Simba: Spatial In-Memory Big Data Analysis

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

ABSTRACT

We present the Simba (Spatial In-Memory Big data Analytics) system, which offers scalable and efficient in-memory spatial query processing and analytics for big spatial data. Simba natively extends the Spark SQL engine to support rich spatial queries and analytics through both SQL and DataFrame API. It enables the construction of indexes over RDDs inside the engine in order to work with big spatial data and complex spatial operations. Simba also comes with an effective query optimizer, which leverages its indexes and novel spatial-aware optimizations, to achieve both low latency and high throughput in big spatial data analysis. This demonstration proposal describes key ideas in the design of Simba, and presents a demonstration plan.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems*

Keywords

Simba, Spatial data anlaysis, Big data, Distributed system

1. INTRODUCTION

The wide presence of smart phones and various sensing devices has created an explosion in the amount of spatial data witnessed by various applications in recent years. What's more, spatial features often play an important role in these applications. For instance, user and driver locations are the most critical information 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 analytics systems are disk oriented (e.g., Oracle Spatial, SpatialHadoop [8], Hadoop GIS [4]). Since they have been optimized for disk-resident data, their performance often deteriorates when scaling to IO intensive workloads.

Increasingly, a popular choice for achieving low latency and high throughput systems is to leverage in-memory computing over a cluster of commodity machines. Systems like Apache Spark [15] have witnessed great success in big data processing by using distributed memory storage and computing. Recently, Spark SQL [5] extends Spark with a SQL-like query interface and the DataFrame API to conduct relational data processing on different underlying data sources, which provides useful abstractions to conduct big data analysis at ease. Moreover, the declarative nature of SQL also enables rich opportunities for query optimization while dramatically simplifying the job of an end user.

Nevertheless, none of the existing *distributed in-memory query engines* (e.g., Spark SQL, MemSQL) provides native support for spatial queries and analytics. In order to use such 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 kernel of a query engine, the underlying system is not able to optimize the workload, which often leads to very expensive query evaluation plans. For example, when we use the original query interface of Spark SQL to conduct a spatial distance join, it has to use the expensive Cartesian product approach, which is not scalable.

This demo presents Simba, (Spatial in-memory big data analytics), as a distributed in-memory spatial analytics engine, which supports compound spatial queries and analysis with the following main objectives: *simple and expressive query interface, low query latency, high analytical throughput, and excellent scalability*. In particular, Simba has the following distinct features:

- Simba extends Spark SQL with a class of spatial operations and offers a simple and expressive programming interface in both SQL and DataFrame API.
-

SIGSPATIAL'16 October 31 - November 03, 2016, Burlingame, CA, USA

ACM ISBN 978-1-4503-4589-7/16/10.

DOI: http://dx.doi.org/10.1145/2996913.2996935

Figure 1: Simba architecture.

2. RELATED WORK

There exist a number of systems that support spatial queries and analytics over distributed spatial data using a cluster of commodity machines. SpatialHadoop [8] and Hadoop GIS [4] are based on Hadoop with the MapReduce [7] framework. Both of them utilize global and local indexing to achieve efficient query processing. Since Simba is based on Spark and Spark SQL that explore in-memory computation using the RDD abstraction, the design and system architecture are fundamentally different. Meanwhile, there exist Spark-based approach like GeoSpark [14] and SpatialSpark [13]. But they stand as external libraries built with Spark APIs to support spatial operations, instead of being a full-fledged query and analytical engine that supports features like native RDD indexing, SQL interface, query planner and optimizer, etc.

Other than these systems, MD-HBase [11] extends HBase to support location-based data and queries. It adds KD-tree and quad-tree indexes to HBase to support range and kNN queries. GeoMesa [9] builds a distributed spatial-temporal database on top of Apache Accumulo. Note that both HBase and Accumulo are modeled after Google's BigTable [6], hence, both MD-HBase and GeoMesa are essentially key-value stores with support for spatial operations. As a result, both the design and objective of the systems are very different from an in-memory spatial analytics engine like Simba, and the systems we mentioned above. Lastly, Oracle Spatial is based on the traditional MPP architecture rather than using a cluster of commodity machines, hence, is not as scalable.

Lastly, note that this demo proposal is developed based on our recent work that has the detailed discussion of the design and technical issues in Simba [12].

3. SYSTEM OVERVIEW

Simba extends Spark SQL and is optimized specially for large scale spatial queries and analytics over multi-dimensional data sets. As an extension of Spark SQL, Simba inherits and extends SQL and the DataFrame API so that users can specify different spatial queries and analytics to interact with the underlying data. A major challenge in this process is to extend both SQL and 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. 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, relational databases, Hive and native RDDs.

```
SELECT * FROM point1
ORDER BY (x - 4) * (x - 4) + (y - 5) * (y - 5)
LIMIT 3.
```

(a) kNN query in Spark SQL.

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

(b) kNN query in Simba.

Figure 2: kNN query in Spark SQL vs. Simba.

An important design choice in Simba is to stay outside Spark's core engine and only introduce changes to the kernel of Spark SQL. This choice has made a few designs and implementations more challenging (e.g. adding spatial indexing without modifying Spark core), but it allows easy migration of Simba into new versions of Spark to be released in the future. In the rest of this section, we describe the main ideas and design choices in Simba.

3.1 Programming Interface

Spark SQL provides two programming interfaces, SQL and the DataFrame API, to facilitate user's expressing of their analytical queries and integrating them into other components of the Spark ecosystem. Simba extends both interfaces to support spatial queries and analytics.

Simba adds spatial keywords and grammar (e.g., `POINT`, `RANGE`, `KNN`, `KNN JOIN`, `DISTANCE JOIN`) to Spark SQL's query parser, so that users can express spatial queries in SQL statements. For example, instead of using SQL statement in Figure 2a, users can ask for the 3-nearest neighbors of point (4, 5) from table `point1` as the query in Figure 2b.

Note that Simba keeps the support for all grammars (including UDFs and UDTs) in Spark SQL. As a result, users can express compound spatial queries in a single SQL statement. For instance, 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.
```

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. 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().
```

3.2 Indexing Support

Spatial operations are expensive to process, especially for data in multi-dimensional space and complex operations like spatial joins. To achieve better query performance, Simba introduces the concept of *indexing* to its kernel.

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

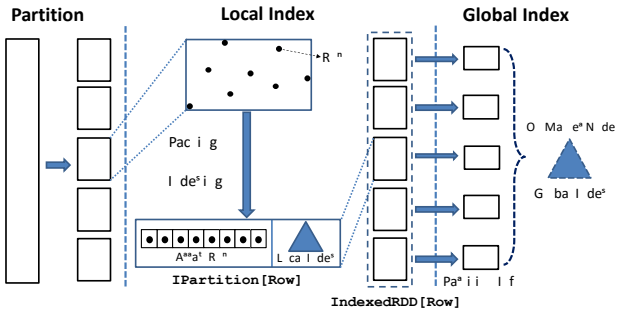


Figure 3: Two-level indexing strategy in Simba.

IndexedRDD. To add index support over an RDD, we pack all elements 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 instantiated with the following implementations: `HashMap`, `TreeMap` (a one-dimensional indexes), and `RTree` (a multi-dimensional index). Finally, `IndexedRDD` is defined as an RDD of `IPartition`:

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

Note that each `IPartition` object contains a *local index* over the elements in that partition, and it also emits the partition boundary and size to construct a *global index* on the master node. As a result, we can naturally fit the element type as records (i.e., `Row`) in `IndexedRDD` so as to index records of a table.

In general, index construction in Simba consists of three phases: *partition*, *local index*, and *global index*, as shown in Figure 3. In the partition phase, Simba partitions the input table according to partition size, data locality and load balancing. Simba defines a partitioner called `STRPartitioner`, which takes a set of random samples from the input table and runs the first iteration of STR Algorithm [10] to determine partition boundaries.

In the local index phase, Simba builds a user-specific index structure (e.g., R-tree) over the data in each partition as its local index. Meanwhile, the storage format of the input table is altered from `RDD[Row]` to `IndexedRDD[Row]`, by converting each RDD partition R_i to an `IPartition[Row]` object. Simba persists each `IndexedRDD` at `MEMORY_AND_DISK_SER` storage level to make it easily reusable.

Finally, in the global index phase, Simba builds a global index at the master node to index all partitions, which enables the pruning of irrelevant partitions for an input query *without invoking many executors* to look at data stored in different partitions. Global indexes are kept in the heap space of the driver program on the master node with no fault tolerance guarantee. Nevertheless, they can be lazily reconstructed from partition boundaries and statistics collected from persisted `IndexedRDD` when required.

3.3 Spatial Operations

Simba introduces new *physical execution plans* to support spatial operations. In particular, the support for local and global indexes enables Simba to explore many efficient implementations for classic spatial operations in the context of Spark.

For range queries, Simba first uses the global index to prune partitions that do not overlap with the query range. Then, for each selected partition, Simba exploits its local index to quickly return matching records from the local data.

An kNN query is carried out in three steps. Firstly, Simba makes use of the global index to generate a loose pruning bound to cover the global kNN results. In particular, Simba finds the nearest partition(s) to the query point q, which is sufficient to cover at least k data points, and we take the maximum distance from the query point to returned partitions. Next, a kNN query over the returned partitions is invoked and we take the k-minimum distance as the new pruning bound which is much tighter. Finally, on each partition that overlaps the pruning bound, Simba executes a local kNN query with the help of the local index, and then merges the results returned from different partitions on the master node.

To process a distance join query, Simba first partitions both input tables using `STRPartitioner`. Then, we start a global join over the partition boundaries of the input tables, where the join condition is that the minimum distance between two partition boundaries (i.e., MBRs) is less than the distance threshold τ in the original join condition. Finally, according to the result of global join, Simba generates a combined partition for each matched pair, and invokes a local join on each combined partition. The final result for the distance join is simply the union of all local join results.

For an kNN join query, we design a hash-join like algorithm. First, Simba partitions one input table R using `STRPartitioner`, and takes a uniform random sample S^θ from another table S. Then, we derive a distance bound γ_i for each partition R_i of R, so that we can use γ_i, R_i, and S^θ to find a subset $S_i \subset S$ such that for any $r \in R_i$, $knn(r, S) = knn(r, S_i)$. Then, we can invoke a local join on each pair of (R_i, S_i), and merge their output as the global kNN join result.

3.4 Optimization

Simba tunes system configurations and optimizes complex spatial queries automatically to make the best use of existing table indexes and statistics. We extend the Catalyst optimizer and the physical planner of Spark SQL with new *logical optimization rules* and *cost-based optimizations*.

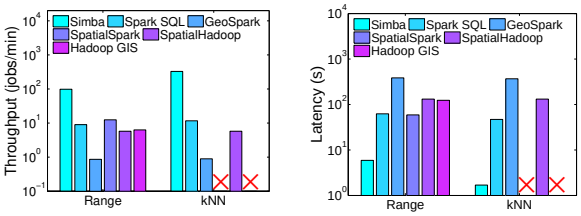
Specifically, to better utilize the indexing support in Simba, we add new rules to the Catalyst optimizer to select the predicates that can be optimized by indexes. We transform the original select condition to *Disjunctive Normal Form* (DNF) and get rid of all predicates which cannot be optimized by 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. Simba also exploits various *geometric properties* to merge predicates in order to reduce the number of physical operations. For example, we merge multiple conjunctive spatial ranges into a single bounding box.

Index optimization improves performance greatly when the predicates are selective, yet it may cause more overheads than savings when the predicate selectivity is low 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 on the fly. Simba also adopts CBO in optimizing join operations. For example, Simba will choose broadcast join when one of the input tables is small. We use CBO to do fine-tuning for kNN join queries as well, by producing logical partition boundaries with finer granularity.

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

4. PERFORMANCE

In this section, we compare the performance of Simba with Spark



(a) Throughput (b) Latency

Figure 4: Single-relation operations on different systems.

SQL and several existing distributed spatial analytics systems including GeoSpark, SpatialSpark, SpatialHadoop, and Hadoop GIS. Our experiments were conducted on a 10-node cluster with 135GB DRAM reserved in total for Simba. A red cross mark in the following bar charts indicates that the corresponding operation is *not supported* in a system.

We first compare the performance of two single-relation operations (namely, range and kNN queries) over a 500 million point subset of the OpenStreetMap (OSM) [3] data set in different systems. As shown in the Figure 4, Simba achieves 5-100x better performance (on both latency and throughput) than all other systems.

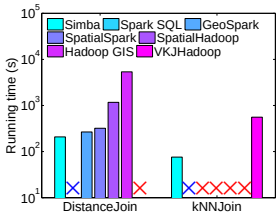


Figure 5: Join operations on different systems.

For join operations, we execute a distance join and a kNN join between two 3-million-record tables, each of which is also a subset of the OSM data set. Note that Spark SQL failed to complete the test on both distance and kNN joins: (1) Spark SQL did not finish the distance join in 10 hours (and crashed) due to the expensive Cartesian product it has to perform. (2) One cannot express a kNN join with the original Spark SQL grammar, and it takes more than a day (and crashed) when trying to do the kNN join using a combination of Spark SQL queries. As shown in Figure 5, Simba runs join queries 1.2x-7x faster than other systems.

5. DEMONSTRATION PLAN

We will present an end-to-end implementation of Simba and show the key features it provides. The demonstration is conducted on the following environments:

Data set: we will have an HDFS instance deployed on a 10-node cluster hosting several data sets of various sizes sampled from OSM and GDELT [2]. The OSM data set will contain up to 500 million records, while GDELT will have up to 75 million records. Data sets of larger size are also available.

Simba cluster: During the demo, we will have a Simba instance running on the 10-node cluster. We will also run different queries on a Spark SQL instance on the same cluster, to make a side-by-side comparison with Spark SQL.

Simba web console: To make Simba more accessible and easy to use for an end-user, we connect it to an open source visualization tool called Zeppelin [1] as shown in Figure 6. Together with the application UI of the underlying Spark system, users can have a clear view of Simba's new grammar, query outputs and internal query execution plans. Users can also issue queries as they wish easily through the GUI in both SQL and Data Frame API, and see the results in various visualized forms.

Our demonstration plan enables attendees to easily explore spatial and temporal features of the underlying data from OSM and GDELT. First, we will show how to import data from different sources and build indexes over them. Then, we will issue range,

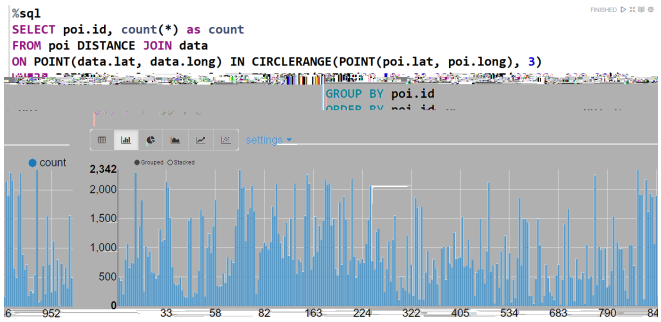


Figure 6: Simba web console.

kNN and join queries to compare Simba's performance with Spark SQL side by side. In addition, we will look into the query execution plan and see how Simba optimizes different queries.

We will also demonstrate the flexibility of Simba in terms of query expression by issuing a number of compound queries involving spatial join, range selections, grouping and aggregations (like the query we have shown in Section 3.1). Finally, another group of examples will present Simba's DataFrame API, and how to use Simba to conduct a detailed analysis on a set of POIs over OSM and GDELT data sets. Attendees will be able to run other ad-hoc queries as well over the datasets through Simba's web console.

6. ACKNOWLEDGMENT

Feifei Li and Dong Xie were supported in part by NSF grants 1200792, 1302663, and 1443046. Bin Yao, Gefei Li, Liang Zhou, Zhongpu Chen 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. The authors are grateful to Dyllon Gagnier for his contribution in developing the front-end GUI for this demo.

7. REFERENCES

[1] http://zeppelin.incubator.apache.org.
[2] Gdelt project. http://www.gdeltproject.org.
[3] Openstreepmap project. http://www.openstreetmap.org.
[4] 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.
[5] 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.
[6] 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. *ACM Trans. Comput. Syst.*, 2008.
[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[8] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, 2015.
[9] 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.
[10] S. T. Leutenegger, M. Lopez, J. Edgington, et al. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*, 1997.
[11] 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.
[12] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, 2016.
[13] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *IEEE CloudDM workshop*, 2015.
[14] J. Yu, J. Wu, and M. Sarwat. Geospark: A cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL GIS*, 2015.
[15] 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.