around the center points. For instance, a KNN query finds the 10 nearest restaurants around the user.

Spatial Indexing: Spatial query processing algorithms usually make use of spatial indexes to reduce the query latency. For instance, R-Tree [39] provides an efficient data partitioning strategy to efficiently index spatial data. The key idea is to group nearby objects and put them in the next higher level node of the tree. R-Tree is a balanced search tree and obtains better search speed and less storage utilization. Quad-Tree [32] recursively divides a two-dimensional space into four quadrants.

2.2 Cluster Computing Systems

Over the years, researchers and practitioners have designed several different cluster computing systems to address the scalability issue on big data. These systems do not come with the functionalities of spatial data processing by default. However, learning their internals is very beneficial for understanding how people extend the core models to support geospatial data.

Hadoop MapReduce. The Hadoop MapReduce system [42] is an open-source implementation of Google MapReduce model [22]. The MapReduce model is inspired by the decades-old Map and Fold operations in functional programming languages such as S [11] and R [52] but extends the idea to the cluster computing environment by incorporating fault tolerance, task scheduling and so on. It is designed to process large datasets with a distributed algorithm on a cluster. This system gives a simple abstraction of complex distributed programs and hides the details of parallelization, fault-tolerance, data distribution and load balancing. A MapReduce program usually consists of three phases: Map, Shuffle, and Reduce. Among them, the Map and Reduce phases can run user-defined functions. The map function takes an input value key/value pair and generates a set of intermediate key/value pairs. The reduce function will merge the values of that key to a smaller set of values, based on the logic written by the user. A complex program may need to repeat the three phases, Map, Shuffle, and Reduce, many times and the intermediate data between two phases are persisted on local disk. The MapReduce system will execute the algorithm by scheduling tasks to distributed machines, running different tasks in parallel, managing data shuffle and providing redundancy and fault tolerance.

Apache Spark. The Spark system is a distributed general-purpose cluster computing framework which allows users to easily write distributed programs without being involved in the details of parallelism. It also can tolerate faults and scale out to many commodity machines. It is an implementation of Resilient Distributed Datasets (RDD) [106]. RDD is an immutable distributed collection of in-memory objects. Each RDD is built using parallelized transformations (filter, join or groupBy). For fault tolerance, Spark rebuilds lost data on failure using lineage: each RDD remembers how it was built from other datasets (through transformations) to recover itself. The lineage among RDDs is represented as a Directed Acyclic Graph which consists of a set of RDDs (points) and directed Transformations (edges).

There are two transformations can be applied to RDDs, (1) narrow transformation: Each partition of the parent RDD is used by at most one partition of the child RDD. A Narrow transformation does not incur any data shuffle. Examples are map and filter.(2) wide transformation: Each partition of the parent RDD is used by multiple child partitions. An example is join. A wide transformation will introduce data shuffle. The dependency between the parent and child is called wide dependency.

Directed Acyclic Graph (DAG) scheduler is deemed one of the most important components of Apache Spark that conducts stage-oriented scheduling. After a complex job is submitted to Spark, the scheduler computes a Directed Acyclic Graph for this job and divides the job into a set of stages based on this graph. This way, Spark can maximize the utilization of in-memory intermediate data and avoid unnecessary data shuffle (aka data transfer among nodes in the cluster).

SparkSQL [9] is an independent Spark module for structured data processing. It provides a higher-level abstraction called DataFrame over Spark RDD. A DataFrame is structured to the format of a table with column information. SparkSQL optimizer leverages the structure information to perform query optimization. SparkSQL supports two kinds of APIs: (1) DataFrame API manipulates DataFrame using Scala and Java APIs; (2) SQL API manipulates DataFrame using SQL statements directly. Unfortunately, Spark and SparkSQL do not provide native support for spatial data and spatial operations. Hence, users need to perform the tedious task of programming their own spatial data exploration jobs on top of Spark.

2.3 Process Geospatial Data in Cluster Environments

Most of the existing cluster computing systems do not come with out-of-box spatial data support. Given the fact that geospatial data analytics may involve a huge amount of geospatial data, researchers and practitioners have been working on extending the state-of-the-art cluster computing systems to support spatial data management.

2.3.1 Geospatial Data Processing in the Hadoop Ecosystem

There exist systems that extend state-of-the-art Hadoop to support massive-scale geospatial data processing. A detailed comparison of the existing Hadoop-based systems is given in Table 1. Although these systems have well-developed functions, all of them are implemented on top of the Hadoop MapReduce framework, which suffers from a large number of reads and writes on disk.

SpatialHadoop [29] provides native support for spatial data in Hadoop. It supports various geometry types, including polygon, point, line string, multi-point and so on, and multiple spatial partitioning techniques [27] including uniform grids, R-Tree, Quad-Tree, KD-Tree, Hilbert curves and so on. Furthermore, SpatialHadoop provides spatial indexes and spatial data visualization [30]. The SQL extension of SpatialHadoop, namely Pigeon [28], allows users to run Spatial SQL queries following the standard SQL/MM-Part 3 [53].

Parallel-Secondo [61] integrates Hadoop with SECONDO, a database that can handle non-standard data types, like spatial data, usually not supported by standard systems. It employs Hadoop as the distributed task manager and performs operations on a multi-node spatial DBMS. It supports the common spatial indexes and spatial queries except KNN . However, it only supports uniform spatial data partitioning techniques, which cannot handle the spatial data skewness problem. In addition, the visualization function in Parallel-Secondo needs to collect the data to the master local machine for plotting, which does not scale up to large datasets.

HadoopGIS [5] utilizes SATO spatial partitioning [89] (similar to KD-Tree) and local spatial indexing to achieve efficient query processing. Hadoop-GIS can support declarative spatial queries with an integrated architecture with HIVE [87]. However,

Feature name	GeoSpark	Simba	Magellan	Spatial Spark	GeoMesa	Spatial Hadoop	Parallel Secondo	Hadoop GIS
RDD API	✓	✗	✗	✓	✓	✗	✗	✗
DataFrame API	✓	✓	✓	✗	✓	✗	✗	✗
Spatial SQL [53, 48]	✓	✗	✗	✗	✓	✓	✗	✗
Complex geometrical operations	✓	✗	✗	✗	✓	✓	✗	✗
Spatial indexing	R-Tree Quad-Tree	R-Tree Quad-Tree	✗	R-Tree	Grid file	R-Tree Quad-Tree	R-Tree	R-tree
Spatial partitioning	Multiple	Multiple	Z-Curve	R-Tree	R-Tree	Multiple	Uniform	SATO
Range / Distance query	✓	✓	✓	✓	✓	✓	✓	✓
KNN query	✓	✓	✗	✗	✗	✓	✗	✓
Range / Distance Join	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Spatial data processing systems in Hadoop MapReduce and Apache Spark

HadoopGIS doesn't offer standard Spatial SQL [53]. In addition, it lacks the support of complex geometry types including convex/concave polygons, line string, multi-point, multi-polygon and so on. HadoopGIS visualizer can plot images on the master local machine.

2.3.2 Spark-based Geospatial Data Processing Systems

Limitations of Spark-based systems I have studied four popular Spark-based systems including their research papers and source code: Simba [98], Magellan [82], SpatialSpark [101] and GeoMesa [50]. Before diving into the details of each system, I want to summarize their common limitations (see Table 1):

- Simple shapes only: Simba only supports simple point objects for KNN and join. Magellan can read polygons but can only process spatial queries on the Minimum Bounding Rectangle (MBR) of the input polygons. SpatialSpark only works with point and polygon. Spatial objects may contain many different shapes and even a single dataset may contain heterogeneous geometrical shapes. Calculating complex shapes is time-consuming but indeed necessary for real life applications.
- Approximate query processing algorithms: Simba does not use the Filter and Refine model [39, 32], which cannot guarantee the query accuracy for complex shapes. The Filter-Refine model is a must in order to guarantee R-Tree/Quad-Tree query accuracy although it takes extra time. Magellan only uses MBR in spatial queries instead of using the real shapes. Simba and GeoMesa do not remove duplicated objects introduced by spatial partitioning (i.e., polygon and line string) and directly returns inaccurate query results.
- RDD only or DataFrame only: Simba and Magellan only provide DataFrame API. However, the Java / Scala RDD API allows users to achieve granular control of their own application. For complex spatial data analytics algorithm such as spatial co-location pattern mining, Simba and Magellan users have to write lots of additional code to convert the data from the DataFrame to the RDD form. That leads to additional execution time as well as coding effort. On the other hand, SpatialSpark only provides RDD API and does not provide support for spatial SQL. The lack of a SQL / declarative interface makes it difficult for the system to automatically optimize spatial queries.

- No standard Spatial SQL [53]: Simba's SQL interface doesn't follow either OpenGIS Simple Feature Access [48] or SQL/MM-Part 3 [53], the two de-facto Spatial SQL standards. Magellan only allows the user to issue queries via its basic SparkSQL DataFrame API and does not allow users to directly run spatial queries in a SQL statement.

Simba [98] extends SparkSQL to support spatial data processing over the DataFrame API. It supports several spatial queries including range query, distance join query, KNN and KNN join query. Simba builds local R-Tree indexes on each DataFrame partition and uses R-Tree grids to perform the spatial partitioning. It also optimizes spatial queries by: (1) only using indexes for highly selective queries. (2) selecting different join algorithms based on data size.

Magellan [82] is a popular industry project that received over 300 stars on GitHub. It extends SparkSQL to support spatial range query and join query on DataFrames. It allows the user to build a "z-curve index" on spatial objects. Magallan's z-curve index is actually a z-curve spatial partitioning method, which exhibits slow spatial join performance [27].

SpatialSpark [101] builds on top of Spark RDD to provide range query and spatial join query. It can leverage R-Tree index and R-Tree partitioning to speed up queries.

GeoMesa [50] is an open-source spatial index extension built on top of distributed data storage systems. It provides a module called GeoMesaSpark to allow Spark to read the pre-processed and pre-indexed data from Accumulo [90] data store. GeoMesa also provides RDD API, DataFrame API and Spatial SQL API so that the user can run spatial queries on Apache Spark. GeoMesa must be running on top of ZooKeeper [51] with 3 master instances. GeoMesa supports range query and join query. In particular, it can use R-Tree spatial partitioning technique to decrease the computation overhead. However, it uses a grid file as the local index per DataFrame partition. Grid file is a simple 2D index but cannot well handle spatial data skewness in contrast to R-Tree or Quad-Tree index. Most importantly, GeoMesa does not remove duplicates introduced by partitioning the data and hence cannot guarantee join query accuracy. In addition, GeoMesa does not support parallel map rendering. Its user has to collect the big dataset to a single machine then visualize it as a low resolution map image.

2.4 Database Indexing Mechanisms

Classic database indexes (e.g., B + -Tree and R-Tree), though speed up queries, suffer from storage overhead and maintenance overhead. There has been a flurry of database indexing mechanism trying to solve this problem. This section studies the existing techniques that index regular data or spatial data (see Table 2).

B^+-Tree B^+-Tree is the most commonly used type of indexes. The basic idea can be summarized as follows: For a non-leaf node, the value of its left child node must be smaller than that of its right child node. Each leaf node points to the physical address of the original tuple. With the help of this structure, searching B^+-Tree can be completed in one binary search time scope. The excellent query performance of B^+-Tree and other tree like indexes is benefited by their well designed structures which consist of many non-leaf nodes for quick searching and leaf nodes for fast accessing parent tuples. This feature incurs two inevitable drawbacks: (1) Storing plenty of nodes costs a huge chunk of disk storage. (2) Index maintenance is extremely time-consuming. For any insertions or deletions occur on parent table, tree like indexes firstly have to traverse themselves for finding proper update locations and then split, merge or re-order one or more nodes which are out of date.

R-Tree An R-Tree index is a balanced search tree that indexes spatial data [39] using a similar data structure like B^{+}-Tree: each non-leaf node has a set of $< key, pointer >$. In a non-leaf node, the pointer points to its child node while in a leaf node, the pointer points to the parent table tuple. The key is 2 dimensional rectangle which is the Minimum Bounding Rectangle (MBR) of data in the corresponding child node. In other words, R-Tree's basic idea is to group nearby spatial objects together and use an upper tree node to store their Minimum Bounding Rectangle (MBR) as well as pointers. R-Tree bulk-loading algorithm runs in a bottom-up fashion, which is indeed faster than inserting tuple one by one. The bulk loading algorithm generates plenty of tree nodes besides tuples pointers and, in practice, it writes many temporary files onto disk for scalability. The index search algorithm takes as input a spatial rectangular range predicate. The algorithm starts at the root node and traverses the child nodes that satisfy the spatial predicate. The algorithm then prunes subtrees in R-Tree, which possess MBRs that do not intersect with the spatial query predicate. The algorithm performs this step recursively until it reaches the tree leaf level and finally returns all spatial objects that lie within the spatial query range. The tree structure offers fast index search on highly selective queries at the cost of excessive indexing and maintenance overhead.

Bitmap Indexes A Bitmap index [57, 83, 113] has been widely applied to low cardinality and read-only datasets. It uses bitmaps to represent values without trading query performance. However, Bitmap index's storage overhead significantly increases when indexing high cardinality attributes because each index entry has to expand its bitmap to accommodate more distinct values. Bitmap index also does not perform well in update-intensive workloads due to tuple-wise index structure.

Compressed Indexes Compressed indexes drop some repeated index information to save space and recover it as fast as possible upon queries but they all have guaranteed query accuracy. These techniques are applied to tree indexes [36, 40]. Though compressed indexes are storage economy, they require additional time for compressing beforehand and decompressing on-the-fly. Compromising on the time of initialization, query and maintenance is not desirable in many time-sensitive scenarios. Hippo on the other hand reduces the storage overhead by dropping redundancy tuple pointers and hence still achieves competitive query response time.

Approximate Indexes Approximate indexes [10, 49, 79] give up the query accuracy and only store some representative information of parent tables for saving indexing and maintenance overhead and improving query speed. They propose many efficient statistics algorithms to figure out the most representative information which is worth to be stored. In addition, some people focus on approximate query processing (AQP)[3, 108] which relies on data sampling and error bar estimating to accelerate query speed directly. However, trading query accuracy makes them applicable to limited scenarios such as loose queries.

Sparse Indexes A sparse index, e.g., as Zone Map [14], Block Range Index [84] (BRIN), Storage Index [68], and Small Materialized Aggregates (SMA) index [63], only stores pointers to disk pages / column blocks in parent tables and value ranges (min and max values) in each page / column block so that it can reduce the storage overhead. For spatial data, the value range is a Minimum Bounding Rectangle of all spatial data in the corresponding page / column block. For a posed query, it finds value ranges which cover the query predicate and then inspects the associated few parent table pages one by one for retrieving truly qualified tuples. However, for unordered data, a sparse index has to spend lots of time on page scanning since the stored value ranges (min and max values) may cover most query predicates. In addition, column imprints [81], a cache-conscious secondary index, significantly enhances the traditional sparse indexes to speed up queries at a reasonably low storage overhead in-memory data warehousing systems. It leverages histograms and bitmap compression but does not support dynamic pages / column block size control to further optimize the storage overhead reduction especially with partially clustered data. The column imprints approach is designed to handle query-intensive workloads and puts less emphasis on efficiently update the index in row stores. BRIN-Spatial index [84, 75] in PostgreSQL extends Block Range Index to support geospatial data. Similar to BRIN, BRIN-Spatial also groups pages into a fixed disk page range unit (128 pages per range by default). Each index entry contains two components: a static disk page range (e.g., page 1 - 10) and a Minimum Bounding Rectangle (MBR) that encloses all spatial data tuples that are recorded in the page range. The index initialization algorithm scans the indexed table only once to generate BRIN-Spatial. For each page range, BRIN-Spatial reads all tuples to construct the MBR for each index entry. Given a spatial range query, the query processor only searches the index entries for which the MBRs intersect with the spatial query predicate. It is highly recommended to ensure that the indexed spatial objects physically maintain their spatial locality in a certain way, e.g., sorting by longitude/latitude coordinate or Hilbert curve. In that case, the index entries keep the minimal MBR overlap between each other.

Index type	Fast query	Guaranteed accuracy	Low storage	Fast maintenance
B+ Tree and R-Tree	✓	✓	✗	✗
Compressed index	✗	✓	✓	✗
Bitmap Indexes	✓	✗	✓	✗
Approximate index	✓	✗	✓	✗
Sparse index	✗	✓	✓	✓
Hippo	✓	✓	✓	✓

Table 2: Compared Indexing Approaches

2.5 Interactive Analytics on Big Spatial Data

To support interactive analytics on big data including geospatial data, practitioners have proposed a variety of techniques such as sampling techniques and data cube relevant techniques.

2.5.1 Sampling Techniques

Data systems using pre-built samples. In the past two decades, several research works studied the implementation of classic sampling techniques such as random sampling, stratified sampling, cluster sampling, systematic sampling [46], and spatial sampling in database systems. However, samples pre-computed by classic sampling techniques may eventually lead to inaccurate results [17]. To enhance the accuracy of pre-built samples, recent systems [2, 24, 91, 1, 76] proposed sampling approaches that take into account different data populations. Sample+Seek [24] applies approximate query processing techniques on the data cube and offers a distribution precision guarantee. BlinkDB [2] and SnappyData [76] support approximate query processing with bounded error over customized HIVE and Spark clusters. They create stratified samples over Query Column Set (QCS) to improve accuracy. But their pre-built stratified samples have no accuracy guarantee so the systems and only support classic OLAP aggregate measures such as SUM, COUNT, AVG.However, all aforementioned approaches assume specific business scenarios, and hence only apply tailored optimization strategies on classic OLAP aggregate measures such as SUM, COUNT, AVG. Such specialized measure-biased sampling techniques cannot be easily extended to other types of analysis such as geospatial visualization.

Sample on the fly (i.e., Query time sampling). Approximate query processing [97, 56, 107] rely on sampling. Some approaches [58, 41] work on placing samplers inside join queries. Many approximate systems such as ABS [107] and Quickr [56] focus more on where to place the online sampler in the query plan. These approaches yield better accuracy and reduce the data-system query time but still inevitably access raw datasets on the fly. A recently proposed approach, namely Dice [55, 54], applies speculative query execution techniques to predict the human next query and prefetch the anticipated query answer upon a data cube [37] that holds pre-computed aggregate measures (e.g., SUM, COUNT, AVG). Such query speculation technique speeds up data-system query over databases. However, Dice still runs an online sampler to return a sample for each query whereas Tabula directly fetches pre-built samples without accessing raw data because it exhausts all query results in advance. Moreover, Tabula provides deterministic sample accuracy loss guarantees while other approaches only guarantee bounded-error with a confidence level.

Spatial sampling. Researchers proposed various approaches to accelerate spatial visualization process on big datasets. POIsam and VAS [38, 71] propose similar visualization-aware sampling approaches with sample accuracy loss guarantee. They shorten map visualization time but, in the case of spatial visualization dashboards, still perform sampling on the fly and have no optimization to reduce online data-system query time. Wang et al. [91] propose a spatial indexing mechanism that indexes spatial data by different levels such that their online sampler can run faster and produce more accurate results. But they do not provide a deterministic sample accuracy loss guarantee and cannot speed up queries that involve non-spatial attributes.

2.5.2 Data Cube Relevant Techniques

Data cubes. Gray et al. [37] proposed the concept of data cubes which prematerializes the answers for all future analytics queries. A data cube is a collection of aggregate measures calculating using all GroupBy queries issued over n "cubed" attributes. The following query can be used to initialize a data cube:

```
SELECT [List of attributes] COUNT(*) AS count
FROM  nyctaxi
GROUPBY CUBE(List of attributes)
```

This query leads to $2^{num\_attributes}$-1 different GroupBy queries such that the result of each GroupBy query is called a cuboid. It is obvious that the cube initialization time grows exponentially with the number of attributes involved in the cube. Later, several papers [4, 45] proposed more advanced techniques to initialize data cubes with distributive and algebraic measures. These algorithms require the aggregate measures in the cube to be distributive or algebraic [37, 64] (1) Distributive: The measure of a cell can be computed solely based on the same measure of its descendant cells. For instance, SUM in Cell$\langle * : SUM\rangle$ is equal to the sum of $\langle Cash : SUM\rangle, \langle Credit : SUM\rangle, \langle Dispute : SUM\rangle$. (2) Algebraic: The measure of a cell can be computed based on several other types of measures in its descendant cells, e.g., AVG(). A distributive measure must be algebraic and an algebraic measure may not be distributive. All other measures are called holistic measures. These techniques focus on allocating cuboid groups to preserve data orders in the same group and avoiding unnecessary raw table accesses. Researchers [12] came up with the iceberg data cube and a Bottom-Up initialization algorithm to compute cube distributive measures with minimal overhead. Instead of persisting all measures, the iceberg cube by nature only stores a small number of aggregate measures. H-Cubing [44] and Star-Cubing [100] propose different iceberg cube initialization algorithms for algebraic measures. However, all aforementioned cubes only work for algebraic measures.

Application-aware data cubes. In the past decade, researchers turn their attention to data cubes with specialized aggregate measures and data types tailored for specific applications. Ranking cube [99] is a data cube to answer Top-K queries. Graph cube [111] propose data cube creation on graph data and textual data. MRcube describes a data cube initialization algorithm for MapReduce framework such that it can produce reducer-friendly cuboid groups for partially-algebraic measures and compute them in parallel. Sampling cube'08 [59] has a similar name to the sampling cube maintained by Tabula but acts very differently; it builds an iceberg data cube on a sample of the raw table to speed up the cube initialization. The aggregate measures of this cube are simple algebraic measures such as AVG. After the first round initialization, the cube will improve some inaccurate AVGs using nearby cells' AVGs. Differently, Tabula is built on the raw table and persists samples in cells to support user defined analysis tasks. Nano cube [60] and its variants [70] pre-materialize heat maps and other types of aggregates to answer online visualization requests. To mitigate the storage overhead, they design a set of complex visual encoding techniques to compress materialized aggregates. To query the cubes, a custom-made front-end visualization tool is required while Tabula is a middleware system which has no requirements on both visualization front-ends and underlying data systems. It is worth noting that the encoding techniques used in these cubes are complementary to Tabula such that they can work in concert with my system to further reduce the memory footprint. Moreover, none of the aforementioned approaches offer a generic system to uphold various user-defined visual analytics including map visualization.

Chapter 3

A CLUSTER COMPUTING SYSTEM FOR PROCESSING LARGE-SCALE SPATIAL DATA

In this chapter, I focus on the details of designing and developing of GeoSpark [105], a cluster computing framework which extends the core engine of Apache Spark and SparkSQL to support spatial data types, indexes, and geometrical operations at scale.

3.1 Introduction

Existing spatial database systems (DBMSs) [74] extend relational DBMSs with new data types, operators, and index structures to handle spatial operations based on

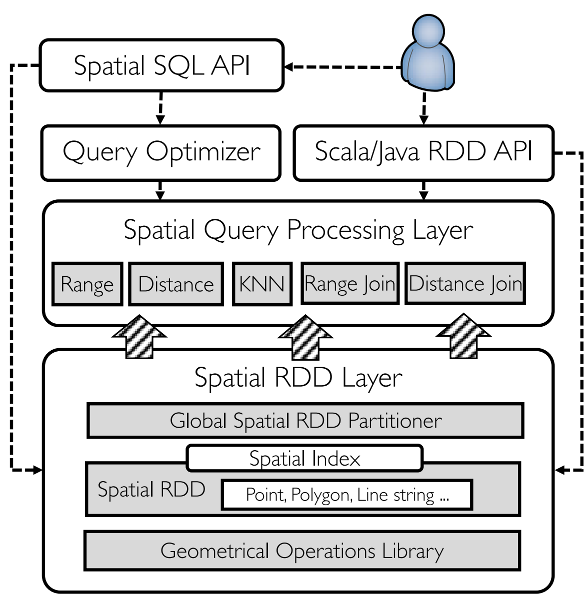


Figure 1: GeoSpark Overview

the Open Geospatial Consortium standards [92]. Even though such systems provide support for spatial data, they suffer from a scalability issue. That happens because the massive scale of available spatial data hinders making sense of it when using traditional spatial query processing techniques. Recent works (e.g., [5, 29]) extend the Hadoop ecosystem to perform spatial analytics at scale. Although the Hadoop-based approach achieves high scalability, it still exhibits slow run time performance and the user will not tolerate such delays.

Apache Spark, on the other hand, provides a novel data abstraction called Resilient Distributed Datasets (RDDs) [106] that are collections of objects partitioned across a cluster of machines. Each RDD is built using parallelized transformations (filter, join or groupBy) that could be traced back to recover the RDD data. In memory RDDs allow Spark to outperform existing models (MapReduce). Unfortunately, the native Spark ecosystem does not provide support for spatial data and operations. Hence, Spark users need to perform the tedious task of programming their own spatial data processing jobs on top of Spark. In fact, large-scale spatial data cannot be easily stored in Spark's native RDD like plain objects because of the following challenges:

Heterogeneous data sources. Different from generic datasets, spatial data is stored in a variety of special file formats that can be easily exchanged among GIS libraries. These formats include CSV, GeoJSON [15], WKT [73], NetCDF/HDF [86] and ESRI Shapefile [31]. Spark does not natively understand the content of these files and straightforward loading of such data formats into Spark may lead to inefficient processing of such data.

Complex geometrical shapes. There are many different types of spatial objects each of which may possess very complex shapes such as concave/convex polygons and multiple sub-shapes. In addition, even a single dataset may contain multiple different objects such as Polygon, Multi-Polygon, and GeometryCollection. These objects cannot be efficiently partitioned across machines, serialized in memory, and processed by spatial query operators. It requires too much effort to handle such spatial objects, let alone optimize the performance in terms of run time cost and memory utilization.

Spatial partitioning. The default data partitioner in Spark does not preserve the spatial proximity of spatial objects, which is crucial to the efficient processing of spatial data. Nearby spatial objects are better stored in the same RDD partition so that the issued queries only access a reduced set of RDD partitions instead of all partitions.

Spatial index support. Spark does not support any spatial indexes such as Quad-Tree and R-Tree. In addition, maintaining a regular tree-like spatial index yields additional 15% storage overhead[104, 102]. Therefore, it is not possible to simply build a global spatial index for all spatial objects of an RDD in the master machine memory.

In this section, I present the details of designing and developing GeoSpark [4], which extends the core engine of Apache Spark and SparkSQL to support spatial data types, indexes, and geometrical operations at scale. In other words, the system extends the Resilient Distributed Datasets (RDDs) concept to support spatial data. Figure 1 gives an overview of GeoSpark. Users can interact with the system using either a Spatial SQL API or a Scala/Java RDD API. The Scala/Java RDD API allows the user to use an operational programming language to write her custom made spatial analytics application. The user can create a Spatial RDD, call the geometrical library and run spatial operations on the created RDDs. The Spatial SQL API follows the SQL/MM Part 3 Standard [53]. Specifically, three types of Spatial SQL interfaces are supported:

[4] source code: https://github.com/DataSystemsLab/GeoSpark

(1) Constructors: initialize a Spatial RDD. (2) Geometrical functions: that represent geometrical operations on a given Spatial RDD (3) Predicates: issue a spatial query and return data that satisfies the given predicate such as Contains, Intersects and Within.

The Spatial Resilient Distributed Dataset (SRDD) Layer extends Spark with Spatial RDDs (SRDDs) that efficiently partition spatial data elements across the Apache Spark cluster. This layer also introduces parallelized spatial transformations and actions (for SRDD) that provide a more intuitive interface for programmers to write spatial data analytics programs. A Spatial RDD can accommodate heterogeneous spatial objects which are very common in a GIS area. Currently, GeoSpark allows up to seven types of spatial objects to co-exist in the same Spatial RDD. The system also provides a comprehensive geometrical operations library on-top of the Spatial RDD.

The Spatial Query Processing Layer allows programmers to execute spatial query operators over loaded Spatial RDDs. Such a layer provides an efficient implementation of the most-widely used spatial query operators, e.g., range filter, distance filter, spatial k-nearest neighbors, range join and distance join.

3.2 Spatial RDD (SRDD) layer

GeoSpark Spatial RDDs are in-memory distributed datasets that intuitively extend traditional RDDs to represent spatial objects in Apache Spark. A Spatial RDD consists of many partitions and each partition contains thousands of spatial objects.The rest of this section highlights the details of the Spatial RDD layer and explains how GeoSpark exploits Apache Spark core concepts for accommodating spatial objects.

3.2.1 SRDD Spatial Objects Support

Spatial RDD supports various input formats (e.g., CSV, WKT, GeoJSON, NetCDF/HDF, and Shapefile), which cover most application scenarios. Line delimited file formats (CSV, WKT and GeoJSON) that are compatible with Spark can be created through the GeoSpark Spatial SQL interface. Binary file formats (NetCDF/HDF and Shapefile) need to be handled by GeoSpark customized Spark input format parser which detects the position of each spatial object.

Since spatial objects have many different types [15, 73, 86, 31], GeoSpark uses a flexible implementation to accommodate heterogeneous spatial objects. Currently, GeoSpark supports seven types of spatial objects, Point, Multi-Point, Polygon, Multi-Polygon, LineString, Multi-LineString, GeometryCollection, and Circle. This means the spatial objects in a Spatial RDD can either belong to the same geometry type or be in a mixture of many different geometry types.

GeoSpark users only need to declare the correct input format followed by their spatial data without any concern for the underlying processing procedure. Complex data transformation, partitioning, indexing, in-memory storing are taken care of by GeoSpark and do not bother users.

A SQL and Scala example of constructing a Spatial RDD from WKT strings is given below.

```
/* Spatial SQL API*/
SELECT ST_GeomFromWKT(TaxiTripRawTable.pickuppointString)
FROM TaxiTripRawTable

/* Scala/Java RDD API */
```

```
var TaxiTripRDD = new SpatialRDD(sparkContext, dataPath)
```

3.2.2 SRDD Built-in Geometrical Library

GeoSpark provides a built-in library for executing geometrical computation on Spatial RDDs in parallel. This library provides native support for over 20 geometrical operations (e.g., Dataset boundary, polygon union and reference system transform) that follow the Open Geospatial Consortium (OGC) [92] standard [5]. Operations in the geometrical computation library can be invoked through either GeoSpark Spatial SQL interface or GeoSpark RDD APIs. Each operation in the geometrical library employs a distributed computation algorithm to split the entire geometrical task into small sub-tasks and execute them in parallel. I explain the algorithms used in Dataset Boundary and Reference System Transformation as examples (SQL is also given); other operations have similar algorithms.

DatasetBoundary (SQL: ST_Envelope_Aggr): This function returns the rectangle boundary of the entire Spatial RDD. In GeoSpark Spatial SQL, it takes as input the geometry type column of the dataset. It uses a Reduce-like algorithm to aggregate the boundary: it calculates the merged rectangular boundary of spatial objects two by two until the boundaries of all the objects are aggregated. This process first happens on each RDD partition in parallel. After finding the aggregated boundary of each partition, it aggregates the boundary of partitions two by two until the end.

For instance, the following function returns the entire rectangular boundary of all taxi trips' pickup points.

[5] A complete list of geometrical operations supported in GeoSpark: https://datasystemslab.github.io/GeoSpark/api/sql/GeoSparkSQL-Overview/

```
/* Spatial SQL */
SELECT ST_Envelope_Aggr(TaxiTripTable.pickuppoint)
FROM TaxiTripTable

/* Scala/Java RDD API */
var envelopeBoundary = TaxiTripRDD.boundary()
```

ReferenceSystemTransform (SQL: ST_Transform) Given a source and a target Spatial Reference System code, this function changes the Spatial Reference System (SRS) [92] of all spatial objects in the Spatial RDD. In GeoSpark Spatial SQL, this function also takes as input the geometry type column of the dataset. It uses a Map-like algorithm to convert the SRS: for each partition in a Spatial RDD, this function traverses the objects in this partition and converts their SRS.

3.2.3 SRDD Partitioning

Apache Spark loads the input data file into memory, physically splits its in-memory copy to many equally sized partitions (using hash partitioning or following HDFS partitioned file structure) and passes each partition to each worker node. This partitioning method doesn't preserve the spatial proximity which is crucial for improving query speed.

GeoSpark automatically repartitions a loaded Spatial RDD according to its internal spatial data distribution. The intuition of Spatial RDD partitioning is to group spatial objects into the same partition based upon their spatial proximity. Spatial partitioning accelerates the query speed of a join query. It achieves that by reducing the data shuffles across the cluster (see Section 3.3.3) and avoiding unnecessary computation on partitions that are impossible to have qualified data. A good spatial partitioning technique keeps all Spatial RDD partitions balanced in terms of memory space and spatial computation, aka. load balancing.

Algorithm 1: SRDD Spatial Partitioning

```
Data: An original SRDD
Result: A repartitioned SRDD
/* Step 1: Build a global grid file at master node */
Take samples from the original SRDD A partitions in parallel;
Construct the selected spatial structure on the collected sample at master
  node;
Retrieve the grids from built spatial structures;
/* Step 2: Assign grid ID to each object in parallel */
foreach spatial object in SRDD A do
    foreach grid do
        if the grid intersects the object then
            Add (grid ID, object) pair into SRDD B;
    // Only needed for R-Tree partitioning
    if no grid intersects the object then
        Add (overflow grid ID, object) pair into SRDD B;
/* Step 3: Repartition SRDD across the cluster */
Partition SRDD B by ID and get SRDD C;
Cache the new SRDD C in memory and return it;
```

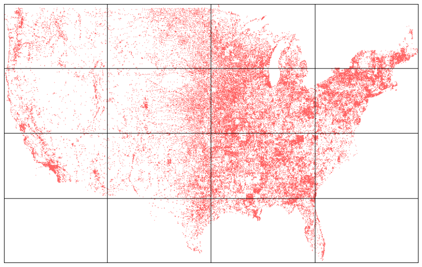
Spatial RDD represents a very large and distributed dataset so that it is extremely time consuming to traverse the entire Spatial RDD for obtaining the spatial distribution and partition it according to its distribution. GeoSpark employs a low overhead spatial partitioning approach to take advantage of global spatial distribution awareness. Hence, GeoSpark swiftly partitions the objects across the cluster. The spatial partitioning technique incorporates three main steps as follows (see Algorithm 1):

Step 1: Building a global spatial grid file: In this step, the system takes samples from each Spatial RDD partition and collects the samples to Spark master node to generate a small subset of the Spatial RDD. This subset follows the Spatial

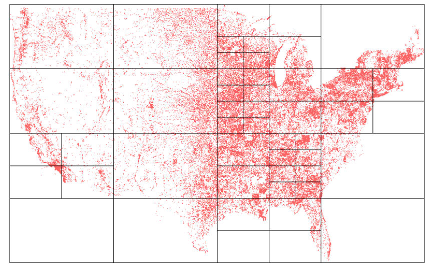
RDD's data distribution. Hence, if I split the subset into several load balanced partitions that contain a similar number of spatial objects and apply the boundaries of partitions to the entire Spatial RDD, the new Spatial RDD partitions should still be load-balanced. Furthermore, the spatial locations of these records should also be of a close spatial proximity to each other. Therefore, after sampling the SRDD, GeoSpark constructs one of the following spatial data structures that splits the sampled data into partitions at the Spark master node (see Figure 2). As suggested by [27], GeoSpark takes 1% percent data of the entire Spatial RDD as the sample:

- **Uniform Grid:** GeoSpark partitions the entire two-dimensional space into equal sized grid cells which have the same length, width and area. The boundaries of these grid cells are applied to the entire Spatial RDD. The partitioning approach generates non-balanced grids which are suitable for uniform data.
- **R-Tree | Quad-Tree | KDB-Tree:** This approach exploits the definition of capacity (fanout) in the classical spatial tree index structures, R-Tree [39], Quad-Tree [32] and KDB-Tree [77]: each tree node contains the same number of child nodes. GeoSpark builds an R-Tree, Quad-Tree or KDB-Tree on the sample subset and collects the leaf node boundaries to a grid file. It is worth noting that: the grids of R-Tree that builds on the sample data do not cover the entire space of the dataset. Thus, I need to have an overflow partition to accommodate the objects that do not fall in any grid of R-Tree partitioning. However, since Quad-Tree and KDB-Tree always start the splitting from the entire dataset space, I don't need the overflow partition.

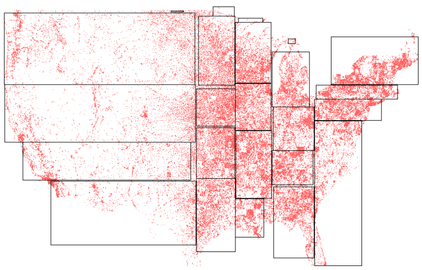
Step 2: Assigning a grid cell ID to each object: After building a global grid file, GeoSpark needs to know the grid inside which each object falls and then repartition the Spatial RDD in accordance with the grid IDs. Therefore, GeoSpark



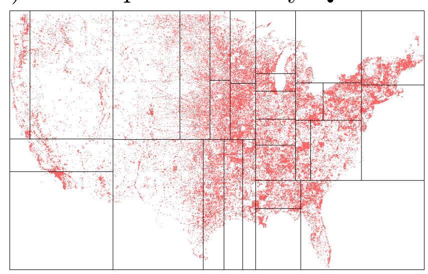
(a) SRDD partitioned by uniform grids



(b) SRDD partitioned by Quad-Tree



(c) SRDD partitioned by R-Tree



(d) SRDD partitioned by KDB-Tree

Figure 2: Grids generated by SRDD spatial partitioning techniques

duplicates the grid files and broadcasts the copies to each Spatial RDD partition. After receiving the broadcasted grid file, each original Spatial RDD partition simultaneously starts to check each internal object against the grid file. The results are stored in a new Spatial RDD whose schema is <Key, Value>. If an object intersects with a grid, the grid ID will be assigned to this object and a <grid ID, object>pair will be added to the result Spatial RDD. Because some objects span across multiple grids and some grids overlap with each other, an object may belong to multiple grids and hence the resulting SRDD may contain duplicates. To guarantee the query accuracy, GeoSpark spatial query processing layer handles this issue using two different techniques to remove duplicates (see Section 3.3.3).

Step 3: Re-partitioning SRDD across the cluster: The Spatial RDD generated by the last step already has <Key, Value>pair schema. The Key represents a grid cell ID. In this step, GeoSpark repartitions the Spatial RDD by Key and then spatial objects which have the same grid cell ID (Key) are grouped into the same partition. These partitions constitute a new Spatial RDD. This step results in massive data being shuffled across the cluster due to passing spatial objects to their assigned worker nodes.

3.2.4 SRDD Indexing

Spatial indexes such as R-Tree and Quad-Tree can speed up a spatial query significantly. That is due to the fact that such indexes group nearby spatial objects together and represent them with a tight bounding rectangle in the next higher level of the tree. A query that does not intersect with the rectangle cannot intersect with any of the objects in the lower levels.

Since many spatial analysis algorithms (e.g., spatial data mining, geospatial statistical learning) have to query the same Spatial RDD many times until convergence, GeoSpark allows the user to build spatial indexes and the built indexes can be cached, persisted and re-used many times. However, building a spatial index for the entire dataset is not possible because a tree-like spatial index yields additional 15% storage overhead [104, 102]. No single machine can afford such storage overhead when the data scale becomes large.

Build local indexes To solve the problem, if the user wants to use a spatial index, GeoSpark will build a set of local spatial indexes rather than a single global index. In particular, GeoSpark creates a spatial index (R-Tree or Quad-Tree) per RDD partition. These local R-Trees / Quad-Trees only index spatial objects in their associated partition. Therefore, this method avoids indexing all objects on a single machine.

To further speed up the query, the indexes in GeoSpark are clustered indexes. In that case, spatial objects in each partition are stored directly in the spatial index of this partition. Given a query, the clustered indexes can directly return qualified spatial objects and skip the I/O of retrieving spatial index according to the qualified object pointers.

Query local indexes When a spatial query is issued by the spatial query processing layer, the query is divided into many smaller tasks that are processed in parallel. In case a local spatial index exists in a certain partition, GeoSpark will force the spatial computation to leverage the index. In many spatial programs, the built indexes will be re-used again and again. Hence, the created spatial indexes may lead to a tremendous saving in the overall execution time and the index construction time can be amortized.

In addition, since spatial indexes organize spatial objects using their Minimum Bounding Rectangle (MBR) instead of their real shapes, any of the queries that leverage spatial indexes have to follow the Filter and Refine model [39, 32] (explained in Section 3.3): In the filter phase, I find candidate objects (MBR) that intersect with the query object (MBR); in the refine phase, I check the spatial relation between the candidate objects and the query object and only return the objects that truly satisfy the required relation (contain or intersect).

Persist local indexes To re-use the built local indexes, GeoSpark users first need to store the indexed spatial RDD using one of the following ways (DataFrame shares similar APIs) - (1) cache to memory: call IndexedSpatialRDD.cache() (2) persist on disk: call IndexedSpatialRDD.saveAsObjectFile(HDFS/S3_PATH). Both methods make use of the same algorithm: (1) go to each partition of the RDD in parallel (2) call SRDD customized serializer to serialize the local index on each partition to a byte array, one array per partition (see Section 3.2.5) (3) write the generated byte array of each partition to memory or disk. The user can directly use the name of the cached indexed SpatialRDD in his program because Spark manages the corresponding memory space and de-serializes byte arrays in parallel to recover the original RDD partition information. For an indexed SpatialRDD on disk, the user needs to call IndexedSpatialRDD.readFromObjectFile(HDFS/S3_PATH) to explicitly read it back to Spark. Spark will read the distributed file partitions in parallel and the byte array of each partition is de-serialized to a local index.

3.2.5 SRDD Customized Serializer

When Spark transfers objects across machines (e.g., data shuffle), all objects have to be first serialized in byte arrays. The receiver machines will put the received data chunk in memory and then de-serialize the data. Spark default serializer can provide a compact representation of simple objects (e.g., integers). However, for objects such as spatial objects that possess very complex geometrical shapes, the default Spark serializer cannot efficiently provide a compact representation of such objects [67]. That may lead to large-scale data shuffled across the network and tremendous memory overhead across the cluster.

To overcome this issue, GeoSpark provides a customized serializer for spatial objects and spatial indexes. The proposed serializer uses a binary format to serialize a spatial object and indexes. The serialized object and index are put in byte arrays.

The way to serialize a spatial object is as follows:

- **Byte 1** specifies the type of the spatial object. Each supported spatial object type has a unique ID in GeoSpark.

- **Byte 2** specifies the number of sub-objects in this spatial object.
- **Byte 3** specifies the type of the first sub-object (only needed for GeometryCollection, other types don't need this byte).
- **Byte 4** specifies the number of coordinates (n) of the first sub-object. Each coordinate is represented by two double type (8 bytes * 2) data X and Y.
- **Byte 5 - Byte 4+16*n** stores the coordinate information.
- **Byte 16*n+1** specifies the number of coordinates (n) of the second sub-object...
- **Until the end** Here all sub-objects have been serialized.

The way to serialize a single local spatial index (Quad-Tree or R-Tree, explained in subsection 3.2.4) is detailed below. It uses the classic N-ary tree serialization/deserialization algorithm. It is also worth noting that, spatial objects are stored inside tree nodes.

Serialization phase It uses the Depth-First Search (DFS) to traverse each tree node from the root following the pre-order strategy (first write current node information then write its children nodes). This is a recursive procedure. In the iteration of each tree node (each recursion), it first serializes the boundary of this node, and then serializes all spatial objects in this node one by one (use the object serializer explained above). Eventually, it goes to the children nodes of the working node. Since each N-ary tree node may have various internal spatial objects and children nodes, it also writes a memo to note the number of spatial objects and children nodes in this node.

De-serialization phase It still utilizes the same traverse strategy as the serialization phase (DFS, pre-order). It starts from the root and runs a recursive algorithm. In each recursion, it first re-constructs the boundary and internal spatial objects of the working node. Then it starts reading the bytes of the children nodes of the working node and hands over the work to the next recursion.

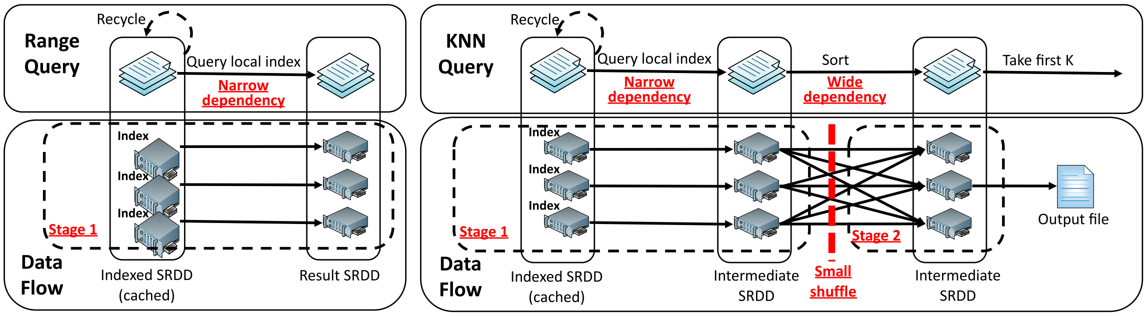


Figure 3: Spatial range query and KNN query DAG and data flow

Based on my experiments in subsection 3.5, GeoSpark serializer is faster than Spark kryo serializer and has smaller memory footprint when running complex spatial operations, e.g., spatial join query.

3.3 Spatial Query Processing Layer

After the Spatial RDD layer loads, partitions, and indexes Spatial RDDs, GeoSpark can run spatial query processing operations on the SRDDs. The spatial query proccessing layer provides support for a wide set of popular spatial operators that include range query, distance query, K Nearest Neighbors (KNN) query, range join query and distance join query.

3.3.1 Processing Spatial Range and Distance Queries

A spatial range query is fast and less resource-consuming since it only returns all the spatial objects that lie within the input query window object (point, polygon, line string and so on). Such a query can be completed by a parallelized Filter transformation in Apache Spark, which introduces a narrow dependency. Therefore,

Algorithm 2: Range query and distance query

Data: A query window A, a Spatial RDD B and spatial relation predicate
Result: A Spatial RDD that contains objects that satisfy the predicate

```
foreach partition in the SRDD B do
    if an index exists then
        // Filter phase
        Query the spatial index of this partition using the window A's MBR;
        // Refine phase
        Check the spatial relation predicate using real shapes of A and
          candidate objects;
    else
        foreach object in this partition do
            Check spatial relation predicate between this object and A;
            Record this object if it is qualified;
Generate the result Spatial RDD;
```

it is not necessary to repartition the Spatial RDD since repartitioning might lead to a wide dependency in the Apache Spark DAG. A more efficient way is to broadcast the query window to all worker nodes and parallelize the computation across the cluster. For non-closed query window objects such as a point or a line string, the range query processing algorithm only checks the "intersect" relation rather than "contain".

The spatial distance query conducts the same operation on the given Spatial RDD but adds an additional distance buffer between the query window and the candidate objects.

SQL API: Spatial predicates such as "ST_Contains" can be used to issue a range query in GeoSpark Spatial SQL. For instance, "ST_Contains (A, B)" returns true if A contains B. "ST_Distance (A, B)$\leq$ d" returns true if the distance between A and B is equal to or less than d. The following two Spatial SQL examples depict (1) return taxi trips that are picked up in Manhattan (2) return taxi trips that are picked up within 1 mile from Manhattan. The other two Scala/Java examples take as input a Spatial RDD and a query window object and then run the same operations.

```
/* Spatial SQL Range query */
SELECT *
FROM TaxiTripTable
WHERE ST_Contains(Manhattan, TaxiTripTable.pickuppoint)

/* Spatial SQL Distance query */
SELECT *
FROM TaxiTripTable
WHERE ST_Distance(Manhattan, TaxiTripTable.pickuppoint) <= 1

/* Scala/Java RDD Range query */
RangeQuery.SpatialRangeQuery(PickupPointRDD, Manhattan)

/* Scala/Java RDD Distance query */
RangeQuery.SpatialDistanceQuery(PickupPointRDD, Manhattan, 1)
```

Algorithm For a given spatial range query, GeoSpark broadcasts the query window to each machine in the cluster. For each Spatial RDD partition (see Algorithm 2), if a spatial index exists, it follows the Filter and Refine model: (1) uses the query window's MBR to query the spatial index and return the candidate results. (2) checks the spatial relation between the query window and candidate objects using their real shapes. The truly qualified spatial objects are returned as the partition of the new resulting Spatial RDD. If no spatial index exists, GeoSpark filters spatial objects using the query window and collects qualified objects to be a partition of the new result Spatial RDD. The result Spatial RDD is sent to the next stage of the Spark program (if needed) or persisted on disk. For a distance query, I added a distance buffer to the query window such that it extends the boundary of the query window to cover more area. The remaining part of distance query algorithm remains the same with range query.

DAG and iterative spatial data mining The DAG and data flow of the range query and the distance query are described in Figure 3. The query processing algorithm only introduces a single narrow dependency, which does not require data shuffle. Thus, all it needs is just one stage. For a compute-intensive spatial data mining program, which executes range queries many times (with different query windows), all queries access data from the same cached indexed Spatial RDD fluently without any interruptions from wide dependencies so that the procedure is very fast.

3.3.2 Spatial K Nearest Neighbors (KNN) Query

The straightforward way to execute a KNN query is to rank the distances between spatial objects and the query location, then pick the top K nearest neighbors. However, ranking these distances in a large SRDD should be avoided if possible to avoid a large amount of data shuffle, which is time-consuming and bandwidth-consuming. In addition, it is also not necessary to spatially partition a Spatial RDD that incurs a single wide dependency.

SQL API: In Spatial SQL, "ST_Neighbors(A, B, K)" issues a spatial KNN query which finds the K nearest neighbors of A from Column B. The SQL example below returns the 100 nearest taxi trip pickup points of New York Time Square. The Scala/Java example performs the same operation.

```
/* Spatial SQL API */
SELECT ST_Neighbors(TimeSquare, TaxiTripTable.pickuppoint, 100)
FROM TaxiTripTable
```

```
/* Scala/Java RDD API */
KnnQuery.SpatialKnnQuery(PickupPointRDD, TimeSquare, 100)
```

Algorithm To parallelize a spatial KNN query more efficiently, GeoSpark modifies a popular top-k algorithm [16] to fit the distributed environment (1) to be able to leverage local spatial indexes if they exist (2) to reduce the data shuffle scale of ranking distances. This algorithm takes an indexed/non-indexed SRDD, a query center object (point, polygon, line string and so on) and a number K as inputs. It contains two phases (see Algorithm 3): selection and sorting.

- **Selection phase** For each SRDD partition, GeoSpark calculates the distances from the given object to each spatial object, then maintains a local priority queue by adding or removing objects based on the distances. Such a queue contains the nearest K objects around the given object and becomes a partition of the new intermediate SRDD. For the indexed Spatial RDDs, GeoSpark can query the local indexes (only R-Tree supports this, see [78]) in partitions to accelerate the distance calculation. Similarly, GeoSpark needs to follow Filter and Refine model to recheck the results returned by the index search using their real shapes.
- **Sorting phase** Each partition in the Spatial RDD generated by the selection phase only contains K objects. GeoSpark sorts this intermediate Spatial RDD in ascending order according to the distances. Sorting the small scale intermediate Spatial RDD is much faster than sorting the original Spatial RDD directly. The sorting phase also outputs an intermediate Spatial RDD. GeoSpark collects the first K objects in the intermediate Spatial RDD across the cluster and returns those objects as the final result.

Algorithm 3: K nearest neighbor (KNN) query

```
Data: A query center object A, a Spatial RDD B, the number K
Result: A list of K spatial objects
/* Step 1: Selection phase                                    */
foreach partition in the SRDD B do
    if an index exists then
        Return K nearest neighbors of A by querying the index of this partition;
    else
        foreach object in this partition do
            Check the distance between this object and A;
            Maintain a priority queue that stores the top K nearest neighbors;
/* Step 2: Sorting phase                                      */
Sort the spatial objects in the intermediate Spatial RDD C based on their
  distances to A;
Return the top K objects in C
```

DAG and iterative spatial data mining Figure 3 depicts the DAG and data flow of spatial KNN query. The query processing algorithm includes two transformations: selection, sorting. The former incurs a narrow dependency and the latter introduces a wide dependency that results in a small data shuffle. These two transformations will be scheduled to two stages without pipeline execution. However, because the intermediate Spatial RDD generated by the first transformation only has K objects per partition, the shuffle caused by transforming this Spatial RDD is very small and will not impact the execution much. For an iterative spatial data mining using KNN query, the two intermediate Spatial RDDs are dropped after each query execution but the indexed Spatial RDD still resides in memory cache. Recycling the indexed Spatial RDDs accelerates the iterative execution to a greater extent.

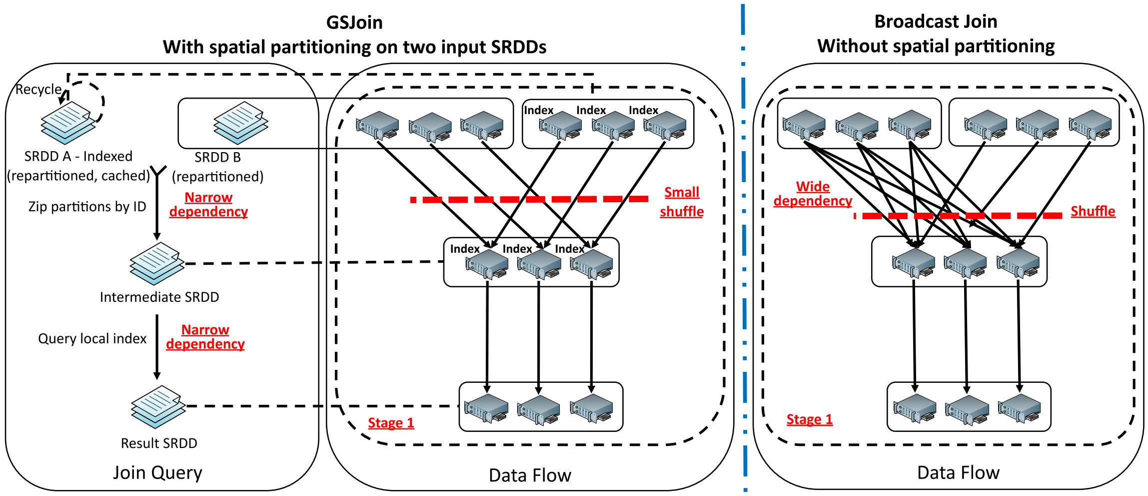


Figure 4: Join query DAG and data flow

3.3.3 Processing Spatial Join Queries

Spatial join, a computation and data intensive operation, incurs high data shuffle in Spark. Taking into account the spatial proximity of spatial objects, GeoSpark partitions Spatial RDDs in advance based on the objects' spatial locations in Spatial RDD layer (Section 3.2.3) and caches the Spatial RDDs. Therefore, GeoSpark join query algorithm re-uses the spatial partitioned RDDs (probably indexed as well) and avoids large scale data shuffle. Moreover, since GeoSpark skips the partitions which are guaranteed not to satisfy the join query predicate, it can significantly accelerate the overall spatial join processing.

***GSJoin* algorithm** GeoSpark combines the approaches proposed by [112, 62, 110] to a new spatial range join algorithm, namely *GSJoin*, that re-uses spatial partitioned RDDs as well as their indexes. This algorithm, which takes two spatial partitioned

RDDs A and B (i.e., TaxiStopStations and TaxiTripTable), consists of three steps as follows (see Algorithm 4 and Figure 4).

- **Zip partitions** This step zips the partitions from A and B (TaxiStopStations and TaxiTripTable) according to their grid IDs. For instance, I merge Partition 1 from A and B to a bigger partition which has two sub-partitions. Both Partition 1 s from A and B have the spatial objects that fall inside Grid 1 (see Section 3.2.3). Partition 1 from A (TaxiStopStations) contains all taxi stop stations that locate in Grid 1 and Partition 1 from B (TaxiTripTable) contains all taxi trips that are picked up in Grid 1. Note that, the data in Partition1 from A is guaranteed to disjoint from other partitions (except 1) of B because they belong to totally different spatial regions (see Section 3.2.3). Thus, *GSJoin* does not waste time on checking other partitions from B with Partition 1 from A. This Zip operation applies to all partitions from A and B and produces an intermediate RDD called C.
- **Partition-level local join (no index)** This step runs a partition-level local range join on each partition of C. Each partition from C has two sub-partitions, one from A and one from B. If no indexes exist on both the sub-partitions, the local join will perform a nested loop join that traverses all possible pairs of spatial objects from the two sub-partitions and returns the qualified pairs. This costs O(n^2) complexity on each C partition, where n is the number of objects in a sub-partition.
- **Partition-level local join (with index)** During the partition-level local join step, if an index exists on either one sub-partition (say, sub-partition from B is indexed), this local join will do an index-nested loop. It uses each object in the sub-partition from A as the query window to query the index of the

sub-partition from B. This costs O(n*log(n)) complexity on each C partition, where n is the number of objects in a sub-partition. It is worth noting that this step also follows the Filter and Refine model which is similar to this part in Range query and Distance query (mentioned in Section 3.2.4). Each query window needs to recheck the real shapes of candidate spatial objects which are obtained by scanning the index.

- **Remove duplicates** This step removes the duplicated spatial objects introduced by spatial partitioning. At that time, I duplicate the spatial objects that intersect with multiple grids and assign different grid IDs to these duplicates and this will lead to duplicated results eventually. Figure 5 illustrates an example. Since both Pa and Pb fall in Grid 1, 2, 3, and 4, the result "Pa intersects Pb" will be reported four times in the final join result set. In order to remove the duplicates, two methods are available. The first method is to use a "GroupBy" operation to collect all objects that intersect with Pa in this cluster then remove the duplicated ones. This method introduces a big data shuffle across the cluster because it needs to do a GroupBy on all results. I should always avoid unnecessary data shuffle. The second method is called "Reference point" [25]. The intuition of the reference point is to establish a rule that, if duplicated results appear, only report it once. When doing a partition-level local join, GeoSpark calculates the intersection of two intersecting objects (one from SRDD A and the other from B) and only reports the pair of objects when the reference point falls in the associated grid of this partition. To find this reference point, I first calculate the intersection shape of the two intersected objects. The X-coordinate/Y-coordinate of a reference point is the max X/Y of all coordinates of the intersection. In Figure 5, I use the red point as the

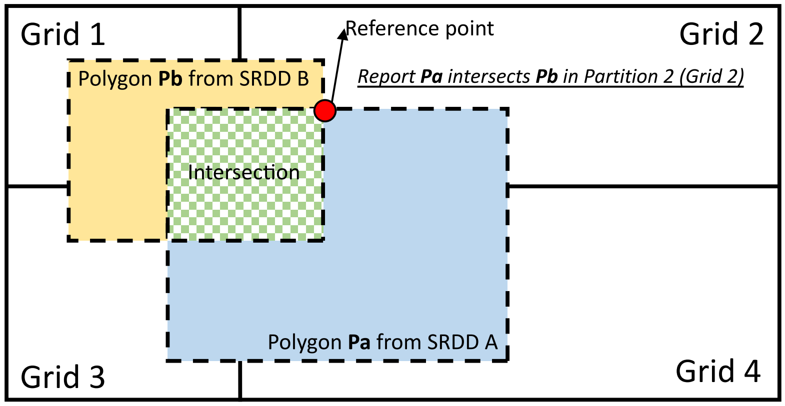


Figure 5: Removing duplicates

reference point and only report "Pa intersects Pb" when doing the local join on Partition 2. Note that, the reference point idea only works for Quad-Tree and KDB-Tree partitioning because their grids don't overlap with each other. R-Tree and other methods that produce overlapped grids have to use the first method because even the reference point can still appear in multiple grids.

Note that the *GSJoin* algorithm can query Spatial RDDs that are partitioned by any GeoSpark spatial partitioning method including Uniform grids, R-Tree grids, Quad-Tree grids, KDB-Tree grids and indexed by any of the indexing methods such as R-Tree index and Quad-Tree index. That is why I also implemented other partitioning methods and indexes in GeoSpark for benchmarking purpose.

Broadcast join algorithm Besides *GSJoin*, GeoSpark also provides a straightforward broadcast range join algorithm, which works well for small scale Spatial RDDs. When at least one of the two input Spatial RDDs is very small, this algorithm broadcasts the small Spatial RDD A to each partition of the other Spatial RDD B. Then, a partition-level local join (see *GSJoin*) happens on all partitions of B in parallel. The Broadcast join algorithm has two important features: (1) it is faster than *GSJoin*

Algorithm 4: *GSJoin* algorithm for range join and distance join query

```
Data: (repartitioned) SRDD A and (repartitioned) SRDD B
Result: PairRDD in schema <Left object from A, right object from B>
/* Step1: Zip partitions                                          */
foreach partition pair from SRDD A and B with the same grid ID i do
    Merge two partitions to a bigger partition that has two sub-partitions;
Return the intermediate SRDD C;
/* Step2: Run partition-level local join                          */
foreach partition P in the C do
    foreach object O_A in the sub-partition from A do
        if an index exists in the sub-partition from B then
            // Filter phase
            Query the spatial index of this partition using the O_A's MBR;
            // Refine phase
            Check the spatial relation using real shapes of O_A and candidate
              objects O_Bs;
            /* Step3: Remove duplicates                               */
            Report <O_A, O_B> pair only if the reference point of this pair is in
              P;
        else
            foreach object O_B in the sub-partition from B do
                Check spatial relation between O_A and O_B;
                /* Step3: Remove duplicates                           */
                Report <O_A, O_B> pair only if the reference point of this pair is
                  in P;
Generate the result PairRDD;
```

for very small datasets because it does not require any spatial partitioning methods and duplicate removal. (2) it may lead to system failure or very long execution time for large datasets because it shuffles an entire SRDD A to each partition of SRDD B.

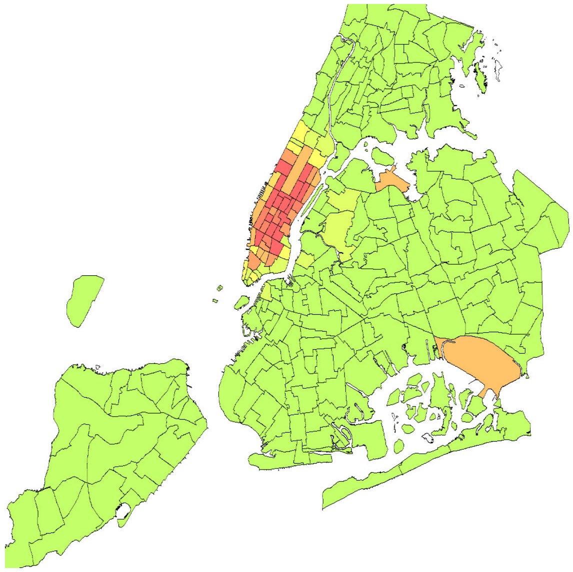
Distance join algorithm The distance join algorithms can be seen as extensions to the range join algorithms, *GSJoin* and Broadcast join. The only difference is that I add a distance buffer to all objects in either Spatial RDD A or B at the very beginning (even before spatial partitioning) to extend their boundaries. The extended spatial objects can be used in both range join algorithms.

Spark DAG: The DAG and data flow of *GSJoin* are shown in Figure 4. The join query introduces two transformations: zip partitions and partition-level local join. Both of them incur narrow dependencies. Therefore, they are pipelined to one stage with fast execution. On the contrary, the broadcast join algorithm (see the right part in Figure 4) introduces two wide dependencies which lead to heavy network traffic and intensive computation. Until now, the effect of spatial partitioning is shown by degrading wide dependencies to narrow dependencies. For spatial analytics program that runs multiple join queries, the repartitioned A and B are cached into memory and recycled for each join query.

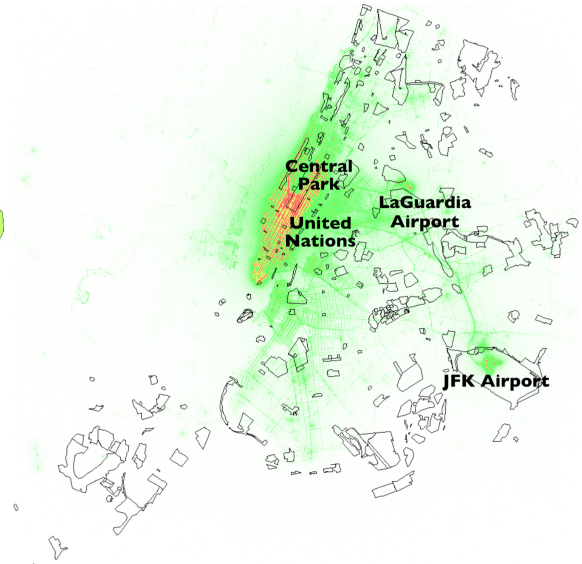
3.4 Application Use Cases

3.4.1 Application 1: Spatial Aggregation

A data scientist in NYC Taxi Company would like to further investigate the distribution of taxi pickup points because he makes an interesting observation from the Region Heat Map: the distribution is very unbalanced. New York City has many taxi zones and each zone may have very different demographic information. Therefore, this time, the scientist wants to know the pickup points distribution per taxi zone. This spatial aggregation can be easily completed by GeoSpark: He first uses `Range Join Query` (SQL interface) to find the taxi trips that are picked up in each taxi zone then calculates the count per taxi zone. The result of this aggregation can be directly visualized by the GeoSpark Choropleth Map. The corresponding pseudo code is available in Figure 7. The SQL statement issues a range join query since the



(a) Spatial Aggregation: Trip pickup points aggregated by taxi zones



(b) Spatial Co-location: Taxi trips co-locate with area landmarks

Figure 6: Visualized spatial analytics

```
var aggregatedDataframe = spark.sql
("SELECT TaxiZone.boundary, COUNT(TaxiTripTable.pickuppoint)
  FROM TaxiZone,TaxiTripTable
  WHERE ST_Contains(TaxiZone.boundary, TaxiTripTable.pickuppoint)
  GROUP BY TaxiZone.boundary")
var vizEffect = new ChoroplethMap(resolutionX, resolutionY)
vizEffect.Visualize(aggregatedDataframe.toRDD)
```

Figure 7: Spatial Aggregation: Range Join Query + Choropleth Map

inputs of “ST_Contains” are from two datasets. The “COUNT()” function counts the taxi trip pickup points per taxi zone. Each row in the join query result is in the form of “TaxiZoneShape,Count”. The Choropleth Map API takes as input the map resolution and its “Visualize” function can plot the given join query result. Figure 6a is the generated Choropleth Map. A red colored zone means that more taxi trips were picked up in that zone. According to the Choropleth map, the scientist finds that the hottest zones are in Manhattan but there are two zones which are in orange colors

```
val PickupPoints = new PointRDD(sparkContext,DataPath1,FileType.CSV)
val AreaLandmarks = ShapefileReader.readToGeometryRDD(DataPath2)

PickupPoints.spatialPartitioning(GridType.KDBTREE)
PickupPoints.buildIndex(IndexTYpe.QuadTree)
PickupPoints.indexedRDD.cache()

for (i <- 1 to 10) {
currentDistance = beginDistance + i*distanceIncrement
AreaLandmarks.spatialPartitioning(PickupPoints.getPartitioner,
    currentDistance)
var adjacencyMatrixCount = JoinQuery.DistanceJoinQueryFlat (
    sparkContext,
  PickupPoints, AreaLandmarks, currentDistance).count
println(RipleyK(coefficient, adjacencyMatrixCount))}
```

Figure 8: Spatial Co-location Pattern Mining: iterative Distance Join Query

but far from Manhattan. Then, he realizes these two zones are La Guardia Airport and JFK Airport, respectively.

3.4.2 Application 2: Spatial Co-location Pattern Mining

After finding many pickup points in La Guardia Airport and JFK Airport, the data scientist in the NYC Taxi Company makes a guess that the taxi pickup points are co-located with the New York area landmarks such as airports, museums, hospitals, colleges and so on. In other words, many taxi trips start from area landmarks (see Figure 6b). He wants to use a quantitative metric to measure the degree of the co-location pattern. This procedure is called spatial co-location pattern mining.

Spatial co-location pattern mining is defined by two kinds of spatial objects that are often located in a neighborhood relationship. Ripley's K function [93] is commonly used in judging co-location. It usually executes multiple times and forms a

2 dimensional curve for observation. To obtain Ripley's K in each iteration, I need to calculate the adjacency matrix of two types of spatial objects given an updated distance restriction. To obtain the adjacency matrix, a time-consuming distance join is necessary. The user can use GeoSpark RDD APIs to assemble this application[6] and migrate the adjacency matrix computation to a Spark cluster.

A snippet of the application source code is given in Figure 8. The user first needs to create two Spatial RDDs, PickupPoints from a CSV file and AreaLandmarks from an ESRI shapefile. Then he should run spatial partitioning and build local index on the larger Spatial RDD (PickupPoints RDD is much larger in this case) and cache the processed Spatial RDD. Since Ripley's K requires many iterations (say, 10 iterations) with a changing distance, the user should write a for-loop. In each loop, he just needs to call `DistanceJoinQuery` API to join PickupPoints (cached) and AreaLandmarks. `DistanceJoinQuery` takes as input two Spatial RDDs and a distance and returns the spatial objects pairs that are located within the distance limitation. Although the distance is changing in each loop, the cached PickupPoints can be quickly loaded from memory to save plenty of time. The experiment verifies this conclusion.

3.5 Experiments

This section presents a comprehensive experiment analysis that experimentally evaluates the performance of GeoSpark and other spatial data processing systems. I compare four main systems:

[6]Runnable example: https://github.com/jiayuasu/GeoSparkTemplateProject

Dataset	Size	Description
OSMpostal	1.4 GB	171 thousand polygons, all postal areas on the planet (from Open Street Map)
TIGERedges	62 GB	72.7 million line strings, all streets in US. Each street is a line string which consists of multiple line segments
OSMobject	90 GB	263 million polygons, all spatial objects on the planet (from Open Street Map)
NYCtaxi	180 GB	1.3 billion points, New York City Yellow Taxi trip information

Table 3: Dataset description

- GeoSpark: I use GeoSpark 1.0.1, the latest release. It includes all functions needed in this experiment. I also open GeoSpark customized serializer to improve the performance.
- Simba [98]: I use Simba's latest GitHub repository which supports Spark 2.1. By default, Simba automatically opens Spark Kryo serializer.
- Magellan [82]: Magellan 1.0.6, the latest version, is used in my experiment. By default, Magellan automatically opens Spark Kryo serializer.
- SpatialHadoop [29]: I use SpatialHadoop 2.4.2 in the main GitHub repository in the experiments.

Datasets. Table 3 summarizes four real spatial datasets used in the experiments, described as follows:

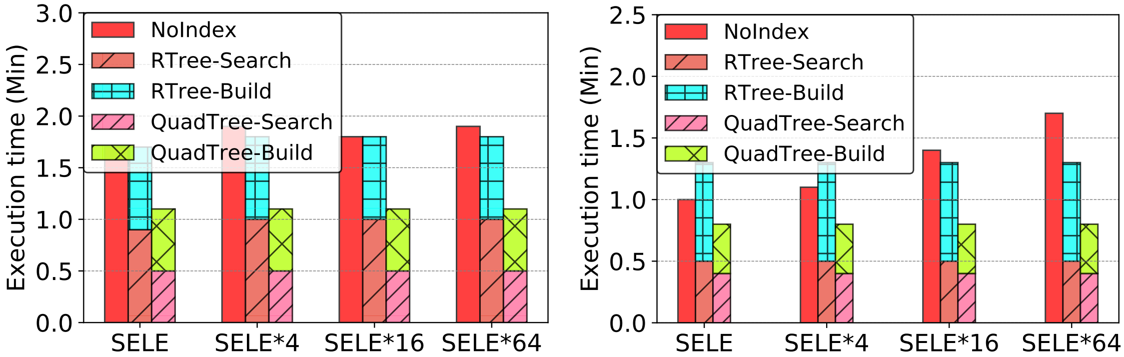
- OSMpostal [43]: contains 171 thousand polygons extracted from Open Street Maps. These polygons are the boundaries of all postal code areas on the planet.
- TIGERedges [88]: contains 72.7 million line strings provided by United States Census Bureau TIGER project. These line strings are all streets in the United States. Each of them consists of many line segments.
- OSMobject [66]: includes the polygonal boundaries of all spatial objects on the planet (from Open Street Maps). There are 263 million polygons in this dataset.

- NYCtaxi [85]: The dataset contains the detailed records of over 1.1 billion individual taxi trips in the city from January 2009 through December 2016. Each record includes pick-up and drop-off dates/times, pick-up and drop-off precise location coordinates, trip distances, itemized fares, and payment methods. But I only use the pick-up points in my experiment.

Workload. I run the following spatial queries on the evaluated systems. Distance query and distance join results are not stated in the evaluation because they follow similar algorithms as the range query and the range join query, respectively.

- Spatial Range query: I run spatial range queries on NYCtaxi, OSMobject and TIGERedges with different query selectivities. Simba can only execute range queries on points and polygons without index. Its local index construction fails on points and polygons due to extremely high memory utilization. Magellan does not support accurate queries on polygons and line strings.
- Spatial KNN query: I test the spatial KNN query by varying K from 1 to 1000 on NYCtaxi, OSMobject and TIGERedges datasets. Only GeoSpark and SpatialHadoop support KNN query. Simba offers a KNN API but doesn't return any results.
- Spatial Range join query ($\bowtie$): I run OSMpostal $\bowtie$ NYCtaxi, OSMpostal $\bowtie$ OSMobject, OSMpostal $\bowtie$ TIGERedges. Magellan only supports the first case (polygons $\bowtie$ points). Simba only offers distance join between points and points (explained later).

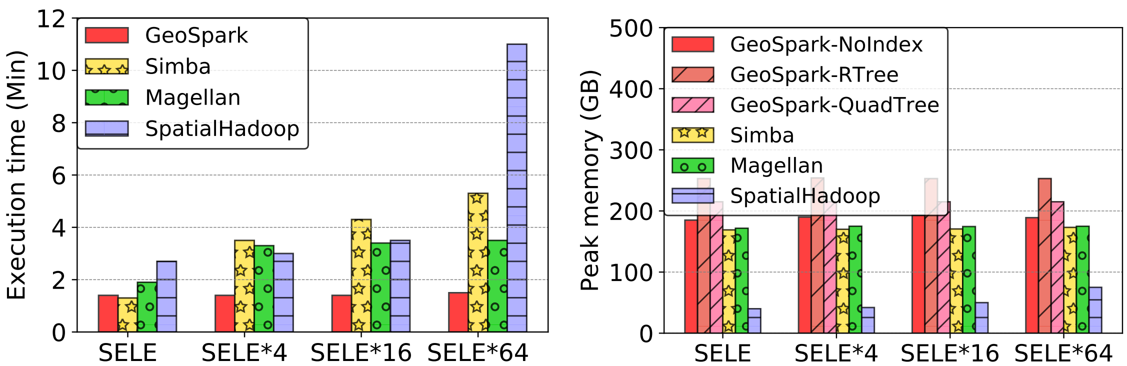
Cluster settings. I conduct the experiments on a cluster which has one master node and four worker nodes. Each machine has an Intel Xeon E5-2687WV4 CPU (12 cores, 3.0 GHz per core), 100 GB memory, and 4 TB HDD. I also install Apache



(a) OSMobject range query

(b) TIGERedges range query

Figure 9: The impact of GeoSpark local index on complex shapes range query



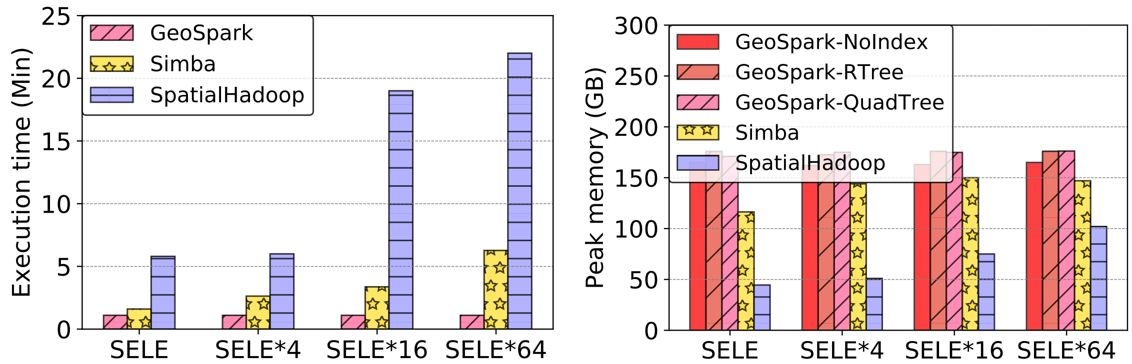
(a) Execution time

(b) Peak memory utilization

Figure 10: Range query with different selectivity (SELE) on NYCtaxi points

Hadoop 2.6 and Apache Spark 2.1.1. I assign 10 GB memory to the Spark driver program that runs on the master machine, which is quite enough to handle any necessary global computation.

Performance metrics. I use two main metrics to measure the performance of the evaluated systems: (1) Execution time: It stands for the total run time the system takes to execute a given job. (2) Peak execution memory: that represents the highest execution memory used by the system when running a given job.



(a) Execution time

(b) Peak memory utilization

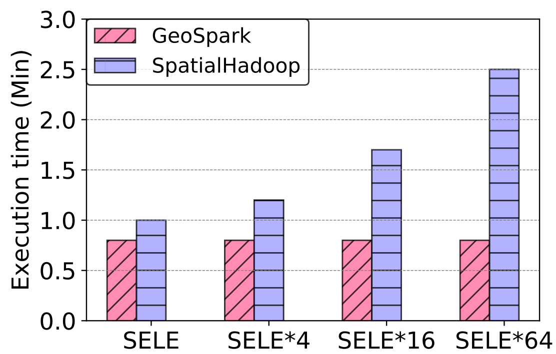
Figure 11: Range query with different selectivity (SELE) on OSMobject polygons

3.5.1 Performance of Range Query

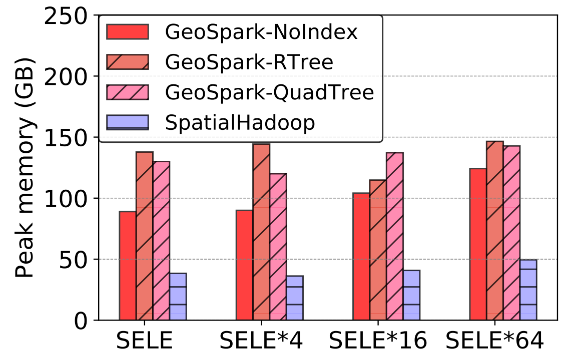
This section evaluates the performance of all four systems on NYCtaxi points, OSMobject polygons and TIGERedges linestrings (see Figure 10, 11, 12). I vary the range query selectivity factor as follows: (1) I use the entire dataset boundary as the largest query window.(2) then, I reduce the window size to its $\frac{1}{4}$th and generate many rectangular windows in this size but at random locations within the dataset boundary. (3) I keep reducing the window size until I get four different query selectivity factors (window size): $\frac{1}{64}$*boundary (SELE*1), $\frac{1}{16}$*boundary, $\frac{1}{4}$*boundary, boundary (SELE*64).

Impact of GeoSpark local indexing For the NYCtaxi point dataset, the fastest GeoSpark method is GeoSpark range query without index (Figure 13). I created different versions of GeoSpark, as follows: (1) NoIndex: GeoSpark running with no local indexes built in each SRDD partition. (2) RTree: GeoSpark running with R-Tree index built on each local SRDD partition. RTree-Search represents the R-tree search time, whereas RTree-Build denotes the R-Tree construction time. (3) QuadTree: Similar to RTree but with Quad-Tree index stored in each local SRDD partition instead. The NoIndex version of GeoSpark has similar range query search time as the indexed version of GeoSpark but the indexed (i.e., RTree or QuadTree) range query needs extra time to build the index. This is because the NYCtaxi point data is too simple and using index to prune data does not save much computation time. The RTree version even has around 5% longer search time than the NoIndex version because GeoSpark follows the Filter and Refine model: after searching local indexes using MBRs, GeoSpark rechecks the spatial relation using real shapes of query window and spatial objects in order to guarantee the query accuracy (the window can be a very complex polygon rather than a rectangle). As it turns out in Figure 9a and 9b (OSMobject and TIGERedges), GeoSpark RTree leads to 2 times less search time than the NoIndex version. GeoSpark QuadTree exhibits 4 times less search time than the NoIndex version. This makes sense because the tested polygon data and line string data have very complex shapes. For instance, a building's polygonal boundary (OSMobject) and a street's shape (TIGERedges) may have more than 20 coordinates. A local index in GeoSpark can prune lots of useless data to save the computation time while the regular range query needs to check all complex shapes in each partition across the cluster.

Comparing different systems I show the range query execution times of the four systems in Figure 10a, 11a and 12a. Simba can run range query on NYCtaxi points and OSMobject polygons without index. I do not use Simba R-Tree index range query because it always runs out of memory on these datasets even though the tested cluster has 400GB memory. Magellan and SpatialHadoop do not use indexes in processing a range query by default. I do not show the results for Magellan on OSMobject polygons and TIGERedges linestrings because it only uses MBRs and hence cannot return accurate query results. I use the most optimized GeoSpark in



(a) Execution time



(b) Peak memory utilization

Figure 12: Range query with different selectivity (SELE) on TIGERedges line strings

the comparison: GeoSpark range query without index in Figure 10a and QuadTree in Figures 11a and 12a.

For NYCtaxi point data (Figure 10a), GeoSpark shows the least execution time. Furthermore, its execution time is almost constant on all query selectivities because it finishes the range query almost right after loading the data. On highly selective queries (ie.e., SELE*1), GeoSpark has similar execution time with Simba and Magellan. Moreover, the execution times of Simba, Magellan and SpatialHadoop increase with the growth of the query window size. On SELE*4, *16 and *64, Simba and Magellan are 2-3 times slower than GeoSpark. SpatialHadoop is 2 times slower than GeoSpark on SELE*1 and 10 times slower than GeoSpark on SELE*64.

For OSMobject polygons and TIGERedges linestrings (Figure 11a and 12a), GeoSpark still has the shortest execution time. On OSMobject polygons, the QuadTree version of GeoSpark is around 2-3 times faster than Simba and 20 times faster than SpatialHadoop.

GeoSpark outperforms its counterparts for the following reasons: (1) on polygon and line string data, GeoSpark's Quad-Tree local index can speed up the query by

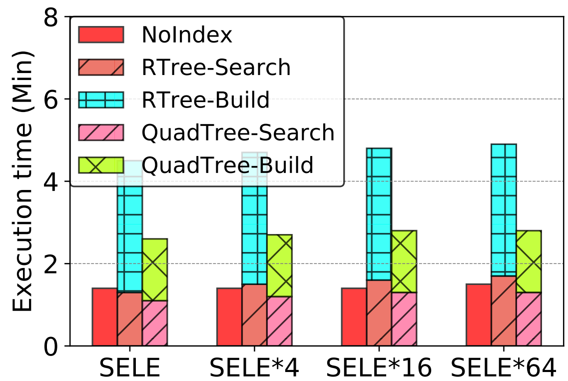
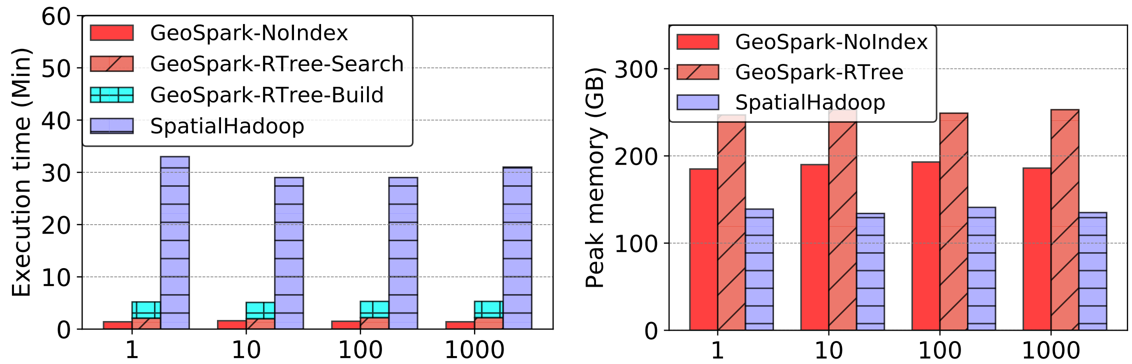


Figure 13: The impact of GeoSpark local index on NYCtaxi range query

pruning a large amount of data. (2) GeoSpark serializer overrides the default Spark generic serializer to use my own spatial data serialization logic. According to range query's DAG (see Figure 3), Spark performs RDD transformations to produce the result RDD and internally calls the serializer. GeoSpark serializer directly tells Spark how to understand and serialize the spatial data quickly while the default Spark generic serializer wastes some time on understanding complex spatial objects. Although the GeoSpark serializer is faster than the Spark default serializer, it is worth noting that in order to support heterogeneous spatial objects in a single Spatial RDD, GeoSpark customized serializer costs some extra bytes to specify the geometry type per spatial object besides the coordinates (see Section 3.2.5): (1) 3 extra bytes on point objects (15% additional memory footprint) (2) 2+n extra bytes on polygons or line strings, where n is the number of coordinates (7% addition memory footprint, if n = 10). My experiment verifies the theoretical values. Figures 10b, 11b and 12b illustrate the peak memory utilization of different systems. Since the range query processing algorithm is similar on evaluated systems (a filter operation on all data partitions), the peak memory utilization is roughly equal to the input data memory footprint. Compared to Simba and Magellan, GeoSpark costs around 10% additional memory on



(a) Time (NYCtaxi)

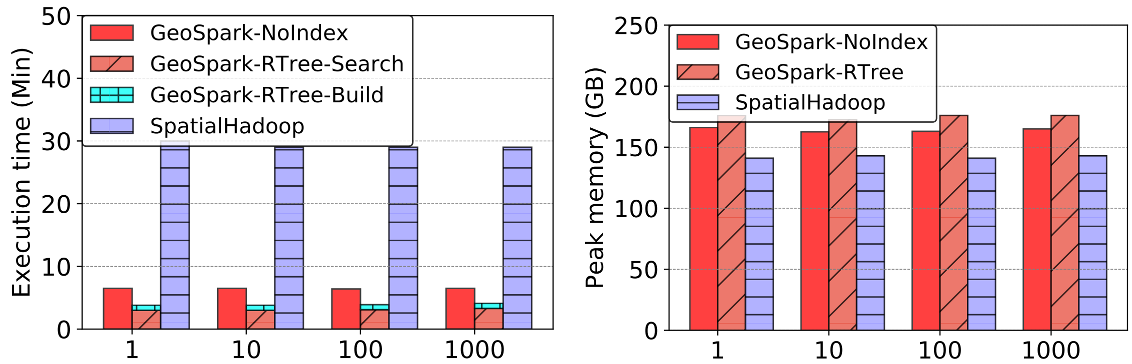
(b) Mem.(NYCtaxi)

Figure 14: KNN query with different K on NYCtaxi points

points and polygons (Figure 10b and 11b). GeoSpark Quad-Tree index range query costs 13% additional memory and R-Tree index range query costs 30% additional memory. This is because a tree index takes 10%-40% additional space to store the tree node information [104, 102]. In general, R-Tree consumes more space because it needs to store MBRs and extra child node information while Quad-Tree always has 4 child nodes. SpatialHadoop always has 2-3 times less memory utilization because Hadoop-based systems don't utilize memory much and all intermediate data is put on the disk.

3.5.2 Performance of K Nearest Neighbors (KNN) query

This section studies the performance of GeoSpark and SpatialHadoop. SpatialHadoop does not use an index for KNN queries by default. Simba offers a KNN API but the source code does not return any results on the tested datasets. I vary K to take the values of 1, 10, 100 and 1000 and randomly pick several query points within the dataset boundaries.



(a) Time (OSMobject)

(b) Mem.(OSMobject)

Figure 15: KNN query with different K on OSMobject polygons

Since the KNN query shows similar performance trends as the range query, I only put the results of KNN on NYCtaxi points and OSMobject polygons. R-Tree [39, 78] data structure supports KNN because it uses MBR to represent tree node boundaries.

As depicted in Figures 14a and 15a, on NYCtaxi points, GeoSpark NoIndex is $\tilde{2}0$ times faster than SpatialHadoop and GeoSpark RTree is $\tilde{6}$ times faster than SpatialHadoop. On OSMobject polygons, GeoSpark NoIndex is 5 times faster than SpatialHadoop and the RTree version of GeoSpark is 7 times faster than SpatialHadoop in terms of total execution time including the index building time. On point data, GeoSpark NoIndex shows similar search time performance as the indexed version (explained in Section 3.5.1).

The execution time of a KNN query in GeoSpark and SpatialHadoop remains constant with different values of K. That happens because the value of K is very small in contrast to the input data and the majority of the time is spent on processing the input data.

The peak memory utilization of GeoSpark is $\tilde{2}$ times higher than SpatialHadoop because Spark stores intermediate data in main memory. It is also worth noting that, although GeoSpark's peak memory utilization remains constant in processing range

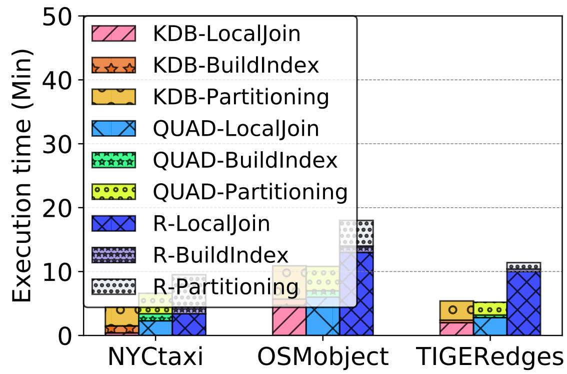


Figure 16: The impact of spatial partitioning on range join in GeoSpark

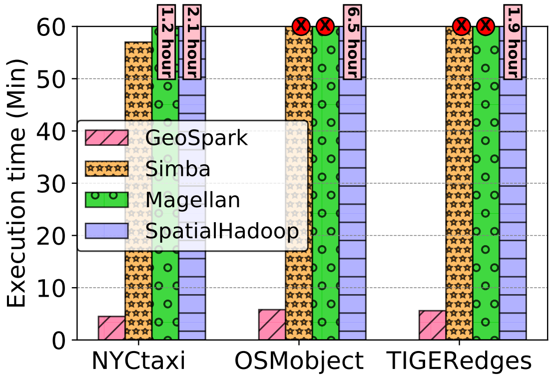
and KNN queries, SpatialHadoop's peak memory utilization of KNN queries is 3 times larger than that of range queries because SpatialHadoop needs to sort the candidate objects across the cluster, which leads to a data shuffle across the network. In practice, heavy network data transfer increases memory utilization.

3.5.3 Performance of Range Join Query

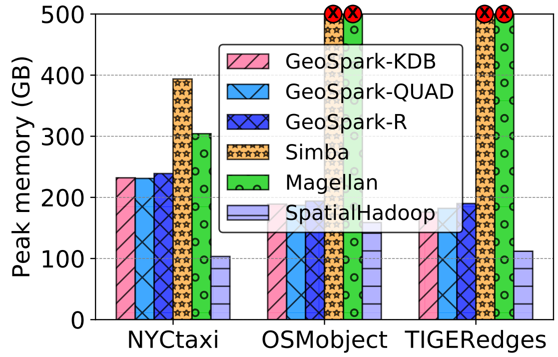
This section evaluates the range join query performance of compared systems when joining the following datasets: (a) OSMpostal $\bowtie$ NYCtaxi, (b) OSMpostal $\bowtie$ OSMobject, (c) OSMpostal $\bowtie$ TIGERedges.

Impact of spatial partitioning. As depicted in Figure 16, GeoSpark KDB-Tree partitioning method exhibits the shortest local join time (KDB-LocalJoin) on all join queries. Quad-Tree partitioning local join time (QUAD-LocalJoin) is 1.5 times slower than KDB-LocalJoin. R-Tree partitioning local join time (R-LocalJoin) is around 2 times slower than KDB-Tree partitioning. This is because KDB-Tree partitioning generates more load-balanced grid cells. For instance, on OSMpostal polygons $\bowtie$ NYCtaxi points, during the spatial partitioning step (see the DAG and data flow of GSJoin in Figure 4): (1) Shuffled serialized data across the cluster by GeoSpark are 12.9GB (KDB), 13.2GB (QUAD) and 13.4GB (R) (2) The min, median and max shuffled SRDD partition sizes are 4MB-6.4MB-8.8MB (KDB), 1MB-3.2MB-10.4MB (QUAD), 5MB-9.3MB-103MB (R). Obviously, the partition size of KBD-Tree partitioning is more balanced. Quad-Tree method is not as balanced as KDB-Tree. R-Tree partitioning has an overflow data partition, which is much larger than other partitions because R-Tree does not usually cover the entire space. According to the example given above, it makes sense that GeoSpark KDB-Tree spatial partitioning has the least local join time and R-Tree partitioning has the slowest local join speed. Another factor that slows down R-Tree partitioning local join is the additional data shuffle step resulting from removing duplicates among overlapped partitions (see Section 3.3.3).

Comparing different systems. Magellan only supports OSMpostal $\bowtie$ NYCtaxi (polygons, points) using Z-Curve spatial partitioning. Simba only supports distance join between points and points. In order to make Simba work with OSMpostal $\bowtie$ NYCtaxi, I take the central point of each postal area in OSMpostal to produce a point dataset and use the average radius of OSMpostal polygon as distance to run the distance join. By default, Simba uses R-Tree spatial partitioning and builds local R-Tree indexes on the smaller dataset of the join query. SpatialHadoop uses Quad-Tree spatial partitioning by default and it also builds Quad-Tree local index on NYCtaxi, OSMobject and TIGERedges. GeoSpark uses three different spatial partitioning methods, KDB-Tree (KDB), Quad-Tree (QUAD), and R-Tree (R) (these partitioning methods are not GeoSpark local indexes). Based on the performance of GeoSpark range query and KNN query, I build GeoSpark Quad-Tree index on



(a) Execution time



(b) Peak memory utilization

Figure 17: Range join query performance on datasets $\bowtie$ OSMpostal

NYCtaxi, OSMobject and TIGERedges by default. The experimental result is shown in Figure 17.

In Figure 17a, I compare GeoSpark KDB-Tree partitioning join query with Quad-Tree local index, Simba, Magellan and SpatialHadoop. As I can see from this figure, in comparison with GeoSpark, Simba is around 10 times slower (OSMpostal $\bowtie$ NYCtaxi), Magellan is 15 times slower (OSMpostal $\bowtie$ NYCtaxi) and SpatialHadoop is more than 25 times slower on all join query scenarios. The execution time contains all parts in a join query including spatial partitioning, local index building and local join. A red cross means that this join data type is not supported by this system.

Simba exhibits higher join time for the following reasons: (1) it uses R-Tree partitioning. As explained above, R-Tree partitioning is not as balanced as KDB-Tree partitioning. (2) GeoSpark customized serializer tells Spark how to serialize spatial data exactly. Simba uses the default Spark kryo serializer which produces many unnecessary intermediate data when serializing and shuffling spatial data. (3) Simba's join algorithm shuffles lots of data across the cluster. Based on my test, Simba shuffles around 70GB data while GeoSpark only shuffles around 13GB.

GeoSpark outperforms Magellan because (1) Magellan uses Z-Curve spatial partitioning which leads to many overlapped partitions [27]. Using Z-Curve to index spatial objects loses lots of spatial proximity information. Thus, this forces the system to put lots of spatial objects that are impossible to intersect together. This wastes a significant amount of execution time and introduces a huge network data transfer. Based on my test, Magellan reads 623GB data through the network during the join query execution. (2) Magellan doesn't support any spatial tree index like R-Tree or Quad-Tree. (3) Magellan doesn't have customized serializer like GeoSpark.

SpatialHadoop is slow because SpatialHadoop has to put intermediate data on the disk and this becomes even worse on spatial join because the spatial partitioning part in the join query shuffles lots of data across the cluster which stresses memory as well as disk.

The peak memory utilization is given in Figure 17b. GeoSpark has the lowest peak memory and Simba has 1.7 times higher peak memory utilization. The peak memory used by Simba is close to the upper limitation of my cluster, almost 400GB. That means if I increase the input data size, Simba will crash. Magellan has 1.3 times higher peak memory utilization than GeoSpark. SpatialHadoop still has the lowest peak memory which is around 1 - 2 times less than GeoSpark.

3.5.4 Performance of Application Use Cases

This section evaluates the three use cases presented earlier in Section 3.4. I use the same GeoSpark source code in Figure 7 and 8 to test the performance: (1) App1: Region heat map: I first run a range query on NYCtaxi to only keep Manhattan region pick-up points then produce a map with OSM L6 zoom level [65] which has

Application	GeoSpark	Simba	Magellan	SpatialHadoop
Spatial aggregation	17 min 234GB	X X	20min 307GB	41min 105GB
Spatial coLocation	1st iter., 8.5min 10 iter., 10.5min execution, 240GB cached, 101GB	Crashed Crashed	X X	X X

Table 4: Performance of GeoSpark applications (min is for execution time, GB is for peak memory)

4096 map tiles and 268 million pixels. (2) App2: Spatial aggregation: I perform a range join between NYCtaxi points and NYC taxi zones (published along with [85], 264 taxi polygonal zones in NYC) then count the taxi trip pickup points in each zone. (3) App3: Spatial co-location pattern mining: I cache the spatial-partitioned Spatial RDD and its local index and iterate co-location pattern mining 10 times. To be precise, I also write the applications using the compared Hadoop-based and Spark-based systems and test their performance.

It is worth noting that SpatialHadoop is able to plot heat maps (used in App1) and range join query (used in App2). SpatialHadoop uses Quad-Tree spatial partitioning by default and it also builds Quad-Tree local index on NYCtaxi. It doesn't support distance join query used in App3. Magellan only supports App2 (NYC taxi zones $\bowtie$ NYCtaxi (polygons, points) using Z-Curve spatial partitioning). It doesn't support distance join query in App3. Simba only supports distance join between points and points (used in App3) but it doesn't support NYC taxi zones $\bowtie$ NYCtaxi (polygons, points) (used in App2). By default, Simba uses R-Tree spatial partitioning and builds local R-Tree indexes on the smaller dataset of the join query. GeoSpark uses KDB-Tree spatial partitioning and builds Quad-Tree index on NYCtaxi. The experimental result is shown in Table 4. An X means that this join type is not supported by this system.

As it turns out in Table 4, region heat map takes GeoSpark 7 minutes because it needs to repartition the pixels across the cluster following the uniform grids. Its peak memory is dominated by the range query part because the repartitioning part only shuffles the Manhattan region data which is smaller than the input Spatial RDD. SpatialHadoop runs 4 times slower than GeoSpark because it puts intermediate on disk. However, its memory utilization is much less than GeoSpark.

Regarding App2 spatial aggregation using NYCtaxizone $\bowtie$ NYCtaxi, the execution time of GeoSpark is $\tilde{1}7$ minutes, which is 2 times longer than that on OSMpostal $\bowtie$ NYCtaxi because there are 10 times more taxi zones than OSMpostal postal areas in New York City. In addition, spatial aggregation executes a CountBy operation to count the pickup point per taxi zone and this leads to data shuffle across the cluster. GeoSpark is faster than Magellan because of KDB-Tree partitioning and Quad-Tree index. Its memory consumption is lower than Magellan due to the help of GeoSpark customized serializer. SpatialHadoop is still around 2 times slower than other systems but its peak memory is lower.

As depicted in Table 4, GeoSpark and Simba support App3 Co-location pattern mining because they support distance joins. However, Simba takes a very long time to run and eventually crashes because of memory overflow in distance join query. GeoSpark is the only system can properly handle this application and finish it in a timely manner. Recalled that I run 10 iterations using GeoSpark in this application. The first iteration of the co-location pattern mining algorithm takes 8.5 minutes and all 9 others take 2.3 minutes. The first iteration takes 30 times more time than the other iterations. This happens because GeoSpark caches the spatial RDD and the corresponding local indexes. Hence, the upcoming iteration directly reads data from the memory cache, which saves a lot of time.

3.6 Summary

In this section, I presented the anatomy of GeoSpark, an in-memory cluster computing framework for processing large-scale spatial data. GeoSpark provides Spatial SQL and Spatial RDD APIs for Apache Spark programmers to easily develop spatial analysis applications. Moreover, the system provides native support for spatial data partitioning, indexing, , and query processing in Apache Spark to efficiently analyze spatial data at scale. Extensive experiments show that GeoSpark outperforms Spark-based systems such as Simba and Magellan up to one order of magnitude and Hadoop-based system such as SpatialHadoop up to two orders of magnitude. The release of GeoSpark stimulated the database community to work on spatial extensions to Spark. I expect that more researchers and practitioners will contribute to GeoSpark code base to support new spatial data analysis applications.

Chapter 4

A FAST, YET LIGHTWEIGHT, DATABASE INDEXING MECHANISM

In this chapter, I first explain the design of Hippo [104, 102], a lightweight database indexing mechanism that reduces the storage overhead and maintenance overhead of existing indices for regular numerical data. Then I show that how I extend the idea of Hippo to support geospatial data indexing.

4.1 Indexing Regular Numerical Data

4.1.1 Introduction

A database system (DBMS) often employs an index structure, e.g., B^+-Tree, to speed up queries issued on the indexed table. Even though classic database indexes improve the query response time, they face the following challenges:

Table 5: Index overhead and storage dollar cost

(a) B^+-Tree overhead

TPC-H	Index size	Initialization time	Insertion time
2 GB	0.25 GB	30 sec	10 sec
20 GB	2.51 GB	500 sec	1180 sec
200 GB	25 GB	8000 sec	42000 sec

(b) Storage dollar cost

HDD	EnterpriseHDD	SSD	EnterpriseSSD
0.04 $/GB	0.1 $/GB	0.5 $/GB	1.4 $/GB

- **Indexing Overhead:** A database index usually yields 5% to 15% additional storage overhead. Although the overhead may not seem too high in small databases, it results in non-ignorable dollar cost in big data scenarios. Table 5a depicts the storage overhead of a B^+-Tree created on the Lineitem table from the TPC-H benchmark [21] (database size varies from 2, 20 and 200 GB). Moreover, the dollar cost increases dramatically when the DBMS is deployed on modern storage devices (e.g., Solid State Drives and Non-Volatile Memory) because they are still more than an order of magnitude expensive than Hard Disk Drives (HDDs). Table 5b lists the dollar cost per storage unit collected from Amazon.com and NewEgg.com. In addition, initializing an index may be a time consuming process especially when the index is created on a large table (see Table 5a).
- **Maintenance Overhead:** A DBMS must update the index after inserting (deleting) tuples into (from) the underlying table. Maintaining a database index incurs high latency because the DBMS has to locate and update those index entries affected by the underlying table changes. For instance, maintaining a B^+-Tree searches the tree structure and perhaps performs a set of tree nodes splitting or merging operations. That requires plenty of disk I/O operations and hence encumbers the time performance of the entire DBMS in big data scenarios. Table 5a shows the B+ Tree insertion overhead (insert 0.1% records) for the TPC-H Lineitem table.

Existing approaches that tackle one or more of the aforementioned challenges are classified as follows: *(1) Compressed indexes:* Compressed B^+-Tree approaches [35, 36, 113] reduce the storage overhead but compromise on the query performance due to the additional compression and decompression time. Compressed bitmap indexes also reduce index storage overhead [40, 57, 83] but they mainly suit low cardinality attributes which are quite rare. For high cardinality attributes, the storage overhead of compressed bitmap indexes significantly increases [96]. *(2) Approximate indexes:* An approximate index [10, 49, 79] trades query accuracy for storage to produce smaller, yet fast, index structures. Even though approximate indexes may shrink the storage size, users cannot rely on their un-guaranteed query accuracy in many accuracy-sensitive application scenarios like banking systems or user archive systems. *(3) Sparse indexes:* A sparse index [14, 63, 84, 68] only stores pointers which refer to disk pages and value ranges (min and max values) in each page so that it can save indexing and maintenance overhead. It is generally built on ordered attributes. For a posed query, it finds value ranges which cover or overlap the query predicate and then rapidly inspects the associated few parent table pages one by one for retrieving truly qualified tuples. However, for unordered attributes which are much more common, sparse indexes compromise too much on query performance because they find numerous qualified value ranges and have to inspect a large number of pages.

In this section, I propose Hippo a fast, yet scalable, sparse database indexing approach. In contrast to existing tree index structures, Hippo stores disk page ranges (each works as a pointer of one or many pages) instead of tuple pointers in the indexed table to reduce the storage space occupied by the index. Unlike existing approximate indexing methods, Hippo guarantees the query result accuracy by inspecting possible qualified pages and only emitting those tuples that satisfy the query predicate. As opposed to existing sparse indexes, Hippo maintains simplified histograms that represent the data distribution for pages no matter how skew it is, as the summaries for these pages in each page range. Since Hippo relies on histograms already created and maintained by almost every existing DBMS (e.g., PostgreSQL), the system does not exhibit a major additional overhead to create the index. Hippo also adopts a page grouping technique that groups contiguous pages into page ranges based on the similarity of their index key attribute distributions. When a query is issued on the indexed database table, Hippo leverages the page ranges and histogram-based page summaries to recognize those pages for which the internal tuples are guaranteed not to satisfy the query predicates and inspects the remaining pages. Thus Hippo achieves competitive performance on common range queries without compromising the accuracy. For data insertion and deletion, Hippo dispenses with the numerous disk operations by rapidly locating the affected index entries. Hippo also adaptively decides whether to adopt an eager or lazy index maintenance strategy to mitigate the maintenance overhead while ensuring future queries are answered correctly.

I implemented a prototype of Hippo inside PostgreSQL 9.5[7]. Experiments based on the TPC-H benchmark as well as real and synthetic datasets show that Hippo occupies up to two orders of magnitude less storage space than that of the B^+-Tree while still achieving comparable query execution performance to that of the B^+-Tree for 0.1% - 1% selectivity factors. Also, the experiments show that Hippo outperforms BRIN, though occupies more storage space, in executing queries with various selectivity factors. Furthermore, Hippo achieves up to three orders of magnitude less maintenance overhead than its counterparts, i.e., B^+-Tree and BRIN. Most importantly, Hippo exhibits up to an order of magnitude higher throughput (measured in terms of the number of tuples processed per second) than both BRIN and B^+-Tree for hybrid query/update workloads.

[7]https://github.com/DataSystemsLab/hippo-postgresql

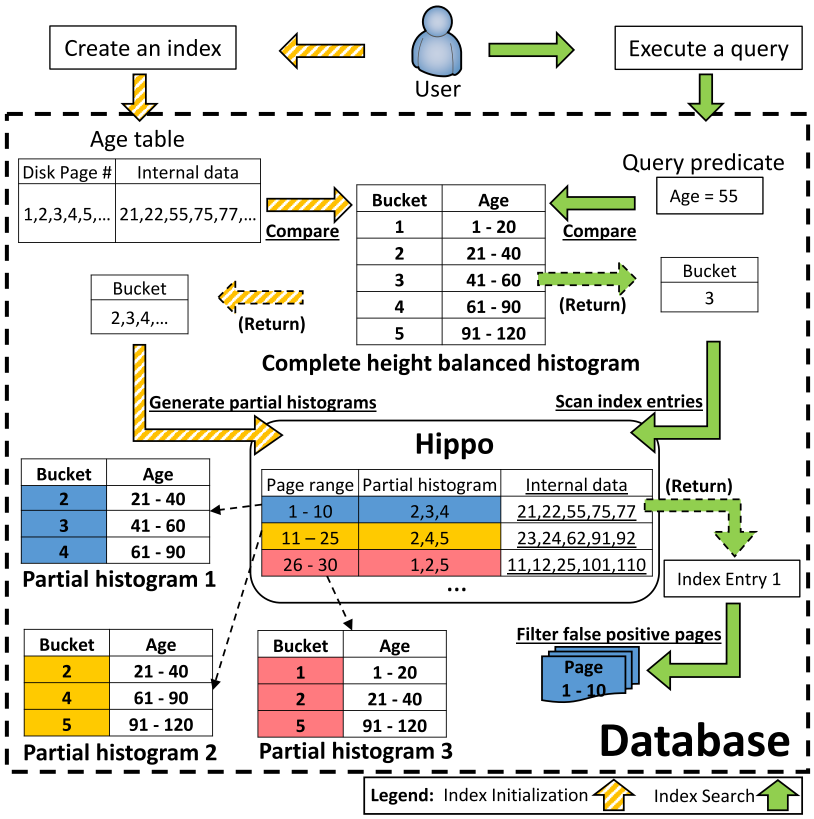


Figure 18: Initialize and search Hippo on age table

4.1.2 Hippo Overview

This section gives an overview of Hippo. Figure 18 depicts a running example that describes the index initialization (left part of the figure) and search (right part of the figure) processes in Hippo. The main challenges of designing an index are to reduce the indexing overhead in terms of storage and initialization time as well as speed up the index maintenance while still keeping competitive query performance. To achieve that, an index should possess the following two main properties: *(1) Less Index Entries:* For better storage space utilization, an index should determine and only store the most representative index entries that summarize the key attribute. Keeping too many

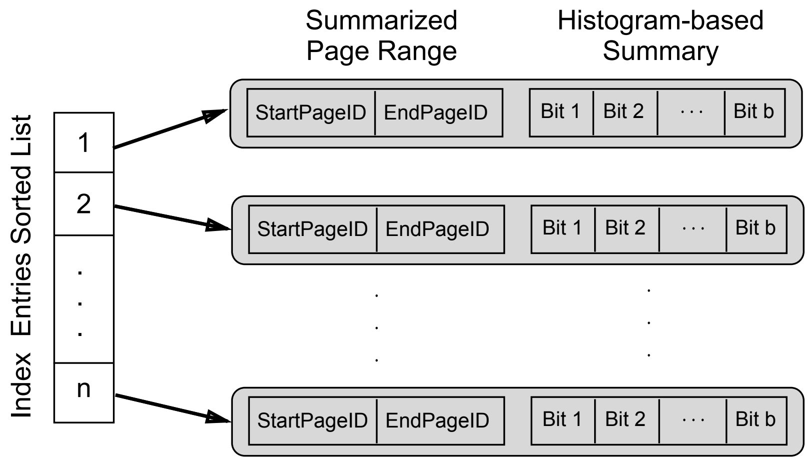


Figure 19: Hippo Index Structure

index entries inevitably results in high storage overhead as well as high initialization time. *(2) Index Entries Independence:* The index entries should be independent from each other. In other words, the range of values that each index entry represents should have minimal overlap with other index entries. Interdependence among index entries, like that in a B$^+$-Tree, results in overlapped tree nodes. That may lead to more I/O operations during query processing and several cascaded updates during index maintenance.

Data Structure. Figure 19 depicts the index structure. To create an index, Hippo scans the indexed table and generates histogram-based summaries for a set of disk page based on the index key attribute. These summaries are then stored by Hippo along with page ranges they summarize. As shown in Figure 19, Hippo consists of n index entries such that each entry consists of the following two components:

- **Summarized Page Range:** represents the IDs of the first and last pages summarized (i.e., StartPageID and EndPageID in Figure 19) by the index entry. The DBMS can load particular pages into buffer according to their IDs. Hippo summarizes more than one physically contiguous pages to reduce the overall index size, e.g., Page 1 - 10, 11 - 25, 26 - 30 in Figure 18. The number of summarized pages in each index entry varies. Hippo adopts a page grouping technique that groups contiguous pages into page ranges based on the similarity of their index attribute distributions, using the partial histogram density (explained in Section 4.1.4).

- **Histogram-based Summary:** A bitmap that represents a subset of the complete height balanced histogram buckets (maintained by the underlying DBMS), aka. partial histogram. Each bucket, if exists, indicates that at least one of the tuples of this bucket exists in the summarized pages. Each partial histogram represents the distribution of the data in the summarized contiguous pages. Since each bucket of a height balanced histogram roughly contains the same number of tuples, each of them has the same probability to be hit by a random tuple from the table. Hippo leverages this feature to handle a variety of data distributions, e.g., uniform, skewed. To reduce the storage footprint, only bucket IDs are kept in partial histograms and partial histograms are stored in a compressed bitmap format. For instance, the partial histogram of the first index entry in Figure 18 is 01110.

Main idea. Hippo solves the aforementioned challenges as follows: (1) Each index entry summarizes many pages and only stores two page IDs and a compressed bitmap.(2) Each page of the parent table is only summarized by one Hippo index entry. Hence, any updates that occur in a certain page only affect a single independent index entry. Finally, during a query, pages whose partial histograms do not have desired buckets are guaranteed not to satisfy certain query predicates and marked as false positives. Thus Hippo only inspects other pages that probably satisfies the query predicate and achieves competitive query performance.

4.1.3 Index Search

The search algorithm takes as input a query predicate and returns a set of qualified tuples. As explained in Section 4.1.2, partial histograms are stored in a bitmap format. Hence, any query predicates for a particular attribute are broken down into atomic units: equality query predicate and range query predicate. Each unit predicate is compared with the buckets of the complete height balanced histogram (discussed in Section 4.1.4). A bucket is hit by a predicate if the predicate fully contains, overlaps, or is fully contained by the bucket. Each unit predicate can hit at least one or more buckets. Afterwards, the query predicate is converted to a bitmap. Each bit in this bitmap reflects whether the bucket that has the corresponding ID is hit (1) or not (0). Thus, the corresponding bits of all hit buckets are set to 1.

The search algorithm then runs in two main steps (see pseudo code in Algorithm 5): (1) Step I: Scanning Hippo index entries and (2) Step II: Filtering false positive pages. The search process leverages the index structure to avoid unnecessary page inspection so that Hippo can achieve competitive query performance.

4.1.3.1 Step I: Scanning Index Entries

Step I finds possible qualified disk pages, which may contain tuples that satisfy the query predicate. Since it is quite possible that some pages may not contain any

Algorithm 5: Hippo index search

Data: A given query predicate Q
Result: A set of qualified tuples R

```
// Step I: Scanning Index Entries;
Set of Possible Qualified Pages P = φ;
foreach Index Entry in Hippo do
    if the partial histogram has joint buckets with Q then
        Add the IDs of pages indexed by the entry to P;
// Step II: Filtering False Positive Pages;
Set of Qualified Tuples R = φ;
foreach Page ID ∈ P do
    Retrieve the corresponding page p;
    foreach tuple t ∈ p do
        if t satisfies the query predicate then
            Add t to R;
return R;
```

qualified tuple especially for highly selective queries, Hippo prunes these index entries (that index these unqualified pages) that definitely do not contain any qualified pages.

In this step, the search algorithm scans the Hippo index. For each index entry, the algorithm retrieves the partial histogram which summarizes the data distribution in the pages indexed by such entry. The algorithm then checks whether the input query predicate has one or more joint (i.e. overlapped) buckets with the partial histogram. To efficiently process that, Hippo performs a nested loop between each partial histogram and the input query predicate to find the joint buckets. Since both the partial histograms and the query predicate are in a bitmap format, Hippo accelerates the nested loop by performing a bitwise 'AND' of the bytes from both sides, *aka. bit-level parallelism.* In case bitwise 'AND'ing the two bytes returns 0, that means there exist no joint buckets between the query predicate and the partial histogram. Entries with partial histograms that do not contain the hit buckets (i.e., the corresponding bits are 0) are guaranteed not to contain any qualified disk pages.

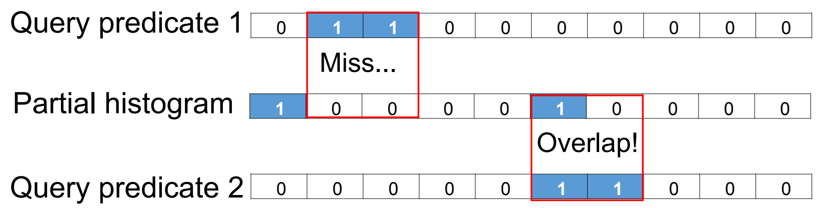


Figure 20: Scan index entries

Hence, Hippo disqualifies these pages and excludes them from further processing. On the other hand, index entries with partial histograms that contain at least one of the hit buckets, i.e., the corresponding bits are 1, may or may not have qualified pages. Hippo deems these pages as possible qualified pages and hence forwards their IDs to the next step.

Figure 20 visualizes the procedure of scanning index entries according to their partial histograms. In Figure 20, buckets hit by the query predicates and the partial histogram are represented in a bitmap format. According to this figure, the partial histogram misses a query predicate if the highlighted area of the predicate falls into the blank area of the partial histogram, whereas a partial histogram is selected if the predicate does not fall completely into the blank area of the histogram.

4.1.3.2 Step II: Filtering False Positive Pages

The previous step identifies many unqualified disk pages that are guaranteed not to satisfy the query predicate. However, not all unqualified pages can be detected by the previous step. The set of possible qualified pages, retrieved from Step I, may still contain false positives (defined below). During the search process, Hippo considers a possible qualified page p a false positive if and only if (1) p lies in the page range summarized by a qualified index entry from Step I and (2) p does not contain any tuple that satisfies the input query predicate. To filter out false positive pages, Step II inspects every tuple in each possible qualified page, retrieves those tuples that satisfy the query predicate, and finally returns those tuples as the answer.

Step II takes as input the set of possible qualified pages IDs, formatted in a separate bitmap. Each bit in this bitmap is mapped to the page at the same position in the original table indexed by Hippo. For each page ID, Hippo retrieves the corresponding page from disk and checks each tuple in that page against the query predicate. In case, a tuple satisfies the query predicate, the algorithm adds this tuple to the final result set. The right part of Figure 18 describes how to search the index using an input query predicate. First, Hippo finds that query predicate age = 55 hits bucket 3. Since the first one of the three partial histograms nicely contains bucket 3, only the disk pages 1 - 10 are selected as possible qualified pages and hence sent for further inspection in step II. It is also worth noting that these partial histograms summarize different number of pages.

4.1.4 Index Initialization

To create an index, Hippo takes as input a database table and the key attribute (i.e., column) in this table. Hippo then performs two main steps (See pseudo code in Algorithm 6) to initialize itself: (1) Generate partial histograms (Section 4.1.4.1), and (2) Group similar pages into page ranges (Section 4.1.4.2), described as follows.

4.1.4.1 Generate Partial Histograms

To initialize the index, Hippo leverages a complete height balanced histogram, maintained by most DBMSs, that represents the data distribution. A histogram consists of a set of buckets such that each bucket represents the count of tuples with attribute value within the bucket range. A partial histogram only contains a subset of the buckets that belongs to the height balanced histogram. The resolution of the complete histogram (H) is defined as the total number of buckets that belongs to this histogram. A histogram will obviously have larger physical storage size if it has higher resolution. The histogram resolution also affects the query response time (see Section 4.1.6 for further details).

Hippo stores a partial histogram for each index entry to represent the data distribution of tuples in one or many disk pages summarized by the entry. Partial histograms allow Hippo to early identify unqualified disk pages and avoid unnecessary page inspection. To generate partial histograms, Hippo scans all disk pages of the indexed table. For each page, the algorithm retrieves each tuple, the key attribute value is extracted from each tuple and then compared to the complete histogram using binary search. Buckets hit by tuples are kept for this page and then compose a partial histogram. A partial histogram only contains distinct buckets. For instance, there is a group of age attribute values like the first entry of Hippo given in Figure 18: 21, 22, 55, 75, 77. Bucket 2 is hit by 21 and 22, bucket 3 is hit by 55 and bucket 4 is hit by 77 (see partial histogram 1 in Figure 18).

Hippo shrinks the storage footprint of partial histograms by dropping all bucket value ranges and only keeping bucket IDs. Actually, as mentioned in Section 4.1.2, dropping value range information does not have much negative impact on the index

```
Algorithm 6: Hippo index initialization
Data: Pages of a parent table
Result: Hippo index
Create a working partial histogram (in bitmap format);
Set StartPage = 1 and EndPage = 1;
foreach page do
    Find distinct buckets hit by its tuples;
    Set associated bits to 1 in the partial histogram;
    if the working partial histogram density > threshold then
        Store the partial histogram and the page range (StartPage and
          EndPage) as an index entry;
        Create a new working partial histogram;
        StartPage = EndPage + 1;
        EndPage = StartPage;
    else
        EndPage = EndPage + 1;
```

search. To further shrink the storage footprint, Hippo stores the histogram bucks IDs in bitmap type format instead of using an integer type (4 bytes or more). Each partial histogram is stored as a bitmap such that each bit represents a bucket at the same position in a complete histogram. Bit value 1 means the associated bucket is hit and stored in this partial histogram while 0 means the associated bucket is not included. The partial histogram can also be compressed by any existing bitmap compression technique. The time for compressing and decompressing partial histograms is ignorable compared to that of inspecting possible qualified pages.

4.1.4.2 Group Pages Into Page Ranges

Generating a partial histogram for each disk page may lead to very high storage overhead. Grouping contiguous pages and merging their partial histograms into a larger partial histogram (in other words, summarizing more pages within one partial histogram) can tremendously reduce the storage overhead. However, that does not mean that all pages should be grouped together and summarized by a single merged partial histogram. The more pages are summarized, the more buckets the partial histogram contains. If the partial histogram becomes a complete histogram and covers any possible query predicates, it is unable to filter the false positives and the disk pages summarized by this partial histogram will be always treated as possible qualified pages.

One strategy is to group a fixed number of contiguous pages per partial histogram. Yet, this strategy is not efficient when a set of contiguous pages have much more similar data distribution than other areas. To remedy that, Hippo dynamically groups more contiguous pages under the same index entry when they possess similar data distribution and less contiguous pages if they do not show similar data distribution. To take the page grouping decision, Hippo leverages a parameter called *partial histogram density*. The density of a partial histogram is defined as the ratio of complete histogram buckets that belongs to the partial histogram. Obviously, the complete histogram has a density value of 1. The definition can be formalized as follows:

$$Partial\ histogram\ density\ (D)\ =\ \frac{\#\ Buckets_{partial\ histogram}}{\#\ Buckets_{complete\ histogram}}$$

The density exhibits an important phenomenon that, for a set of contiguous disk pages, their merged partial histogram density will be very low if these pages are very similar, and vice versa. Therefore, a partial histogram with a certain density may summarize more pages if these contiguous pages have similar data, vice versa. Making use of this phenomenon enables Hippo to dynamically group pages and merge partial histograms into one. In addition, it is understandable that a lower density partial histogram (summarizes less pages) has the high probability to be excluded from further processing.

Users can easily set the same density value for all partial histograms as a threshold. Hippo can automatically decide how many pages each partial histogram should summarize. Algorithm 6 depicts how Hippo initializes the index and summarizes more pages within a partial histogram by means of the partial histogram density. The basic idea is that new pages will not be summarized into a partial histogram if its density is larger than the threshold and a new partial histogram will be created for the following pages.

The left part of Figure 18 depicts how the initialization process for an index create on the age attribute. In the example, the partial histogram density is set to 0.6. All tuples are compared with the complete histogram and IDs of distinct buckets hit by all tuples are generated as partial histograms along with their page range. So far, as Figure 18 and 19 shows, each index entry has the following parameters: a partial histogram in compressed bitmap format and two integers that stand for the first and last pages summarized by this histogram (summarized page range). Each entry is then serialized and stored on disk.

4.1.5 Index Maintenance

Inserting (deleting) tuples into (from) the table requires maintaining the index. That is necessary to ensure that the DBMS can retrieve the correct set of tuples that match the query predicate. However, the overhead introduced by frequently maintaining the index may preclude system scalability. This section explains how Hippo handles updates.

Algorithm 7: Update Hippo for data insertion

Data: A newly inserted tuple that belongs to Page a
Result: Updated Hippo

Find the bucket hit by the inserted tuple;
Locate a Hippo index entry which summarizes Page a;
if *an index entry is located*
then
    Fetch the located Hippo index entry;
    Update the retrieved entry if necessary;
else
    Retrieve the entry that summarizes the last page;
    if *the partial histogram density* $<$ *threshold* **then**
        Summarize Page a into the retrieved index entry;
    else
        Summarize Page a into a new index entry;

4.1.5.1 Data Insertion

Hippo adopts an eager update strategy when a new tuple is inserted to the indexed table. An eager strategy instantly updates or checks the index at least when a new tuple is inserted. Otherwise, all subsequent queries might miss the newly inserted tuple. Data insertion may change the physical structure of a table (i.e., heap file). The new tuple may belong to any pages of the indexed table. The insertion procedure (See Algorithm 7) performs the following steps: (I) Locate the affected index entry, and (II) Update the index entry if necessary.

Step I: Locate the affected index entry: After retrieving the complete histogram, the algorithm checks whether a newly inserted tuple hits one or more of the histogram buckets. The newly inserted tuple belongs to a disk page. This page may be a new page has not been summarized by any partial histograms before or an old page which has been summarized. However, because the numbers of pages summarized by each histogram are different, searching Hippo index entries to find the one contains this target page is inevitable. From the perspective of disk storage, in a Hippo, all partial histograms are stored on disk in a serialized format. It will be extremely time-consuming if every entry is retrieved from disk, de-serialized and checked against the target page. The algorithm then searches the index entries by means of the index entries sorted list explained in Section 4.1.5.3.

Step II: Update the index entry: In case the inserted tuple belongs to a new page and the partial histogram density which summarizes the last disk page is smaller than the density threshold set by the system user, the algorithm summarizes the new page into this partial histogram in the last index entry. Otherwise, the algorithm creates a new partial histogram to summarize this page and stores them in a new index entry. In case a new tuple belongs to a page that is already summarized by Hippo, the partial histogram in the associated index entry will be updated if the inserted tuple hits a new bucket.

It is worth noting that: (1) Since the compressed bitmaps of partial histograms may have different size, the updated index entry may not fit the space left at the old location. Thus the updated one may be put at the end of Hippo. (2) After some changes (replacing old or creating new index entry) in Hippo, the corresponding position of the sorted list might need to be updated.

The I/O cost incurred by eagerly updating the index due to a newly inserted tuple is equal to *log(# of index entries) + 4*. Locating the affected index entry yields *log(# of index entries)* I/Os, whereas Step II consumes 4 I/Os to update the index entry. Section 4.1.6 gives more details on how to estimate the number of index entries in Hippo.

4.1.5.2 Data Deletion

The eager update strategy is deemed necessary for data insertion to ensure the correctness of future queries issued on the indexed table. However, the eager update strategy is not necessary after deleting data from the table. That is due to the fact that Hippo inspects possible qualified pages during the index search process and pages with qualified deleted tuples might be still marked as possible qualified page in the first phase of the search algorithm. Even if these pages contain deleted tuples, such pages will be discarded during the "Step II: filtering false positive pages" phase of the search algorithm. However, not maintaining the index at all may introduce a lot of false positives during the search process, which may takes its toll on the query repossess time.

Hippo still ensures the correctness of queries even if it does not update the index at all after deleting tuples from a table. To achieve that, Hippo adopts a periodic lazy update strategy for data deletion. The deletion strategy maintains the index after a bulk of delete operations are performed to the indexed table. In such case, Hippo traverses all index entries. For each index entry, the system inspects the header of each summarized page for seeking notes made by DBMSs (e.g., PostgreSQL makes notes in page headers if data is removed from pages). Hippo re-summarizes the entire index entry instantly within the original page range if data deletion on one page is detected. The re-summarization follows the same steps in Section 4.1.4.

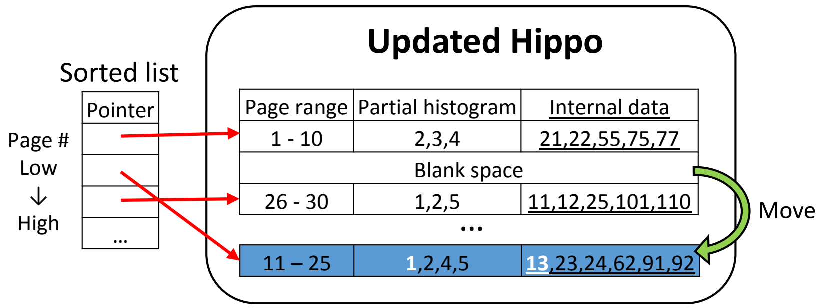


Figure 21: Hippo Index Entries Sorted List

4.1.5.3 Index Entries Sorted List

When a new tuple is inserted, Hippo executes a fast binary search (according to the page IDs) to locate the affected index entry and then updates it. Since the index entries are not guaranteed to be sorted based on the page IDs (noted in data insertion section), an auxiliary structure for recording the sorted order is introduced to Hippo.

The sorted list is initialized after all steps in Section 4.1.4 with the original order of index entries and put at the first several index pages of Hippo. During the entire Hippo life time, the sorted list maintains a list of pointers of Hippo index entries in the ascending order of page IDs. Actually each pointer represents the fixed size physical address of an index entry and these addresses can be used to retrieve index entries directly. That way, the premise of a binary search has been satisfied. Figure 21 depicts the Hippo index entries sorted list. Index entry 2 in Figure 18 has a new bucket ID 1 due to a newly inserted tuple in its internal data and hence this entry becomes the last index entry in Figure 21. The sorted list is still able to record the ascending order and help Hippo to perform a binary search on the index entries. In addition, such sorted list leads to slight additional maintenance overhead: Some index updates need to modify the affected pointers in the sorted list to reflect the new physical addresses.

Table 6: Notations used in Cost Estimation

Term	Definition
H	Complete histogram resolution which means the number of buckets in this complete histogram
pageH	Average number of histogram buckets hit by a page
D	Partial histogram density, which is a user supplied parameter
P	Average number of pages summarized by a partial histogram for a certain attribute
T	Average number of tuples summarized by a partial histogram for a certain attribute
Card	Total number of tuples of the indexed table
pageCard	Average number of tuples per page
SF	The selectivity factor of the issued query

4.1.6 Cost Model

This section deduces a cost model for Hippo. Table 6 summarizes the main notations. Given a database table R with a number of tuples $Card$ and average number of tuples per disk page $pageCard$, a user may create a Hippo index on attribute (i.e., column) a_i of R. Let the complete histogram resolution be H (it has H buckets in total) and the partial histogram density be D. Assume that each Hippo index entry on average summarizes P data pages and T tuples. Queries executed against the index have an average selectivity factor SF. To calculate the query I/O cost, I need to consider: (1) I/O$_{scanning\ index}$ represents the cost of scanning the index entries (Phase I in the search algorithm) and (2) I/O$_{filtering\ false\ positives}$ represents the I/O cost of filtering false positive pages (Phase II).

Estimating the number of index entries. Since all index entries are scanned in the first phase, the I/O cost of this phase is equal to the total pages the index spans on disk (I/O$_{scanning\ index}$=$\frac{\#\ of\ index\ entries}{pageCard}$). To estimate the number of Hippo index entries, I have to estimate how many disk pages (P) are summarized by a partial histogram in general, or how many tuples (T) are checked against the complete histogram to generate a partial histogram. This problem is very similar to the Coupon Collector's Problem[33]. This problem can be described like that: "A vending machine sells H types of coupons (a complete histogram with H buckets). Alice is purchasing coupons from this machine. Each time (each tuple) she can get a random type coupon (a bucket) but she might already have a same one. Alice keeps purchasing until she gets $D * H$ types of coupons (distinct buckets). How many times (T) does she need to purchase?" Therefore, the expectation of T is determined by the following equation:

$$T = H\times(\frac{1}{H} + \frac{1}{H-1} + ... + \frac{1}{H - D\times H + 1}) \tag{4.1}$$

$$= H\times \sum_{i=0}^{D\times H-1} \frac{1}{H-i} \tag{4.2}$$

Note that the partial histogram density $D \in [\frac{pageH}{H}, 1]$. That means the global density should be larger than the ratio of average hit histogram buckets per page to all histogram buckets because page is the minimum unit when grouping pages based on density. Estimating $pageH$ is also a variant of Coupon Collector's Problem: How many types of coupons (distinct buckets) will Alice get if she purchases $pageCard$ coupons (tuples)? Given Equation 4.2, the mathematical expectation of $pageH$ can be easily found as follows:

$$pageH = H\times(1 - (1 - \frac{1}{H})^{pageCard}) \tag{4.3}$$

The number of Hippo index entries is equivalent to the total number of tuples in the indexed table divided by the average number of tuples summarized by each index entry, i.e., $\frac{Card}{T}$. Hence, the number of index entries is given in Equation 4.4. The index size is equal to the product of the number of index entries and the size of a single entry. The size of each index entry is roughly equal to each partial histogram size.

$$\#\ of\ Index\ entries\ = Card/(H \times \sum_{i=0}^{D \times H-1} \frac{1}{H-i}) \tag{4.4}$$

Given Equation 4.4, I observe the following: (1) For a certain H, the higher the value of D, the less Hippo index entries there exist. (2) For a certain D, the higher H there is, the less Hippo index entries there are. Meanwhile, the size of each index entry increases with the growth of the complete histogram resolution. The final I/O cost of scanning the index entries is given in Equation 4.5.

$$I/O_{scanning\ index} = \frac{Card}{H \times pageCard} \times (\sum_{i=0}^{D \times H-1} \frac{1}{H-i})^{-1} \tag{4.5}$$

Estimating the number of read data pages. Data pages summarized by each index entry are likely to be checked in the second phase of the search algorithm, filtering false positive pages, if their associated partial histogram has joint buckets with the query predicate. Determining the probability of having joint buckets contributes to the query I/O cost estimation. The probability that a partial histogram in an index entry has joint buckets with a query predicate depends on how likely a predicate overlaps with the highlighted area in partial histograms (see Figure 20). The probability is determined by the equation given below:

$$\begin{aligned} Prob &= (Average\ buckets\ hit\ by\ a\ query\ predicate) \times D \\ &= SF \times H \times D \end{aligned} \tag{4.6}$$

To be precise, $Prob$ follows a piecewise function as follows:

$$Prob = \begin{cases} SF \times H \times D & SF \times H \leqslant \frac{1}{D} \\ 1 & SF \times H > \frac{1}{D} \end{cases}$$

Given the aforementioned discussion, I observe that (1) when SF and H are fixed, the smaller D is, the smaller $Prob$ is. (2) when H and D are fixed, the smaller SF is, the smaller $Prob$ is. (3) when SF and D are fixed, the smaller H is, the smaller $Prob$ is. It is obvious that the probability in Equation 4.6 is equivalent to the probability that pages in an index entry are checked in the second phase, i.e., filtering false positive pages. Since the total pages in Table R is $\frac{Card}{pageCard}$, the mathematical expectation of the number of pages in R checked by the second part, as known as the I/O cost of second part, is:

$$I/O_{filtering\ false\ positives} = (Prob \times \frac{Card}{pageCard}) \tag{4.7}$$

By adding up I/O$_{scanning\ index}$ (Equation 4.5) and I/O$_{filtering\ false\ positives}$ (See Equation 4.7), the total query I/O cost is as follows:

$$Query\ I/O = \frac{Card}{pageCard} \times ((H \times \sum_{i=0}^{D \times H - 1} \frac{1}{H - i})^{-1} + Prob) \tag{4.8}$$

4.1.7 Experiments

This section provides a comprehensive experimental evaluation of Hippo. All experiments are run on an Ubuntu Linux 14.04 64 bit machine with 8 cores CPU (3.5 GHz per core), 32 GB memory, and 2 TB magnetic hard disk. I install PostgreSQL 9.5 (128 MB default buffer pool) on the test machine.

Compared Approaches. During the experiments, I study the performance of the following indexing schemes: (1) Hippo: A complete prototype of my proposed indexing approach implemented inside the core engine of PostgreSQL 9.5. Unless mentioned otherwise, the default partial histogram density is set to 20% and the default histogram resolution (H) is set to 400. (2) B^+-Tree: The default implementation of the B^+-Tree in PostgreSQL 9.5 (with a default fill factor of 90), (3) BRIN: A sparse Block Range Index implemented in PostgreSQL 9.5 with 128 default pages per range. I also consider other BRIN settings, i.e., BRIN-32 and BRIN-512, with 32 and 512 pages per range respectively.

Datasets. I use the following four datasets:

- TPC-H : A 207 GB decision support benchmark that consists of a suite of business oriented ad-hoc queries and data modifications. Tables populated by TPC-H follow a uniform data distribution. For evaluation purposes, I build indexes on Linitem table PartKey, SuppKey or OrderKey attribute. PartKey attribute has 40 million distinct values while SuppKey has 2 million distinct values and the values of OrderKey attribute are sorted. For TPC-H benchmark queries, I also build indexes on L_Shipdate and L_Receiptdate when necessary.
- Exponential distribution synthetic dataset (abbr. Exponential): This 200 GB dataset consists of three attributes, IncrementalID, RandomNumber, Payload.

RandomNumber attribute follows exponential data distribution which is highly skewed.

- Wikipedia traffic (abbr. Wikipedia) [94]: A 231 GB Wikipedia article traffic statistics covering seven months period log. The log file consists of 4 attributes: PageName, PageInfo, PageCategory, PageCount. For evaluation purposes, I build index on the PageCount attribute which stands for hourly page views.
- New York City taxi dataset (abbr. NYC Taxi) [85]: This dataset contains 197 GB New York City Yellow and Green Taxi trips published by New York City Taxi & Limousine Commission website. Each record includes pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, and itemized fares. I reduce the dimension of pick-up location from 2D (longitude, latitude) to 1D (integer) using a spatial dimension reduction method, Hilbert Curve, and build indexes on pick-up location attribute.

Implementation details. I have implemented a prototype of Hippo inside PostgreSQL 9.5 as one of the main index access methods by leveraging the underlying interfaces which include but not limited to "ambuild", "amgetbitmap", "aminsert" and "amvacuumcleanup". A PostgreSQL 9.5 user creates and queries the index as follows:

```
CREATE INDEX hippo_idx ON lineitem USING hippo(partkey)


SELECT * FROM lineitem

WHERE partkey > 1000 AND partkey < 2000


DROP INDEX hippo_idx
```

The final implementation has slight differences from the aforementioned details due to platform-dependent features. For instance, Hippo only records possible qualified

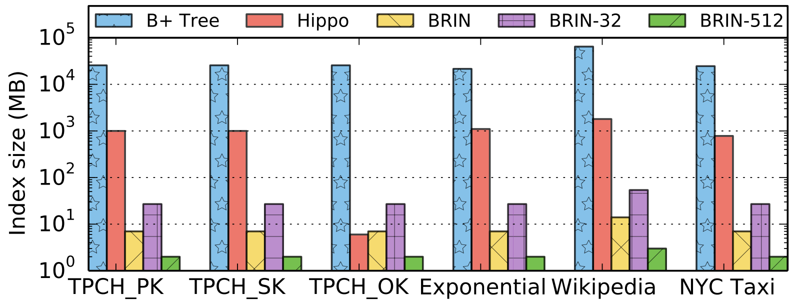


Figure 22: Index size on different datasets (log. scale)

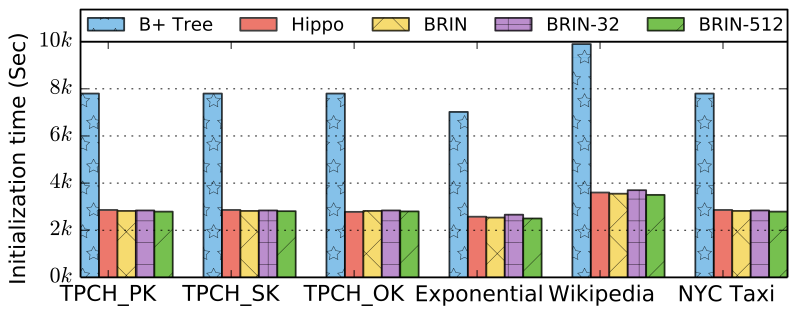


Figure 23: Index initial. time on different datasets

page IDs in a tid bitmap format and returns it to the kernel. PostgreSQL automatically inspects pages and checks each tuples against the query predicate. PostgreSQL `DELETE` command does not really remove data from disk unless a `VACUUM` command is called automatically or manually. Hippo then updates the index for data deletion when a `VACUUM` command is invoked. In addition, it is better to rebuild Hippo index if there is a huge change of the parent attribute's histogram. Furthermore, a script, running as a background process, drops the system cache during the experiments.

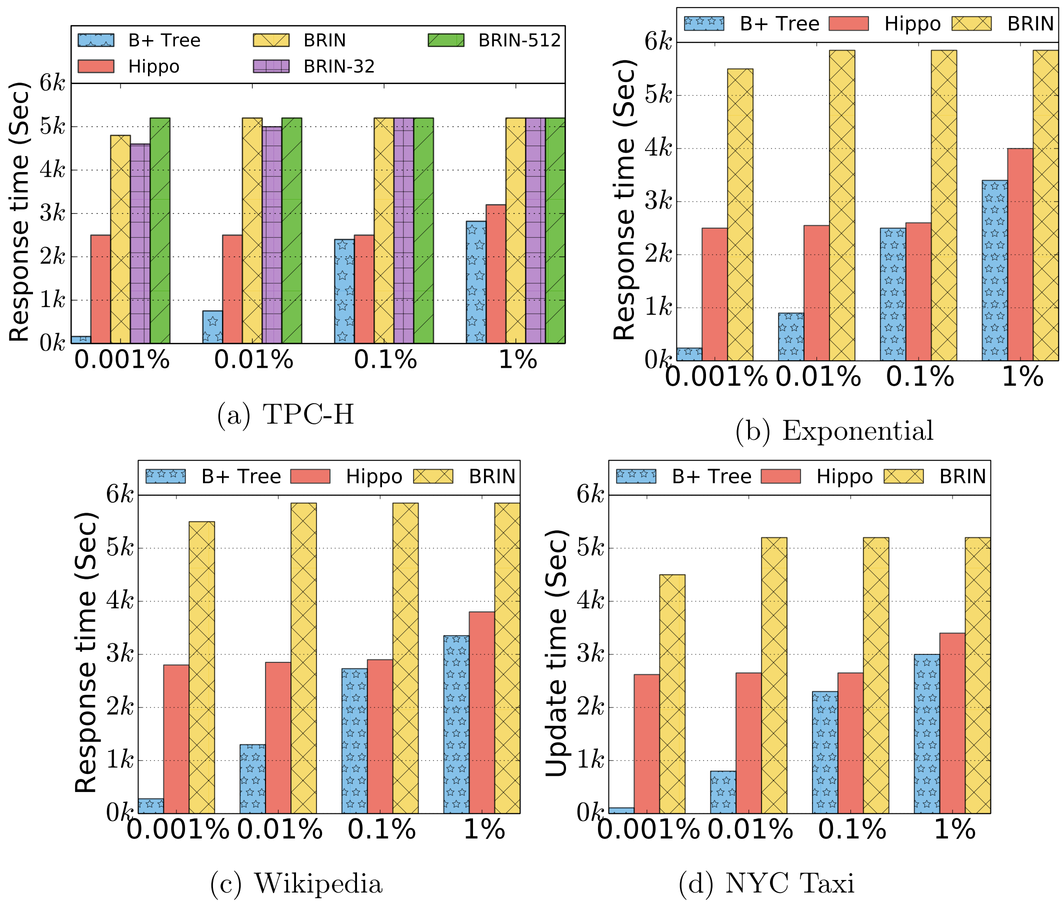


Figure 24: Query response time at different selectivity factors

4.1.7.1 Tuning Hippo Parameters

This section evaluates the performance of Hippo by tuning two main system parameters: partial histogram density D (Default value is 20%) and complete histogram resolution H (Default value is 400). For these experiments, I build Hippo on PartKey attribute in Lineitem table of 200 GB TPC-H benchmarks. I then evaluate the index size, initialization time, and query response time.

Impact of partial histogram density The following experiment compares the default Hippo density (20%) with two different densities (40% and 80%) and tests

Table 7: Tuning Parameters

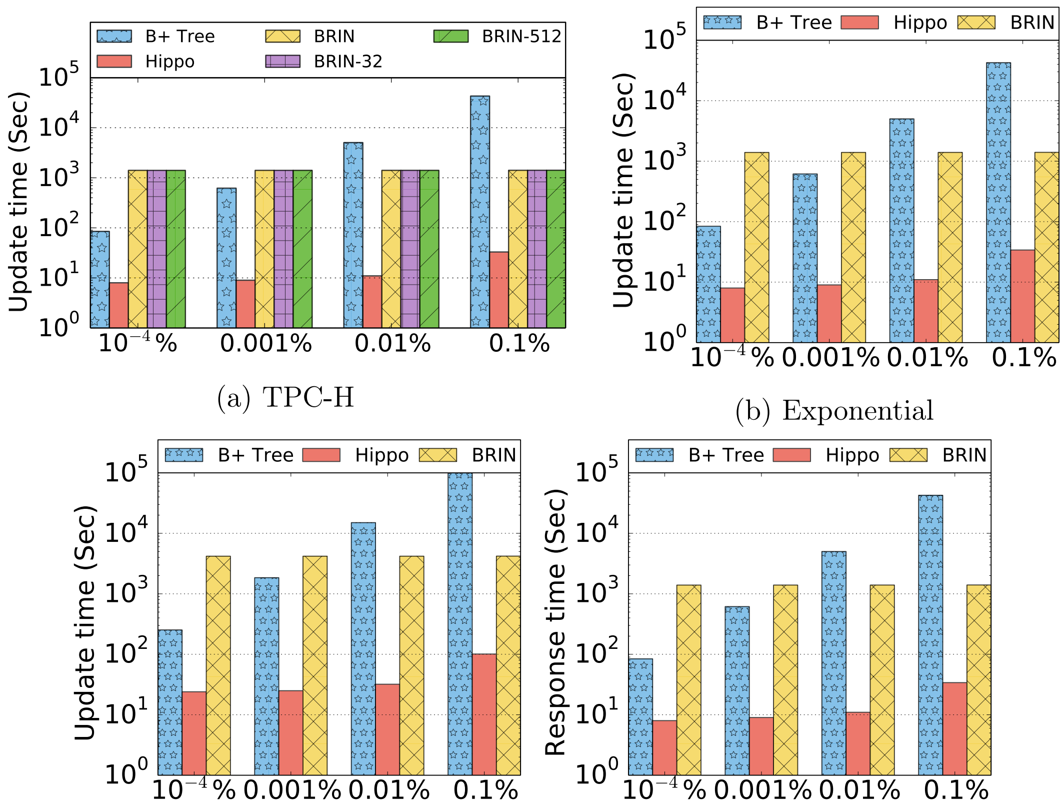
Parameter	Value	Size	Initial. time	Query time
Default	D=20% R=400	1012 MB	2765 sec	2500 sec
Density (D)	40%	680 MB	2724 sec	3500 sec
	80%	145 MB	2695 sec	4500 sec
Resolution (R)	800	822 MB	2762 sec	3000 sec
	1600	710 MB	2760 sec	3500 sec

their query time with selectivity factor 0.1%. As given in Table 7, when I increase the density Hippo exhibits less indexing overhead as expected. That happens due to the fact that Hippo summarizes more pages per partial histogram and write less index entries on disk. Similarly, higher density leads to more query time because it is more likely to overlap with query predicates and result in more pages are selected as possible qualified pages.

Impact of histogram resolution This section compares the default Hippo histogram resolution (400) to two different histogram resolutions (800 and 1600) and tests their query time with selectivity factor 0.1%. The density impact on the index size, initialization time and query time is given in Table 7 .

As given in Table 7, with the growth of histogram resolution, Hippo size decreases moderately. The explanation is that higher histogram resolution leads to less partial histograms and each partial histogram in the index may summarize more pages. However, the partial histogram (in bitmap format) has larger physical size because the bitmap has to store more bits.

As Table 7 shows, the query response time of Hippo varies with the growth of histogram resolution. This is because for the large histogram resolution, the query



(a) TPC-H

(b) Exponential

(c) Wikipedia

(d) NYC Taxi

Figure 25: Data update time (logarithmic scale) at different update percentage

predicate may hit more buckets so that this Hippo is more likely to overlap with query predicates and result in more pages are selected as possible qualified pages.

4.1.7.2 Indexing Overhead

This section studies the indexing overhead (in terms of index size and index initialization time) of the B$^+$-Tree, Hippo, BRIN (128 pages per range by default), BRIN-32, and BRIN-512. The indexes are built on TPC-H Lineitem table PartKey (TPCH_PK), SuppKey (TPCH_SK), OrderKey (TPCH_OK) attributes, Exponential table RandomNumber attribute, Wikipedia table PageCount attribute, NYC Taxi table pick-up location attribute.

As given in Figure 22, Hippo occupies 25 to 30 times smaller storage space than the B^+-Tree on all datasets (except on TPC-H OrderKey attribute). This happens because Hippo only stores disk page ranges along with page summaries. Furthermore, Hippo on TPC-H PartKey incurs the same storage space as that of the index built on the SuppKey attribute (which has 20 times less distinct attribute values). That means the number of distinct values does not actually impact Hippo index size as long as it is larger than the number of complete histogram buckets. Each attribute value has the same probability to hit a histogram bucket no matter how many distinct attribute values there are. This is because the complete histogram leveraged by Hippo summarizes the data distribution of the entire table. Hippo still occupies small storage space on tables with different data distributions, such as Exponential, Wikipedia and NYC Taxi data. That happens because the complete histogram, which is height balanced makes sure that each tuple has the same probability to hit a bucket and then avoid the effect of data skewness. However, it is worth noting that Hippo has a significant size reduction when the data is sorted on TPC-H OrderKey attribute. In this case, Hippo only contains five index entries and each index entry summarizes thousands of pages. When data is totally sorted, Hippo keeps summarizing pages until the first 20% of the complete histogram buckets (No.1 - 80) are hit, then the next 20% (No. 81 - 160), and so forth. Therefore, Hippo cannot achieve competitive query time in this case.

In addition, BRIN exhibits the smallest index size among the three indexes since it only stores page ranges and corresponding value ranges (min and max values). Among different versions of BRIN, BRIN-32 exhibits the largest storage overhead while BRIN-512 shows the lowest storage overhead because the latter can summarize more pages per entry.

On the other hand, as Figure 23 depicts, Hippo and BRIN consume less time to initialize the index as compared to the B^+-Tree. That is due to the fact that the B^+-Tree has numerous index entries (tree nodes) stored on disk while Hippo and BRIN have just a few index entries. Moreover, since Hippo has to compare each tuple to the complete histogram which is kept in memory temporarily during index initialization, Hippo may take more time than BRIN to initialize itself. Different versions of BRIN spends most of the initialization time on scanning the data table and hence do not show much time difference.

4.1.7.3 Query Response Time

This section studies the query response time of the three indexes, B^+-Tree, Hippo and BRIN (128 pages by default). I first evaluate the query response time of the three indexes when different query selectivity factors are applied. Then, I further explore the performance of each index using the TPC-H benchmark queries which deliver industry-wide practical queries for decision support.

Queries with different selectivity factors This experiment studies the query execution performance while varying the selectivity factor from 0.001%, 0.01%, 0.1% to 1%. According to the Hippo cost model, the corresponding query time costs in this experiment are $0.2Card$, $0.2Card$, $0.2Card$ and $0.8Card$. The indexes are built on TPC-H Lineitem table PartKey attribute (TPC-H), Exponential table RandomNumber attribute, Wikipedia table PageCount attribute, NYC Taxi table pick-up location attribute. I also compare different versions of BRIN (BRIN-32 and BRIN-512) on TPC-H PartKey attribute.

As the results shown in Figure 24, all the indexes consume more time to query data on all datasets with the increasing of query selectivity factors. All versions of BRIN are two or more times worse than B^+-Tree and Hippo at almost all selectivity factors. They have to scan almost the entire tables due to their very insufficient page summaries - value ranges. Moreover, B^+-Tree is not better than Hippo at 0.1% query selectivity factor although it is faster than Hippo at low query selectivity factors like 0.001% and 0.01%. Actually, the performance of Hippo is very stable on all datasets including highly skewed data and real life data. In addition, Hippo consumes much more time at the last selectivity factor 1% because it has to scan many more pages as predicted by the cost model. Compared to the B^+-Tree, Hippo maintains a competitive query response time performance at selectivity factor 0.1% but consumes 25 - 30 times less storage space. In contrast to BRIN, Hippo achieves less query response time at the small enough index size. Therefore, I may conclude that Hippo makes a good tradeoff between query response time and index storage overhead at medium query selectivity factors, i.e, 0.1%.

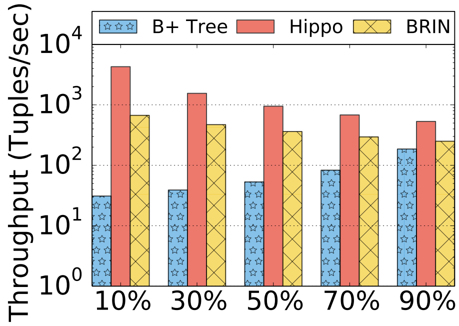
Evaluating the cost model accuracy This section conducts a comparison between the estimated query I/O cost and the actual I/O cost of running a query on Hippo. In this experiment, I vary the query selectivity factors to take the values of 0.001%, 0.01%, 0.1%, and 1%. Hence, the average number of buckets hit by the query predicate ($SF * H$) should be 0.004, 0.04, 0.4, and 4 respectively. However, in practice, no in-boundary queries can hit less than 1 bucket. Therefore, the average hit buckets by predicates are 1, 1, 1 and 4. Given $H = 400$ and $D = 20\%$, the query I/O cost estimated by Equation 4.8 is $\frac{Card}{pageCard} * (0.05\% + 20\%|20\%|20\%|80\%)$. I observe that: (1) Queries for the first three SF values yields pretty similar I/O cost. That matches the experimental results depicted in Figure 24. (2) The I/O cost of scanning index entries consumes $\frac{Card}{pageCard} * 0.05\%$ which is at least 40 times less than that of filtering false positive pages.

Table 8: The estimated query I/O deviation from the actual query I/O for different selectivity factors

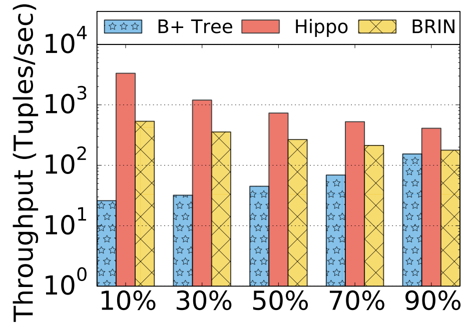
SF	0.001%	0.01%	0.1%	1%
TPC-H	0.02%	0.02%	0.21%	6.18%
Exponential	0.37%	0.37%	0.35%	12.69%
Wikipedia	0.91%	0.91%	1.19%	9.10%
NYC Taxi	0.87%	0.87%	0.51%	13.39%

As Table 8 shows, the cost model exhibits high accuracy. Furthermore, the cost model accuracy is stable especially for the first three lower selectivity factors. However, when $SF = 1\%$, the accuracy is relatively lower especially on Exponential and NYC Taxi table. The reason behind that is two-fold: (1) The 1% selectivity factor query predicate may hit more buckets than the other lower SF values. That leads to quite different overlap situations with partial histograms. (2) The complete height balanced histogram, maintained by the DBMS, does not perfectly reflect the data distribution since it is created periodically using some statistical approaches. Exponential and NYC Taxi tables exhibit relatively more clustered/skewed data distribution. That makes it more difficult to reflect their data distribution accurately. On the other hand, the histogram of a uniformly distributed TPC-H table is very accurate so that predicated I/O cost is more accurate in this case.

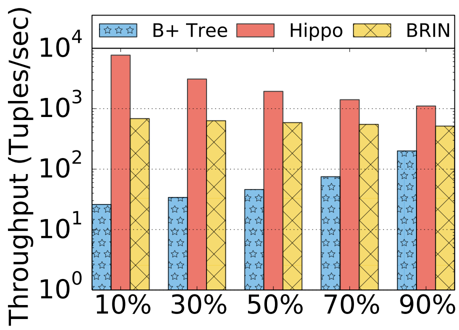
TPC-H benchmark queries This section compares Hippo to the B$^+$-Tree and BRIN using the TPC-H benchmark queries. I select all TPC-H benchmark queries



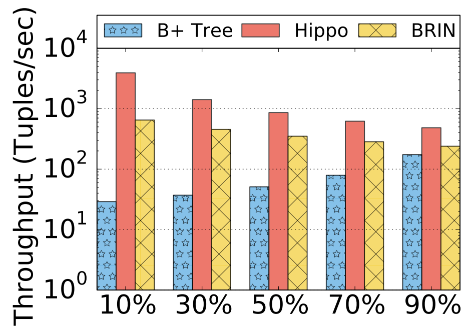
(a) TPC-H



(b) Exponential



(c) Wikipedia



(d) NYC Taxi

Figure 26: Throughput on different query / update workloads (logarithmic scale)

that contain typical range queries and hence need to access an index. I then adjust their selectivity factors to 0.1% (i.e., one week reports). I build the three indexes on the L_ShipDate (Query 6, 7, 14, 15 and 20) and L_ReceiptDate (Query 12) attributes in the Lineitem table as required by the queries. The qualified queries, Query 6, 7, 12, 14, 15 and 20, perform at least one index search (Query 15 performs twice) on the evaluated indexes.

Table 9: Query response time (Sec) on TPC-H

Index type	Q6	Q7	Q12	Q14	Q15	Q20
B^+-Tree	2450	259000	2930	2670	4900	3500
Hippo	2700	260400	3200	3180	5400	3750
BRIN	5600	276200	6200	6340	11300	6700

As Table 9 depicts, Hippo achieves similar query response time to that of the B^+-Tree and runs around two times faster than BRIN on all selected TPC-H benchmark queries. It is also worth noting that the time difference among all three indexes becomes non-obvious for Query 7. That happens because the query processor spends most of the time joining multiple tables, which dominates the execution time for Q7.

4.1.7.4 Maintenance Overhead

This experiment investigates the index maintenance time of three kinds of indexes, B^+-Tree, Hippo and BRIN, on all datasets when insertions or deletions. The indexes are built on TPC-H Lineitem table PartKey attribute, Exponential table RandomNumber attribute, Wikipedia table PageCount attribute, and NYC Taxi table Pick-up location attribute. This experiment uses a fair setting which counts the batch maintenance time after randomly inserting or deleting a certain amount (0.0001% , 0.001%, 0.01%, and 0.1%) of tuples. In addition, after inserting tuples into the parent table, the indexes' default update operations are executed because they adopt an eager strategy to keep indexes up to date. However, after deleting the certain amount of tuples from the parent table, I rebuild BRIN from scratch because BRIN does not have any proper update strategies for deletion. I also compare different versions of BRIN (BRIN-32 and BRIN-512) on TPC-H PartKey attribute.

As depicted in Figure 25 (in a logarithmic scale), Hippo costs up to three orders of magnitude less time to maintain the index than the B^+-Tree and up to 50 times less time than all versions of BRIN. This happens because the B^+-Tree spends more time on searching proper index entry insert / delete location and adjusting tree nodes. On the other hand, BRIN's maintenance is very slow after deletion since it has to rebuild the index after a batch of delete operations.

4.1.7.5 Performance on Hybrid Workloads

This section studies the performance of Hippo in hybrid query/update workloads. In this experiment, I build the considered indexes on TPC-H Lineitem table PartKey attribute, Exponential table RandomNumber attribute, Wikipedia table PageCount attribute, and NYC Taxi table Pick-up location attribute. I use five different hybrid query/update workloads: 10%, 30%, 50%, 70% and 90%. The percentage here stands for the percentage of queries in the entire workload. For example, 10% means 10% of the operations that access the index are queries and 90% are updates. The average selectivity factor is 0.1%. The index performance is measured by throughput (Tuples/second) defined as the number of qualified tuples queried or updated per a given period of time. The results are given in Figure 26 (in logarithmic scale).

As it turns out in Figure 26, Hippo has the highest throughput on all workloads. Hippo and BRIN can have higher throughput at update-intensive workloads like 10% and 30%. That happens since Hippo and BRIN have less index maintenance time that that of the B^+-Tree. On the other hand, B^+-Tree achieves higher throughput on query-intensive workloads like 70% and 90%. This is due to the fact that B^+-Tree costs less or same query response time compared to Hippo. Therefore, I can conclude that Hippo performs orders of magnitudes better than BRIN and B^+-Tree for update-intensive workload. Furthermore, for query intensive workloads, Hippo still can exhibit slightly better throughput than that of the B^+-Tree at a much small index storage overhead.

4.1.8 Summary

In this section, I introduced Hippo a data-aware sparse indexing approach that efficiently and accurately answers database queries. Hippo occupies up to two orders of magnitude less storage overhead than de-facto database indexes, i.e., B^+-tree while achieving comparable query execution performance. To achieve that, Hippo stores page ranges instead of tuples in the indexed table to reduce the storage space occupied by the index. Furthermore, Hippo maintains histograms, which represent the data distribution for one or more pages, as the summaries for these pages. This structure significantly shrinks index storage footprint without compromising much performance on high and medium selectivity queries. Moreover, Hippo achieves about three orders of magnitudes less maintenance overhead compared to the B^+-tree and BRIN. Such performance benefits make Hippo a very promising alternative to index high cardinality attributes in big data application scenarios. Furthermore, the simplicity of the proposed structure makes it practical for DBMS vendors to adopt Hippo as an alternative indexing technique.

4.2 Extending Hippo to Index Big Spatial Data

To make sense of geospatial data, the first step is to digest the dataset in a database system. The user can issue spatial queries using SQL, e.g., find all Taxi trips to Laguardia airport. To speed up such queries, a user may build a spatial index, e.g., R-tree, on the location or geometry attribute. However, spatial index structures suffer from the same two issues as regular database indexing mechanisms: huge indexing overhead and slow maintenance speed. Even though classic database indexes [20, 39] improve the query response time, they usually yield close to 15% additional storage overhead. It results in non-ignorable cost in massive-scale spatial database scenarios, e.g., Taxi trips locations. Moreover, existing database systems take a lot of time in initializing and bulk loading the spatial index (e.g, R-Tree or Quad-Tree [39, 32]) especially when the size of indexed spatial data reaches hundreds of Gigabytes or more. Furthermore, spatial indexes supported by state-of-the-art spatial database systems, e.g., PostGIS [74], are designed with the implicit assumption that the underlying spatial data does not change much. However, many modern applications constantly insert new spatial data into the database, e.g., inserting a new taxi trip record. Maintaining a database index incurs high latency since the DBMS has to locate and update those index entries affected by the underlying table changes. For instance, maintaining an R-Tree searches the tree structure and perhaps performs a set of tree nodes splitting or merging operations. That requires plenty of disk I/O operations and hence encumbers the time performance of the entire DBMS in update intensive application scenarios.

In this section, I first present my effort on extending the regular Hippo index to index geospatial data (denoted as Hippo-Spatial). I then present a comprehensive experimental analysis of classic and state-of-the-art spatial database indexing schemes supported in PostgreSQL (a popular open source database system) [75]. This includes a popular spatial tree indexing scheme (i.e., the GIST [47] implementation of R-Tree [39]), a Block Range Index for spatial data [13] (denoted as BRIN-Spatial) provided by PostgreSQL as well as the new indexing scheme, Hippo-Spatial. The results emphasize the fact that there is no one size that fits all when it comes to indexing massive-scale spatial data. The results also prove that modern database systems can maintain a lightweight index (in terms of storage and maintenance overhead) that is also fast enough for spatial data analytics applications.

4.2.1 Hippo-Spatial Overview

Index structure. Hippo is a data-aware sparse index. In context of spatial data, each Hippo (denoted as Hippo-Spatial) index entry is composed of two components: a dynamic disk page range and a histogram-based page range summary (depicted in Figure 27). In the summary, specifically, the simplified histogram (called partial histogram), each bit shows whether the corresponding two dimensional bucket presents (1) in this page range or not (0). The histogram-based summary is extracted from the two complete load balanced 1D histograms on X and Y axises, respectively (visualized histograms given in Figure 27). Such histograms are widely supported and naturally maintained by most existing DBMSs and execute with no much extra cost. Two 1D histogram buckets, one from X axis and one from Y, represent a 2D bucket. I number a 2D histogram bucket by its 1D buckets on X and Y. For example, bucket (1,1) represents the bucket on the lower-left corner of Figure 27 histogram. Hippo-Spatial iterates each parent table tuple and groups as ranges contiguous similar pages (in terms of data distribution). In the partial histogram of each page range, distinct histogram buckets hit by tuples are marked as 1 in corresponding bits. Hippo-Spatial ensures that the partial histogram in each index entry has the same density:

$$Partial\ histogram\ density\ (D)\ =\ \frac{\#\ Buckets_{value=1}}{\#\ Buckets_{complete\ histogram}}$$

Index search. When a spatial range query is issued, the system first locates the histogram buckets cover / intersect / covered by the query predicate and outputs a

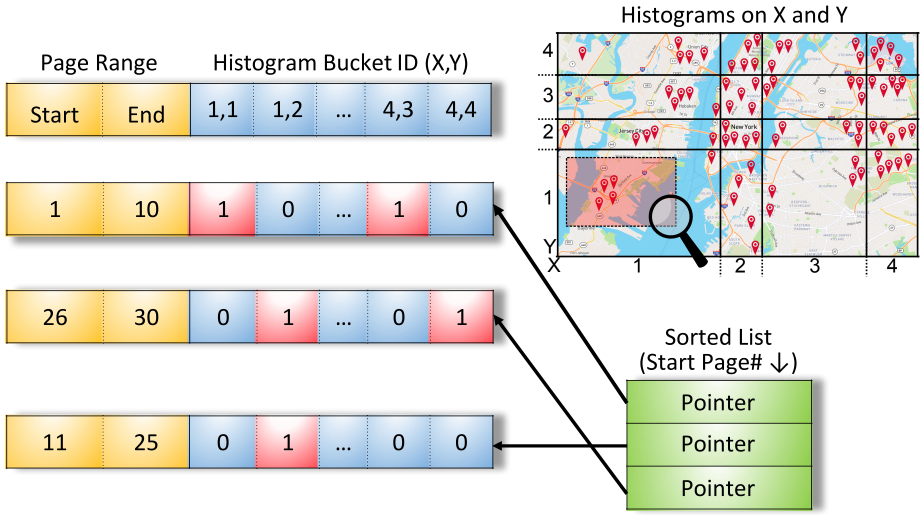


Figure 27: Hippo-Spatial Index Structure

partial histogram similar to the histogram-based summary maintained for each index entry. Then, the search algorithm reads each index entry and filters out the index entries for which the histogram-based summary has no common buckets with the query predicate. For all index entries that match the query predicate, the search algorithm inspects the corresponding disk pages and the qualified data tuples are returned.

Index maintenance. When a new tuple is inserted, Hippo-Spatial updates the index entries in an eager manner. It first finds the 2D histogram bucket where the tuple falls in and then runs a binary search on index entry sorted list to locate the page range which the tuple belongs to. The sorted list maintains a list of index entry pointers that are sorted in the ascending order of their start page ID. If this tuple hits a distinct histogram bucket, the partial histogram in Hippo-Spatial index entry will set the corresponding bit to 1; if no distinct buckets hit, Hippo-Spatial does nothing instead. On the other hand, Hippo-Spatial deletion runs in a lazy manner. This means Hippo-Spatial updates the index entries only for a batch of deletion operations.

During the update, Hippo-Spatial scans the index entries and in case some tuples in a certain page range are deleted, Hippo-Spatial will re-summarize all pages in this page range and update the index entry.

4.2.2 Experimental Analysis

I run all experiments on an Ubuntu 16.04 64 bit machine with 12 cores CPU (3.5 GHz per core), 128 GB memory, and 4 TB magnetic hard disk. I conduct the experiments on PostgreSQL 9.6 and PostGIS 2.3 with 128 MB default buffer pool. After fully loading the NYC city taxi trip dataset into PostgreSQL, the corresponding NYC Taxi trips table occupies 25 million PostgreSQL disk pages on the test machine. The size of PostgreSQL default buffer pool is rather small while the operating system memory is too large to be ignored. To avoid the impact of pre-cached data, I clear OS cache before each single transaction. I leverage the `EXPLAIN ANALYZE`, a PostgreSQL built-in performance analysis tool, to capture the execution time of all transactions and count the disk I/O operations. I use the default PostgreSQL 9.6 settings in all experiments. I use the `(CREATE INDEX)` (given below) to build the specified index on top of the NYC taxi trip table in PostgreSQL:

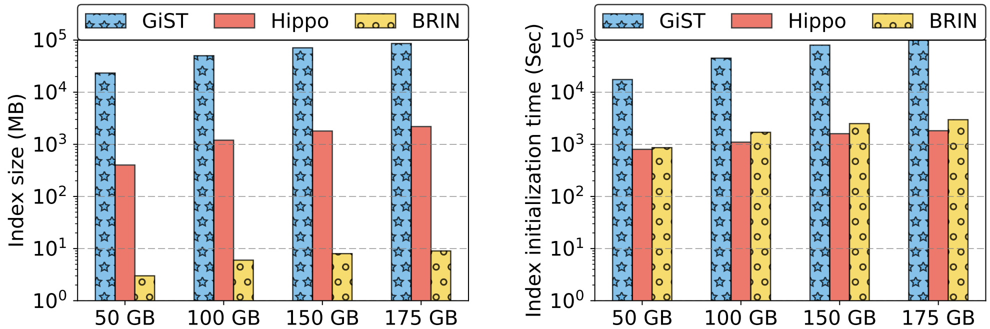
```
CREATE INDEX hippo_idx ON NYCTaxi USING Hippo (PickUpLocation);
```

All indexes are built on the NYC Taxi dataset pick-up location (i.e., latitude and longitude coordinate) attribute. For the sake of GIST and BRIN-Spatial, the latitude and longitude coordinates are represented by a single coordinate attribute in PostgreSQL compatible geometry format. BRIN-Spatial allows a parameter called Pages Per Range (P) which specifies the number of parent table pages summarized by each index entry. I use 32, 128 (default) and 512 to tune BRIN-Spatial. Hippo-Spatial accepts a parameter named Density (D) to control the partial histogram density inside each index entry. Its performance is also impacted by the number of buckets in the complete histogram (H). I choose three parameter combinations to tune Hippo-Spatial: (1) D = 20% H = 400 (default setting) (2) D = 40% H = 400 (3) D = 20% H = 800. All indexes use their default settings unless otherwise stated.

I issue a spatial range query on NYC Taxi table with a particular query window and qualified tuples are returned to the psql front-end. The format used in the experiments is:

```
EXPLAIN ANALYZE SELECT count(*) FROM NYCTaxi WHERE <predicate>;
```

The predicate represents a spatial range query window targeted at the pick-up attribute written in an index-dependent format. All insertions work in an eager manner to ensure the query correctness. In the experiments, I use the `(INSERT INTO NYCTaxi VALUES (aTrip))` SQL command to inserts a new Taxi trip tuple in the NYC taxi trips table. I also use `(COPY NYCTaxi FROM aFile)` command to insert a batch of tuples in a single operation in order to avoid unnecessary I/O. Nonetheless, it still performs the insertion/index update tuple by tuple. In PostgreSQL, a `DELETE` operation just makes a note on the deleted tuples and hides them from the output instead of immediately removing them physically. That is due to the fact that clearing and recycling deleted tuples' physical space is a time-consuming process. All physical deletions and corresponding index updates only happen when the `VACUUM` command is invoked. The `VACUUM` command runs periodically but also accepts manual invocation from the user.



(a) Index size

(b) Index initialization time

Figure 28: Indexing overhead on different data scales (logarithmic scale)

4.2.2.1 Studying the Indexing Overhead

This section studies the indexing overhead incurred by the three compared indexing schemes. I build three indexes on different sizes of the New York Taxi Trip data and record the corresponding overhead (Figure 28) including index size and index initialization time. Results of using different index parameters are described in Figure 29 and Figure 30.

Index Size As depicted in Figure 28a, Hippo-Spatial occupies close to two orders of magnitude less storage space than GIST. That happens due to the fact that GIST stores the pointers of hundreds of millions of Taxi trips in the table and maintains a Minimum Bounding Rectangle in each tree node. On the other hand, Hippo-Spatial only stores disk page ranges and MBR summaries. A tuple pointer is a physical address that consists of a disk page ID and slot ID. Once the index search is completed, GIST collects the pointers and passes them to the DBMS. Given a tuple pointer, the DBMS directly jumps to the specified address and retrieves the embedded tuple without any rechecks. Retrieving a small amount of pointers during queries is fast, yet storing 1.1 billion tuple pointers in an index is very space-consuming. In addition, each MBR

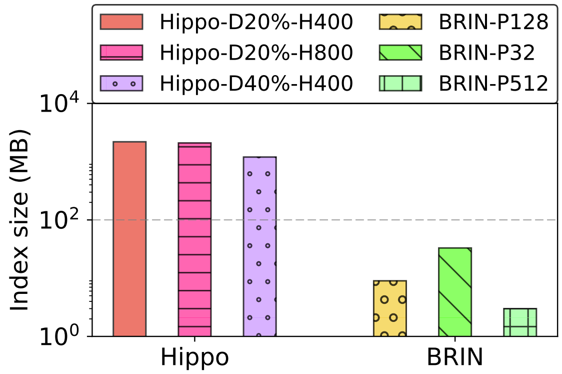


Figure 29: Index size (log. scale)

is represented by four double values, minimum X and Y, maximum X and Y, also occupies non-negligible storage space. On the contrary, each Hippo-Spatial index entry only contains a disk page range and a concise summary. Generally speaking, a disk page may store 50 - 100 tuples, and that is why Hippo-Spatial incurs much less storage overhead.

As given in Figure 28a, Hippo-Spatial leads to more storage overhead than BRIN-Spatial. That happens because Hippo-Spatial, as opposed to BRIN-Spatial, is data-aware and hence speeds up the search process. Each Hippo-Spatial index entry stores a histogram-based page summary instead of a simple MBR. Nonetheless, the extra storage space occupied by Hippo-Spatial is relatively small since its size is less than 1% of the indexed table.

Figure 29 studies the storage overhead of both BRIN-Spatial and Hippo-Spatial using different parameter settings. For instance, Hippo-D20%-H400 denotes a hippo index with density set to 20% and the number of histogram buckets set to 400 and BRIN-P128 denotes a BRIN-Spatial index with 128 pages per range. Hippo-Spatial occupies 100 times larger disk space than BRIN-Spatial. That makes sense because each index entry in Hippo-Spatial maintains a histogram-based summary of a dynamic

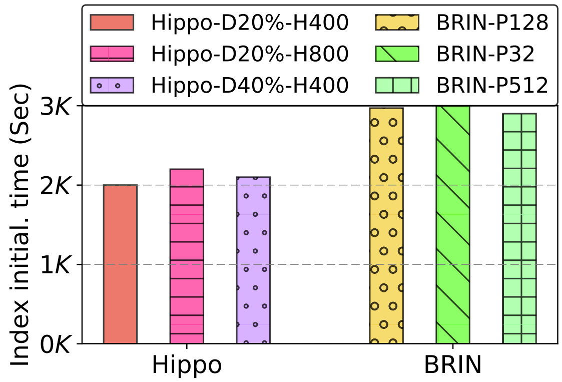


Figure 30: Initialization time

page range while BRIN-Spatial only stores the Minimum Bounding Rectangle per each page range. Each summary in Hippo-Spatial represents a partial histogram and each bucket in this histogram is represented by a single bit. Although Hippo-Spatial compresses these partial histograms, they are still much larger than a simple MBR. As the number of pages per range increases, BRIN-Spatial occupies less disk space since it summarizes more pages within one range at the cost of slower query response time. For different Hippo-Spatial parameter combinations, The higher the histogram density, the more pages each Hippo-Spatial index entry summarizes. That will also lead to more tuples being summarized by each index entry. Maintaining the same density but increasing the total number of histogram buckets leads to an increase in the storage space occupied by Hippo-Spatial. That happens because more complete histogram buckets also leads to more tuples hitting more distinct buckets in each partial histogram.

Index initialization time Figure 28b depicts the index initialization time incurred by creating each of the three indexes in PostgreSQL. The system takes the same time to bulk load Hippo-Spatial and BRIN-Spatial because each of them scans the indexed table tuple by tuple and summarizes each encountered tuple using an in-memory validation operation. The only difference is that, given a tuple, Hippo-Spatial finds the histogram bucket to which the tuple belongs using binary search while BRIN-Spatial checks whether the retreived tuple is covered by the temporary MBR and updates the MBR if needed. Moreover, PostgreSQL spends two orders of magnitude more time to bulk load GIST compared to BRIN-Spatial and Hippo-Spatial. This happens because the initialization algorithm in GIST is rather complex and requires a large number of temporary disk files to decide the boundries of the minimum bounding rectangles. Hence, the intensive disk I/O cost encumbers the initialization performance of GIST.

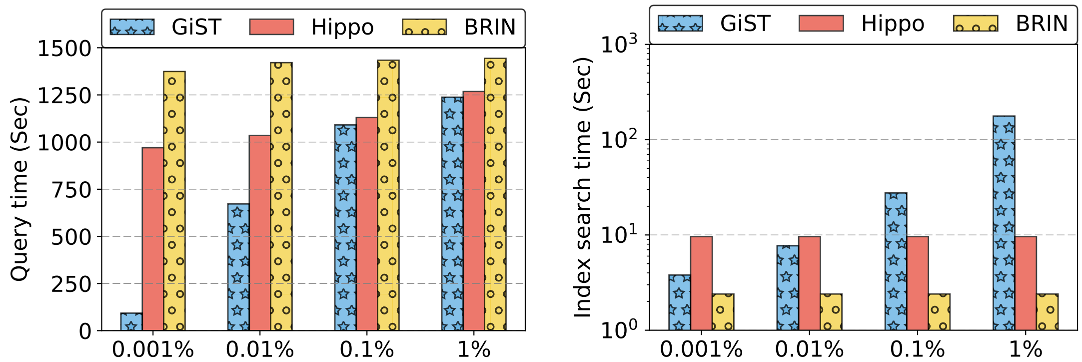
Figure 30 depicts how a variety of parameters settings impact the initialization time of both BRIN-Spatial and Hippo-Spatial. Hippo-Spatial takes 30% less initialization time than BRIN-Spatial. That happens due to the fact that the index initialization algorithm makes use of a temporary in-memory data structure (denoted TmpEntry) to store the to-be-persisted index entry. For BRIN-Spatial and Hippo-Spatial, TmpEntry keeps summarizing new incoming tuples and updates MBR for BRIN-Spatial (partial histogram for Hippo-Spatial) if needed. This process continues until BRIN-Spatial reaches pages per range limit or Hippo-Spatial reaches the density limit. Then, TmpEntry will be serialized and persisted to disk. However, in most cases of Hippo-Spatial, the TmpEntry data structure is rarely updated because TmpEntry only notes distinct histogram buckets hit by the scanned tuples. Unlike Hippo-Spatial, BRIN-Spatial initialization algorithm keeps updating the MBR as long as the newly summarized tuple not fully covered by the MBR. Such frequent TmpEntry updates lead to the gap in the initialization time.

4.2.2.2 Evaluating the Query Response Time

This section studies the query execution performance time using each of the three considered indexing indexes. To identify the proper scenarios for different indexes, I define two metrics of spatial range query: spatial range query selectivity and query range area size. The categorized results are given in Figures 31 and 33.

Varying the spatial range query selectivity factor This section studies the impact of varying the spatial range query selectivity factor on the query response time. The selectivity factor of a given spatial range query is calculated as the ratio of the total NYC taxi trips returned by running the spatial range query over the total number trips stored in the database. I vary the average spatial range query selectivity from 0.001%, 0.01%, 0.1% to 1%. To generate the query workload with average selectivity, I first create GIST index on the pick-up/drop-off location and randomly select a set of query points from the table. Then, I use each query point to issue a K Nearest Neighbors (KNN) searches on the NYC taxi table. The number K refers to the number of tuples returned by 0.001% - 1% selectivity queries. For each KNN query, the returned K^{th} nearest neighbor and its mirror point against the query point represent a query range window that has the specified range selectivity. The generated spatial range queries are then used to run the experiments and the reported query execution time in Figure 31 represents the average time PostgreSQL took to run the query workload.

As shown in Figure 31, GIST exhibits two orders of magnitude faster query execution performance than Hippo-Spatial and BRIN-Spatial on highly selective queries (0.001% selectivity factor). As the spatial range query selectivity factor becomes higher (lower selectivity), the query execution time gap between GIST and



(a) Query time (b) Index probe time (log. scale)

Figure 31: Varying the spatial range query selectivity factor

Hippo-Spatial diminishes. For 0.1% and 1% selectivity factors, Hippo-Spatial is able to achieve similar query execution performance to that of GIST. That happens due to the fact that, for highly selective queries (e.g., 0.001% selectivity), GIST's balanced tree structure is able to prune disjoint subtrees and retrieve only a small amount of qualified NYC taxi tuples to recheck. On the other hand, Hippo-Spatial still has many more possible qualified page to inspect. For less selective queries (selectivity factor 0.1% and 1%), GIST also has to retrieve more tuples for further inspection and that is why it has similar performance to that of Hippo-Spatial. However, BRIN-Spatial exhibits the slowest query execution performance as compared to GIST and Hippo-Spatial. The main reason is that all Minimum Bounding Rectangles store with each index entry in BRIN-Spatial span the entire New York City metropolitan area and BRIN-Spatial actually inspects almost all disk pages occupied by the NYC taxi table to process queries with different selecitvities.

Figure31b describes the index probe time on different selectivity factors. The index probe time refers in particular to the time these indexes spend on searching index entries when a query is issued. That excludes the time the database system takes to read the data pages. For GIST, the index probe time stands for the time GIST

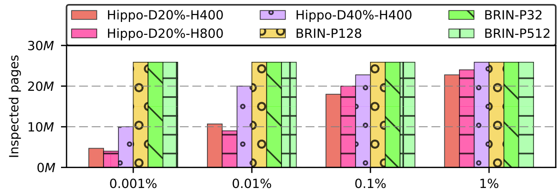
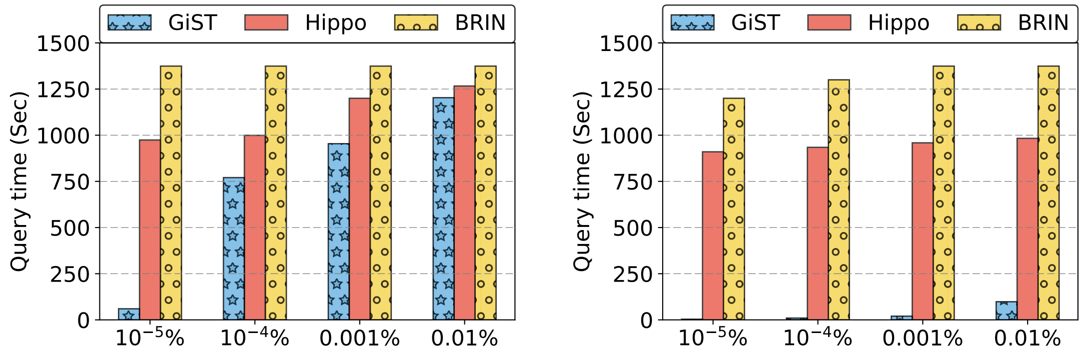


Figure 32: Inspected data pages on different query selectivities

used to find all qualified tuple pointers. The upcoming GIST refine and data page retrieval phase is taken care of by PostgreSQL. For BRIN-Spatial and Hippo-Spatial, the index probe time represents the time these indexes spend on traversing all index entries. It is obvious that the index probe time for BRIN-Spatial and Hippo-Spatial is constant for all spatial range selectivity factors. That happens due to the fact that BRIN-Spatial and Hippo-Spatial always scan all index entries. On the other hand, for higher selectivity factors, GIST have to expand its probe range and go to lower tree levels. Figure 31b shows that the index probe time of GIST, in fact, increases exponentially.

Figure 32 depicts the total number of inspected data pages using different index parameters. Both BRIN-Spatial and Hippo-Spatial need to inspect possible qualified pages for retrieving the truly qualified tuples. As given in Figure 32, Hippo-Spatial inspects less pages than BRIN-Spatial. To be precise, Hippo-Spatial with 20% density inspects up to 6 times less NYC taxi data pages on 0.001% and 0.01% selectivity factors and BRIN-Spatial inspects up to 40% more disk pages for queries with 0.1% and 1% selectivity factors. That happens because Hippo-Spatial is able to prune more data pages since it only inspects page ranges which have joint histogram buckets with the spatial query predicate. Hippo-Spatial with 40% density and Hippo-Spatial with 800 histogram buckets (i.e., Hippo-D40%-H800) experience slower query execution time as compare to Hippo-Spatial. The partial histograms of Hippo-D40%-H800 are



(a) Queries in dense locations

(b) Queries in random locations

Figure 33: Query time issued in different spatial areas

too full and too many bits set to 1. That increases the probability that each index entry in Hippo-Spatial has joint buckets with the spatial query predicate. It is also worth noting that BRIN-Spatial in general (with various parameters setting) inspects the same number of data pages since it always inspects the entire table due to its data-agnostic nature.

Varying the spatial range area size This section studies the impact of varying the size of the spatial range area. The range area represents the area covered by the issued spatial range query. I have discussed the query response time for different query selectivity factors. However, users rarely issue spatial queries in strict accordance to the selectivity factor. Assume that a user observes the NYC taxi dataset on a web browser. The user usually searches dense areas. In fact, spatial data is alway highly skewed and sparse areas such as deserts are less interesting for analysts. I define two types of queries:

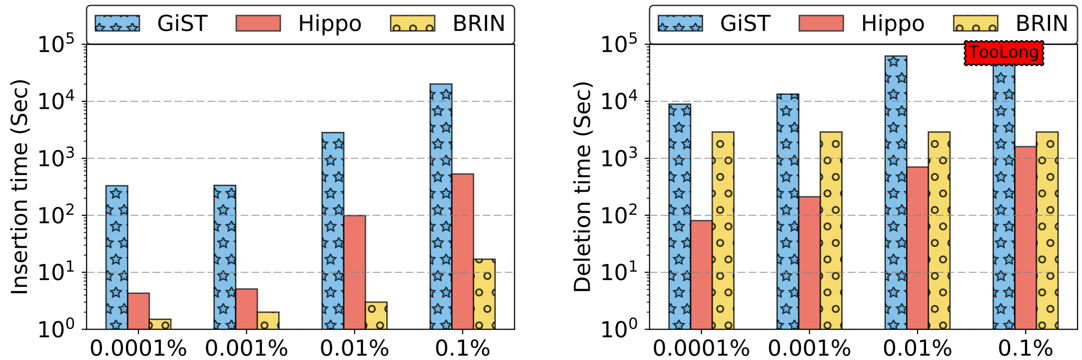
- random area spatial query (studied in Figure 33b): To generate such queries, I issue spatial range queries in random locations that lie within the New York City region.

- dense area spatial queries (studied in Figure 33a): To generate this workload, I limit the spatial queries to dense locations (e.g., Manhattan). A dense location contains a large number of Taxi trips. For instance, the hottest/densest data areas in New York Taxi dataset are Times Square, JFK airport and Laguardia airport.

Furthermore, I vary the range area size from $10^{-5}\%$ to 0.01%. Larger range area such as 0.001% or 0.01% exposes the region of a city while smaller range area such $10^{-5}\%$ exhibits the nearby businesses of my current location. Results are given in Figure 33. As it turns out in Figure 33b, GIST achieves the best query execution performance for queries generated in random locations within NYC. That happens because spatial data is always skewed and most spatial range queries only return few tuples. On the contrary, in Figure 33a, GIST takes much more time for queries issued in dense areas of NYC. That is due to the fact that the number of taxi trip records in the Manhattan (i.e., dense) area are far more than other areas in New York City. Moreover, Hippo-Spatial exhibits just a bit slower query execution performance than GIST. BRIN-Spatial, on the other hand, exhibits the slowest query execution performance since it has to inspect a large fraction of data pages.

4.2.2.3 Studying the Index Maintenance Overhead

This section studies the index maintenance overhead of all considered indexing schemes. I study the overhead incurred by two main index maintenance operations, i.e., insertion (see Figure 34a) and deletion time (see Figure 34b).



(a) Data insertion time

(b) Data deletion time

Figure 34: Index maintenance performance on different data update percentage

Insertion time This section studies the time the database system takes to update the index when new taxi trip inserted in the NYC taxi table. Note that updating the index due to tuple insertion is deemed necessary to ensure the correctness of future queries. This section compares the three indexing schemes after inserting a certain amount of tuples in the NYC taxi table. I vary the number of inserted tuples as ratio of the original data size, i.e., 0.0001%, 0.001%, 0.01% and 0.1% tuples of the index NYC taxi table and insert them using the `COPY FROM` SQL clause.

As depicted in Figure 34a, GIST exhibits the highest index maintenance overhead when new tuples are inserted. That happens because GIST spends too much time on locating the proper tree node. Furthermore, GIST spends a non-ignorable amount of time on splitting the tree nodes to accommodate the newly inserted key. Frequent tree structure traverse and adjustments result in tremendous disk I/Os. Hippo-Spatial and BRIN-Spatial exhibit more than two orders of magnitude less maintenance overhead for insertion. That is due to the fact that both Hippo-Spatial and BRIN-Spatial possess a flat index structure which is relatively less complex than GISTand hence easier to maintain. A newly inserted tuple leads to updating at most a single index entry. On the other hand, Hippo-Spatial takes more time time to insert a new tuple in contrast to BRIN-Spatial. That happens because Hippo-Spatial checks each new tuple against the complete histogram and updates the corresponding on-disk partial histogram if this new tuple hits a new distinct histogram bucket. On-disk updates happens more frequently in Hippo-Spatial since BRIN-Spatial only does physical entry updates when the new tuple is outside the corresponding MBR.

Deletion time In this section, I evaluate the time PostgreSQL takes to maintain each of the three tested index structures in response to deleting a tuple(s) from the NYC taxi trip table. I vary the percentage of deleted tuple to take 0.0001%, 0.001%, 0.01% and 0.1% values.

As shown in Figure 34b, Hippo-Spatial achieves close to two orders of magnitude better performance than GIST) in handling the `DELETE` operation. For the sake of batch deletion, Hippo-Spatial re-summarizes an index entry that contain many deleted tuples in one go meanwhile GIST searches for the affected tree nodes and sometimes merges the affected tree nodes in response to tuple deletion. On the other hand, BRIN-Spatial follows a naive lazy update strategy that rebuilds the entire index after a fixed number of tuples is deleted from the indexed table. That explains why Hippo-Spatial achieves close to an order of magnitude better performance BRIN-Spatial on low deletion percentages. The performance gap slightly decreases when a large percentage of the table is deleted because Hippo-Spatial has to re-summarize most index entries in that case, which is equivalent to re-building the whole index.

Hybrid workload performance Figure 35 compares the performance of three indexes in hybrid query/update workloads. I generated five query / update workloads that vary the percentage of issued search operations as compared to the update operations, named after the percentage of search operations in the entire workload: 10%, 30%, 50%, 70% and 90%. Each workload consists of a thousand operations,

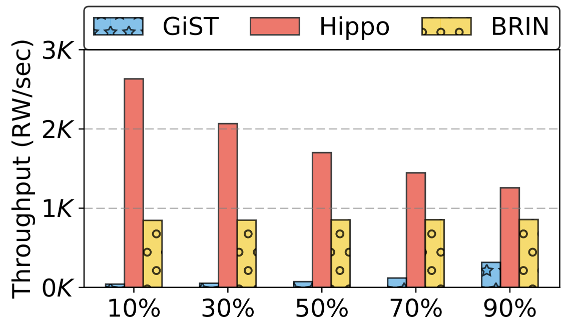


Figure 35: Throughput

which represent either index search or data update operations. In the experiments, I measure the system throughput achieved for each workload. The throughput is measured in terms of the number of operations per second. In each workload, the average spatial query selectivity factor is set to 0.01% while the average number of updated tuples is set to 0.01%

As it turns out in Figure 35, GIST yields the lowest system throughput. That happens because GIST spends too much time on index maintenance. BRIN-Spatial works faster than GIST due to fast index maintenance although it incurs high latency when performing search operations. Hippo-Spatial consistently achieves the highest system throughput, as compared to BRIN-Spatial and GIST. That is explained by the fact that Hippo-Spatial exhibits better index maintenance performance than GIST and also exhibits a competitive query response time. Although Hippo-Spatial is outperformed by BRIN-Spatial when performing insertion operations, Hippo-Spatial still achieves higher throughput than BRIN-Spatial given its relatively superior query execution performance and fast data deletion operations. In summary, I can conclude that Hippo-Spatial and BRIN-Spatial are more suitable for update-involved workloads while Hippo-Spatial outperforms BRIN-Spatial due to better query response time and faster data deletion.

4.2.3 Summary

Through extensive experiments, I presented a comprehensive analysis of classic and state-of-the-art spatial database indexing schemes supported in PostgreSQL, GIST, Hippo-Spatial and BRIN-Spatial. Below, I share my key insights through the following learned lessons:

- ***Do not create GIST (i.e., spatial tree index) when the database system is deployed on a storage device with high $ per GB.*** The storage overhead introduced by GIST created over the NYC taxi dataset is 84 GB, which is close to 50% of the original data size. Note that the dollar cost increases dramatically when the DBMS is deployed on modern storage devices (e.g., SSD and Non-Volatile-Ram) since they are still more than an order of magnitude expensive than classic Hard Disk Drives (HDDs). As per Amazon.com and NewEgg.com, the dollar cost per storage unit for HDD and SSD are 0.04 and 1.4 $/GB, respectively. Instead, the user may consider Hippo-Spatial and BRIN-Spatial. to reduce the overall storage cost since these indexes only occupy between 0.1 and 1 % as compared to the original dataset.
- ***Do not use BRIN-Spatial. or Hippo-Spatial for Yelp-like applications.*** Applications like Yelp usually issue very highly selective spatial range queries that retrieve point-of-interests (e.g., 0.001% range query selectivity) and present them to the end-user. As per the experiments, GIST is deemed a perfect indexing scheme for Yelp-like applications given its superior performance in executive highly selective spatial range queries. Furthermore, spatial data (i.e., Point-of-Interests) in Yelp are not dense. That is due to the fact that every longitude

and latitude location on the surface of the earth contains a few (usually one) Point-of-Interests (or buildings).

- ***Use Hippo-Spatial for spatial analytics applications over dynamic and dense spatial data.*** NASA constantly collects Earth science data (e.g., weather, pollution, socioeconomic data) [26]. Earth science data is quite dense and new data is inserted into the system on a daily basis. Furthermore, since geospatial data in such applications is typically consumed as aggregate visualizations (e.g., Heatmap, Cartogram), spatial range queries on such data are not quite selective (selectivity factor between 0.1% and 1%) as in Yelp-like applications. Having said that, Hippo-Spatial is deemed the perfect for such data given: (1) its small storage footprint and low maintenance overhead compared to GIST and (2) its superior query execution performance over selective queries and higher throughput compared to BRIN-Spatial.

Chapter 5

A SAMPLING MIDDLEWARE FOR INTERACTIVE GEOSPATIAL VISUALIZATION DASHBOARDS

In this chapter, I first illustrate the need of interactive geospatial data analytics. Then I explain the design of Tabula [103], a sampling middleware that makes the user experience with the visualization dashboard more seamless and interactive. Finally, I conducted extensive experiments to study the performance of Tabula and several other systems.

5.1 Introduction

When a user explores a spatial dataset using a visualization dashboard, such as Tableau and ArcGIS, that often involves several interactions between the dashboard and the underlying data system. In each interaction, the dashboard application first issues a query to extract the data of interest from the underlying data system (e.g., PostGIS and Apache Spark SQL), and then runs the visual analysis task (e.g., heat maps and statistical analysis) on the selected data. Based on the visualization result, the user may iteratively go through such steps several times to explore various subsets of the database.

Running example. Figure 36 depicts a dashboard that visualizes 700 Million taxi rides data stored as a 100 GB database table; each tuple represents a taxi ride with various attributes (i.e., columns) such as the pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances (denoted as **D**), passenger count (denoted

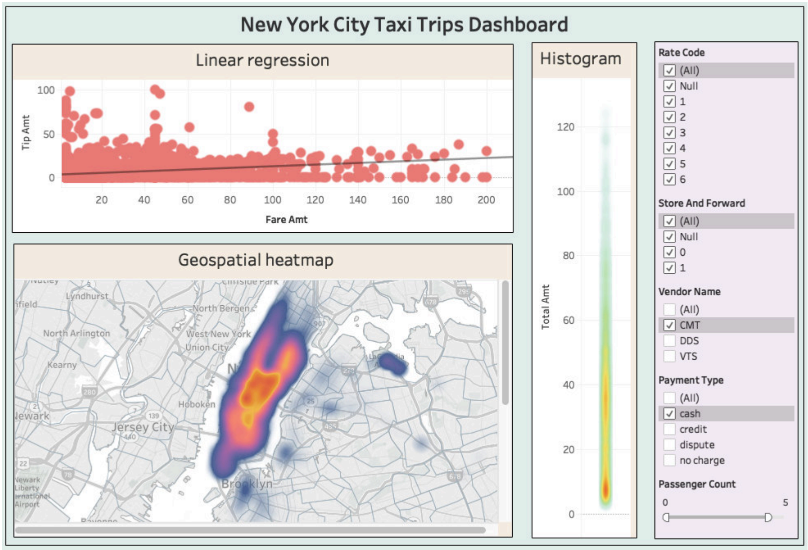


Figure 36: A real interactive spatial visualization dashboard in Tableau

as **C**), payment method (denoted as **M**), itemized fare amount, tip and so on. The user can first select all taxi rides paid by cash using filters on the right pane and then plots the pickup location of such rides on a heat map. She may then select taxi rides paid by credit card and render another heat map to visually compare the difference between the two maps.

Every interaction between the visualization dashboard and the underlying data system may take a significant amount of time (denoted as <u>data-to-visualization time</u>) to run, especially over large-scale data. The reason is two-fold: (1) The data-system query time proportionally increases with the volume of the underlying data table. Even scalable data processing systems such as Apache Spark and Hadoop, which parallelize the query execution, still exhibit non-negligible latency on large scale data. (2) Existing spatial visualization dashboards such as Tableau, ArcGIS and Apache Zeppelin work well for small to medium size data but do not scale to large datasets. Furthermore, since the user may perform various visualization effects on the same dashboard (e.g., 3 different tasks in Figure 36), practitioners would prefer to use a more generic approach to reduce the data-to-visualization time rather than install several different and isolated systems.

To remedy that, one approach that practitioners use is to draw a smaller sample of the entire data table (e.g., 1 million tuples) and materialize the sample in the database. This approach then keeps executing the dashboard on the materialized sample instead of the actual data set. The caveat is that running queries on the sample may lead to inaccurate visualization results since the query answer may significantly deviate from the actual answer especially for some small data populations. As shown in Figure 37, the approach that runs a query on the pre-built sample (denoted as *SampleFirst*) generates different visualization results in Tableau (Figure 37b) as compared to the approach that runs the query on the entire data table (Figure 37a). The *SampleFirst* approach even misses important visual patterns (the taxi rides from an airport, the red circle), and hence may mislead the user.

Recent research works such as Sample+Seek [24], BlinkDB [2], and SnappyData [76] address the problem of enhancing the accuracy of pre-built samples for approximate query processing. These approaches create stratified samples over multi-dimensional data to improve accuracy with a given confidence level. However, the pre-built stratified samples have no **deterministic** accuracy guarantee. So these systems may still need to perform some queries over the entire underlying table in an online fashion. Most importantly, all aforementioned approaches only support classic OLAP aggregate measures, such as COUNT, AVG, and cannot be easily extended to other types of data analysis (e.g., linear regression and most spatial visual effects in Figure 36). Instead of creating pre-built samples, an alternative approach runs data-system queries over the entire table for every iteration, draws a sample of the extracted population and sends

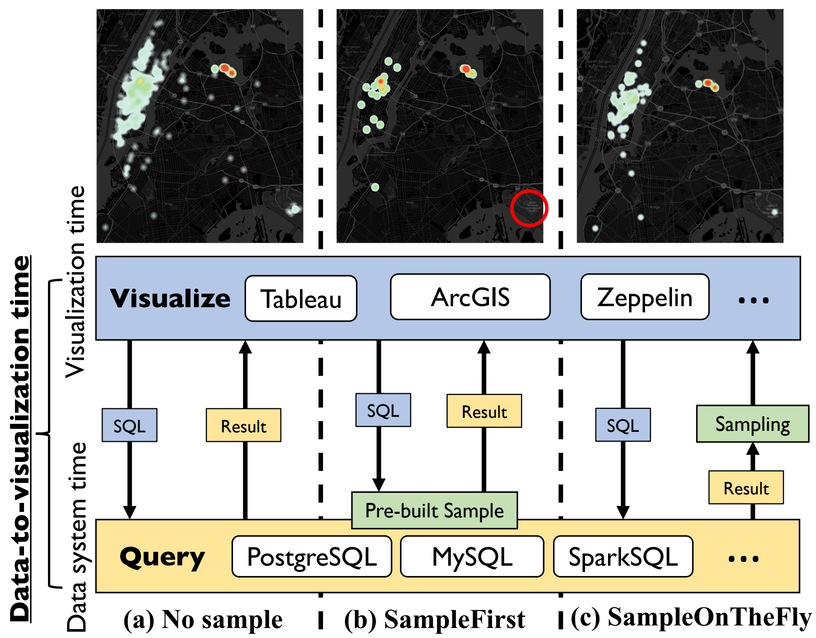


Figure 37: Iterations between spatial visualization dashboards and data systems with different sampling approaches

it back to the visualization dashboard to shorten the visualization time. Although this approach (denoted as *SampleOnTheFly*) can certainly achieve higher and deterministic accuracy for the selected population, it is prohibitively expensive since it has to query the original table to prepare the sample for every user interaction.

In this chapter, I present Tabula, a middleware framework that sits between a SQL data system and a spatial visualization dashboard to make the user experience with the dashboard more seamless and interactive. Tabula can seamlessly integrate with the existing data system infrastructure (e.g., PostgreSQL, SparkSQL). As opposed to Nanocube and its variants [60], Tabula (given its inherent design as a middleware system) can work in concert with existing visual exploration tools such as Tableau and ArcGIS. Similar to Tabula, POIsam [38] and VAS [71] propose an online sampling technique to produce samples specifically optimized for spatial visual analysis. However, as opposed to Tabula, POIsam and VAS resort to the SampleOnTheFly approach to guarantee the sampling quality, which takes its toll on the overall data-to-visualization (as I prove in Section 5.5).

Tabula adopts a materialized sampling cube approach, which pre-materializes samples, not for the entire table as in the SampleFirst approach, but for the results of potentially unforeseen queries (represented by an OLAP cube cell). Note that Tabula stores the sampling cube in the underlying data system. In each dashboard interaction, the system fetches a readily materialized sample for a given SQL query, which mitigates the data-system time. To scale, Tabula employs two strategies to reduce the sampling cube initialization time and memory utilization: (1) a partially materialized cube which only materializes local samples of those queries for which the global sample (the sample drawn from the entire dataset) exceeds the required accuracy loss threshold. (2) a sample selection technique to further reduce memory footprint. It finds similarities between different local samples, only persists a few representative samples, then uses the representative sample as an answer to many queries.

Since the dashboard application may show several types of visualization effects (see Figure 36), Tabula allows users to extend the system's functionality by declaring their own user-defined accuracy loss function that fits each specific visualization effect. The system automatically incorporates the user-defined accuracy loss function in the sampling cube initialization and representative sample selection algorithms. Moreover, it always ensures that the accuracy loss due to using the sample never exceeds a user-specified deterministic accuracy loss threshold (100% confidence). That happens because Tabula efficiently examines the accuracy loss for all unforeseen queries when initializing the sampling cube.

I built a prototype of Tabula on top of SparkSQL and conducted extensive experiments to study the performance of Tabula and several other systems such as SampleFirst, SampleOnTheFly, SnappyData [76] and POIsam [38]. Based on the experiments, Tabula can bring down the total data-to-visualization time (including both data-system and visualization times) of a heat map generated over 700 million taxi rides to 600 milliseconds with 250 meters user-defined accuracy loss. It could be up to 20 times faster than its counterparts. Besides, Tabula costs up to two orders of magnitude less memory footprint (e.g., only 800 MB for the running example) and one order of magnitude less initialization time than the fully materialized sampling cube approach.

It is worth noting that the techniques proposed in this chapter may be applied to both geospatial data and regular data visual analysis. For example, Section 5.2 shows that the generic user-defined accuracy loss function can be about statistical mean and geospatial heat maps. Having said that, I believe that geospatial visualization is the most important scenario on which Tabula has a direct impact.

5.2 Using Tabula

Figure 38 gives an overview of Tabula. A user must initialize Tabula by providing the following system parameters as input: (1) User-defined accuracy loss function (abbr. `loss()`): This function determines how to calculate the accuracy loss due to using the sample as opposed to the original query answer. (2) Accuracy loss threshold θ: this parameter decides the acceptable accuracy for all queries processed by Tabula (3) Target attribute *attr* on which `loss()` measures the accuracy loss (4) Cubed attributes: the set of attributes that will be used to build the sampling cube (e.g.,

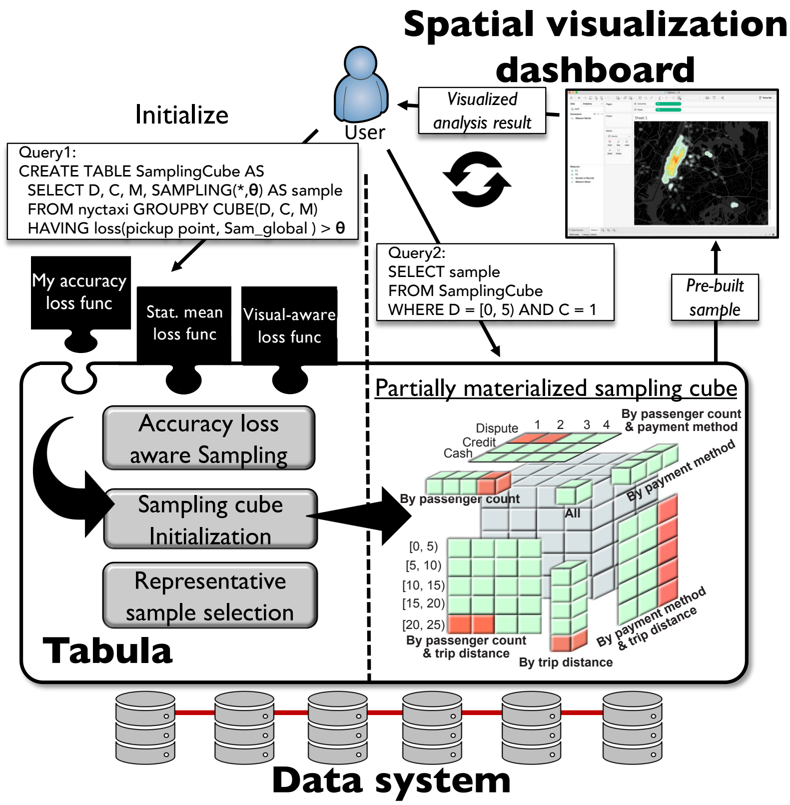


Figure 38: Tabula overview. Samples in red cells are materialized. DCM cuboid doesn't show due to visualization limitation.

attributes D, C and M). Data-system SQL queries will use these attributes in `WHERE` clause predicates. The user feeds such parameters to Tabula as follows:

```
CREATE TABLE [sampling cube name] AS
SELECT [cubed attributes], SAMPLING(*,[θ]) AS sample
FROM  [table name]
GROUPBY CUBE([cubed attributes])
HAVING [loss function name]([attr], Sam_global) > [θ]
```

where Sam_{global} represents a sample constructed by Tabula over the entire table using random sampling. `SAMPLING()` is a Tabula-specific function that takes a dataset represented as a set of tuples and produces a sample of that dataset such that the accuracy of the produced sample, compared to the original dataset, does not exceed the accuracy loss threshold θ. Using the above SQL query, Tabula leverages the underlying data system to initialize a materialized sampling cube. The data system can be any system that supports the `CUBE` operator. Query1 in Figure 38 is an initialization query for the running example.

Once the sampling cube is initialized, the user, via the spatial visualization dashboard, can issue SQL queries to Tabula, as follows:

```
SELECT sample  FROM  [sampling cube name]
WHERE [conditions]
```

After receiving this query, Tabula directly fetches a materialized sample from the sampling cube and returns it back to the visualization dashboard. This way, Tabula significantly reduces both the data-system time and visualization time. Besides, the system always guarantees with 100% confidence level that the accuracy loss from using the returned sample, as compared to the original query answer, does not exceed the accuracy loss threshold θ.

It is worth noting that the attributes in the `WHERE` clause must be a subset of the cubed attributes specified in the initialization query. Query2 in Figure 38 is an example query which asks for a sample of tuples that satisfy `D = [0, 5) AND C = 1`. The user can then issue subsequent queries with a different set of attributes to run the same or different analysis on various populations.

User-defined accuracy loss function. The visual analytics result obtained from a sample should be very close to that from the raw data. In this section, I formalize the difference as accuracy loss. There are many ways to compute accuracy loss, which serve different purposes. In fact, one size does not fit all. The accuracy loss highly depends on the type of analysis the user plans to perform. That is the main reason why Tabula provides a generic approach for the user to declaratively define her custom made accuracy loss function that suits the analytics task at hand. The body of this function is a user-defined scalar expression over several aggregate functions. The standard SQL syntax is given below (supported by databases such as PostgreSQL):

```
CREATE AGGREGATE loss(Raw, Sam)
RETURN decimal_value AS
BEGIN  scalar_expression END
```

Such a function takes raw data and sample data as input, then returns a decimal value which is the accuracy loss. For instance, consider an analytics task which requires a low relative error between the statistical mean of the sample and the statistical mean of the raw data. Such an accuracy loss function is implemented in Tabula as follows:

Function 1. `BEGIN` `ABS(`$\frac{AVG(Raw)\ -\ AVG(Sam)}{AVG(Raw)}$`)` `END`

Another example is the geospatial heat map on taxi pickup locations in Figure 36, where the accuracy loss function can stem from recent work on spatial visualization-aware sampling (VAS [71] and POIsam [38]). In that case, the user may implement the accuracy loss as the average minimum distance between the sample and raw data, as follows:

Function 2. `BEGIN` $\frac{1}{|Raw|}\sum_{x \in Raw} MIN_{s \in Sam}(loss_{pair}(x,\ s))$ `END`

where $loss_{pair}$(x, s) is the Euclidean distance, Manhattan distance or any distance metric between two data objects.

The third example is the linear regression analysis on trip tip amount VS. fare amount in Figure 36, where the accuracy loss function calculates the angle difference between the regression lines of raw data and sample data, as follows:

Function 3. `BEGIN ABS(`$angle(Raw) - angle(Sam)$`) END`

Given n tuples each of which has a 2D attribute (x_i, y_i), I use the following function to calculate the slope [34]:

$$slope = \frac{n\Sigma(x_i * y_i) - \Sigma x_i * \Sigma y_i}{n\Sigma x_i^2 - (\Sigma x_i)^2}$$

I then convert slope to angle (unit: degree °). Eventually, the corresponding visual analysis task plots the regression line of data of interest: $y = slope * x + intercept$. In this example, the loss function uses fare amount as x, tip amount as y.

Tabula requires that the accuracy loss function must be algebraic (see definitions in Section 2.5.2). To achieve that, all aggregate functions and mathematical operators involved in calculating loss(Raw, Sam) must be distributive or algebraic. In fact, many common aggregations satisfy this restriction [64] including SUM, COUNT, AVG, STD_DEV, MIN, MAX, DISTINCT, TOP-K, excluding MEDIAN.

Once the user defines the accuracy loss function, Tabula embeds such a function in the core components of the sampling cube. For instance, if the user defines the statistical mean-aware accuracy loss function (discussed previously) and sets the value of the accuracy loss threshold $\theta = 10\%$, Tabula will guarantee with 100% confidence that the relative error due to using the statistical mean of every sample in the cube will never exceed 10%. On the other hand, if the user uses geospatial heat map-aware sampling accuracy loss and sets the value of θ to an absolute loss value, 1 meter, then Tabula guarantees that the average min distance between the raw query result and the returned sample will never exceed 1 meter.

5.3 Materialized Sampling Cube

After the user issues the initialization query (presented in Section 5.2), Tabula builds a *partially* materialized sampling cube and stores it in the underlying data system. The system only materializes local samples for a selected set of cells in the sampling cube, namely iceberg cells. A cell that satisfies loss(cell data, Sam_{global}) $> \theta$ (SQL equivalent: `loss([target attribute], Sam`$_{global}$`)` $> \theta$) is called an iceberg cell. Otherwise, Tabula will use the global sample to answer a query corresponding to a non-iceberg cell. Figure 38 gives the layout of a sampling cube, which contains a cube table (see Figure 39(a)) and a sample table (see Figure 39(b)). All cells depicted in Figure 39(a) are iceberg cells. A cell in the materialized sampling cube is defined as $\langle a_1, a_2, ..., a_n : sample\_id \rangle$, where n is the number of cubed attributes. sample_id points to a sample in the sample table and many cells may share the same sample_ids because of Tabula's optimization in Section 5.4. Cell $\langle [0, 5), null, null : 1\rangle$ and $\langle [0, 5), 1, credit : 1\rangle$ both share the same sample set whose id is 1. 'null' indicates *. In the rest of this section, I will first explain how the sampling module of Tabula can harness the user-defined accuracy loss function to draw samples. I will then explain how the system finds iceberg cells and efficiently constructs the sampling cube using the sampling module.

5.3.1 Accuracy Loss-Aware Sampling

The sampling function (i.e., `SAMPLING(*,[`θ`])` in Section 5.3) aims at generating a sample with the objective to minimize the sample size while guaranteeing $loss(Raw, Sam) \leq \theta$. Since classic sampling algorithms such as random and stratified

D.	C.	M.	Sample ID
[0, 5)	(null)	(null)	1
[0, 5)	1	credit	1
[0, 5)	1	dispute	2
[0, 5)	1	(null)	1
[0, 5)	2	cash	3
[0, 5)	2	credit	2
[0, 5)	2	(null)	2
[0, 5)	3	dispute	3
[0, 5)	4	cash	3
…			

(a) A part of the cube table. This table only stores iceberg cells.

ID	Sample
1	{tuple,tuple,...}
2	{tuple,tuple,...}
3	{tuple,tuple,...}

(b) Sample table

Figure 39: Tabula sampling cube physical layout

sampling do not handle a user-defined accuracy loss function, the sampling module in Tabula employs a generic sampling algorithm, which works for a generic accuracy loss function. The sampling problem can be formally defined as follows:

Definition 4 (Sampling problem)**.** *Given a dataset T, an accuracy loss function ($loss()$), and an accuracy loss threshold θ, select a subset t from T such that: (1) $loss(T, t) \leq \theta$ and (2) The size of t is minimized.*

The sampling module in Tabula employs a greedy algorithm similar to the algorithm proposed by POIsam [38]. However, my algorithm guarantees that $loss(T, t) \leq \theta$ but the sample size may not be minimal. Algorithm 8 depicts the major steps of this algorithm: it first creates an empty sample set t which has $loss(T, t) = \infty$. In each greedy selection round, for every remaining tuple tp in T, it computes $loss(originalT, t + tp)$. OriginalT is the original raw dataset T. It always picks from T (without replacement) the tuple which has the minimum $loss$ and adds it to t. This algorithm keeps picking tuples from T and adds them into t until $loss(originalT, t) <= \theta$. Tabula further accelerates the greedy algorithm using the lazy-forward strategy of POIsam (not shown here). The final complexity of each

Algorithm 8: Greedy algorithm for sampling

```
Data: A dataset T, loss(), θ
Result: A sample t
Create an empty list t;
Create a copy of T originalT;
loss=∞;
while loss> θ do
    minTuple = NULL;
    foreach tuple tp in T do
        tp_loss= loss(originalT, t+tp);
        if tp_loss< loss then
            loss= tp_loss;
            minTuple = tp
    t.add(minTuple);
    T.remove(minTuple);
return t
```

greedy round is O(k*N) where N is the size of input data and k is much smaller than N.

Lemma 5.3.1. *The sampling function* `SAMPLING()` *can produce a sample whose loss*$<= \theta$*, in limited iterations.*

Proof. This lemma can be proved by contradiction. Let us assume that `SAMPLING()` cannot find the sample whose loss$<= \theta$ and keep running forever. According to the definition, the algorithm in `SAMPLING()` picks at least one tuple from the remaining tuples without replacement. It will definitely put all tuples into the sample set in the worst case scenario and have no candidate for the next round. However, at this moment, the sample set will be identical to the original dataset and the loss must be 0 which is absolutely $<= \theta$. So the assumption is contradictory to the algorithm definition in `SAMPLING()`. □

Lemma 5.3.2. *The sampling function* `SAMPLING()` *is a* **holistic** *aggregate function.*

Proof. The definition of holistic functions is given in Section 2.5.2. Since the sampling function embeds an arbitrary accuracy loss function and a threshold θ, I can assume that a user defines the loss function as Equation 1 and $\theta = 10\%$. Thus, for a cube cell, Tabula's sampling function needs to draw a local sample such that loss(Raw, Sam$_{local}$) $<= \theta$. I can also assume that an ancestor cell has grouped raw data $\{1, 2, 3, 4, 6, 7, 8, 9\}$ and two descendant cells have $\{1, 2, 3, 4\}$ and $\{6, 7, 8, 9\}$, respectively. If Tabula directly applies the sampling function to descendant cells, the qualified samples for descendants could be $\{2, 3\}$ and $\{7\}$, respectively. Next, Tabula applies the sampling function to the union of descendant samples - $\{2, 3, 7\}$ rather than the raw data of the ancestor. The produced sample is still $\{2, 3, 7\}$. The sample actually is not acceptable to the ancestor's raw data because loss(ancestor, $\{2, 3, 7\}$) $= 20\% > \theta$. To be short, applying the sampling function to descendant samples cannot always return a correct sample for the ancestor. This violates the definitions of both distributive and algebraic measures. $\square$

Any data cube with such a holistic aggregate function cannot leverage state-of-the-art cube construction approaches [37].

5.3.2 Sampling Cube Initialization

The straightforward way to initialize a sampling cube is as follows: First, Tabula draws a global random sample, called Sam$_{global}$, from the entire raw dataset. Second, it builds the sampling cube by running a set of GroupBy queries to calculate all cuboids in the cube (a cuboid [12] is a GroupBy query). This can be done via using the SQL CUBE operator in the underlying DBMS (see Query 1 in Figure 38). Given the grouped raw data of each cube cell, if applying the global sample to this cell satisfies the iceberg condition, i.e., loss(cell data, Sam_{global}) $> \theta$, Tabula will identify this cell as an iceberg cell and generate / materialize a local sample (denoted as Sam_{local}) for it.

However, the cost of using the classic CUBE operator to build the sampling cube increases exponentially with the number of cubed attributes. In Figure 36, each record may have five attributes (filters) and running the `CUBE` operator on these attributes requires $(2^5 - 1)$ GroupBy operations over the entire table. Tabula avoids that by learning which cuboid of the sampling cube really contains iceberg cells before actually building the cube, then all unnecessary GroupBys can be avoided. To achieve that, after drawing the global sample, the algorithm runs in two main stages, namely: Stage 1: Dry run for iceberg cell lookup and Stage 2: Real run for sampling cube construction.

5.3.2.1 Dry Run Stage: Iceberg Cell Lookup

In this stage, the system identifies all iceberg cuboids (cuboids that have iceberg cells), by scanning the raw table data only once. According to the aforementioned straightforward initialization query, Tabula applies two aggregate functions to each iceberg cell in the sampling cube: `SAMPLING()` and `loss()`. Based on the literature in OLAP data cube (see Section 2.5.2), I know that: if an aggregate measure is distributive or algebraic, existing algorithms only need to run the full table GroupBy operation once to build an initial cuboid and other cuboids can be built upon it. For a holistic aggregate measure, there are no better algorithms to materialize the aggregate measure for each cell other than building every cuboid from the raw data [37].

Table 10: Example tables generated in the dry run stage

D.	C.	M.
(null)	(null)	credit
[0, 5)	(null)	cash
[5, 10)	(null)	(null)
[0, 5)	1	(null)
[5, 10)	1	(null)
...		

(a) Iceberg cell table

D.	C.	M.
[0,5)	1	credit
[5,10)	1	credit
[15,20)	2	cash
[15,20)	3	cash

(b) Cuboid D,C,M

D.	C.	M.
[0, 5)	1	(null)
[5, 10)	1	(null)

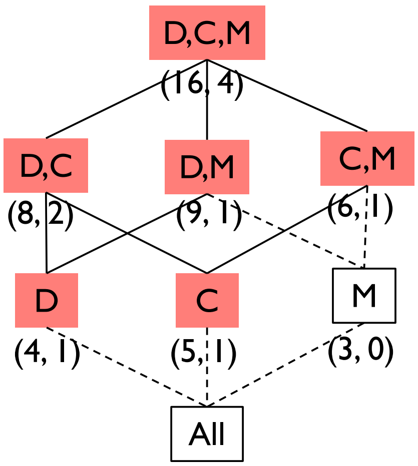
(c) Cuboid D,C

D.	C.	M.
[5, 10)	(null)	(null)

(d) Cuboid D

Since the sampling function is holistic, although Tabula only materializes local samples for some cells of the cube (iceberg cells), the underlying data system has to run (2^n - 1) full table GroupBy operations because it cannot speculate which cells are iceberg cells beforehand (n is the number of cubed attributes). However, since the loss function is algebraic, Tabula leverages such property in this stage by utilizing any existing cube initialization algorithms in Section 2.5.2 to efficiently build a partially materialized sampling cube. Such a cube only uses accuracy loss as the aggregate measure. This way, Tabula only accesses the raw data once to build the top/bottom cuboid and then all other cuboids can be derived from the cuboid itself.

The output of the dry run stage is an iceberg cell table (Table 10a). Tabula then derives iceberg cell tables for each cuboid (e.g., Table 10b,10c,10d). In addition, Tabula can also know the approximate number of all cells and iceberg cells in each cuboid by checking the global sample. Therefore, based on these outcomes, I can draw the lattice structure of Tabula (Figure 40a) without even computing any local samples for now. Each vertex of the lattice is a cuboid (i.e., GroupBy query) and letters indicate the attributes that appear on the grouping list of this cuboid. For instance, the top vertex DCM is the cuboid that has trip distance, passenger count



(a) Actual cube lattice

Distance	Count
[0, 5)	1
[0, 5)	2
[5, 10)	1
[5, 10)	3
[10, 15)	1
[10, 15)	2
[15, 20)	2
[15, 20)	3

(b) (D,C) status

Figure 40: Major steps in the initialization algorithm

and payment method. The bottom vertex "All" actually is not a GroupBy query because it has no attributes on the grouping list. Two cuboids 1 and 2 are connected by an edge only if the grouping list of cuboid 1 is a subset of the grouping list of cuboid 2. All cells in cuboid 1 can find their descendant cells in 2. As depicted in Figure 40a, every colored cuboid contains at least one iceberg cell. The first number indicates all cells and the second number indicates iceberg cells.

Global sample size. The size of a sample affects its accuracy loss. Since Tabula checks the global random sample against every single cube cell during the dry run stage (builds local samples later if necessary), the size of the global sample has no effect on Tabula's error bound which is the loss threshold. However, a too small global sample may unnecessarily introduce too many iceberg cells. Therefore, Tabula utilizes Serfling's Inequality [80, 38] (a lemma of the law of large numbers) to determine a proper global sample size. Let $x_1, x_2, ...x_n$ be a finite set of numbers in [0, 1] with a mean μ, for any $\epsilon > 0$ and $1 \leq k \leq n$, I have

Algorithm 9: Initialization: Real Run Stage

```
Data: The raw table tbl and results of the dry run stage
Result: A cube table including samples
foreach cuboid cbd in all cuboids do
    if its iceberg cell table is not null then
        if it statisfies Inequation 5.1 then
            Run equality join tbl with the iceberg cells of cbd to retrieve data;
        Build cbd via a GroupBy on tbl or retrieved data;
        Draw a local sample for each iceberg cell;
    else
        Skip this cuboid;
```

$$\mathbb{P}\left[\max_{k\leq m\leq n-1}|\frac{1}{m}\sum_{i=1}^{m}x_i-\mu|\geq\epsilon\right]\leq 2exp(-\frac{2k\epsilon^2}{1-\frac{k-1}{n}})=\delta$$

where k is the sample size. Therefore, given any relative error ϵ of μ and confidence level δ, I have $k\approx\frac{ln\frac{2}{\delta}}{2\epsilon^2}$. By default, Tabula uses $\epsilon=0.05$ and $\delta=0.01$. Given the NYCtaxi dataset (700 million records) used in Section 5.5, the global sample has around 1000 tuples. This makes sure that this sample can represent the distribution of the raw dataset.

5.3.2.2 Real Run Stage: Sampling Cube Construction

Based on the iceberg cell information learned in the dry run stage, Tabula constructs a sampling cube that only contains iceberg cuboids. For each cell in this cuboid, the algorithm draws a local sample if the cell is an iceberg cell. The algorithm performs the same step for all iceberg cuboids until it eventually builds the sampling cube (see Figure 41).

Algorithm 9 gives the detailed pseudocode of the real run stage. The dry run stage has shown the number of iceberg cells in each cuboid (e.g., Table 10a), so Tabula can easily skip these non-iceberg cuboids (uncolored cuboids in Figure 40a) and work on iceberg cuboids that have at least one iceberg cell (red-colored cuboids in Figure 40a). For each iceberg cuboid, the algorithm then fetches the raw data that correspond to each iceberg cell in this cuboid. That can be done in two different ways: (1) Run a GroupBy operation using the cuboid attributes on the raw data and check the iceberg condition before drawing the sample for a cell (2) Run an equi-join operation between the cuboid iceberg cell table and the raw data to find the data corresponding to iceberg cells (see Figure 40b), then run the GroupBy operation on the retrieved raw data of iceberg cells, and finally draw a local sample for such cells. The second way is obviously more efficient when the iceberg cuboid only has a few iceberg cells. To decide that, Tabula employs the following cost model:

$$\begin{aligned} Cost_{Prune} + Cost_{GroupPrunedData} &< Cost_{GroupAllData} \\ N * i + \frac{i}{k} N * log_k(\frac{i}{k} N) &< N * log_k(N) \end{aligned} \tag{5.1}$$

where N is the cardinality of the table, i is the number of iceberg cells, k is the number of all cells in this cuboid. If the inequality holds, Tabula will use the second way mentioned above. Note that this condition assumes that each cell has the same amount of grouped raw data.

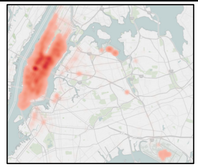
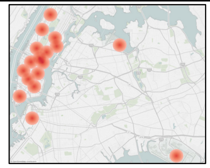
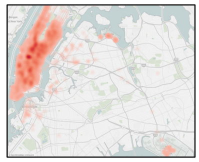
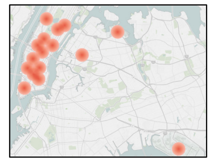
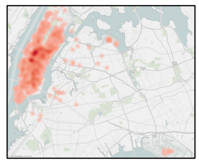
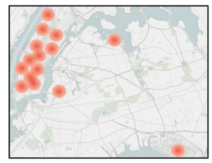
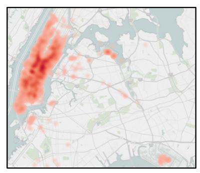
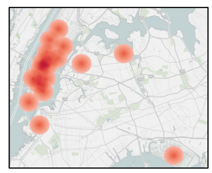
D.	C.	M.	Cell raw data	Sample
[0, 5)	1	credit		
[0, 5)	1	dispute		
[0, 5)	2	(null)		
[0, 5)	2	cash		
		...		

Figure 41: Output cube table of the initialization algorithm

5.4 Sample Selection

After the cube initialization, the partially materialized sampling cube may still possess a large memory footprint. That is because: (1) the number of cuboids and cube cells increases exponentially as the number of cubed attributes increase (2) for every iceberg cell, Tabula materializes a sample dataset (not just a single aggregate value), which may still consist of hundreds or thousands of tuples.

I observed that a sample in an iceberg cell can actually be re-used to represent the samples of other iceberg cells. That happens when applying the sample to those cells still ensures that loss(raw, sam) $\leq$ threshold θ. For example, in Figure 41, the sample stored in Iceberg Cell $\langle [0, 5), 1, dispute \rangle$ is similar to the raw data of

Iceberg Cell $\langle [0,5), 2, null \rangle$. In this case, I can let Cell $\langle [0,5), 2, null \rangle$ use the sample of $\langle [0,5), 1, dispute \rangle$ instead of materializing its own local sample. The sample of $\langle [0,5), 1, dispute \rangle$ is the representative sample for these two iceberg cells' samples. Therefore, to further reduce the memory footprint, Tabula only persists a representative set of samples from the cube table, and re-uses the representative samples in many iceberg cells rather than persisting every individual local sample. I define the representation relationship between two samples as follow:

Definition 5 (Sample Representation relationship)**.** *Given the raw data of an iceberg cell $Cell_A$ (format: tuple, tuple, ...) and its local sample Sam_A (format: tuple, tuple, ...), the raw data of another iceberg cell $Cell_B$ and its sample Sam_B. Sam_A can represent Sam_B only if $loss(Cell_B, Sam_A) \leq$ loss threshold θ.*

To select representative samples, Tabula first evaluates the relationships among different iceberg cells, which are described by a graph, namely sample representation graph (abbr. SamGraph). See the example in Figure 42.

Definition 6 (SamGraph)**.** *SamGraph(V, E) is a directed graph, where V and E represent the set of vertexes and edges, respectively. Each $v \in V$ represents a local sample stored in an iceberg cell. A directed edge from vertex v to u indicates that the sample v can represent the sample u. A bi-directed edge between a vertex v and u means that both samples v and u can represent each other.*

To build the SamGraph, Tabula performs an inner join on the cube table generated by the initialization algorithm (Section 5.3.2). The join condition is the representation relationship depicted above. I can express the inner join query using a SQL query as follows:

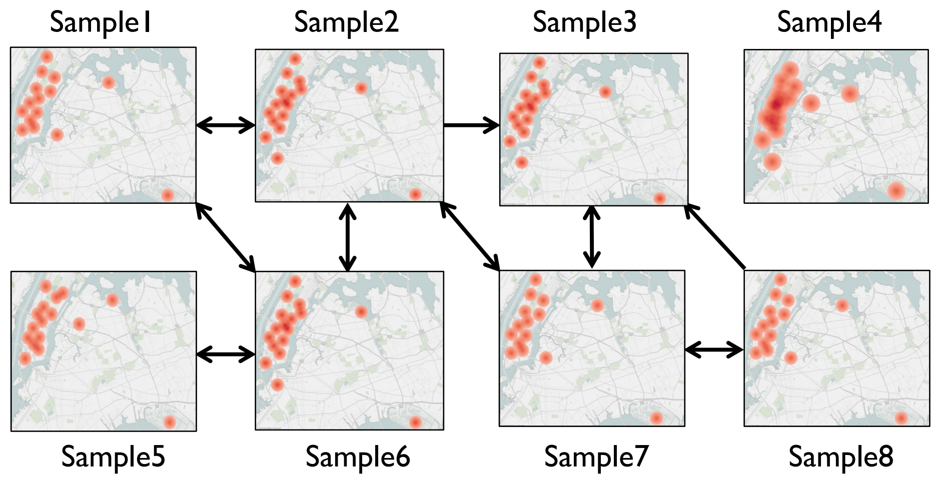


Figure 42: Select the representative samples

```
SELECT t1.D, t1.C, t1.M, t2.D, t2.C, t2.M
FROM  cube_table t1, cube_table_no_rawdata t2
WHERE loss(t1.cellrawdata, t2.sample) ≤ threshold
```

where cube_table is the cube table in Figure 41 and cube_table_no_rawdata is the same table but without raw data. Because the loss function may need some measures of the cell raw data (e.g., AVG), the cube table from the initialization algorithm needs to carry the raw data for each iceberg cell. Therefore, the cube table generated by the real run stage has a column named Cell Raw Data. Note that this join can be accelerated by any existing image/data similarity join algorithms. In addition, this join result does not have to exhaust all possible representation relationships. Sample selection on a non-exhaustive SamGraph may not minimize Tabula's memory footprint but still ensures Tabula's bounded-error guarantee.

Tabula traverses the SamGraph to select the representative samples, only persists selected samples, and drops the rest. I formally define the Representative Sample Selection (abbr. RepSamSel) problem as follows:

Definition 7 (Representative Sample Selection)**.** *Given a SamGraph = (V, E), select a subset D of V such that: (1) For every vertex $v \notin D$, v is represented by at least a vertex $u \in D$ and (2) The size of D is minimized.*

As depicted above, the main objective is to persist a minimal set of local samples and every unpersisted local sample can be represented by a persisted local sample. The first condition guarantees that if a local sample is not selected to be persisted, each iceberg cell can still use one of the persisted samples to answer queries. The second condition ensures that Tabula selects the minimum number of samples to persist/maintain, and hence reduces the overall memory space occupied by the sampling cube.

Lemma 5.4.1. *The representative sample selection (RepSamSel) problem is NP-Hard.*

Proof. That can be proved by reducing the Minimum Dominating Set (MDS) problem which is known to be NP-hard [6] to RepSamSel problem. I first relax the representation relationship: if sample A can represent sample B, then sample B can also represent A. Then, I can change all edges in SamGraph (V, G) to bi-directed edges. Now RepSamSel problem is identical to the MDS problem. The RepSamSel problem on bi-directed SamGraphs is a subset of that on directed SamGraphs. Therefore, RepSamSel problem is also NP-hard. The full details of the reduction algorithm is omitted due to space limitation. □

Representative Sample Selection Algorithm. Since RepSamSel problem is NP-hard, I resort to a greedy selection strategy. The algorithm (see Algorithm 10) takes as input the SamGraph(V, E) and keeps selecting a sample $v \in V$ and inserts it into D based on a greedy strategy, until every remaining sample in V has at least one representative in D. The greedy strategy always picks the sample $v \in V$

Algorithm 10: Representative Sample Selection

Data: SamGraph(V, E), each edge is denoted as $\langle head, tail \rangle$. head and tail are sample IDs

Result: A set D which consists of many sample IDs

Group edges E by head sample IDs;

`// Sort head sample IDs by their outdegress`

Sort groups in the descending order of the group counts;

Create a LinkedHashMap $HM\langle head, \{tail, tail, ..\}\rangle$;

Insert sorted groups one by one into HM;

Create a representative set D $= \emptyset$;

while HM $!= \emptyset$ **do**

  `// Pick sample ID by outdegress`

  Remove the top map $\langle head, \{tail, tail, ..\}\rangle$ from HM;

  Put *head* in D;

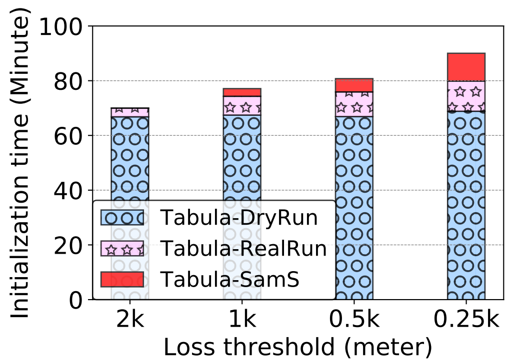
  `// Remove samples that are represented by` *head*

  foreach *tail in* $\{tail, tail, ..\}$ **do**

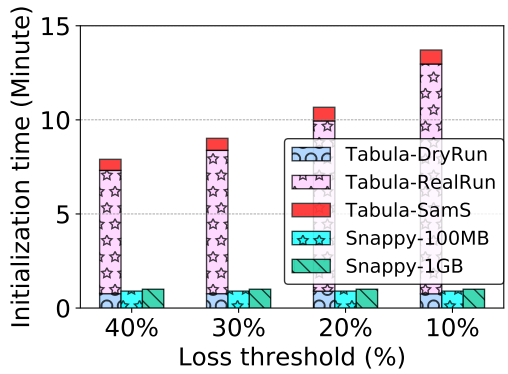
    Remove the map whose key is tail, from HM

such that v has the highest number of edges directed from v to other samples. In other words, the algorithm always selects the most representative sample among all remaining samples in a greedy fashion. Given the SamGraph in Figure 42, the greedy representative sample selection algorithm will pick Sample2 (represents 1,2,3,6,7), Sample8 (represents 3,7,8), Sample5 (represents 5,6) and Sample4 (represents itself), in this particular order. These four samples compose the representative samples set and are persisted in a sample table as depicted in Figure 39(b). The resulting cube table is normalized to the final cube table such as Figure 39(a) and each iceberg cell of the final cube table links to a sample id. If the local sample of an iceberg cell can be linked to multiple samples, I randomly pick one link to keep.

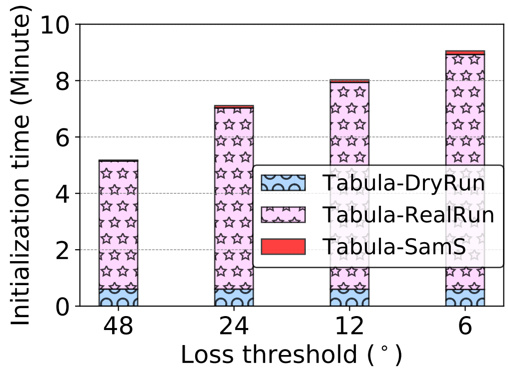
Compared approaches. All pre-built samples are cached into the cluster's memory: (1) SampleFirst (SamFirst): This approach creates a random sample of the entire dataset before accepting any query (see the definition in Section 5.1). I use two SampleFirst versions: 100MB and 1GB pre-built sample sizes. (2) SampleOnTheFly (SamFly): This approach has no pre-built samples (see the definition in Section 5.1). It uses the greedy sampling algorithm (Algorithm 8) to ensure the deterministic accuracy guarantee. (3) POIsam [38]: It is similar to SampleOnTheFly but has an extra random sampling step. After executing every query, it first creates a random sample on the query result then applies Algorithm 8. Please note that this greedy algorithm modifies the original POIsam's algorithm which fixes returned sample size and minimizes accuracy loss. POIsam supports visualization-aware sampling accuracy loss function including 1 dimension and geospatial data. In the experiments, I use POIsam's default theoretical error bound (5%) and confidence level (10%). This means that the sample produced by POIsam for every online query can have 5% or more error than Sample on the fly, at 10% chance. (4) SnappyData [76]: It applies data-system queries on the stratified samples, then returns an AVG of the query result. I use the cubed attributes as Query Column Set (QCS) in the experiments. Two versions of SnappyData are tested: 100MB and 1GB size pre-built samples. (5) Tabula: this is the system proposed in this chapter. (6) Tabula*: this is Tabula but does not have the sample selection technique. (7) Sampling cube (FullSamCube): this approach creates a fully materialized data cube which holds a local sample for every cell. (8) Partially materialized sampling cube(PartSamCube): this approach



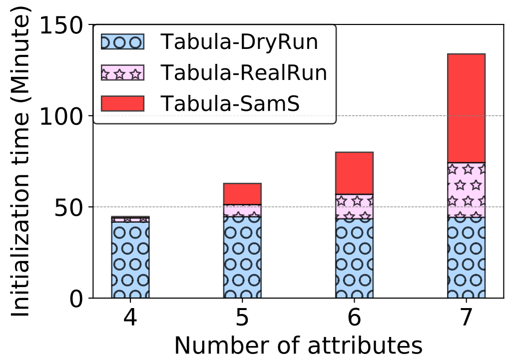
(a) Geospatial heatmap-aware



(b) Statistical mean



(c) Linear regression



(d) Num. attrs. (use histogram loss)

Figure 43: Initialization time of different loss functions and different cubed attributes

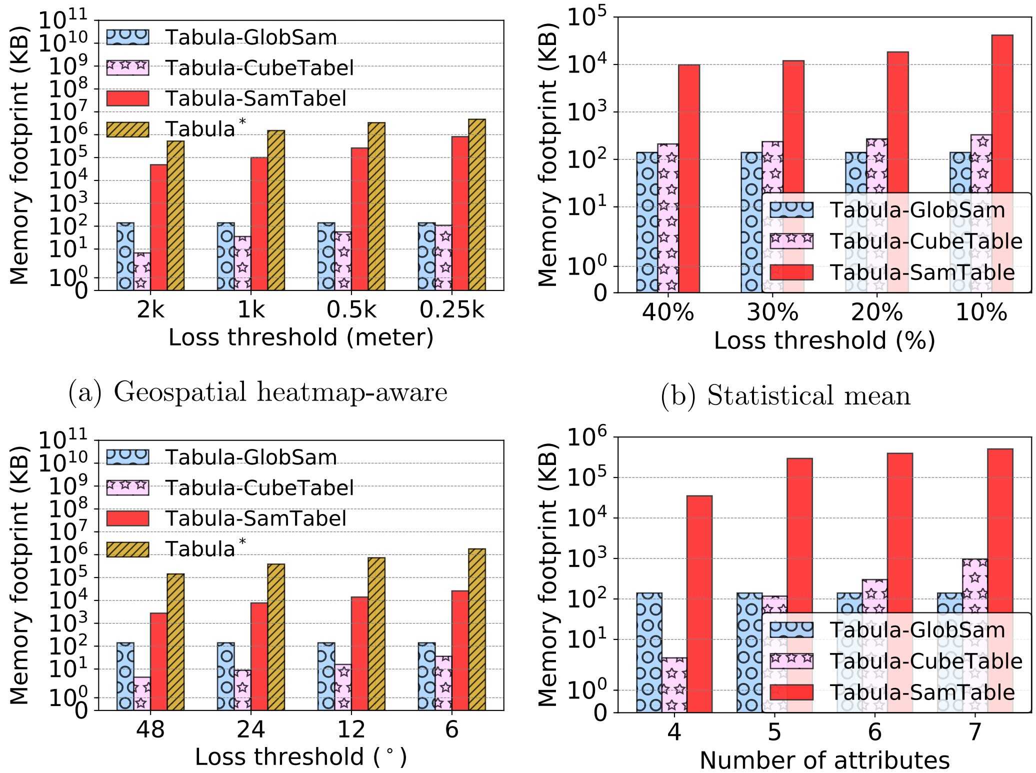
directly executes the initialization query as shown in Section 5.2. It does not use the initialization algorithm in Section 5.3.2 and sample selection in Section 5.4.

Evaluation metrics. I use the following metrics to measure the performance of each approach: (1) Initialization time: The time used to initialize the systems. I show the initialization time of Tabula, FullSamCube, PartSamCube, and SnappyData. (2) Memory footprint: The physical memory occupied by the pre-built / materialized samples in different approaches. SampleOnTheFly and POIsam do not incur extra memory space because they always draw samples on the fly. (3) Data-to-visualization time: it consists of (a) data-system: executing data-system queries and running online sampling (only for SamFly and POIsam). (b) sample visualization: performing visual analysis tasks (exclude SnappyData). SnappyData has no visual analysis time because it takes a query and directly renders a conclusion, which is AVG. (4) Actual accuracy loss: the actual accuracy loss of the returned sample, calculated by the user-defined loss function. (5) Query answer size: the number of tuples sent to /processed by the dashboard.

Dataset and query attributes. I use the New York City taxi trips real dataset (NYCtaxi) [85] which is mentioned in the running example. I pre-cache the entire dataset into the cluster's memory before initializing or using any approach. There are 7 categorical attributes used in the experiments: vendor name, pickup weekday, passenger count, payment type, rate code, store and forward, dropoff weekday. I use the first 4, 5, 6, 7 attributes in the predicates of data-system queries. Full data cubes built upon these attributes have 3 thousand, 17 thousand, 47 thousand and 151 thousand cells, respectively. The first 5 attributes are used by default.

User defined accuracy loss functions. (1) Statistical mean loss: this is Function 1 which checks against fare amount attribute of NYCtaxi data. (2) Geospatial heatmap-aware loss function: this is Function 2 (3) Linear regression loss: this is Function 3 (4) Histogram-aware loss: this is Function 2 but it is calculated on 1-dimension data (using Euclidean distance). The corresponding analysis task is shown in Figure 36. This function checks against NYCtaxi fare amount attribute so the distance unit is US dollar.

Analytics workload. I build a full data cube on n attributes then randomly pick 100 SQL queries (cells) from the cube. All compared approaches will then run these queries. Returned query answers are passed to the visualization dashboard in files. To quantitatively measure the visual analysis performance, I use two well-known analysis tools to record the corresponding visual analysis time: (1) Matlab: a renowned



(a) Geospatial heatmap-aware

(b) Statistical mean

(c) Linear regression

(d) Num. attrs. (use histogram loss)

Figure 44: Memory footprint of different loss functions and different attributes, in logarithm scale

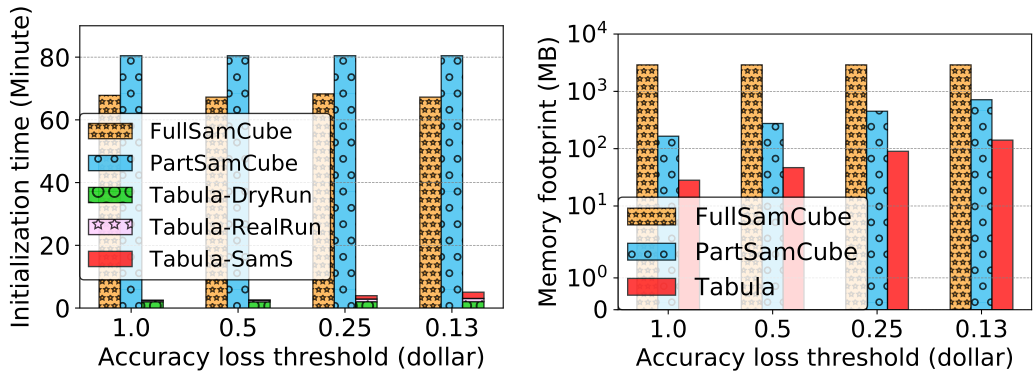
scientific computing software. I leverage it to draw histogram and geospatial heatmaps on results returned in corresponding accuracy loss based experiments. (2) Scikit Learn: a widely used machine learning Python library. I use it to calculate statistical means and linear regression functions of returned query answers. All analysis tasks are executed on the master machine of the cluster.

Cluster settings. All compared approaches are implemented with Apache Spark. I conduct the experiments on a cluster which has one master node and four worker nodes. Each machine has an Intel Xeon E5-2687WV4 CPU (12 cores, 3.0 GHz per core), 100 GB memory, and 4 TB HDD. I also install Apache Hadoop 2.6, Apache Spark 2.3, and SnappyData Enterprise 1.0.2.1 (column store).

5.5.1 Initialization Time

In this section, I study the initialization time of different approaches (see Figure 43 and 45a). I vary the value of the user specified loss threshold θ. SampleFirst's initialization time is omitted because the random sampling time is negligible compared to other approaches. I compare Tabula against FullSamCube and PartSamCube on a small dataset, 5GB NYCtaxi (see Figure 45a) using histogram-aware loss function, because FullSamCube and PartSamCube incur high initialization time and cannot scale to the full NYCtaxi dataset. I also show the execution time of the dry run stage, real run stage and sample selection (denoted as SamS) of Tabula.

As shown in Figure 45a, Tabula takes around 40 times less initialization time compared to FullSamCube and PartSamCube. This makes sense because Tabula utilizes the dry run stage to skip many unnecessary GroupBys while other approaches run $2^n - 1$ GroupBy operations (n is the number of attributes). As depicted in Figure 43, the dry run stage execution time remains the same for different user specified loss thresholds but the overall initialization time of Tabula increases with the decrease of loss threshold. This is because a lower value of θ introduces more iceberg cells. Tabula always spends the same amount of time on the dry run stage to identify iceberg cells. However, if there are more iceberg cells, Tabula will take more time to draw local samples for iceberg cells in the real run stage and select representative samples in sample selection. It is also worth noting that geospatial heatmap-aware loss functions lead to Tabula consuming more time on the dry run stage while statistical mean costs the least time on that. This makes sense because the visualization-aware loss function involves complex tuple-to-tuple calculation compared to the statistical mean loss function.



(a) Initialization time

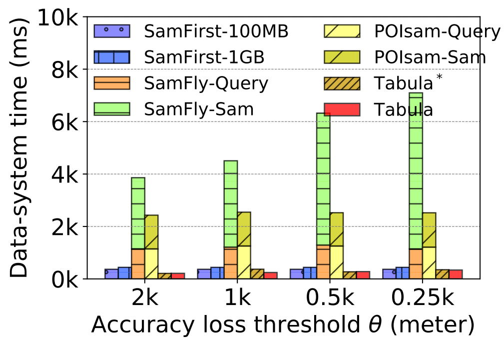
(b) Memory (log. scale)

Figure 45: Cubing overhead on 5GB NYCtaxi data

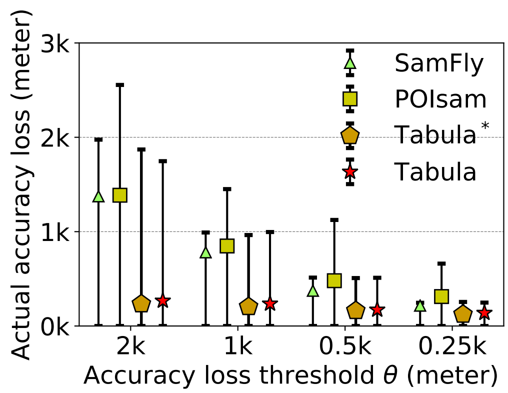
5.5.2 Memory Footprint

In this section, I study the memory footprint of Tabula for different accuracy loss functions. Tabula consists of three physical components in memory, global sample, cube table (Figure 39a) and sample table (Figure 39b). Tabula* does not have the sample table. As depicted in Figure 44, decreasing the value of θ leads to more memory space occupied by Tabula. That happens because a smaller θ results in more iceberg cells and more materialized local samples. Among the three components of Tabula, the global sample size remains the same for different θ values because this size is only related to the raw dataset scale according to Section 5.3.2.1. Both the cube and sample tables increase for smaller thresholds but the sample tables are at least 100 times larger than the cube tables. This makes sense because the cube table only contains simple iceberg cell information without any materialized samples. Tabula* is around 50 times larger than Tabula because it does not employ the sample selection technique.

As depicted in Figure 45b, the size of FullSamCube remains the same for different thresholds because it always materializes local samples for all cube cells regardless of thresholds while PartSamCube only materializes samples for iceberg cells. FullSamCube is around 50-100 times larger than Tabula while PartSamCube is around 5 - 8 times larger than Tabula because PartSamCube does not contain the sample selection technique.

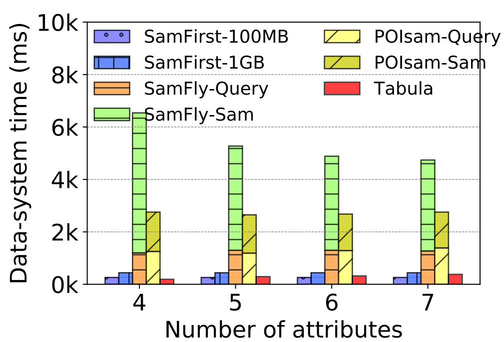


(a) data-system time

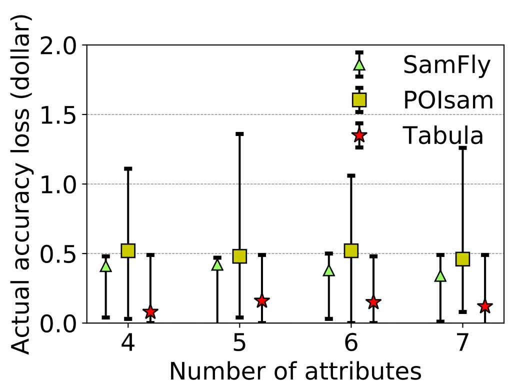


(b) actual loss

Figure 46: Performance of geospatial heatmap-aware loss (unit: meter), 0.25 kilo meter ≈ 0.004 (normalized distance)

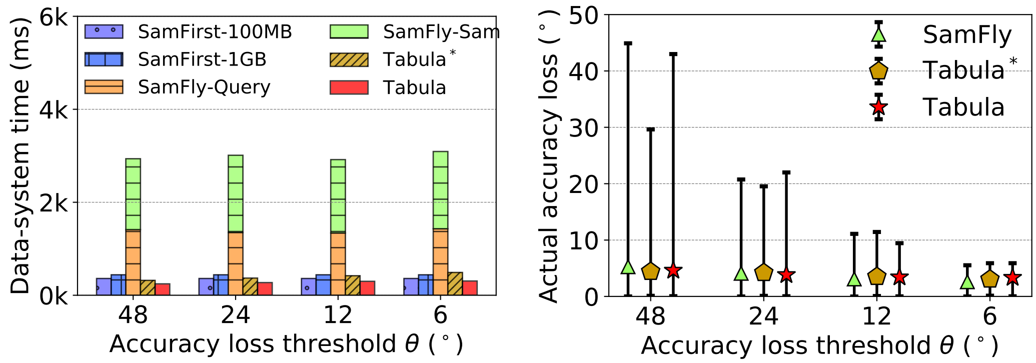


(a) data-system time



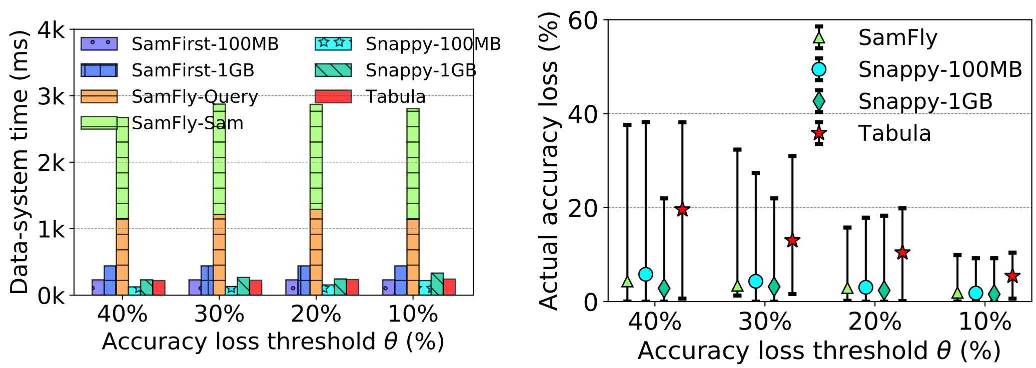
(b) Actual loss

Figure 47: Performance of different numbers of attributes in data-system queries, with histogram-aware loss function



(a) data-system time (b) actual loss

Figure 48: Performance of linear regression loss (loss unit: degree °)



(a) data-system time (b) actual loss

Figure 49: Performance of statistical mean loss (loss unit: percentage)

5.5.3 Data-to-visualization Time

I study the effect of various accuracy loss functions and threshold values on the total data-to-visualization time. I only run analysis tasks using the smallest accuracy loss thresholds (θ) for all accuracy loss functions and report the time in Table 11.

In terms of data-system time, as shown in Figures 46a,49a and 48a, the data-system time of SamFirst remains the same for all threshold values and loss functions because the only factor that can affect SamFirst is the size of its pre-built sample. POIsam and SampleOnTheFly are 10 times and 20 times slower than Tabula. This is because Tabula quickly returns either the materialized global or a local sample while POIsam and SampleOnTheFly always query the entire dataset and draw samples on the fly. Although I pre-cached the entire dataset and ran queries in parallel, the query time is still significantly large. POIsam can reduce the online sampling time, but its data-system time is still non-negligible.

Table 11: Sample visualization time of different approaches

Approach	Geospatial heat map	Statistical mean	Regression
SamFirst-100MB	29 ms	0.02 ms	0.07 ms
SamFirst -1GB	59 ms	0.13 ms	0.15 ms
SamFly	146 ms	0.01 ms	0.29 ms
POIsam	143 ms	-	-
Tabula	390 ms	0.13 ms	1.33 ms
No sampling	330 sec	1.8 sec	1.9 sec

In terms of the sample visualization time, as shown in Table 11, Tabula has the highest visual analysis time among all compared approaches because it sometimes returns the global sample (around 1000 tuples) for queries which hit non-iceberg cells while other approaches such as SampleOnTheFly and POIsam only return around 100 tuples for geospatial heat map loss function. However, analysis tools can still easily render results for Tabula within several hundred milliseconds because of the small size of global samples. Please note that it takes around 3 orders of magnitude more time on analyzing the raw query result (without any sampling).

5.5.4 Studying the Actual Accuracy Loss

In this section, I vary the accuracy loss threshold value (error bound in SnappyData) and evaluate the actual loss of samples returned by different approaches. The results are depicted in Figures 46b,49b and 48b. Error bars indicate the minimum, average and maximum actual accuracy loss. I omit the actual accuracy loss of two SamFirst approaches in figures because their average accuracy loss is 20 times larger than other approaches for geospatial heatmap-aware loss functions and 4 times larger than others. As shown in the figures, as I decrease the threshold value, the actual loss of POIsam, SampleOnTheFly, SnappyData, and Tabula decreases. SampleOnTheFly, SnappyData and Tabula never exceed the thresholds. The actual accuracy loss of POIsam is around 1%-5% percent larger than SampleOnTheFly and sometimes, it exceeds the threshold. This makes sense because POIsam runs the greedy sampling function over a random sample. Tabula* has similar actual accuracy loss to Tabula because the sample selection technique does not necessarily increase accuracy loss. Also, SnappyData can guarantee the error-bound since the actual accuracy loss exceeds the threshold value, it accesses the raw table and runs queries and aggregation on-the-fly. Since SnappyData implements its own optimized block-based column store, its data-system time is still comparable to Tabula.

5.5.5 Impact of the Number of Attributes

In this section, I evaluate the impact of the number of attributes. I initialized Tabula on 4, 5, 6 and 7 attributes of NYCtaxi dataset and use these attributes in data-system queries. Histogram-aware loss function with "0.5 dollar" threshold value is used in the experiment. 4 metrics are reported in Figure 43d, 44d and 47, respectively. The result of actual accuracy loss is omitted because the number of attributes has no effect on actual accuracy loss.

As depicted in the figures, in terms of initialization time, using more cubed attributes leads to higher execution time for all three initializing stages of Tabula because this introduces more cube cells as well as iceberg cells. Cube cells increase exponentially with more cubed attributes. But the number of attributes has relatively small impact on the dry run stage because the dominating part in this stage is building the first cuboid which requires a full table GroupBy. Other cuboids are derived from the first one.

In terms of memory footprint, the global sample size of Tabula remains the same because it is only related to the cardinality of the raw dataset. The sizes of the cube table and sample table increase with more cubed attributes. But the growing speed of the sample table becomes slower because, even though there are more and more iceberg cells and local samples, Tabula still can only materialize a small number of local samples as the representatives.

In terms of data-to-visualization time, using more attributes slightly increases the data-system time of Tabula because of larger cube tables and sample tables. SampleFirst approaches have the constant data-system time since they always perform a full sequential filtering on pre-built samples. The query time of SampleOnTheFly and POIsam remains the same for different numbers of attributes because they always perform a full sequential scan on the raw table. However, the visual analysis time of SampleFirst drops while using more attributes. This is because the queries will contain more predicates and lead to smaller query results. Similarly, the sampling time of SampleOnTheFly drops significantly while using more attributes. The online sampling time of POIsam does not change much because it first draws a random sample of the raw query result and the random sample size does not change much (controlled by the law of larger numbers [38]). The visual analysis time of Tabula slightly reduces while using more cubed attributes because Tabula returns materialized local sample for more queries in this situation.

5.6 Summary

In this section, I presented Tabula as a middleware system to accelerate the spatial visualization dashboard. It can be easily extended, thanks to its generic user-defined accuracy loss function, to support various visual analytics. Tabula adopts a materialized sampling cube approach, which pre-materializes sampled answers for a set of potentially unforeseen queries. To achieve scalability, the system employs a partially materialized cube to only materialize local samples of iceberg cells based on the accuracy loss function and a sample selection technique to selectively materialize representative local samples. The system ensures the difference between the sample and the raw query answer never exceeds a user-specified accuracy loss threshold. According to the experiments, Tabula upholds different user-defined visual analysis: for complex analysis such as geospatial visual analytics and linear regression, it achieves up to 20 times less data-to-visualization time than SampleOnTheFly-like approaches; for OLAP analytics such as statistical mean (AVG), it exhibits similar performance to column-store based SnappyData. The proposed middleware system also occupies up to two orders of magnitude less memory footprint and an order of magnitude less initialization time than the fully materialized sampling cube approach. That makes Tabula a very practical and scalable approach to deploy in real geospatial data visualization dashboards.

Chapter 6

CONCLUSION AND FUTURE WORK

This chapter concludes the dissertation by summarizing the contributions of the work and highlighting the future directions.

6.1 Conclusion

In this dissertation, I designed database systems to accelerate large-scale geospatial data analytics. In particular, I worked on several tasks in this direction: (1) scalable analytics systems for geospatial data (2) lightweight database indexing mechanisms (3) interactive analytics on big spatial data. The proposed approaches are as follows:

For scalable analytics systems, I first presented GeoSpark, an in-memory cluster computing framework for processing large-scale spatial data. Moreover, the system provides native support for spatial data partitioning, indexing, , and query processing in Apache Spark to efficiently analyze spatial data at scale. Extensive experiments show that GeoSpark outperforms Spark-based systems such as Simba and Magellan up to one order of magnitude and Hadoop-based system such as SpatialHadoop up to two orders of magnitude. I then introduced GeoSparkViz, a cluster computing system for visualizing massive-scale geospatial data. The GeoSparkViz approach allows users to declaratively define geosaptial visual analytics (GeoViz) tasks. The system adopts a GeoViz-aware spatial partitioning scheme and execution strategies that co-optimize the map visualization operations with spatial query operators (in GeoSpark). Experiments based on real spatial data show that GeoSparkViz can achieve up to one order of magnitude less data-to-visualization time compared to its counterparts.

For lightweight database indexing mechanism, I introduced Hippo, a data-aware sparse indexing approach that efficiently and accurately answers database queries. Hippo occupies up to two orders of magnitude less storage overhead than de-facto database indexes, i.e., B^+ -tree while achieving comparable query execution performance. Moreover, Hippo achieves about three orders of magnitudes less maintenance overhead compared to the B^+-tree and BRIN. To achieve that, Hippo stores page ranges instead of tuples in the indexed table to reduce the storage space occupied by the index. Furthermore, Hippo maintains histograms, which represent the data distribution for one or more pages, as the summaries for these pages. I then extend Hippo to support spatial data indexing. Hippo-Spatial adopts a two-dimension histogram as the page summaries. The proposed approach outperforms R-Tree index by showing two orders of magnitude less storage overhead and competitive query performance.

For interactive analytics on big spatial data, I designed Tabula as a middleware system to accelerate the spatial visualization dashboard. Tabula adopts a materialized sampling cube approach, which pre-materializes sampled answers for a set of potentially unforeseen queries. The system ensures the difference between the sample and the raw query answer never exceeds a user-specified accuracy loss threshold. For complex analysis such as geospatial visual analytics and linear regression, it achieves up to 20 times less data-to-visualization time than competitors.

6.2 Future work

In the future, I will continue developing open-source high-performance data management systems to make sense of "Big Spatial Data". My current work raises a number of challenging research problems in this direction that I plan to address immediately.

Large-scale spatial streaming data analytics. The unprecedented popularity of GPS-equipped mobile devices and Internet of Things (IoT) sensors has led to continuously generating large-scale location information combined with the status of surrounding environments. Such data has a streaming nature and keeps evolving at a staggering rate over time. Precisely digesting the massive spatial streaming data that swarms into the database systems in a short time window requires a well-designed system architecture. It will be greatly beneficial to spatial data scientists in a variety of real-world scenarios. For instance, to make timely planning strategies, the city of Chicago started installing sensors across its road intersections to monitor the environment, air quality, and traffic. Furthermore, making sense of real-time streaming data from these "never sleeping" GPS-equipped devices may even bolster autonomous city governance (i.e., City Brain) including AI-based climate control and traffic planning. Unfortunately, existing streaming data management systems are not scalable and efficient to handle spatial streaming data. They either require tedious programming tasks or suffer from a significant performance drop. To remedy that, I plan to address the scalability issue of large-scale spatial streaming data analytics by developing a full-fledged distributed spatial streaming analytics system. To be specific, I am going to investigate the applicability of Spark Streaming and Apache Flink to continuous spatial queries such as range query, join query and nearest-neighbor query.

I also plan to carry out research in inventing distributed spatial data structures in the streaming environment.

Interactive visual analysis of dynamic geospatial data. Geospatial visual analytics is the science of analytical reasoning assisted by geo-visual map interfaces. In my current work Tabula, I have shown that interactive spatial visual analytics can help users easily find interesting insights. Although there is a flurry of research projects tackling this problem from different angles, the existing work mainly focuses on static data rather than dynamic data, with the latter becoming more popular recently. Consider an example user who wants to set up a real-time heat map of millions of GPS-installed vehicles in New York City. As time goes on, this heat map should change every minute or even every second to reflect the actual movement of vehicles from place to place. Moreover, the user may impose filters on numerical or textual attributes of moving vehicles or zoom in to a particular region for more details. The interactive nature of geospatial visual analytics requires an immediate response from visualization systems. I plan to address several challenges in this topic that include the co-optimization between underlying database systems and front-end visualization frameworks and materialized visualization maintenance.

Machine Learning-enhanced spatial data structures. Machine Learning techniques have introduced significant performance improvement to several classic database components such as data indices and query optimizers. Therefore, the user can see analysis results with lower storage cost yet at a higher speed. In this ML revolution, spatial data has got little attention and is merely treated as a second-class citizen. However, spatial data has several special features that deserve better designs. First, spatial data usually consists of heterogeneous objects including points, polygons, and trajectories. Second, spatial queries are often clustered to certain hot geographical regions. Third, other attributes in a dataset are commonly correlated to the spatial attribute (e.g., spatial auto-correlation: income - education -> location). Thus, I plan to design a set of ML-enhanced spatial data structures such as indices or new physical data layouts to facilitate spatial query processing. The newly invented spatial data structures can also be embedded into my future projects derived from other research topics that I plan to pursue.

REFERENCES

[1] Swarup Acharya, Phillip B Gibbons, and Viswanath Poosala. "Congressional samples for approximate answering of group-by queries". In: *ACM SIGMOD Record*. Vol. 29. ACM. 2000, pp. 487–498.

[2] Sameer Agarwal et al. "BlinkDB: queries with bounded errors and bounded response times on very large data". In: *The European Conference on Computer Systems, EuroSys*. 2013, pp. 29–42.

[3] Sameer Agarwal et al. "Knowing when you're wrong: Building fast and reliable approximate query processing systems". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2014, pp. 481–492.

[4] Sameet Agarwal et al. "On the Computation of Multidimensional Aggregates". In: *Proceedings of the International Conference on Very Large Data Bases, VLDB*. 1996, pp. 506–521.

[5] Ablimit Aji et al. "Demonstration of Hadoop-GIS: A spatial data warehousing system over MapReduce". In: *Proceedings of the ACM International Conference on Advances in Geographic Information Systems, SIGSPATIAL* 6.11 (2013), pp. 518–521.

[6] Robert B Allan and Renu Laskar. "On domination and independent domination numbers of a graph". In: *Discrete Mathematics* 23.2 (1978), pp. 73–76.

[7] *Apache Flink*. https://flink.apache.org/.

[8] *Apache Zeppelin*. https://zeppelin.apache.org/.

[9] Michael Armbrust et al. "Spark SQL: Relational data processing in spark". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2015, pp. 1383–1394.

[10] Manos Athanassoulis and Anastasia Ailamaki. "BF-tree: Approximate tree indexing". In: *Proceedings of the VLDB Endowment, PVLDB*. Vol. 7. 14. 2014, pp. 1881–1892.

[11] Richard A Becker. "A brief history of S". In: *Computational statistics* (1994), pp. 81–110.

[12] Kevin S Beyer and Raghu Ramakrishnan. “Bottom-Up Computation of Sparse and Iceberg CUBEs”. In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 1999, pp. 359–370.

[13] *Block Range Index*. https://www.postgresql.org/docs/9.6/static/brin.html.

[14] Charles Bontempo and George Zagelow. “The IBM Data Warehouse Architecture”. In: *Communications of the ACM* 41.9 (1998), pp. 38–48.

[15] Howard Butler et al. “GeoJSON”. In: *Electronic. URL: http://geojson. org* (2014).

[16] Pei Cao and Zhe Wang. “Efficient Top-K query calculation in distributed networks”. In: *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*. Vol. 23. 2004, pp. 206–215.

[17] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. “Approximate Query Processing: No Silver Bullet”. In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2017, pp. 511–519.

[18] Chuansheng Chen et al. “Population migration and the variation of dopamine D4 receptor (DRD4) allele frequencies around the globe”. In: *Evolution and Human Behavior* 20.5 (1999), pp. 309–324.

[19] Ralph J. Cicerone et al. *Climate Change: An Analysis of Some Key Questions*. The National Academies Press, 2001.

[20] Douglas Comer. “Ubiquitous B-tree”. In: *ACM Computing Surveys, CSUR* 11.2 (1979), pp. 121–137.

[21] Transaction Processing Performance Council. “TPC-H benchmark specification”. In: *Published at http://www. tcp. org/hspec. html* (2011), pp. 1–134.

[22] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[23] Subhankar Dhar and Upkar Varshney. “Challenges and business models for mobile location-based services and advertising”. In: *Communications of the ACM* 54.5 (2011), pp. 121–129.

[24] Bolin Ding et al. “Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee”. In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2016, pp. 679–694.

[25] Jens Peter Dittrich and Bernhard Seeger. "Data redundancy and duplicate detection in spatial join processing". In: *Proceedings of the International Conference on Data Engineering, ICDE*. 2000, pp. 535–546.

[26] *Earth Science Data on AWS With NASA / NEX Public Data Sets*. 2019.

[27] Ahmed Eldawy, Louai Alarabi, and Mohamed F. Mokbel. "Spatial partitioning techniques in spatialhadoop". In: *Proceedings of the VLDB Endowment, PVLDB* 8.12 (2015), pp. 1602–1605.

[28] Ahmed Eldawy and Mohamed F. Mokbel. "Pigeon: A spatial MapReduce language". In: *Proceedings of the International Conference on Data Engineering, ICDE*. 2014, pp. 1242–1245.

[29] Ahmed Eldawy and Mohamed F. Mokbel. "SpatialHadoop: A MapReduce framework for spatial data". In: *Proceedings of the International Conference on Data Engineering, ICDE*. 2015, pp. 1352–1363.

[30] Ahmed Eldawy, Mohamed F. Mokbel, and Christopher Jonathan. "HadoopViz: A MapReduce framework for extensible visualization of big spatial data". In: *Proceedings of the International Conference on Data Engineering, ICDE*. 2016, pp. 601–612.

[31] Environmental Systems Research Institute. "ESRI & StatSci statistical software integrated with ARC/INFO GIS". In: *Computational Statistics & Data Analysis* 16.3 (1993), pp. 370–371.

[32] R. A. Finkel and J. L. Bentley. "Quad trees a data structure for retrieval on composite keys". In: *Acta Informatica* 4.1 (1974), pp. 1–9.

[33] Philippe Flajolet, Danièle Gardy, and Loÿs Thimonier. "Birthday paradox, coupon collectors, caching algorithms and self-organizing search". In: *Discrete Applied Mathematics* 39.3 (1992), pp. 207–229.

[34] David A Freedman. *Statistical models: theory and practice*. Cambridge University Press, 2009.

[35] Francesco Fusco, Marc Ph Stoecklin, and Michail Vlachos. "NET-FLi: On-the-fly compression, archiving and indexing of streaming network traffic". In: *Proceedings of the VLDB Endowment, PVLDB* 3.2 (2010), pp. 1382–1393.

[36] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. "Compressing relations and indexes". In: *Proceedings of the International Conference on Data Engineering, ICDE*. 1998, pp. 370–379.

[37] Jim Gray et al. "Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals". In: *Data Mining Knowledge Discovery* 1.1 (1997), pp. 29–53.

[38] Tao Guo et al. "Efficient selection of geospatial data on maps for interactive and visualized exploration". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2018, pp. 567–582.

[39] Antonin Guttman. "R-trees: A dynamic index structure for spatial searching". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 1984, pp. 47–57.

[40] Gheorghi Guzun et al. "A tunable compression framework for bitmap indices". In: *Proceedings of the International Conference on Data Engineering, ICDE*. 2014, pp. 484–495.

[41] Peter J Haas and Joseph M Hellerstein. "Ripple Joins for Online Aggregation". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 1999, pp. 287–298.

[42] Apache Hadoop. "Apache Hadoop". In: *Encyclopedia of Big Data Technologies*. Cham: Springer International Publishing, 2019, pp. 58–58.

[43] Mordechai Haklay and Patrick Weber. "OpenStreet map: User-generated street maps". In: *IEEE Pervasive Computing* 7.4 (2008), pp. 12–18.

[44] Jiawei Han et al. "Efficient Computation of Iceberg Cubes with Complex Measures". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2001, pp. 1–12.

[45] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. "Implementing Data Cubes Efficiently". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 1996, pp. 205–216.

[46] W L Hays. *Statistics*. College Publishing, 1981.

[47] Joseph M Hellerstein. "Generalized Search Tree". In: *Encyclopedia of Database Systems*. Springer, 2009, pp. 1222–1224.

[48] J R Herring. “OpenGIS: Implementation Specification for Geographic information-Simple feature access-Part 1: Common architecture”. In: *Open Geospatial Consortium* (2006), p. 95.

[49] Michael E. Houle and Jun Sakuma. “Fast approximate similarity search in extremely high-dimensional data sets”. In: *Proceedings of the International Conference on Data Engineering, ICDE*. 2005, pp. 619–630.

[50] James N Hughes et al. “Geomesa: a distributed architecture for spatio-temporal fusion”. In: *SPIE Defense+ Security*. International Society for Optics and Photonics. 2015, 94730F–94730F.

[51] Patrick Hunt et al. “ZooKeeper: Wait-free coordination for internet-scale systems”. In: *Proceedings of the 2010 USENIX Annual Technical Conference, USENIX ATC 2010*. 2019, pp. 145–158.

[52] Ross Ihaka and Robert Gentleman. “R: a language for data analysis and graphics”. In: *Journal of computational and graphical statistics* 5.3 (1996), pp. 299–314.

[53] International Organization for Standardization. *Information technology — Database languages — SQL Multimedia and application packages — Part 3: Spatial - ISO 13249-3:2006*. Standard. Geneva, Switzerland: International Organization for Standardization, 2009.

[54] Prasanth Jayachandran et al. “Combining User Interaction, Speculative Query Execution and Sampling in the DICE System”. In: *Proceedings of the VLDB Endowment, PVLDB* 7.13 (2014), pp. 1697–1700.

[55] Niranjan Kamat et al. “Distributed and interactive cube exploration”. In: *Proceedings of the International Conference on Data Engineering, ICDE*. Ed. by Isabel F. Cruz et al. 2014, pp. 472–483.

[56] Srikanth Kandula et al. “Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters”. In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2016, pp. 631–646.

[57] Daniel Lemire, Owen Kaser, and Kamel Aouiche. “Sorting improves word-aligned bitmap indexes”. In: *Data and Knowledge Engineering* 69.1 (2010), pp. 3–28.

[58] Feifei Li et al. “Wander Join: Online Aggregation via Random Walks”. In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2016, pp. 615–629.

[59] Xiaolei Li et al. “Sampling cube: a framework for statistical olap over sampling data”. In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2008, pp. 779–790.

[60] Lauro Lins, James T. Klosowski, and Carlos Scheidegger. “Nanocubes for real-time exploration of spatiotemporal datasets”. In: *IEEE Transactions on Visualization and Computer Graphics* 19.12 (2013), pp. 2456–2465.

[61] Jiamin Lu and Ralf Hartmut Güting. “Parallel SECONDO: Boosting database engines with Hadoop”. In: *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*. 2012, pp. 738–743.

[62] Gang Luo, Jeffrey F Naughton, and Curt J Ellmann. “A non-blocking parallel spatial join algorithm”. In: *Proceedings of the International Conference on Data Engineering, ICDE*. 2002, pp. 697–705.

[63] Guido Moerkotte. “Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing”. In: *Proceedings of the International Conference on Very Large Data Bases, VLDB*. 1998, pp. 476–487.

[64] Arnab Nandi et al. “Distributed cube materialization on holistic measures”. In: *Proceedings of the International Conference on Data Engineering, ICDE*. 2011, pp. 183–194.

[65] OpenStreetMap. *Open Street Map zoom levels*. http://wiki.openstreetmap.org/wiki/Zoom_levels.

[66] *OpenStreetMap*. http://www.openstreetmap.org/. 2019.

[67] L. Opyrchal and A. Prakash. “Efficient object serialization in Java”. In: *Electronic Commerce and Web-based Applications/Middleware, 1999. Proceedings. 19th IEEE International Conference on Distributed Computing Systems Workshops on*. IEEE. 2003, pp. 96–101.

[68] An Oracle and White Paper. “A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server”. In: *Oracle* June (2012).

[69] Bernd Uwe Pagel et al. "Towards an analysis of range query performance in spatial data structures". In: *Proceedings of the ACM Symposium on Principles of Database Systems, PODS*. 1993, pp. 214–221.

[70] Cícero A.L. Pahins et al. "Hashedcubes: Simple, Low Memory, Real-Time Visual Exploration of Big Data". In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 671–680.

[71] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. "Visualization-aware sampling for very large databases". In: *Proceedings of the International Conference on Data Engineering, ICDE*. 2016, pp. 755–766.

[72] Jignesh M. Patel and David J. DeWitt. "Partition based spatial-merge join". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 1996, pp. 259–270.

[73] Matthew Perry and John Herring. "OGC GeoSPARQL-A geographic query language for RDF data". In: *OGC Candidate Implementation Standard* (2012), p. 57.

[74] PostGIS. *PostGIS*. http://postgis.net/. 2019.

[75] *PostgreSQL database management system*. www.postgresql.org.

[76] Jags Ramnarayan et al. "SnappyData: A Hybrid Transactional Analytical Store Built On Spark". In: 2016, pp. 2153–2156.

[77] John T. Robinson. "The K-D-B-tree: A search structure for large multidimensional dynamic indexes". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 1981, pp. 10–18.

[78] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. "Nearest neighbor queries". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 1995, pp. 71–79.

[79] Yasushi Sakurai et al. "The A-tree: An index structure for high-dimensional spaces using relative approximation". In: *Proceedings of the International Conference on Very Large Data Bases, VLDB*. 2000, pp. 516–526.

[80] Robert J Serfling. "Probability inequalities for the sum in sampling without replacement". In: *The Annals of Statistics* (1974), pp. 39–48.

[81] Lefteris Sidirourgos and Martin Kersten. "Column imprints: A secondary index structure". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2013, pp. 893–904.

[82] Ram Sriharsha. *Magellan on Spark*. https://github.com/harsha2010/magellan. 2019.

[83] Kurt Stockinger and Kesheng Wu. "Bitmap indices for data warehouses". In: *Data Warehouses and OLAP: Concepts, Architectures and Solutions* (2006), pp. 157–178.

[84] Michael Stonebraker and Lawrence A. Rowe. "The design of postgres". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 1986, pp. 340–355.

[85] New York City Taxi and Limousine Commission. *NYC TLC Trip data*. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml. 2019.

[86] The HDF Group. *Hierarchical data format version 5, 2000-2010*. 2010.

[87] Ashish Thusoo et al. "Hive - A warehousing solution over a map-reduce framework". In: *Proceedings of the VLDB Endowment, PVLDB* 2.2 (2009), pp. 1626–1629.

[88] *TIGER/Line files*. http://www.census.gov/geo/www/tiger/. 2019.

[89] Hoang Vo, Ablimit Aji, and Fusheng Wang. "SATO: A spatial data partitioning framework for scalable query processing". In: *Proceedings of the ACM International Conference on Advances in Geographic Information Systems, SIGSPATIAL*. Vol. 04-07-Nove. 2014, pp. 545–548.

[90] Michael Wall, Aaron Cordova, and Billie Rinaldi. *Apache Accumulo*. 2012.

[91] Lu Wang et al. "Spatial Online Sampling and Aggregation". In: *Proceedings of the VLDB Endowment, PVLDB*. Vol. 9. 3. 2016, pp. 84–95.

[92] Barney Warf. *Open Geospatial Consortium (OGC)*. 2014.

[93] Barney Warf. *Spatial Statistics*. Vol. 575. John Wiley & Sons, 2014.

[94] Wikimedia Foundation. *Page view statistics for Wikimedia projects*. https://dumps.wikimedia.org/other/pagecounts-raw/. 2017.

[95] Philip L. Woodworth, Melisa Menéndez, and W. Roland Gehrels. "Evidence for Century-Timescale Acceleration in Mean Sea Levels and for Recent Changes in Extreme Sea Levels". In: *Surveys in Geophysics* 32.4-5 (2011), pp. 603–618.

[96] K WU, E OTOO, and A SHOSHANI. "On the Performance of Bitmap Indices for High Cardinality Attributes1". In: *Proceedings of the International Conference on Very Large Data Bases, VLDB*. 2004, pp. 24–35.

[97] Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. "Continuous sampling for online aggregation over multiple queries". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2010, pp. 651–662.

[98] Dong Xie et al. "Simba: Efficient In-Memory Spatial Analytics". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2016.

[99] Dong Xin et al. "Answering Top-k Queries with Multi-Dimensional Selections: The Ranking Cube Approach". In: *Proceedings of the International Conference on Very Large Data Bases, VLDB*. 2006, pp. 463–475.

[100] Dong Xin et al. "Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration". In: *Proceedings of the International Conference on Very Large Data Bases, VLDB*. 2003, pp. 476–487.

[101] Simin You, Jianting Zhang, and Le Gruenwald. "Spatial Join Query Processing in Cloud: Analyzing Design Choices and Performance Comparisons". In: *Proceedings of the International Conference on Parallel Processing Workshops*. Vol. 2015-Janua. IEEE, 2015, pp. 90–97.

[102] Jia Yu and Mohamed Sarwat. "Indexing the pickup and drop-off locations of NYC taxi trips in PostgreSQL – Lessons from the road". In: *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD*. 2017, pp. 145–162.

[103] Jia Yu and Mohamed Sarwat. "Turbocharging Geospatial Visualization Dashboards via a Materialized Sampling Cube Approach". In: *Proceedings of the International Conference on Data Engineering, ICDE*. 2020, to appear.

[104] Jia Yu and Mohamed Sarwat. "Two birds, one stone: A fast, yet lightweight, indexing scheme for modern database systems". In: *Proceedings of the VLDB Endowment, PVLDB* 10.4 (2016), pp. 385–396.

[105] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. "Spatial data management in apache spark: the GeoSpark perspective and beyond". In: *GeoInformatica* 23.1 (2019), pp. 37–78.

[106] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. 2012, pp. 2–2.

[107] Kai Zeng et al. "ABS: a system for scalable approximate queries with accuracy guarantees". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2014, pp. 1067–1070.

[108] Kai Zeng et al. "The analytical bootstrap: A new method for fast error estimation in Approximate Query Processing". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2014, pp. 277–288.

[109] Ning Zeng, Robert E. Dickinson, and Xubin Zeng. "Climatic impact of Amazon deforestation - A mechanistic model study". In: *Journal of Climate* 9.4 (1996), pp. 859–883.

[110] Shubin Zhang et al. "SJMR: Parallelizing spatial join with MapReduce on clusters". In: *Proceedings - IEEE International Conference on Cluster Computing, ICCC*. 2009, pp. 1–8.

[111] Peixiang Zhao et al. "Graph cube: on warehousing and OLAP multidimensional networks". In: *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. 2011, pp. 853–864.

[112] Xiaofang Zhou, David J. Abel, and David Truffet. "Data partitioning for parallel spatial join processing". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1262 LNCS.2 (1997), pp. 178–196.

[113] Marcin Zukowski et al. "Super-scalar RAM-CPU cache compression". In: *Proceedings of the International Conference on Data Engineering, ICDE*. Vol. 2006. 2006, p. 59.