Simba: Efficient In-Memory Spatial Analytics

Dong Xie[1], Feifei Li[1], Bin Yao[2], Gefei Li[2], Liang Zhou[2], Minyi Guo[2]
[1]University of Utah [2]Shanghai Jiao Tong University
{dongx, lifeifei}@cs.utah.edu {yaobin@cs., oizz01@, nichozl@, guo-my@cs.}sjtu.edu.cn

ABSTRACT

Large spatial data becomes ubiquitous. As a result, it is critical to provide *fast, scalable, and high-throughput* spatial queries and analytics for numerous applications in location-based services (LBS). Traditional spatial databases and spatial analytics systems are disk-based and optimized for IO efficiency. But increasingly, data are stored and processed in memory to achieve low latency, and CPU time becomes the new bottleneck. We present the Simba (Spatial In-Memory Big data Analytics) system that offers scalable and efficient in-memory spatial query processing and analytics for big spatial data. Simba is based on Spark and runs over a cluster of commodity machines. In particular, Simba extends the Spark SQL engine to support rich spatial queries and analytics through both SQL and the DataFrame API. It introduces indexes over RDDs in order to work with big spatial data and complex spatial operations. Lastly, Simba implements an effective query optimizer, which leverages its indexes and novel spatial-aware optimizations, to achieve both low latency and high throughput. Extensive experiments over large data sets demonstrate Simba's superior performance compared against other spatial analytics system.

1. INTRODUCTION

There has been an explosion in the amount of spatial data in recent years. Mobile applications on smart phones and various internet of things (IoT) projects (e.g., sensor measurements for smart city) generate humongous volume of data with spatial dimensions. What's more, spatial dimensions often play an important role in these applications, for example, user and driver locations are the most critical features for the Uber app. How to query and analyze such large spatial data with low latency and high throughput is a fundamental challenge. Most traditional and existing spatial databases and spatial analytics systems are disk-oriented (e.g., Oracle Spatial, SpatialHadoop, and Hadoop GIS [11, 22]). Since they have been optimized for IO efficiency, their performance often deteriorates when scaling to large spatial data.

A popular choice for achieving low latency and high throughput nowadays is to use in-memory computing over a cluster of commodity machines. Systems like Spark [38] have witnessed great success in big data processing, by offering low query latency and high analytical throughput using distributed memory storage and computing. Recently, Spark SQL [13] extends Spark with a SQL-like query interface and the DataFrame API to conduct relational processing on different underlying data sources (e.g., data from DFS). Such an extension provides useful abstraction for supporting easy and user-friendly big data analytics over a distributed memory space. Furthermore, the declarative nature of SQL also enables rich opportunities for query optimization while dramatically simplifying the job of an end user.

However, none of the existing *distributed in-memory* query and analytics engines, like Spark, Spark SQL, and MemSQL, provide native support for spatial queries and analytics. In order to use these systems to process large spatial data, one has to rely on UDFs or user programs. Since a UDF (or a user program) sits outside the query engine kernel, the underlying system is not able to optimize the workload, which often leads to very expensive query evaluation plans. For example, when Spark SQL implements a spatial distance join via UDF, it has to use the expensive cartesian product approach which is not scalable for large data.

Inspired by these observations, we design and implement the Simba (Spatial In-Memory Big data Analytics) system, which is a distributed in-memory analytics engine, to support spatial queries and analytics over big spatial data with the following main objectives: *simple and expressive programming interfaces, low query latency, high analytics throughput, and excellent scalability*. In particular, Simba has the following distinct features:

- Simba extends Spark SQL with a class of important spatial operations and offers simple and expressive programming interfaces for them in both SQL and DataFrame API.
- Simba supports (spatial) indexing over RDDs (resilient distributed dataset) to achieve low query latency.
- Simba designs a SQL context module that executes multiple spatial queries in parallel to improve analytical throughput.
- Simba introduces spatial-aware optimizations to both logical and physical optimizers, and uses cost-based optimizations (CBO) to select good spatial query plans.

Since Simba is based on Spark, it inherits and extends Spark's fault tolerance mechanism. Different from Spark SQL that relies on UDFs to support spatial queries and analytics, Simba supports such operations natively with the help of its index support, query optimizer, and query evaluator. Because these modules are tailored towards spatial operations, Simba achieves excellent scalability in answering spatial queries and analytics over large spatial data.

The rest of the paper is organized as follows. We introduce the necessary background in Section 2 and provide a system overview of Simba in Section 3. Section 4 presents Simba's programming interfaces, and Section 5 discusses its indexing support. Spatial operations in Simba are described in Section 6, while Section 7 explains

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2915237

Simba's query optimizer and fault tolerance mechanism. Extensive experimental results over large real data sets are presented in Section 8. Section 9 summarizes the related work in addition to those discussed in Section 2, and the paper is concluded in Section 10.

2. BACKGROUND AND RELATED SYSTEMS

2.1 Spark Overview

Apache Spark [38] is a general-purpose, widely-used cluster computing engine for big data processing, with APIs in Scala, Java and Python. Since its inception, a rich ecosystem based on Spark for in-memory big data analytics has been built, including libraries for streaming, graph processing, and machine learning [6].

Spark provides an efficient abstraction for in-memory cluster computing called Resilient Distributed Datasets (RDDs). Each RDD is a distributed collection of Java or Python objects partitioned across a cluster. Users can manipulate RDDs through the functional programming APIs (e.g. `map`, `filter`, `reduce`) provided by Spark, which take functions in the programming language and ship them to other nodes on the cluster. For instance, we can count lines containing "ERROR" in a text file with the following scala code:

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(l => l.contains("ERROR"))
println(errors.count())
```

This example creates an RDD of strings called `lines` by reading an HDFS file, and uses `filter` operation to obtain another RDD `errors` which consists of the lines containing "ERROR" only. Lastly, a `count` is performed on `errors` for output.

RDDs are fault-tolerant as Spark can recover lost data using lineage graphs by rerunning operations to rebuild missing partitions. RDDs can also be cached in memory or made persistent on disk explicitly to accelerate data reusing and support iteration [38]. RDDs are evaluated *lazily*. Each RDD actually represents a "logical plan" to compute a dataset, which consists of one or more *"transformations"* on the original input RDD, rather than the physical, materialized data itself. Spark will wait until certain output operations (known as *"action"*), such as `collect`, to launch a computation. This allows the engine to do some simple optimizations, such as pipelining operations. Back to the example above, Spark will pipeline reading lines from the HDFS file with applying the filter and counting records. Thanks to this feature, Spark never needs to materialize intermediate `lines` and `errors` results. Though such optimization is extremely useful, it is also limited because the engine does not understand the structure of data in RDDs (that can be arbitrary Java/Python objects) or the semantics of user functions (which may contain arbitrary codes and logic).

2.2 Spark SQL Overview

Spark SQL [13] integrates relational processing with Spark's functional programming API. Built with experience from its predecessor Shark [35], Spark SQL leverages the benefits of relational processing (e.g. declarative queries and optimized storage), and allows users to call other analytics libraries in Spark (e.g. SparkML for machine learning) through its DataFrame API.

Spark SQL can perform relational operations on both external data sources (e.g. JSON, Parquet [5] and Avro [4]) and Spark's built-in distributed collections (i.e., RDDs). Meanwhile, it evaluates operations lazily to get more optimization opportunities. Spark SQL introduces a highly extensible optimizer called *Catalyst*, which makes it easy to add data sources, optimization rules, and data types for different application domains such as machine learning.

Spark SQL is a full-fledged query engine based on the underlying Spark core. It makes Spark accessible to a wider user base and offers powerful query execution and optimization planning.

Despite the rich support of relational style processing, Spark SQL does not perform well on spatial queries over multi-dimensional data. Expressing spatial queries is inconvenient or even impossible. For instance, a relational query as below is needed to express a 10 nearest neighbor query for a query point $q = (3.0, 2.0)$ from the table `point1` (that contains a set of 2D points):

```
SELECT * FROM point1
ORDERED BY (point1.x - 3.0) * (point1.x - 3.0) +
           (point1.y - 2.0) * (point1.y - 2.0)
LIMIT 10.
```

To make the matter worse, if we want to retrieve (or do analyses over) the intersection of results from multiple kNN queries, more complex expressions such as nested sub-queries will be involved. In addition, it is impossible to express a kNN join succinctly over two tables in a single Spark SQL query. Another important observation is that most operations in Spark SQL require scanning the entire RDDs. This is a big overhead since most computations may not actually contribute to the final results. Lastly, Spark SQL does not support optimizations for spatial analytics.

2.3 Cluster-Based Spatial Analytics Systems

There exists a number of systems that support spatial queries and analytics over distributed spatial data using a cluster of commodity machines. We will review them briefly next.

Hadoop based system. SpatialHadoop [22] is an extension of the MapReduce framework [18], based on Hadoop, with native support for spatial data. It enriches Hadoop with spatial data awareness in language, storage, MapReduce, and operation layers. In the language layer, it provides an extension to Pig [30], called Pigeon [21], which adds spatial data types, functions and operations as UDFs in Pig Latin Language. In the storage layer, SpatialHadoop adapts traditional spatial index structures, such as Grid, R-tree and R+-tree, to a two-level index framework. In the MapReduce layer, SpatialHadoop extends MapReduce API with two new components, *SpatialFileSplitter* and *SpatialRecordReader*, for efficient and scalable spatial data processing. In the operation layer, SpatialHadoop is equipped with several predefined spatial operations including box range queries, k nearest neighbor (kNN) queries and spatial joins over geometric objects using conditions such as *within* and *intersect*. However, only two-dimensional data is supported, and operations such as circle range queries and kNN joins are not supported as well (according to the latest open-sourced version of SpatialHadoop). SpatialHadoop does have good support on different geometric objects, e.g. segments and polygons, and operations over them, e.g. generating convex hulls and skylines, which makes it a distributed geometric data analytics system over MapReduce [20].

Hadoop GIS [11] is a scalable and high performance spatial data warehousing system for running large scale spatial queries on Hadoop. It is available as a set of libraries for processing spatial queries and an integrated software package in Hive [33]. In its latest version, the SATO framework [34] has been adopted to provide different partition and indexing approaches. However, Hadoop GIS only supports data up to two dimensions and two query types: box range queries and spatial joins over geometric objects with predicates like $within$ (within distance).

GeoSpark. GeoSpark [37] extends Spark for processing spatial data. It provides a new abstraction called Spatial Resilient Distributed Datasets (SRDDs) and a few spatial operations. GeoSpark supports range queries, kNN queries, and spatial joins over SRDDs. Besides, it allows an index (e.g. quad-trees and R-trees) to be the object inside each local RDD partition. Note that Spark allows developers to build RDDs over objects of any user-defined types outside Spark kernel. Thus essentially, an SRDD simply encapsu-

Core Features	Simba	GeoSpark	SpatialSpark	SpatialHadoop	Hadoop GIS
Data dimensions	multiple	$d \leq 2$	$d \leq 2$	$d \leq 2$	$d \leq 2$
SQL	✓	×	×	Pigeon	×
DataFrame API	✓	×	×	×	×
Spatial indexing	R-tree	R-/quad-tree	grid/kd-tree	grid/R-tree	SATO
In-memory	✓	✓	✓	×	×
Query planner	✓	×	×	✓	×
Query optimizer	✓	×	×	×	×
Concurrent query execution	thread pool in query engine	user-level process	user-level process	user-level process	user-level process
query operation support					
Box range query	✓	✓	✓	✓	✓
Circle range query	✓	✓	✓	×	×
k nearest neighbor	✓	✓	only 1NN	✓	×
Distance join	✓	✓	✓	via spatial join	✓
kNN join	✓	×	×	×	×
Geometric object	×[1]	✓	✓	✓	✓
Compound query	✓	×	×	✓	×

Table 1: Comparing Simba against other systems.

Notation	Description
R (resp. S)	a table of a point set R (resp. S)
r (resp. s)	a record (a point) $r \in R$ (resp. $s \in S$)
$\|r, s\|$	L_2 distance from r to s
$\text{maxdist}(q, B)$	$\max_{p \in B} \|p, q\|$ for point q and MBR B
$\text{maxdist}(A, B)$	$\max_{q \in A, p \in B} \|p, q\|$ for MBRs A and B
$\text{mindist}(q, B)$	$\min_{p \in B} \|p, q\|$ for point q and MBR B
$\text{mindist}(A, B)$	$\min_{q \in A, p \in B} \|p, q\|$ for MBRs A and B
$\text{range}(A, R)$	records from R within area A
$\text{knn}(r, S)$	k nearest neighbors of r from S
$R \bowtie_\tau S$	R distance join of S with threshold τ
$R \bowtie_{\text{knn}} S$	kNN join between R and S
R_i, S_j	i-th (resp. j-th) partition of table R (resp. S)
$\text{mbr}(R_i)$	MBR of R_i
cr_i	centroid of $\text{mbr}(R_i)$
u_i	$\max_{r \in R_i} \|cr_i, r\|$

Table 2: Frequently used notations.

lates an RDD of spatial objects (which can be points, polygons or circles) with some common geometry operations (e.g. calculating the MBR of its elements). Moreover, even though GeoSpark is able to use index to accelerate query processing within each SRDD partition, it does not support flexible global index inside the kernel. GeoSpark only supports two-dimensional data, and more importantly, it can only deal with spatial coordinates and does not allow additional attributes in spatial objects (e.g. strings for description). In other words, GeoSpark is *a library running on top of and outside Spark without a query engine*. Thus, GeoSpark does not provide a user-friendly programming interface like SQL or the DataFrame API, and has neither query planner nor query optimizer.

SpatialSpark. SpatialSpark [36] implements a set of spatial operations for analyzing large spatial data with Apache Spark. Specifically, SpatialSpark supports range queries and spatial joins over geometric objects using conditions like `intersect` and `within`. SpatialSpark adopts data partition strategies like fixed grid or kd-tree on data files in HDFS and builds an index (outside the Spark engine) to accelerate spatial operations. Nevertheless, SpatialSpark only supports two-dimensional data, and does not index RDDs natively. What's more, same as GeoSpark, it is also *a library running on top of and outside Spark without a query engine* and *does not* support SQL and the DataFrame API.

Remarks. Note that all the systems mentioned above does not support concurrent queries natively with a multi-threading module. Thus, they have to rely on user-level processes to achieve this, which introduce non-trivial overheads from the operating system and hurt system throughput. In contrast, Simba employs a thread pool inside the query engine, which provides much better performance on concurrent queries. There are also systems like GeoMesa and MD-HBase that are related and reviewed in Section 9. Table 1 compares the core features between Simba and other systems.

2.4 Spatial Operations

In this paper, we focus on spatial operations over *point objects*. Our techniques and frameworks also support rectangular objects (such as MBRs), and can be easily extended to support general geometric objects. Table 2 lists the frequently used notations.

Formally, consider a data set $R \subset \mathbb{R}^d$ with N records, where $d \geq 1$ is the dimensionality of the data set and each record $r \in R$ is a point in $\mathbb{R}^d$. For any two points $p, q \in \mathbb{R}^d$, $|p, q|$ denotes their L_2 distance. We consider the following spatial operations in this paper, due to their wide applications in practice [32].

Definition 1 (Range Query) Given a query area Q (either a rectangle or a circle) and a data set R, a range query (denoted as $range(Q, R)$) asks for all records within Q from R. Formally,

$$range(Q, R) = \{r | r \in R, r \in Q\}.$$

Definition 2 (kNN Query) Given a query point $q \in \mathbb{R}^d$, a data set R and an integer $k \geq 1$, the k nearest neighbors of q w.r.t. R, denoted as $\text{knn}(q, S)$, is a set of k records from R where

$$\forall o \in \text{knn}(q, R), \forall r \in R - \text{knn}(q, R), |o, q| \leq |r, q|.$$

Definition 3 (Distance Join) Given two data sets R and S, and a distance threshold $\tau > 0$, the distance join between R and S, denoted as $R \bowtie_\tau S$, finds all pairs (r, s) within distance τ such that $r \in R$ and $s \in S$. Formally,

$$R \bowtie_\tau S = \{(r, s) | (r, s) \in R \times S, |r, s| \leq \tau\}.$$

Definition 4 (kNN Join) Given two data sets R and S, and an integer $k \geq 1$, the kNN join between R and S, denoted as $R \bowtie_{\text{knn}} S$, pairs each object $r \in R$ with each of its kNNs from S. Formally,

$$R \bowtie_{\text{knn}} S = \{(r, s) | r \in R, s \in \text{knn}(r, S)\}.$$

3. Simba ARCHITECTURE OVERVIEW

Simba builds on Spark SQL [13] and is optimized specially for large scale spatial queries and analytics over multi-dimensional data sets. Simba inherits and extends SQL and the DataFrame API, so that users can easily specify different spatial queries and analytics to interact with the underlying data. A major challenge in this process is to extend both SQL and the DataFrame API to support a rich class of spatial operations *natively inside the Simba kernel*.

Figure 1 shows the overall architecture of Simba. Simba follows a similar architecture as that of Spark SQL, but introduces new features and components across the system stack. In particular, new modules different from Spark SQL are highlighted by orange boxes in Figure 1. Similar to Spark SQL, Simba allows users to interact with the system through command line (CLI), JDBC, and scala/python programs. It can connect to a wide variety of data sources, including those from HDFS (Hadoop Distributed File System), relational databases, Hive, and native RDDs.

An important design choice in Simba is to stay outside the core spark engine and only introduce changes to the kernel of Spark SQL. This choice has made a few implementations more challenging (e.g. adding the support for spatial indexing without modifying Spark core), but it allows easy migration of Simba into new version of Spark to be released in the future.

Programming interface. Simba adds spatial keywords and grammar (e.g. `POINT`, `RANGE`, `KNN`, `KNN JOIN`, `DISTANCE JOIN`) in Spark SQL's query parser, so that users can express spatial queries in SQL-like statements. We also extend the DataFrame API with a similar set of spatial operations, providing an alternative programming interface for the end users. The support of spatial operations

[1]Simba is being extended to support general geometric objects.

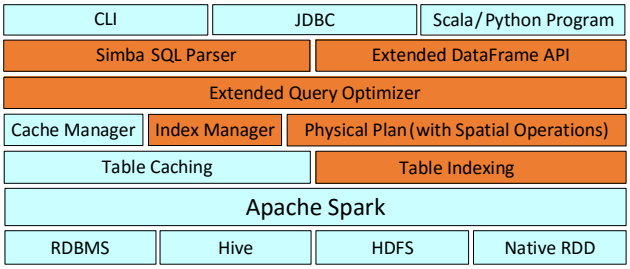


Figure 1: Simba architecture.

in DataFrame API also allows Simba to interact with other Spark components easily, such as MLlib, GraphX, and Spark Streaming. Lastly, we introduce index management commands to Simba's programming interface, in a way which is similar to that in traditional RDBMS. We will describe Simba's programming interface with more details in Section 4 and Appendix A.

Indexing. Spatial queries are expensive to process, especially for data in multi-dimensional space and complex operations like spatial joins and kNN. To achieve better query performance, Simba introduces the concept of *indexing* to its kernel. In particular, Simba implements several classic index structures including hash maps, tree maps, and R-trees [14, 23] over RDDs in Spark. Simba adopts a two-level indexing strategy, namely, *local* and *global* indexing. The global index collects statistics from each RDD partition and helps the system prune irrelevant partitions. Inside each RDD partition, local indexes are built to accelerate local query processing so as to avoid scanning over the entire partition. In Simba, user can build and drop indexes anytime on any table through index management commands. By the construction of a new abstraction called *IndexRDD*, which extends the standard RDD structure in Spark, indexes can be made persistent to disk and loaded back together with associated data to memory easily. We will describe the Simba's indexing support in Section 5.

Spatial operations. Simba supports a number of popular spatial operations over point and rectangular objects. These spatial operations are implemented based on native Spark RDD API. Multiple access and evaluation paths are provided for each operation, so that the end users and Simba's query optimizer have the freedom and opportunities to choose the most appropriate method. Section 6 discusses how various spatial operations are supported in Simba.

Optimization. Simba extends the Catalyst optimizer of Spark SQL and introduces a cost-based optimization (CBO) module that tailors towards optimizing spatial queries. The CBO module leverages the index support in Simba, and is able to optimize complex spatial queries to make the best use of existing indexes and statistics. Query optimization in Simba is presented in Section 7.

Workflow in Simba. Figure 2 shows the query processing workflow of Simba. Simba begins with a relation to be processed, either from an abstract syntax tree (AST) returned by the SQL parser or a DataFrame object constructed by the DataFrame API. In both cases, the relation may contain unresolved attribute references or relations. An attribute or a relation is called *unresolved if we do not know its type or have not matched it to an input table*. Simba resolves such attributes and relations using Catalyst rules and a `Catalog` object that tracks tables in all data sources to build logical plans. Then, the logical optimizer applies standard rule-based optimization, such as constant folding, predicate pushdown, and spatial-specific optimizations like distance pruning, to optimize the logical plan. In the physical planning phase, Simba takes a logical plan as input and generates one or more physical plans based on its spatial operation support as well as physical operators inherited

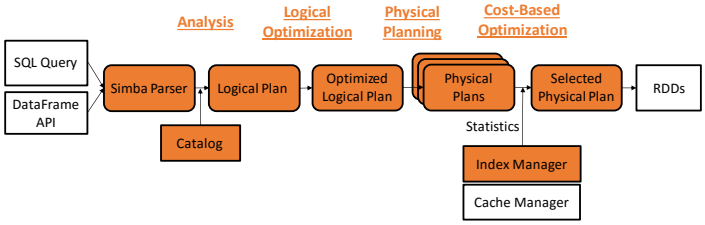


Figure 2: Query processing workflow in Simba.

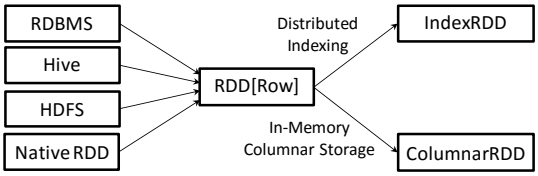


Figure 3: Data Representation in Simba.

from Spark SQL. It then applies cost-based optimizations based on *existing indexes and statistics collected in both Cache Manager and Index Manager* to select the most efficient plan. The physical planner also performs rule-based physical optimization, such as pipelining projections and filters into one Spark `map` operation. In addition, it can push operations from the logical plan into data sources that support predicate or projection pushdown. In Figure 2, we highlight the components and procedures where Simba extends Spark SQL with orange color.

Simba supports analytical jobs on various data sources such as CVS, JSON and Parquet [5]. Figure 3 shows how data are represented in Simba. Generally speaking, each data source will be transformed to an RDD of records (i.e., `RDD[Row]`) for further evaluation. Simba allows users to materialize (often referred as "cache") hot data in memory using columnar storage, which can reduce memory footprint by an order of magnitude because it relies on columnar compression schemes such as dictionary encoding and run-length encoding. Besides, user can build various indexes (e.g. hash maps, tree maps, R-trees) over different data sets to accelerate interactive query processing.

Novelty and contributions. To the best of our knowledge, Simba is the first full-fledged (i.e., support SQL and DataFrame with a sophisticated query engine and query optimizer) in-memory spatial query and analytics engine over a cluster of machines. Even though our architecture is based on Spark SQL, achieving efficient and scalable spatial query parsing, spatial indexing, spatial query algorithms, and a spatial-aware query engine in an in-memory, distributed and parallel environment is still non-trivial, and requires significant design and implementation efforts, since Spark SQL is tailored to relational query processing. In summary,

- We propose a system architecture that adapts Spark SQL to support rich spatial queries and analytics.
- We design the two-level indexing framework and a new RDD abstraction in Spark to build spatial indexes over RDDs natively inside the engine.
- We give novel algorithms for executing spatial operators with efficiency and scalability, under the constraints posed by the RDD abstraction in a distributed and parallel environment.
- Leveraging the spatial index support, we introduce new logical and cost-based optimizations in a spatial-aware query optimizer; many such optimizations are not possible in Spark SQL due to the lack of support for spatial indexes. We also exploit partition tuning and query optimizations for specific spatial operations such as kNN joins.

4. PROGRAMMING INTERFACE

Simba offers two programming interfaces, SQL and the DataFrame API [13], so that users can easily express their analytical queries and integrate them into other components of the Spark ecosystem. Simba's full programming interface is discussed in Appendix A.

Points. Simba introduces the point object in its engine, through a scala class. Users can express a multi-dimensional point using keyword `POINT`. Not only constants or attributes of tables, but also arbitrary arithmetic expressions can be used as the coordinates of points, e.g., `POINT(x + 2, y - 3, z * 2)` is a three-dimensional point with the first coordinate as the sum of attribute `x`'s value and constant 2. This enables flexible expression of spatial points in SQL. Simba will calculate each expression in the statement and wrap them as a point object for further processing.

Spatial predicates. Simba extends SQL with several new predicates to support spatial queries, such as `RANGE` for box range queries, `CIRCLERANGE` for circle range queries, and `KNN` for k nearest neighbor queries. For instance, users can ask for the 3-nearest neighbors of point $(4, 5)$ from table `point1` as below:

```
SELECT * FROM point1
WHERE POINT(x, y) IN KNN(POINT(4, 5), 3).
```

A box range query as follows asks for all points within the two-dimensional rectangle defined by point $(10, 5)$ (lower left corner) and point $(15, 8)$ (top right corner) from table `point2` :

```
SELECT * FROM point2
WHERE POINT(x, y) IN RANGE(POINT(10, 5), POINT(15, 8)).
```

Spatial joins. Simba supports two types of spatial joins: distance joins and kNN joins. Users can express these spatial joins in a θ-join like manner. Specifically, a 10-nearest neighbor join between two tables, `point1` and `point2`, can be expressed as:

```
SELECT * FROM point1 AS p1 KNN JOIN point2 AS p2
ON POINT(p2.x, p2.y) IN KNN(POINT(p1.x, p1.y), 10).
```

A distance join with a distance threshold 20, between two tables `point3` and `point4` in three-dimensional space, is expressed as:

```
SELECT * FROM point3 AS p3 DISTANCE JOIN point4 AS p4
ON POINT(p4.x, p4.y, p4.z) IN
   CIRCLERANGE(POINT(p3.x, p3.y, p3.z), 20).
```

Index management. Users can manipulate indexes easily with index management commands introduced by Simba. For example, users can build an R-Tree index called `pointIndex` on attributes `x`, `y`, and `z` for table `sensor` using command:

```
CREATE INDEX pointIndex ON sensor(x, y, z) USE RTREE.
```

Compound queries. Note that Simba keeps the support for all grammars (including UDFs and UDTs) in Spark SQL. As a result, we can express compound spatial queries in a single SQL statement. For example, we can count the number of restaurants near a POI (say within distance 10) for a set of POIs, and sort locations by the counts, with the following query:

```
SELECT q.id, count(*) AS c
FROM pois AS q DISTANCE JOIN rests AS r
ON POINT(r.x, r.y) IN CIRCLERANGE(POINT(q.x, q.y), 10.0)
GROUP BY q.id ORDER BY c.
```

DataFrame support. In addition to SQL, users can also perform spatial operations over DataFrame objects using a domain-specific language (DSL) similar to data frames in R [10]. Simba's DataFrame API supports all spatial operations extended to SQL described above. Naturally, all new operations are also compatible with the exiting ones from Spark SQL, which provides the same level flexibility as SQL. For instance, we can also express the last SQL query above in the following scala code:

```
pois.distanceJoin(rests, Point(pois("x"), pois("y")),
                  Point(rest("x"), rest("y")), 10.0)
    .groupBy(pois("id"))
    .agg(count("*").as("c")).sort("c").show().
```

5. INDEXING

Indexing is important for the efficient processing of spatial queries and analytics, especially for multi-dimensional data and complex spatial operations such as kNN and spatial joins. In particular, indexing is a critical component towards building an effective optimizer in Simba. Since Simba is an in-memory analytical engine, reducing disk IOs is not a main focus of indexing. Rather, Simba leverages indexes to reduce query latency and improve query throughput via cutting down CPU costs. For example, indexing can help Simba prune irrelevant RDD partitions when processing a range query, which frees more CPU resources for the underlying Spark engine, leading to higher query throughput.

Simba builds (spatial) indexes directly over RDDs to speed up query processing. Specifically, tables are represented as RDDs of records (i.e., `RDD[Row]`). Thus, indexing records of a table becomes indexing elements in an RDD. However, RDDs are designed for sequential scan, thus random access is very expensive as it may become a full scan on the RDD. An extra complexity is that we want to introduce indexing support *without changing the Spark core* for the reasons explained in Section 3. To overcome these challenges, we change the storage format of an indexed table by introducing a new abstraction called `IndexRDD[Row]`, and employ a two-level indexing strategy which can accommodate various index structures to support different queries in Simba.

IndexRDD. Recall that records from a table are stored as `Row` objects (Row objects), and each tables is stored as a RDD of `Row` that contains multiple partitions. To add index support over a table, we pack all records (i.e., `Row` objects) within an RDD partition into an array, which gives each record a unique subscript as its index. Such change makes random access inside RDD partitions an efficient operation with $O(1)$ cost. To achieve this in Simba, we introduce the `IPartition` data structure as below:

```
case class IPartition[Type](Data: Array[Type], I: Index)
```

`Index` is an abstract class that we have designed, and can be instantiated as: `HashMap` (a hash index), `TreeMap` (a one-dimensional index), and `RTree` (a multi-dimensional index). `IndexRDD[Row]` is simply defined as `RDD[IPartition[Row]]` by setting `Type = Row` in the following type declaration:

```
type IndexRDD[Type] = RDD[IPartition[Type]]
```

Records from a table are partitioned by a *partitioner* (that will be described next), and then packed into a set of `IPartition` objects. Since we pack all records within a partition to an array object inside `IPartition`, Simba does introduce small space overhead which may slow down table scanning operation compared to Spark SQL's in-memory columnar storage and the original `RDD[Row]` structure. Experiments in Appendix D.3 have validated that the overhead of our design is insignificant.

Each `IPartition` object contains a *local index* over the records in that partition. Furthermore, each `IPartition` object emits the partition boundary information to construct a *global index*. In general, index construction in Simba consists of three phases: *partition*, *local index*, and *global index*, as shown in Figure 4.

Partition. In the partition phase, Simba partitions the input table (i.e., `RDD[Row]`) according to three main concerns: (1) *Partition Size.* Each partition should have a proper size so as to avoid memory overflow. (2) *Data Locality.* Records that locate close to each other (with respect to the indexing attributes) should be assigned to the same partition. (3) *Load Balancing.* Partitions should be roughly of the same size.

Spark allows users to define their own partition strategies by implementing an abstract class called `Partitioner`. A customized partitioner should specify how many partitions it will generate and how an element maps to a partition ID according to its partition

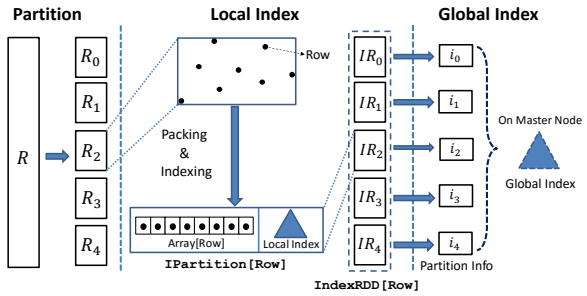


Figure 4: Two-level indexing strategy in Simba.

key. Spark provides two predefined partitioners, range partitioner and hash partitioner, which is sufficient when the partition key is one-dimensional, but does not fit well to multi-dimensional cases.

To address this problem, we have defined a new partitioner named `STRPartitioner` in Simba. `STRPartitioner` takes a set of random samples from the input table and runs the first iteration of Sort-Tile-Recursive (STR) algorithm [26] to determine partition boundaries. Note that the boundaries generated by the STR algorithm are minimum bounded rectangles (MBRs) of the samples. Thus, we need to extend theses MBRs so that they can properly cover the space of the original data set. Finally, according to these extended boundaries, `STRPartitioner` specifies which partition each record belongs to.

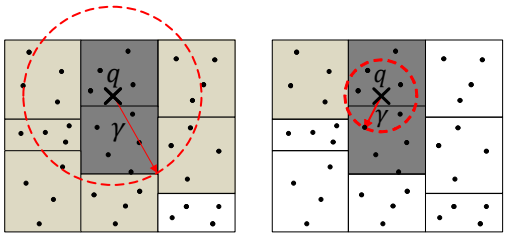
Simba makes no restriction on partition strategies. Instead of using `STRPartitioner`, an end user can always supply his/her own partitioner to Simba. We choose `STRPartitioner` as the default partitioner of Simba due to its simplicity and proven effectiveness by existing studies [19]. As shown in Figure 4, we assume the input table R is partitioned into a set of partitions $\{R_1, \ldots, R_m\}$ by the partitioner. The number of partitions, m, is determined by Simba's optimizer which will be discussed in Section 7.

Local index. In this phase, Simba builds a user-specified index (e.g. R-tree) over data in each partition R_i as its *local* index. It also alters the storage format of the input table from `RDD[Row]` to `IndexRDD[Row]`, by converting each RDD partition R_i to an `IPartition[Row]` object, as shown in Figure 4.

Specifically, for each partition R_i, records are packed into an `Array[Row]` object. Then, Simba builds a *local* index over this `Array[Row]`, and co-locates them in the `IPartition[Row]` object. As we can see, the storage format of the input table is no longer an RDD of `Row` objects, but an RDD of `IPartition[Row]` objects, which is an `IndexRDD` of `Row` objects by the definition above. While packing partition data and building local indexes, Simba also collects several statistics from each partition, such as the number of records and the partition boundaries, to facilitate the construction of the global index as illustrated in Figure 4.

Global index. The last phase of index construction is to build a *global* index which indexes all partitions. The global index enables us to prune irrelevant partitions for an input query *without invoking many executors* to look at data stored in different partitions.

In this phase, partition boundaries generated by the partitioner and other statistics collected in the local index phase are sent to the *master node*. Such information is utilized to bulk load an in-memory index, which is stored in the *driver program* on the master node. Users can specify different types of global indexes: When indexing one-dimensional data, a sorted array of the range boundaries is sufficient (the record count and other statistics for each partition are also stored in each array element). In multi-dimensional cases, more complex index structures, such as R-tree [14, 23] or KD-tree, can be used. By default, Simba keeps the global indexes for different tables in the memory of the master node at all time (i.e., in its driver program). Nevertheless, Simba also allows users to persist global indexes and corresponding statistics to the file system.



(a) Loose Pruning Bound (b) Refined Pruning Bound

Figure 5: Pruning bound for kNN query at global index.

Even for big data, the number of partitions is not very large (from several hundreds to tens of thousands). Thus, global index can easily fit in the memory of the master node. As shown in Figure 9(b) of our experiments, the global index only consumes less than 700KB for our largest data set with 4.4 billion records.

6. SPATIAL OPERATIONS

Simba introduces new *physical execution plans* to Spark SQL for spatial operations. In particular, the support for local and global indexes enables Simba to explore and design new efficient algorithms for classic spatial operations in the context of Spark. Since range query is the simplest, we present its details in Appendix B.

6.1 kNN Query

In Spark SQL, a kNN query can be processed in two steps: (1) calculate distances from all points in the table to the query point; (2) take k records with minimum distances. This procedure can be naturally expressed as an RDD action `takeOrdered`, where users can specify a comparison function and a value k to select k minimum elements from the RDD. However, this solution involves distance computation for every record, a top k selection on each RDD partition, and shuffling of large intermediate results.

In Simba, kNN queries achieve much better performance by utilizing indexes. It leverages two observations: (1) inside each partition, fast kNN selection over the local data is possible by utilizing the local index; (2) a tight pruning bound that is sufficient to cover the global kNN results can help pruning irrelevant partitions using the global index. The first observation is a simple application of classic kNN query algorithms using spatial index like R-tree [31]. The second observation deserves some discussion.

Intuitively, any circle centered at the query point q covering at least k points from R is a safe pruning bound. To get a tighter bound, we shrink the radius γ of this circle. A loose pruning bound can be derived using the global index. Simba finds the nearest partition(s) to the query point q, which are sufficient to cover at least k data points. More than one partition will be retrieved if the nearest partition does not have k points (recall global index maintains the number of records in each partition). The distance between q and a partition R_i is defined as $\text{maxdist}(q, \text{mbr}(R_i))$ [31]. Without loss of generality, assume the following MBRs are returned: $\{\text{mbr}(R_1), \ldots, \text{mbr}(R_\ell)\}$. Then $\gamma = \max\{\text{maxdist}(q, \text{mbr}(R_1)), \ldots, \text{maxdist}(q, R_\ell)\}$ is a safe pruning bound. Figure 5(a) shows an example of this bound as a circle centered at q. The dark boxes are the nearest MBRs retrieved to cover at least k points which help deriving the radius γ. The dark and gray boxes are partitions selected by the global index according to this pruning bound.

To tighten this bound, Simba issues a kNN query on the ℓ partitions selected from the first step (i.e., two partitions with dark boxes in Figure 5(a)), and takes the k-th minimum distance from q to the $k\ell$ candidates returned from these partitions as γ. Figure 5(b) shows this new pruning bound which is much tighter in prac-

tice. Note that ℓ is small for typical k values (often $\ell = 1$), thus, this step has very little overhead.

Global index returns the partition IDs whose MBRs intersect with the circle centered at q with radius γ, Simba marks these partitions using `PartitionPruningRDD` and invokes local kNN queries using the aforementioned observation (1). Finally, it merges the k candidates from each of such partitions and takes the k records with minimum distances to q using RDD action `takeOrdered`.

6.2 Distance Join

Distance join is a θ-join between two tables. Hence, we can express a distance join $R \bowtie_{10.0} S$ in Spark SQL as:

```
SELECT * FROM R JOIN S
ON (R.x - S.x) * (R.x - S.x) + (R.y - S.y) * (R.y - S.y)
    <= 10.0 * 10.0
```

Spark SQL has to use the Cartesian product of two input tables for processing θ-joins. It then filters from the Cartesian product based on the join predicates. Producing the Cartesian product is rather costly in Spark: If two tables are roughly of the same size, it leads to $O(n^2)$ cost when each table has n partitions.

Most systems reviewed in Section 2 did not study distance joins. Rather, they studied *spatial joins*: it takes two tables R and S (each is a set of geometric objects such as polygons), and a spatial join predicate θ (e.g., `overlaps`, `contains`) as input, and returns the set of all pairs (r, s) where $r \in R$, $s \in S$ such that $\theta(r, s)$ is true; $\theta(r, s)$ is evaluated as object r 'θ' (e.g., overlaps) object s.

That said, we design the *DJSpark* algorithm in Simba for distance joins. DJSpark consists of three steps: data partition, global join, and local join, as shown in Figure 6.

Data partition. Data partition phase is to partition R and S so as to preserve data locality where records that are close to each other are likely to end up in the same partition. We also need to consider partition size and load balancing issues. Therefore, we can re-use the *STRPartitioner* introduced in Section 5. The main difference is how we decide the partition size for R and S. Simba has to ensure that it can keep two partitions (one from R and one from S) rather than one (when handling single-relation operations like range queries) in executors' heap memory at the same time.

Note that the data partition phase can be skipped for R (or S) if R (or S) has been already indexed.

Global join. Given the partitions of table R and table S, this step produces all pairs (i, j) which may contribute any pair (r, s) such that $r \in R_i$, $s \in S_j$, and $|r, s| \leq \tau$. Note that for each record $s \in S$, s matches with some records in R_i *only if* $\text{mindist}(s, R_i) \leq \tau$. Thus, Simba only needs to produce the pairs (i, j) such that $\text{mindist}(R_i, S_j) \leq \tau$. After generating these candidate pairs of partition IDs, Simba produces a combined partition $P = \{R_i, S_j\}$ for each pair (i, j). Then, these combined partitions are sent to workers for processing local joins in parallel.

Local join. Given a combined partition $P = \{R_i, S_j\}$ from the global join, local join builds a local index over S_j on the fly (if S is not already indexed). For each record $r \in R_i$, Simba finds all $s \in S_j$ such that $|s, r| \leq \tau$ by issuing a circle range query centered at r over the local index on S_j.

6.3 kNN Join

Several solutions [27,39] for kNN join were proposed on MapReduce. In Simba, we have redesigned and implemented these methods with the RDD abstraction of Spark. Furthermore, we design a new hash-join like algorithm, which shows the best performance.

6.3.1 Baseline methods

The most straightforward approach is the *BKJSpark-N* method (block nested loop kNN join in Spark).

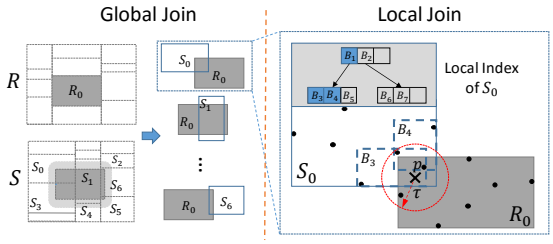


Figure 6: The DJSpark algorithm in Simba.

Producing buckets. R (and S) is divided into n_1 (n_2) equal-sized blocks, which simply puts every $|R|/n_1$ (or $|S|/n_2$) records into a block. Next, every pair of blocks, i.e., (R_i, S_j) for $i \in [1, n_1], j \in [1, n_2]$ are shuffled to a bucket (so a total of $n_1 n_2$ buckets).

Local kNN join. Buckets are processed in parallel. Within each bucket (R_i, S_j), Simba performs $\text{knn}(r, S_j)$ for every $r \in R_i$ through a nested loop over R_i and S_j.

Merge. This step finds the global kNN of every $r \in R$ among its $n_2 k$ local kNNs found in the last step (a total of $|R| n_2 k$ candidates). Simba runs `reduceByKey` (where the key is a record in R) in parallel to get the global kNN of each record in R.

A simple improvement is to build a local R-tree index over S_j for every bucket (R_i, S_j), and use the R-tree for local kNN joins. We denoted it as *BKJSpark-R* (block R-tree kNN join in Spark).

6.3.2 Voronoi kNN Join and z-Value kNN Join

The baseline methods need to check roughly n^2 buckets (when $O(n_1) = O(n_2) = O(n)$), which is expensive for both computation and communication in distributed systems. A MapReduce algorithm leveraging a Voronoi diagram based partitioning strategy was proposed in [27]. It executes only n local joins by partitioning both R and S into n partitions respectively, where the partition strategy is based on the Voronoi diagram for a set of pivot points selected from R. In Simba, We adapt this approach and denote it as the VKJSpark method (Voronoi kNN join in Spark).

Another MapReduce based algorithm for kNN join was proposed in [39], which exploits z-values to map multi-dimensional points into one dimension and uses *random shift* to preserve spatial locality. This approach also produces n partitions for R and S respectively, such that for any $r \in R_i$ and $i \in [1, n]$, $\text{knn}(r, S) \subseteq \text{knn}(r, S_i)$ *with high probability*. Thus, it is able to provide high quality approximations using only n local joins. The partition step is much simpler than the Voronoi kNN join, but it only provides approximate results and there is an extra cost for producing exact results in a post-processing step. We adapt this approach in Simba, and denote it as ZKJSpark (z-value kNN join in Spark).

6.3.3 R-Tree kNN Join

We leverage indexes in Simba to design a simpler yet more efficient method, the RKJSpark method (R-tree kNN join in Spark).

RKJSpark partitions R into n partitions $\{R_1, \ldots, R_n\}$ using the STR partitioning strategy (as introduced in Section 5), which leads to balanced partitions and preserves spatial locality. It then takes a set of random samples S' from S, and builds an R-tree T over S' in the driver program on the master node.

The key idea in RKJSpark is to derive a distance bound γ_i for each R_i, so that we can use γ_i, R_i, and T to find a subset $S_i \subset S$ such that for any $r \in R_i$, $\text{knn}(r, S) = \text{knn}(r, S_i)$. This partitioning strategy ensures that $R \bowtie_{\text{knn}} S = (R_1 \bowtie_{\text{knn}} S_1) \bigcup (R_2 \bowtie_{\text{knn}} S_2) \bigcup \cdots \bigcup (R_n \bowtie_{\text{knn}} S_n)$, and allows the parallel execution of only n local kNN joins (on each (R_i, S_i) pair for $i \in [1, n]$)).

We use cr_i to denote the centroid of $\text{mbr}(R_i)$. First, for each R_i, Simba finds the distance u_i from the furthest point in R_i to cr_i

(i.e., $u_i = \max_{r \in R_i} |r, cr_i|$) in parallel. Simba collects these u_i values and centroids in the driver program on the master node.

Next, Simba finds $\text{knn}(cr_i, S')$ using the R-tree T for $i \in [1, n]$. Without loss of generality, suppose $\text{knn}(cr_i, S') = \{s_1, \ldots, s_k\}$ (in ascending order of their distances to cr_i). Finally, Simba sets:

$$\gamma_i = 2u_i + |cr_i, s_k| \text{ for partition } R_i, \quad (1)$$

and finds $S_i = \{s | s \in S, |cr_i, s| \leq \gamma_i\}$ using a circle range query centered at cr_i with radius γ_i over S. This guarantees the desired property described above, due to:

Theorem 1 *For any partition R_i where $i \in [1, n]$, we have:* $\forall r \in R_i, \text{knn}(r, S) \subset \{s | s \in S, |cr_i, s| \leq \gamma_i\}$, *for γ_i defined in* (1).

The proof is shown in Appendix C. This leads to the design of RKJSpark. Specifically, for every $s \in S$, RKJSpark includes a copy of s in S_i if $|cr_i, s| \leq \gamma_i$. Theorem 1 guarantees that for any $r \in R_i$, $\text{knn}(r, S) = \text{knn}(r, S_i)$.

Then, Simba invokes a `zipPartitions` operation for each $i \in [1, n]$ to place R_i and S_i together into one combined RDD partition. In parallel, on each combined RDD partition, Simba builds an R-tree over S_i and executes a local kNN join by querying each record from R_i over this tree. The union of these n outputs is the final answer for $R \bowtie_{\text{knn}} S$.

7. OPTIMIZER AND FAULT TOLERANCE

7.1 Query Optimization

Simba tunes system configurations and optimizes complex spatial queries automatically to make the best use of existing indexes and statistics. We employed a cost model for determining a proper number of partitions in different query plans. We also add *new logical optimization rules* and *cost-based optimizations* to the Catalyst optimizer and the physical planner of Spark SQL.

The number of partitions plays an important role in performance tuning for Spark. A good choice on the number of partitions not only guarantees no crashes caused by memory overflow, but also improves system throughput and reduces query latency. Given the memory available on a slave node in the cluster and the input table size, Simba partitions the table so that each partition can fit into the memory of a slave node and has roughly equal number of records so as to ensure load balancing in the cluster.

Simba builds a cost model to estimate the partition size under a given schema. It handles two cases: tables with fixed-length records and tables with variable-length records. The first case is easy and we omit its details. The case for variable-length record is much harder, since it is difficult to estimate the size of a partition even if we know how many records are going into a partition. Simba resolves this challenge using a sampling based approach, i.e., a small set of samples (using sampling without replacement) is taken to build estimators for estimating the size of different partitions, where the sampling probability of a record is proportional to its length. This is a coarse estimation depending on the sampling rate; but it is possible to formally analyze the performance of this estimator with respect to a given partitioning strategy, and in practice, this sampling approach is quite effective.

Using the cost model and a specified partition strategy, Simba then computes a good value for the number of partitions such that: 1) partitions are balanced in size; 2) size of each partition fits the memory size of a worker node; and 3) the total number of partitions is proportional to the number of workers in the cluster. Note that the size of different partitions should be close but not greater than a threshold, which indicates how much heap memory Spark can reserve for query processing. In Spark, on each slave node, a fraction of memory is reserved for RDD caching, which is specified as a system configuration `spark.storage.memoryFraction`

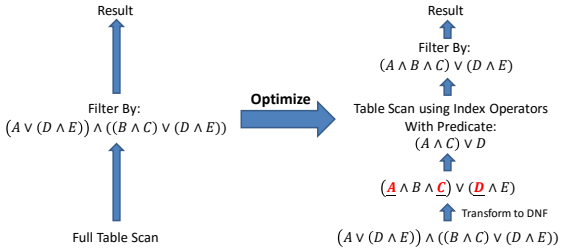


Figure 7: Index scan optimization in Simba.

(we denote this value as α). In our cost model, the remaining memory will be evenly split to each processing core.

Suppose the number of cores is c and total memory reserved for Spark on each slave node is M. The partition size threshold β is then calculated as: $\beta = \lambda((1 - \alpha)M/c)$, where λ is a system parameter, whose default value is 0.8, to take memory consumption of run-time data structures into consideration. With such cost model, Simba can determine the number of records for a single partition, and the numbers of partitions for different data sets.

To better utilize the indexing support in Simba, we add new rules to the Catalyst optimizer to select those predicates which can be optimized by indexes. First, we transform the original select condition to *Disjunctive Normal Form* (DNF), e.g. $(A \wedge B) \vee C \vee (D \wedge E \wedge F)$. Then, we get rid of all predicates in the DNF clause which cannot be optimized by indexes, to form a new select condition θ. Simba will filter the input relation with θ first using index-based operators, and then apply the original select condition to get the final answer.

Figure 7 shows an example of the index optimization. The select condition on the input table is $(A \vee (D \wedge E)) \wedge ((B \wedge C) \vee (D \wedge E))$. Assuming that A, C and D can be optimized by utilizing existing indexes. Without index optimization, the engine (e.g., Spark SQL) simply does a full scan on the input table and filters each record by the select condition. By applying index optimization, Simba works out the DNF of the select condition, which is $(A \wedge B \wedge C) \vee (D \wedge E)$, and invokes a table scan using index operators under a new condition $(A \wedge C) \vee D$. Then, we filter the resulting relation with original condition once more to get the final results.

Simba also exploits various *geometric properties* to merge spatial predicates, to reduce the number of physical operations. Simba merges multiple predicates into segments or bounding boxes, which can be processed together without involving expensive intersections or unions on intermediate results. For example, `x > 3 AND x < 5 AND y > 1 AND y < 6` can be merged into a range query on `(POINT(3, 1), POINT(5, 6))`, which is natively supported in Simba as a single range query. Simba also merges query segments or bounding boxes prior to execution. For instance, two conjunctive range queries on `(POINT(3, 1), POINT(5, 6))` and `(POINT(4, 0), POINT(9, 3))` can be merged into a single range query on `(POINT(4, 1), POINT(5, 3))`.

Index optimization improves performance greatly when the predicates are selective. However, it may cause more overheads than savings when the predicate is not selective or the size of input table is small. Thus, Simba employs a new cost based optimization (CBO), which takes existing indexes and statistics into consideration, to choose the most efficient physical plans. Specifically, we define the selectivity of a predicate for a partition as the percentage of records in the partition that satisfy the predicate.

If the selectivity of the predicate is higher than a user-defined threshold (by default 80%), Simba will choose to scan the partition rather than leveraging indexes. For example, for range queries, Simba will first leverage the local index to do a fast estimation on the predicate selectivity for each partition whose boundary intersects the query area, using *only the top levels of local indexes* for

selectivity estimation (we maintain a count at each R-tree node u that is the number of leaf level entries covered by the subtree rooted at u, and assume uniform distribution in each MBR for the purpose of selectivity estimation,). Selectivity estimation for other types of predicates can be similarly made using top levels of local indexes.

CBO is also used for joins in Simba. When one of the tables to be joined is small (much smaller than the other table), Simba's optimizer will switch to *broadcast join*, which skips the data partition phase on the small table and broadcasts it to every partition of the large table to do local joins. For example, if R is small and S is big, Simba does local joins over $(R, S_1), \ldots, (R, S_m)$. This optimization can be applied to both distance and kNN joins.

For kNN joins in particular, CBO is used to tune RKJSpark. We increase the sampling rate used to generate S' and/or partition table R with finer granularity to get a tighter bound for γ_i. Simba's optimizer adjusts the sampling rate of S' according to the master node's memory size (and the max possible query value $k_{\max}$ so that $|S'| > k_{max}$; k is small for most kNN queries). Simba's optimizer can also invoke another STR partitioner *within each partition* to get partition boundaries *locally on each worker* with finer granularity without changing the global partition and physical data distribution. This allows RKJSpark to compute tighter bounds of γ_i using the refined partition boundaries, thus reduces the size of S_i's.

Lastly, Simba implements a thread-safe SQL context (by creating thread-local instances for conflict components) to support the concurrent execution of multiple queries. Hence, multiple users can issue their queries concurrently to the same Simba instance.

7.2 Fault Tolerance

Simba's fault tolerance mechanism extends that of Spark and Spark SQL. By the construction of `IndexRDD`, table records and local indexes in Simba are still encapsulated in RDD objects! They are persisted at the storage level of `MEM_AND_DISK_SER`, thus *are naturally fault tolerant because of the RDD abstraction of Spark*. Any lost data (records or local indexes) will be automatically recovered by RDD's fault tolerance mechanism (based on the lineage graph). Thus, Simba can tolerate any worker node failures at the same level that Spark provides. This also allows local indexes to be reused when data are loaded back into memory from disk, and avoids repartitioning and rebuilding of local indexes.

For the fault tolerance of the master node (or the driver program), Simba adopts the following mechanism. A Spark cluster can have multiple masters managed by zookeeper [7]. One will be elected "leader" and the others will remain in standby mode. If current leader fails, another master will be elected, recover the old master's state, and then continue scheduling. Such mechanism would recover all system states and global indexes that reside in the driver program, thus ensures that Simba can survive after a master failure.

In addition, user can also choose to persist `IndexRDD` and global indexes to the file system, and have the option of loading them back from the disk. This enables Simba to load constructed index structures back to the system even after power failures.

Lastly, note that all user queries (including spatial operators) in Simba will be scheduled as RDD transformations and actions in Spark. Therefore, fault tolerance for query processing is naturally guaranteed by the underlying lineage graph fault tolerance mechanism provided in Spark kernel and consensus model of zookeeper.

8. EXPERIMENT

8.1 Experiment Setup

All experiments were conducted on a cluster consisting of 10 nodes with two configurations: (1) 8 machines with a 6-core Intel Xeon E5-2603 v3 1.60GHz processor and 20GB RAM; (2) 2 machines with a 6-core Intel Xeon E5-2620 2.00GHz processor and 56GB RAM. Each node is connected to a Gigabit Ethernet switch and runs Ubuntu 14.04.2 LTS with Hadoop 2.4.1 and Spark 1.3.0. We select one machine of type (2) as the master node and the rest are slave nodes. The Spark cluster is deployed in standalone mode. Our cluster configuration reserved up to 180GB of main memory for Spark. We used the following real and synthetic datasets:

OSM is extracted from OpenStreetMap [9]. The full OSM data contains 2.2 billion records in 132GB, where each record has a record ID, a two-dimensional coordinate, and two text information fields (with variable lengths). We took uniform random samples of various sizes from the full OSM, and also duplicated the full OSM to get a data set 2×OSM. These data sets range from 1 million to 4.4 billion records, with raw data size up to 264GB (in 2×OSM).

GDELT stands for Global Data on Events, Language and Tone [8], which is an open database containing 75 million records in total. In our experiment, each record has 7 attributes: a timestamp and three two-dimensional coordinates which represent the locations for the start, the terminal, and the action of an event.

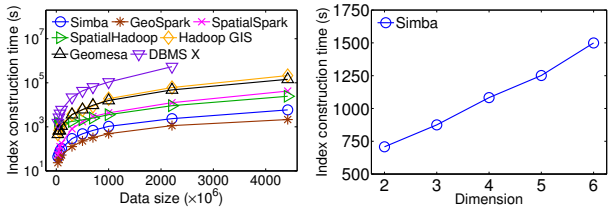
RC: We also generated synthetic datasets of various sizes (1 million to 1 billion records) and dimensions (2 - 6 dimensions) using *random clusters*. Specifically, we randomly generate different number of clusters, using a d-dimensional Gaussian distribution in each cluster. Each record in d-dimension contains $d+1$ attributes: namely, a record ID and its spatial coordinates.

For single-relation operations (i.e. range and kNN queries), we evaluate the performance using *throughput and latency*. In particular, for both Simba and Spark SQL, we start a thread pool of size 10 in the driver program, and issue 500 queries to the system to calculate query throughput. For other systems that we reviewed in Section 2, since they do not have a multi-threading module, we submit 20 queries at the same time and run them as 20 different processes to ensure full utilization of the cluster resources. The throughput is calculated by dividing the total number of queries over the running time. Note that this follows the same way these systems had used to measure their throughput [22]. For all systems, we used 100 randomly generated queries to measure the average query latency. For join operations, we focus on the average running time of 10 randomly generated join queries for all systems.

In all experiments, HDFS block size is 64MB. By default, $k = 10$ for a kNN query or a kNN join. A range query is characterized by its *query area*, which is a percentage over the entire area where data locate. The default query area for range queries is 0.01%. The default distance threshold τ for a distance join is set to 5 (each dimension is normalized into a domain from 0 to 1000). The default partition size is 500×10^3 (500k) records per partition for single-relation operations and 30×10^3 (30k) records per partition for join operations. The default data set size for single-relation operations is 500 million records on OSM data sets and 700 million on RC data sets. For join operations, the default data size is set to 3 million records in each input table. The default dimensionality is 2.

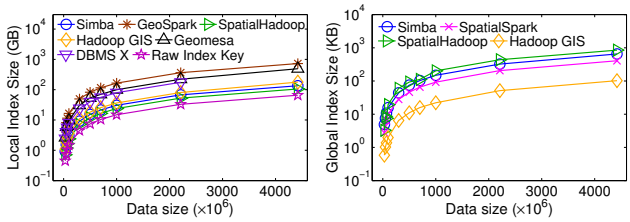
8.2 Cost of Indexing

We first investigate the cost of indexing in Simba and other distributed spatial analytics systems, including GeoSpark [37], SpatialSpark [36], SpatialHadoop [22], Hadoop GIS [11], and Geomesa [24]. A commercial single-node parallel spatial database system (denoted as DBMS X) running on our master node is also included. Figure 8(a) presents the index construction time of different systems on the OSM data set when the data size varies from 30 million to 4.4 billion records (i.e., up to 2×OSM). Note that DBMS X took too much time when building an R-tree index over 4.4 billion records in 2×OSM; hence its last point is omitted.



(a) Effect of data size, OSM. (b) Effect of dimension, RC.

Figure 8: Index construction time.



(a) Local index. (b) Global index.

Figure 9: Index storage overhead on OSM.

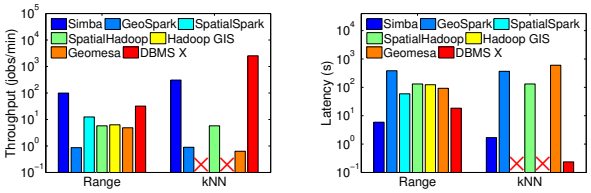
Simba and GeoSpark show the best index construction time, and both scale linearly to the data size. For example, Simba builds its index (which uses R-tree for both local indexes and the global index) over 1 billion records (60GB in file size) in around 25 minutes, which is 2.5x faster than SpatialHadoop, 3x faster than SpatialSpark, 12x faster than Hadoop GIS, and 15x faster than Geomesa. GeoSpark is slightly faster than Simba in building its indexes, because it uses a sampling partition strategy and it only builds a local index for each partition without building global indexes. Simba indexes RDDs natively, while Hadoop based systems index HDFS file blocks. Even though SpatialSpark is also based on Spark, it indexes HDFS files rather than native RDDs. Geomesa builds a hash based index over Apache Accumulo.

We then investigated Simba's indexing support in higher dimensions. Figure 8(b) shows the index construction time in Simba against number of dimensions on the RC data set. The cost increases linearly with the increase in dimensionality. Note that all *other systems can only support up to two dimensions*.

Next, we investigate the storage overhead (memory footprint or disk space used) of indexes in different systems, using OSM data of various sizes. Since Geomesa utilizes a hash based indexing strategy without global indexing, and DBMS X constructs its spatial index on a single node, we show their results with the local index size of the other systems in Figure 9(a), as local indexes dominate the index size of indexing in distributed systems. As a reference, we included the total size of all indexed keys (i.e., spatial coordinates) in Figure 9(a), denoted as "Raw Index Key". SpatialSpark was not included in Figure 9(a) since local indexes are not supported.

As shown in Figure 9(a), Simba, SpatialHadoop and Hadoop GIS have the lowest storage overhead in their local indexes, while those of GeoSpark, Geomesa and DBMS X are much higher (roughly 4-5x). Global indexing is only supported by Simba, SpatialSpark, SpatialHadoop and Hadoop GIS. As shown in Figure 9(b), all systems have small global index storage overhead (only in the order of KB). In particular, the global index size in Simba is very small, which can easily fit in the memory of the master node. For example, it only consumes 653KB for the largest dataset (4.4 billion records with 264GB in raw data) in Simba. This is because that the number of partitions is not a large number (in the order of hundreds to tens of thousands) even for very large data.

Overall, Simba's index storage overhead is acceptable, for example, for 1×OSM with 2.2 billion indexed keys, Simba's local indexes use only 67GB, and its global index uses only 323KB, while the raw key size is about 40GB and the data itself is 132GB.



(a) Throughput (b) Latency

Figure 10: Single-relation operations on different systems.

8.3 Comparison with Existing Systems

In this section, we tested different systems using various spatial operations with the default settings described in Section 8.1 over the 0.25×OSM data set (500 million records). A red cross mark in the following bar charts indicates that the corresponding operation is *not supported* in a system.

Figure 10 shows the performance of single-relation operations on both range and kNN queries. Simba achieves 5-100x better performance (on both throughput and latency) than other existing cluster-based spatial analytics systems and the single-node parallel spatial database. Only DBMS X has outperformed Simba on kNN queries, as when k is small a single-node approach with index support is very efficient for finding kNNs. But its range query performance is worse than Simba.

Note that the performance of GeoSpark in this case is even worse than Hadoop based systems because GeoSpark has not utilized global indexes for query processing, which can help pruning large numbers of useless partitions (more than 90%) before scanning them.

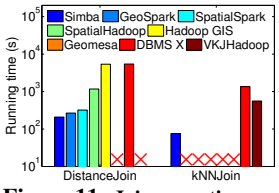
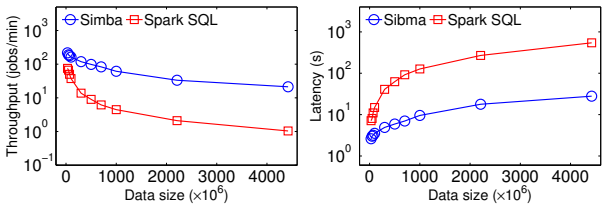


Figure 11: Join operations on different systems.

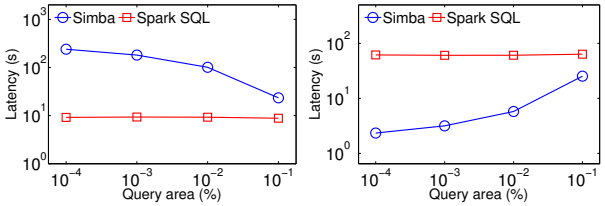
For join operations (using 3 million records in each table), as shown in Figure 11, Simba runs distance join 1.5x faster than SpatialSpark, 25x faster than Hadoop GIS, and 26x faster than DBMS X. Note that distance join over point objects is not natively supported in SpatialHadoop. Nevertheless, SpatialHadoop supports spatial joins over geometric objects. We use $RC(\tau/2) \bowtie_{\text{intersects}} SC(\tau/2)$ to work out the original distance join (by mapping two points to two circles with $\tau/2$ radius centered at them). kNN join is only supported by Simba and DBMS X. For a better comparison with existing distributed methods, we also compared Simba against the Voronoi-based kNN join implemented on Hadoop [1] (denoted as VKJHadoop). By Figure 11, Simba (using its RKJSpark algorithm) is 18x faster than DBMS X and 7x faster than VKJHadoop.

Simba is more efficient than GeoSpark [37] and SpatialSpark [36] because of its indexing support inside the query engine and its query optimizer. GeoSpark and SpatialSpark are only libraries running on top of Spark without a query engine. Compared with Hadoop based systems like SpatialHadoop [22] and Hadoop GIS [11], Simba extends the engine of Spark SQL to support spatial operations natively with a sophisticated, RDBMS-like query engine, and uses Spark for in-memory computation, hence, is much more efficient. For example, Simba provides 51x lower latency and 45x higher throughput than SpatialHadoop for kNN queries. Hadoop based systems will be useful when data is so large that they cannot fit into the cluster's memory space.

Geomesa is a distributed key-value storage system with support for spatial operations. Thus, its analytical performance is not as nearly good as Simba. DBMS X is a single-node parallel database, thus, does not scale well for expensive join operations and large datasets (as evident from its index construction cost).



(a) Effect of data size.



(b) Effect of query area size (percentage of total area).

Figure 12: Range query performance on OSM.

Simba is also much more user-friendly with the native support on both SQL and the DataFrame API over *both spatial and non-spatial operations within a single query*, which are *not* supported by other systems (except that SpatialHadoop supports a SQL-like query language Pigeon). Simba is a full-fledged query engine with native support for query planning, query optimization, and concurrent query processing through thread pooling. Whereas, GeoSpark and SpatialSpark are user programs running on top of Spark (i.e., Spark libraries). What's more, all these systems can only support up to two dimensions. Hence, in the following, we focus on comparing Simba with Spark SQL, which is also a full fledged analytics engine based on Spark but without native support for spatial operations, to see the benefit of extending Spark SQL to Simba for spatial analytics. We support various spatial operations in Spark SQL by directly expressing a spatial operation in its standard SQL syntax if it is possible, or using UDFs in SQL when it is not possible.

8.4 Comparison against Spark SQL

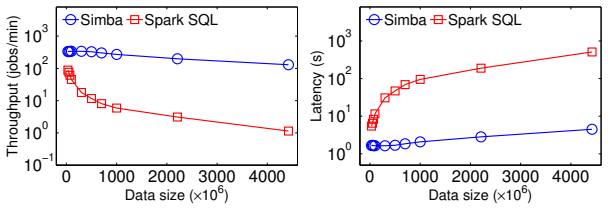
In this comparison we assume that tables are pre-loaded (i..e, we excluded the time of loading tables in both Simba and Spark SQL).

Range queries. Figure 12 shows the performance of range queries in both engines using the OSM data set. Queries are centered at random points sampled from the input data. As a result, the query workload fits well to the distribution of the data, where dense areas will be queried with higher probabilities.

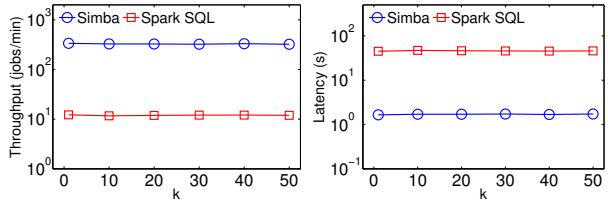
Figure 12(a) shows that Simba outperforms Spark SQL by about one order of magnitude on both query throughput and query latency when data size increases from 30 million to 4.4 billion records (2×OSM). The performance of both Simba and Spark SQL drops when the data size increases while that of Simba drops much slower, which implies Simba has much better scalability. This is due to the spatial index pruning and spatial-aware optimizations in Simba: larger data size does lead to more partitions to process and larger output size, but it also brings more pruning and optimization opportunities for its indexes and optimizer. In contrast, Spark SQL has to scan the whole table regardless.

Figure 12(b) shows how throughput and latency are influenced by the size of the query area. As the query area enlarges, performance of Simba becomes closer to that of Spark SQL. The reason for this is the result size becomes so large that there are less optimization opportunities for Simba's query optimizer. Spark SQL has to scan the whole table regardless.

kNN queries. Figure 13 shows the performance of kNN queries on Spark SQL and Simba over the OSM data set, where query points are also randomly sampled from the input data as range queries.

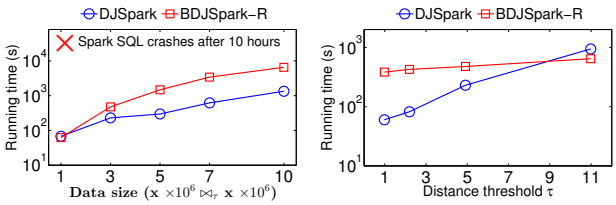


(a) Effect of data size.



(b) Effect of k.

Figure 13: kNN query performance on OSM.



(a) Effect of data size. (b) Effect of τ.

Figure 14: Distance join performance on OSM.

Figure 13(a) measures system throughput and query latency when data size increases from 30 million to 4.4 billion records (2×OSM). Simba achieves one to two orders of magnitude better performance on both metrics. Spark SQL's performance drops significantly as the data size grows, since it requires scanning the whole table for each kNN query. In contrast, Simba's performance is almost not affected by the data size, because Simba is able to narrow the input table down quickly to just a few indexed partitions (typically one or two), which are sufficient to cover the global kNN.

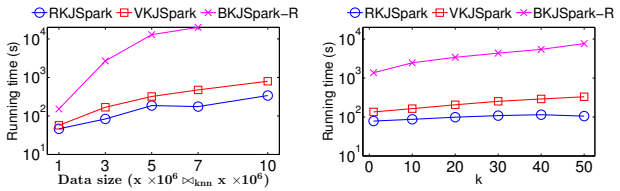
Next, we study the impact of k. As k varies from 1 to 50 in Figure 13(b), Simba maintains a speedup of two orders of magnitude. Both Simba and Spark SQL's performance are not really affected by k: Spark SQL needs to scan the data regardless of k values; whereas Simba's performance will not change much when the change on k is much smaller compared with the partition size.

Distance join. Figure 14 shows the results for distance join using two tables sampled from the OSM data set. We tried expressing and running a distance join as a θ-join in Spark SQL (an example was shown in Section 6.2). However, *it did not finish in 10 hours when joining two tables of only 1 million records, and crashed, due to the expensive Cartesian product it has to perform.*

For Simba, we compared Simba's DJSpark algorithm with a nested loop join approach BDJSpark-R leveraging R-trees as local indexes (which is a simple variation of BKJSpark-R discussed in Section 6.3 by replacing each local kNN join with a local distance join).

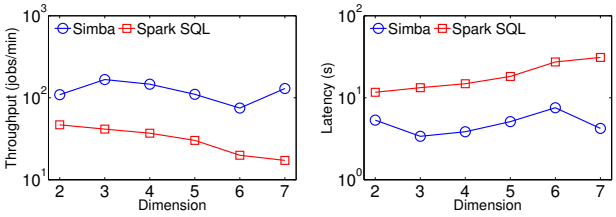
Naturally, the cost of both algorithms increase with larger input size (Figure 14(a)) and larger distance threshold (Figure14(b)). Note that the performance of DJSpark drops fast as the distance threshold grows. This is because the power of global pruning becomes weaker, which will finally cause more overheads than savings when compared with BDJSpark-R. Nevertheless, DJSpark is always more efficient than the baseline method BDJSpark-R, unless the threshold grows to a relatively large value (say $x = 11$ in this case; roughly 1% of the space).

kNN join. It is *impossible to express kNN join in Spark SQL in a*



(a) Effect of data size. (b) Effect of k.

Figure 15: kNN join performance on OSM.



(a) Throughput (b) Latency

Figure 16: Range query performance on GDELT: dimensionality.

single SQL statement. One has to union N SQL statements where N is the number of records in the first input table, which is clearly not a practical solution for large tables. Hence, we focused on comparing different kNN join algorithms in Simba.

Figure 15 shows the performance over OSM data (default is 3 million records in each input table and $k = 10$). Clearly, RKJSpark shows the best performance and the best scalability (w.r.t. both data size and k). As an example, for a kNN join between two tables with 5 million records, RKJSpark join is 3x faster than VKJSpark and 70x faster than BKJSpark-R. Note that BKJSpark-R strictly dominates BKJSpark-N, hence, the latter is omitted.

Remarks. Spark still works when the dataset does not fit in main memory due to its disk persistence mechanism, Simba still has reasonable performance when data does not fit in memory. Clearly, performance will hurt in this case, but in principle, as long as each individual partition can fit in the heap memory of an executor, Spark is able to handle it, so is Simba (and its indexing support is even more important in these cases).

8.5 Support for Multi-Dimensions

In this section, we evaluate the performance of Simba and Spark SQL when handling data in higher dimensions. Note that other spatial analytics systems (GeoSpark, SpatialSpark, SpatialHadoop, and Hadoop GIS) *do not* support more than two dimensions.

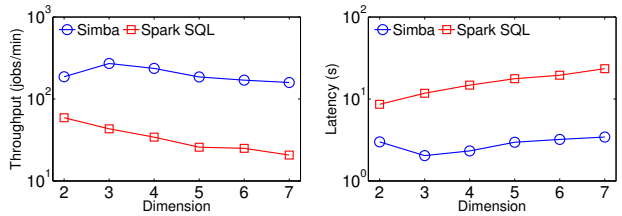
For single-relation operations, as shown in Figures 16 and 17, both Simba and Spark SQL shows higher query latency and lower system throughput on the GDELT data set, as dimensionality increases from 2 to 6. Nevertheless, Simba outperforms Spark SQL by 1-2 orders of magnitude in all cases.

Figure 18 shows the impact of dimensionality on different join algorithms in Simba, by joining two tables with 3 million records from the RC data set as dimensionality goes from 2 to 6. DJSpark and RKJSpark remain the best performance in all dimensions.

9. RELATED WORK

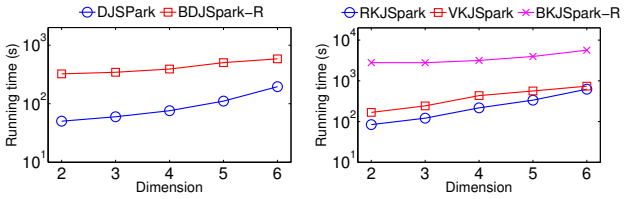
We have already reviewed the most closely related systems in Sections 2 and 8, which Simba has significantly outperformed.

In addition, MD-HBase [29] extends HBase to support location services. It adds KD-tree and quad-tree indexes to HBase to support range and kNN queries. GeoMesa [24] builds a distributed spatial-temporal database on top of Apache Accumulo [3]. It uses GeoHash indexes to provide spatial queries over data stored in Apache Accumulo. Both HBase and Accumulo are modeled after Google's BigTable [16], hence, both MD-HBase and GeoMesa are essen-



(a) Throughput (b) Latency

Figure 17: kNN query performance on GDELT: dimensionality.



(a) Distance join (b) kNN join

Figure 18: Join operations performance on RC: dimensionality.

tially key-value stores with support for spatial operations. As a result, both the design and the objective of the system are very different from an in-memory spatial analytical engine like Simba.

Most existing systems design indexing structures for the MapReduce framework (e.g., Hadoop and HDFS), and they work with indexed data from HDFS. An open source project [2] provides an approach that builds index directly on Spark's RDD abstraction. However, it only supports one-dimensional ART index [25] on key-value RDDs, which doesn't extend to spatial query processing.

In addition to system efforts, indexing and query processing for spatial data in MapReduce were explored, for example, z-value based indexing [15], range queries and kNN queries [12,28,40,40], kNN joins [27, 39], and spatial join over geometric objects [41].

Various spatial partitioning strategies were explored in MapReduce (using Hadoop). A survey and performance comparison can be found in [19,34]. Simba primarily explored the Sort-Tile-Recursive (STR) partitioning scheme [26] for its indexing module, which has shown to be one of the best spatial partitioning methods (e.g., see latest results in [19]). That said, other spatial partitioning methods can be easily employed and supported by Simba.

Lastly, the general principle of Simba's query optimizer is no different from those found in classic relational query engines [17], such as selectivity estimation, CBO, and pushing down predicates. But we have to adopt those principles to an in-memory, distributed and parallel environments, and to spatial operations. The cost models developed for partitioning and cost-based spatial query evaluation in the context of Spark are new. Merging and pushing down spatial predicates based on geometric properties in such a distributed and parallel in-memory query engine was also not explored before.

10. CONCLUSION

This paper describes Simba, a distributed in-memory spatial query and analytics engine based on Spark. Simba offers simple and expressive query language in both SQL and DataFrame API. Simba extends Spark SQL with native support to spatial operations, introduces indexes on RDDs, and adds spatial-aware (logical and cost-based) optimizations to select good query plans. Extensive experiments reveal its superior performance compared to other systems.

For future work, we plan to add native support on more geometric objects (e.g., polygons) and spatial operators (e.g., spatial join over polygons), and data in very high dimensions. We will also design more sophisticated CBOs for effective auto-tuning.

11. ACKNOWLEDGMENT

Feifei Li and Dong Xie were supported in part by NSF grants 1200792, 1302663, 1443046. Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo were supported by the National Basic Research Program (973 Program, No.2015CB352403), and the Scientific Innovation Act of STCSM (No.13511504200, 15JC1402400). Feifei Li and Bin Yao were also supported in part by NSFC grant 61428204.

12. REFERENCES

[1] http://www.comp.nus.edu.sg/ dbsystem/source.html.

[2] https://github.com/amplab/spark-indexedrdd.

[3] Apache accumulo. http://accumulo.apache.org.

[4] Apache avro project. http://avro.apache.org.

[5] Apache parquet project. http://parquet.incubator.apache.org.

[6] Apache spark project. http://spark.apache.org.

[7] Apache zookeeper. https://zookeeper.apache.org/.

[8] Gdelt project. http://www.gdeltproject.org.

[9] Openstreepmap project. http://www.openstreetmap.org.

[10] R project for statistical computing. http://www.r-project.org.

[11] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop gis: a high performance spatial data warehousing system over mapreduce. In *VLDB*, 2013.

[12] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-based geospatial query processing with mapreduce. In *CouldCom*, 2010.

[13] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *SIGMOD*, 2015.

[14] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[15] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *Scientific and Statistical Database Management*, 2009.

[16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *TOCS*, 2008.

[17] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[18] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[19] A. Eldawy, L. Alarabi, and M. F. Mokbel. Spatial partitioning techniques in spatial hadoop. *PVLDB*, 2015.

[20] A. Eldawy, Y. Li, M. F. Mokbel, and R. Janardan. Cg_hadoop: computational geometry in mapreduce. In *SIGSPATIAL*, 2013.

[21] A. Eldawy and M. F. Mokbel. Pigeon: A spatial mapreduce language. In *ICDE*, 2014.

[22] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, 2015.

[23] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[24] J. N. Hughes, A. Annex, C. N. Eichelberger, A. Fox, A. Hulbert, and M. Ronquest. Geomesa: a distributed architecture for spatio-temporal fusion. In *SPIE Defense+ Security*, 2015.

[25] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, 2013.

[26] S. T. Leutenegger, M. Lopez, J. Edgington, et al. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*, 1997.

[27] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. In *VLDB*, 2012.

[28] Q. Ma, B. Yang, W. Qian, and A. Zhou. Query processing of massive trajectory data based on mapreduce. In *Proceedings of the first international workshop on Cloud data management*, 2009.

[29] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi. $\mathcal{MD}$-hbase: design and implementation of an elastic data infrastructure for cloud-scale location services. In *DAPD*, 2013.

[30] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[31] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.

[32] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., 2005.

[33] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. In *PVDLB*, 2009.

[34] H. Vo, A. Aji, and F. Wang. Sato: A spatial data partitioning framework for scalable query processing. In *SIGSPATIAL*, 2014.

[35] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD*, 2013.

[36] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *IEEE CloudDM workshop (To Appear)*, 2015.

[37] J. Yu, J. Wu, and M. Sarwat. Geospark: A cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL GIS*, 2015.

[38] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[39] C. Zhang, F. Li, and J. Jestes. Efficient parallel knn joins for large data in mapreduce. In *EDBT*, 2012.

[40] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. Spatial queries evaluation with mapreduce. In *ICGCC*, 2009.

[41] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *IEEE ICCC*, 2009.

APPENDIX

A. FULL PROGRAMMING INTERFACE

A.1 SQL

Points. Simba introduces the point object inside the Spark SQL engine, through a scala class. Users can use keyword `POINT` followed by a list of expressions to express multi-dimensional points, where the number of expressions indicates the number of dimensions. For example, we can use `Point(x - 1, y * 2)` to express a two dimensional point with the value of $x - 1$ as the first dimension and the value of $y * 2$ as the second dimension. Note that it is easy to extend Simba with additional spatial and geometric objects by adding more scala classes inside its engine.

Range predicates. Simba supports box range query and circle range query. For a box range query, user uses the following predicate to check if points are inside a bounding box:

```
p IN RANGE(low, high)
```

In the predicate above, parameters `p`, `low` and `high` represent three point objects expressed by the grammar for points described above. Specifically, `p` indicates an input point for the predicate to check, while `low` and `high` specify the bounding box (lower-left and higher-right points) for the range query. Points `p`, `low` and `high` can reside in arbitrary dimensions.

Similarly, user specifies a circle range query as below:

```
p IN CIRCLERANGE(c, rd)
```

Note that `p` is again an input point for the predicate to check, while the point object `c` and the constant `rd` indicate a circle centered at point `c` with radius `rd` which specifies a circle range.

kNN predicates. Similar to the range query predicates, Simba provides the kNN predicate as follows:

```
p IN KNN(q, k)
```

It checks if `p` is in the set of k nearest neighbors of a query point q, with respect to an input table. The input table is indicated by the relation after the `FROM` clause. A kNN predicate is a select condition in the `WHERE` clause. A kNN predicate can also serve as the join condition for a kNN join operation.

Distance joins. User expresses a distance join between two tables R and S with a distance threshold τ as follows:

```
R DISTANCE JOIN S ON s IN CIRCLERANGE(r, τ)
```

Here s (resp. r) are points built from a record in table S (resp. R).

kNN joins. A kNN join between table R and table S is expressed as follows:

```
R KNN JOIN S ON s IN KNN(r, k)
```

Similar to distance join, s (resp. r) are built from a record in S (resp. R). User can also invoke approximate kNN join using ZKJSpark (Section 6.3.2) by replacing KNN JOIN with ZKNN JOIN.

Index management. Simba allows users to manipulate indexes easily with its index management keywords. Specifically, user is able to create an index through:

```
CREATE INDEX idx_name ON R(x_1, ..., x_m) USE
    idx_type
```

It builds a new index over m attributes $\{x_1, \ldots, x_m\}$ in table R using a specific index type as indicated by idx_type (which can be R-tree, tree map or hash map). The index is named as idx_name. User can show the indexes already built on table R at anytime:

```
SHOW INDEX ON R
```

In addition, user can drop an index through the index name, or drop all indexes on a table through the table name:

```
DROP INDEX idx_name ON table_name
DROP INDEX table_name
```

Lastly, Simba's SQL commands are fully integrated with Spark SQL's SQL query interface. In other words, user can easily use these keywords and commands together with arbitrary sql predicates from the standard SQL library supported by Spark SQL.

A.2 DataFrame API

Points. Simba also introduces the point object in the DataFrame API, which wraps a list of expressions into a multi-dimensional point that can be processed by Simba's query engine. For example, we can express a three dimensional point as follows:

```
Point(tb("x"), tb("y"), tb("z"))
```

Specifically, the code above wraps three attributes x, y and z from table tb into a point object for further processing.

Single-relation operations. User can apply (box or circle) range queries and kNN queries directly on data frames. Specifically, in the DataFrame API, Simba provides the following functions:

```
range(base: Point, low: Point, high: Point)
circleRange(base: Point, center: Point, r: Double)
knn(base: Point, q: Point, k: Int)
```

In the APIs described above, base indicates the point objects to be filtered, while the other parameters give filter conditions.

Join operations. In the DataFrame API of Simba, distance joins and kNN joins can be expressed with following functions:

```
distanceJoin(target: DataFrame, left_key: Point,
             right_key: Point, τ: Double)
knnJoin(target: DataFrame, left_key: Point,
        right_key: Point, k: Int)
```

These functions will join the current data frame with another data frame target using the join condition (either distance join with threshold τ, or kNN join with value k) over left_key (a point from the first data frame) and right_key (a point from the second data frame).

Index management. User can easily create and drop indexes on an input data frame through:

```
index(index_type: IndexType, index_name: String,
      attrs: Seq[Attribute])
dropIndex()
```

Finally, note that Simba's data frame API is integrated with, and can call the data frame API, from Spark SQL, and other spark-based systems that support the data frame API such as MLLib.

A.3 Limitations of the Programming Model

Simba's main programming and user interface is SQL and the DataFrame API, both of which only provide declarative semantics. As a result, Simba does not have native support for conditional branching (e.g. If. . .Else), looping (For and While loops), and recursion. That said, because Simba is based on Spark which allows users to develop and submit a customized Java or Scala program for execution, users can always overcome the aforementioned hurdles in Simba through a user program in Java or Scala and submit the program to Simba for execution.

B. RANGE QUERY

Two types of range queries are supported in Simba, namely box and circle range queries. They can be expressed by standard SQL queries in Spark SQL by using filters with properly constructed range conditions. But without Simba's programming interfaces described in Section 4, the statement becomes clumsy and error-prone. More importantly, without indexes, the engine has to scan all records, which is quite inefficient and resource intensive.

In contrast, Simba allows users to express range queries natively as shown in Section 4, and it can handle range queries using indexes (assuming that the spatial attributes are indexed):

Global filtering. In this step, Simba uses the global index to prune partitions that do not overlap with the query range. In particular, Simba inspects the global index to obtain IDs of the partitions which intersect the query area. Next, Simba calls a Spark internal developer API PartitionPruningRDD, which can filter partitions by ID, to mark required partitions.

Local processing. For each selected partitions from global filtering, Simba use its local index to quickly return matching records from local data. If $\mathrm{mbr}(R_i)$ is completely inside the query area, all records in R_i can be returned even without checking the index.

C. PROOF OF THEOREM 1

Theorem 1 *For any partition R_i where $i \in [1, n]$, we have:*

$\forall r \in R_i, \mathrm{knn}(r, S) \subset \{s | s \in S, |cr_i, s| \leq \gamma_i\}$, *for γ_i defined in* (1).

PROOF. Recall that $\mathrm{knn}(cr_i, S') = \{s_1, \ldots, s_k\}$ (in ascending order of their distances to cr_i), and $u_i = \max_{r \in R_i} |r, cr_i|$. Hence, for any $r \in R_i$, and for any $t \in [1, k]$, we have:

$$\begin{aligned} |r, s_t| &\leq |r, cr_i| + |cr_i, s_t| \quad \text{(triangle inequality)} \\ &\leq u_i + |cr_i, s_k|. \quad \text{(by construction)} \end{aligned}$$

This implies that a circle centered at r with radius $(u_i + |cr_i, s_k|)$ will cover at least k points (i.e., at least $s_1, s_2, s_3, ..s_k$) in $S' \subset S$. In other words, $\mathrm{knn}(r, S) \subset S_r = \{s | s \in S, |r, s| \leq u_i + |cr_i, s_k|\}$. We denote this set as the *cover set* of record r.

For each element $e \in S_r$, we have:

$$\begin{aligned} |e, cr_i| &\leq |e, r| + |r, cr_i| \quad (\forall r \in R_i, \text{triangle inequality}) \\ &\leq u_i + |cr_i, s_k| + u_i \\ &= \gamma_i, \end{aligned}$$

which implies S_r is a subset of $\{s | s \in S, |cr_i, s| \leq \gamma_i\}$. Thus, $\forall r \in R_i, \mathrm{knn}(r, S) \subset S_r \subset \{s | s \in S, |cr_i, s| \leq \gamma_i\}$. □

D. ADDITIONAL EXPERIMENTS

D.1 Impact of partition size

Figure 19 shows that Simba's index construction cost grows roughly linearly with respect to partition size. This is because it is more costly to build local indexes over larger partitions (which outweighs the savings resulted from processing less number of partitions).

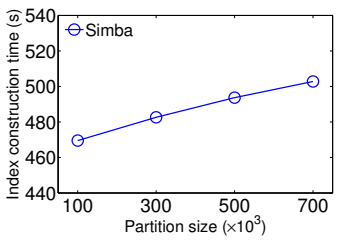
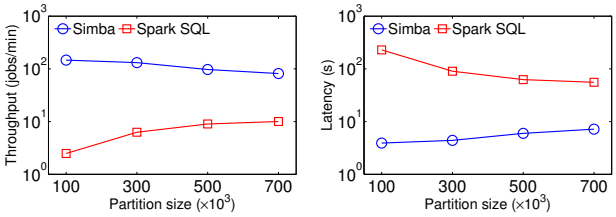


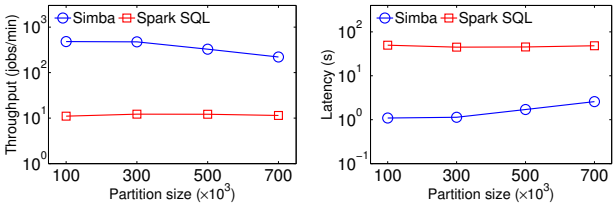
Figure 19: Index construction cost: effect of partition size.



(a) Throughput (b) Latency

Figure 20: Range query performance on OSM: partition size.

Figure 20 shows the effect of partition size (from 1×10^5 to 7×10^5 records per partition). As it increases, the pruning power of global index in Simba shrinks and so does Simba's performance. Spark SQL's performance slightly increases as it requires fewer partitions to process. Nevertheless, Simba is still much more efficient due to its local indexes and spatial query optimizer.

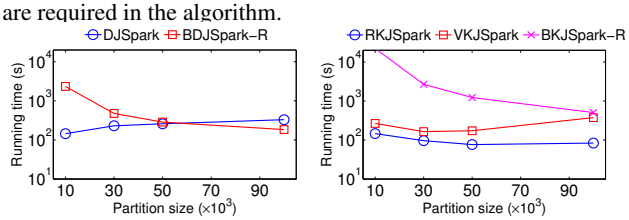


(a) Throughput (b) Latency

Figure 21: kNN query performance on OSM: partition size.

In Figure 21, as the partition size increases, the performance of Simba decreases as the pruning power of global index drops when the partitions become larger.

Figure 22(a) shows the effect of partition size on different distance join algorithms. As the partition size grows, the running time of DJSpark increases slightly, as the pruning power of global join phase reduces when the partition granularity is decreasing. In contrast, BDJSpark-R becomes faster because fewer local join tasks are required in the algorithm.



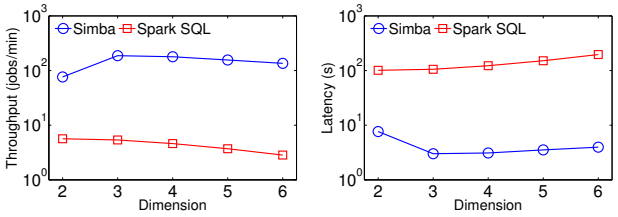
(a) Distance Join (b) kNN Join

Figure 22: Effect of partition size for join operations.

Figure 22(b) presents how partition size affects the performance of different kNN join approaches. With the increase of partition size, BKJSpark-R and RKJSpark grow faster since fewer local join tasks are required for join processing. VKJSpark becomes slightly slower as the partition size increases because the power of its distance pruning bounds weakens when the number of pivots decreases.

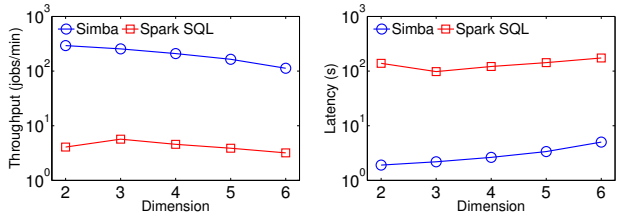
D.2 Influence of Dimensionality on RC

Figures 23 and 24 show the results for range and kNN queries when dimensionality increases from 2 to 6 on the RC data set. The



(a) Throughput (b) Latency

Figure 23: Range query performance on RC: dimensionality.



(a) Throughput (b) Latency

Figure 24: kNN query performance on RC: dimensionality.

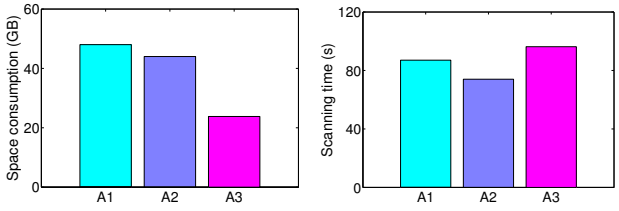
trends are similar to that on the GEDLT data set as in Figures 16 and 17: Simba significantly outperforms Spark SQL in all dimensions.

D.3 Impact of the Array Structure

In this section, we experimentally validated our choice of packing all records in a partition to an array object. Specifically, we compare three different data representation strategies as below:

- Alternative 1 (A1): packing all Row objects in a partition into an array.
- Alternative 2 (A2): cache `RDD[Row]` directly.
- Alternative 3 (A3): in-memory columnar storage strategy.

We demonstrate their space consumption and scanning time on our default dataset (OSM data with 500 million records) in Figure 25. Note that these records are variable-length records (as each record contains two variable-length string attributes, the two text information fields in OSM).



(a) Space Consumption (b) Scanning Time

Figure 25: Comparison between table representations.

Figure 25(a) shows the space consumption of different strategies. Packing all `Row` objects within a partition to an array object (A1 in Figure 25) consumes slightly more space than directly caching the `RDD[Row]` in Spark SQL (A2 in Figure 25). Such overhead is caused by additional meta information kept in array objects. On the other hand, the in-memory columnar storage (A3 in Figure 25) clearly saves space due to its columnar storage with compression.

The table scan time is shown in Figure 25(b). Caching `RDD[Row]` directly shows the best performance. The array structure adopted by Simba has slightly longer scanning time, due to the small space overhead and the additional overhead from flattening members out of the array objects. Lastly, the in-memory columnar storage gives the worst performance since it requires joining multiple attributes to restore the original `Row` object (and the potential overhead from decompression), whose overhead outweighs the savings resulted from its less memory footprint.

That said, Simba can also choose to use the in-memory columnar storage (and with compression) to represent its data.