Sphinx: Empowering Impala for Efficient Execution of SQL Queries on Big Spatial Data

Ahmed Eldawy[1], Ibrahim Sabek[2], Mostafa Elganainy[3], Ammar Bakeer[3],
Ahmed Abdelmotaleb[3], and Mohamed F. Mokbel[2]

[1] University of California, Riverside
eldawy@cs.ucr.edu
[2] University of Minnesota, Twin Cities
{sabek,mokbel}@cs.umn.edu
[3] KACST GIS Technology Innovation Center, Saudi Arabia
{melganainy,abakeer,aothman}@gistic.org

Abstract. This paper presents Sphinx, a full-fledged open-source system for big spatial data which overcomes the limitations of existing systems by adopting a standard SQL interface, and by providing a high efficient core built inside the core of the Apache Impala system. Sphinx is composed of four main layers, namely, *query parser*, *indexer*, *query planner*, and *query executor*. The *query parser* injects spatial data types and functions in the SQL interface of Sphinx. The *indexer* creates spatial indexes in Sphinx by adopting a two-layered index design. The *query planner* utilizes these indexes to construct efficient query plans for range query and spatial join operations. Finally, the *query executor* carries out these plans on big spatial datasets in a distributed cluster. A system prototype of Sphinx running on real datasets shows up-to three orders of magnitude performance improvement over plain-vanilla Impala, SpatialHadoop, and PostGIS.

1 Introduction

There has been a recent marked increase in the amount of spatial data produced by several devices including smart phones, space telescopes, medical devices, among others. For example, space telescopes generate up to 150 GB weekly spatial data, medical devices produce spatial images (X-rays) at 50 PB per year, NASA satellite data has more than 1 PB, while there are 10 Million geotagged tweets issued from Twitter every day as 2% of the whole Twitter firehose. Meanwhile, various applications and agencies need to process an unprecedented amount of spatial data including brain simulation [1], cancer detection [2], the track of infectious disease [3], climate studies [4], and geo-tagged advertising [5].

As a result of that rise of big spatial data and its applications, several attempts have been made to extend big data systems to support big *spatial* data. This includes HadoopGIS [6], SpatialHadoop [7], $\mathcal{MD}$-HBase [8], Distributed Secondo [9], GeoMesa [10], GeoSpark [11], Simba [12], and ESRI Tools for Hadoop [13]. However, these systems suffer from at least one of these limitations: (1) The lack of the ANSI-standard SQL interface, and (2) performance limitations for interactive SQL-like queries due to significant startup time and disk IO overhead.

This paper introduces Sphinx[4], a highly-scalable distributed spatial database with a standard SQL interface. Sphinx is an open source project[5] which extends Impala [15] to efficiently support big spatial data. Impala is a distributed system designed from the ground-up to efficiently answer SQL queries on disk-resident data. It achieves orders of magnitude speedup [15–17] on standard TPC-H and TPC-DS benchmarks as compared to its competitors such as Hive [18] and Spark-SQL [19]. To achieve this impressive performance, Impala is built *from scratch* with several key design points including: (1) ANSI-standard SQL interface, (2) DBMS-like query optimization, (3) C++ runtime code generation, and (4) direct disk access. While Sphinx utilizes this design to achieve orders of magnitude speedup over competitors, it also makes it very challenging due to the fundamental differences over the existing research in this area such as SpatialHadoop [7], GeoSpark [11], and Simba [12].

Even though there has been a body of research in implementing parallel spatial indexes and query processing in traditional DBMS [20, 21], this work is not directly applicable in Impala due to three fundamental differences in their underlying architectures. First, Impala relies on the Hadoop Distributed File System (HDFS) which, unlike traditional file systems, only supports sequential file writing and cannot modify existing files. Second, Impala adopts standard formats of big data frameworks, such as CSV, RCFile, and Parquet, which are more suitable to the big data environment where the underlying data should be accessible to other systems, e.g., Hadoop and Spark. This is totally different from DBMS which has exclusive access to its data giving it more space for optimizing the underlying format. Third, the query processing of Impala relies on runtime code generation where the master node generates an optimized code which is then run by the worker nodes. However, traditional DBMS relies on a precompiled fixed code that cannot be changed at runtime. As a collateral result of these differences, Impala misses key features in traditional databases including indexes which is one of the features we introduce in Sphinx.

Sphinx consists of four layers that are implemented inside the core of Impala, namely, *query parser*, *indexing*, *query planner*, and *query executor*. In the *query parser*, Sphinx introduces spatial data types (e.g., Point and Polygon), functions (e.g., Overlap and Touch), and spatial indexing commands. In the *indexing* layer, Sphinx constructs two-layer spatial indexes, based on grid, R-tree and Quad tree, in HDFS. Sphinx also extends the *query planner* by adding new query plans for two spatial operations, *range* and *spatial join* queries. Finally, the *query executor* provides two core components which utilize *runtime code generation* to efficiently execute the *range* and *spatial join* queries on both indexed and non-indexed tables. Extensive experiments on real datasets of up to 2.7 billion points, show that Sphinx achieves up-to three orders of magnitude speedup over traditional

[4] The idea of Sphinx was first introduced as a poster here [14]

[5] Project home page is http://www.spatialworx.com/sphinx/ and source code is available at https://github.com/gistic/SpatialImpala

Impala [15] and PostGIS and three times faster than SpatialHadoop [7]. We believe that Sphinx will open new research directions for implementing more spatial queries using Impala.

The rest of this paper is organized as follows. Section 2 gives a background on Impala to make the paper self contained. Section 3 provides an overview of Sphinx. The details of the *query parser*, *indexing*, *query planner*, and *query executor* layers are described in Sections 4-7. Section 8 gives an experimental evaluation of Sphinx using real data. Section 9 reviews related work. Finally, Section 10 concludes the paper.

2 Background on Impala

Impala [15,16] is an open-source engine for distributed execution of SQL queries in the Hadoop ecosystem. It achieves real-time response for analytic queries on gigabytes of data and runs much faster than other comparable SQL engines, including Hive [18] and SparkSQL [19]. In this section, we describe the key features of Impala that are needed to understand the contributions of Sphinx.

Similar to other big data engines (e.g., Hadoop and Spark), Impala adopts the Hadoop Distributed File System (HDFS) as its main storage engine. Impala provides an ANSI-standard SQL interface which deals with input files as tables and enforces the schema only as the file is read to reduce the loading time of new data. This *schema-on-read* feature also allows Impala to use a wide range of storage methods including text files in HDFS, HBase tables, RC Files, and Parquet compressed column store. In Impala, users can provide user-defined functions (UDF) and user-defined aggregate functions (UDAF), which Impala integrates with SQL queries.

SQL queries in Impala are executed through the following four steps: (1) The *query parser* decodes the input query and checks for any syntax errors. (2) The *query planner* applies simple optimization rules (e.g., pushing the selection and projection down) to generate a *single-node* query plan as an initial *logical* plan that is never executed. (3) The *query planner* continues its job by converting the initial *logical* plan to a *distributed physical* plan, which has enough details to be executed on cluster nodes. The physical plan consists of *query fragments* where each fragment can run independently on one machine, which minimizes the movement of data across machines. (4) The *query executor* completes the job by running the physical plan on cluster nodes. The query executor is implemented mainly in C++ and uses runtime code generation, which makes it faster and more memory efficient compared to other engines, e.g., Java [16].

3 Architecture

Sphinx is implemented in the core of Impala to provide orders of magnitude speedup with spatial queries while maintaining backward compatibility with non-spatial queries. Figure 1 provides a high level overview of Sphinx which shows

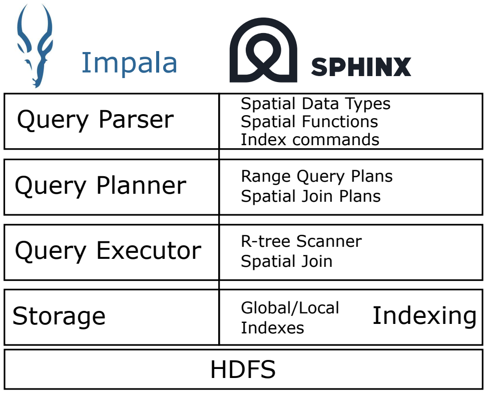


Fig. 1. Overview of Sphinx

the four layers that have been extended from Impala, namely, *query parser*, *query planner*, *query executor*, and *storage/indexing* layers, described briefly below.

Query parser. (Section 4) Sphinx modifies the *query parser* layer by adding spatial data types (e.g., `Point` and `Polygon`), spatial predicates (e.g., `Overlap` and `Touch`), and spatial functions (e.g., `Intersect` and `Union`). It also adds a new `Create Index` command to construct a spatial index. This *syntactic sugar* makes the system user-friendly, especially, for non-technical users.

Storage. (Section 5) In the *storage/indexing* layer, Sphinx employs a two-layer spatial index of one *global index* that partitions the data into blocks and *local indexes* that organize records in each block. Users can build this index using the `Create Index` command in Sphinx or import an existing index from SpatialHadoop. This allows Sphinx to inherit the various indexes [22] supported by SpatialHadoop.

Query planner. (Section 6) In the *query planner* layer, Sphinx adds two new query plans for range query and three plans for spatial join. Sphinx automatically chooses the most efficient plan depending on the existence of spatial indexes.

Query executor. (Section 7) In the *query executor* layer, the query plan, created by the planner, is physically executed on the worker nodes of the cluster. Sphinx introduces two new components, *R-tree scanner* and *spatial join*, which are both implemented in C++ for efficiency. These new components use run-time code generation to optimize the generated machine code based on query selectivity and the types of constructed indexes, if any.

4 Query Parser

Sphinx modifies the *query parser* of Impala to allow users to express spatial queries in an easy way. In particular, Sphinx adds a new `Geometry` data type that expresses spatial objects, and four sets of spatial functions that manipulate geometry objects. In addition, Sphinx introduces the `CREATE INDEX` command to build spatial indexes. Finally, we extend the `CREATE EXTERNAL TABLE` command to import SpatialHadoop indexes into Sphinx.

4.1 Geometry Data Type

Impala supports only the primitive relational data types, such as numbers, Boolean, and string. In addition, it does not provide a way to add user-defined abstract data types (ADT). Therefore, Sphinx modifies the query parser to introduce the new primitive datatype `Geometry`, which represents shapes of various types including `Point`, `Linestring`, `Polygon`, and `MultiPolygon`, as defined by the Open Geospatial Consortium (OGC) [23]. Furthermore, we adopt the Well-Known Text (WKT) format as a default textual representation of shapes for better integration with existing systems, such as PostGIS and Oracle Spatial.

4.2 Spatial Functions

Sphinx adds OGC-compliant spatial functions which are categorized into four groups, namely, *basic functions*, *spatial predicates*, *spatial analysis*, and *spatial aggregates*. Functions in the first three categories are implemented as user-defined functions (UDF), while those in the last one are implemented as user-defined aggregate functions (UDAF). It is imperative to mention that all those functions only work in Sphinx as the input and/or the output of each function is of the `Geometry` datatype, which is supported only in Sphinx.

Basic Spatial Functions which are used to either construct shapes, e.g., `MakePoint`, or retrieve basic information out of them, e.g., `Area`.

Spatial Predicates test the spatial relationship of two `Geometry` objects, such as, `Touch`, `Overlap`, and `Contain`. The return value is always `Boolean`, which allows these functions to be used in the `WHERE` clause of the SQL query.

Spatial Analysis functions are used to manipulate `Geometry` objects, such as `Centroid`, `Intersection`, and `Union`. These functions are usually combined together to perform spatial analysis on a large dataset.

Spatial Aggregate Functions take as input a set of geometries and return one value that summarizes all of the input shapes, e.g., `Envelope` returns the minimum-bounding rectangle (MBR) of a set of objects.

4.3 Spatial Operations

Users can use the above spatial data types and functions to express several spatial operations such as the range query and spatial join operations. The following example show how the range query finds all points in the table P that lie in the rectangle defined by two corner points (x_1, x_2) and (y_1, y_2).

```
SELECT COUNT(*) FROM Points AS P
WHERE Contains(MakeBox(x1, y1, x2, y2), P.coords)
```

The following example uses spatial join to associate points with ZIP code boundaries using the point-in-polygon predicate.

```
SELECT * FROM Points JOIN ZIPCodes
  ON Contains(ZIPCodes.geom, Points.coords)
```

4.4 Spatial Indexing

Sphinx introduces the new `CREATE INDEX` command which the user can use to build spatial indexes. The following example builds an R-tree index on the the field `coordinates` in the table `Points`.

```
CREATE INDEX PointsIndex
  ON Points USING RTREE (coordinates)
```

In addition, we extend the `CREATE EXTERNAL TABLE` command in Impala to allow users to import existing indexes built using SpatialHadoop [7] as shown in the example below.

```
CREATE EXTERNAL TABLE OSM_Points
(
... /* Columns definitions */
)
INDEXED ON coords AS RTREE
LOCATION('/osm_points')
```

5 Spatial Indexing

In this section, we describe how Sphinx constructs spatial indexes on HDFS-resident tables. The main goal of spatial indexing is to store the records in a spatial-aware manner by grouping nearby records and storing them physically together in the same HDFS block. Sphinx employs the two-layered index previously employed by other major big spatial data systems including SpatialHadoop [22] and Simba [12] where one global index partitions records into blocks and several local indexes organize records in each block. Figure 2 shows an example of an R+-tree global index build in Sphinx for a 130 GB of geotagged tweets. Each rectangle in the figure represents a block of 128 MB of records.

This section shows two methods of using indexes in Sphinx. The first method constructs the index within Sphinx while the second method imports imports an existing index constructed by SpatialHadoop. The next two sections show how these indexes are used with two fundamental spatial queries, range query and spatial join.

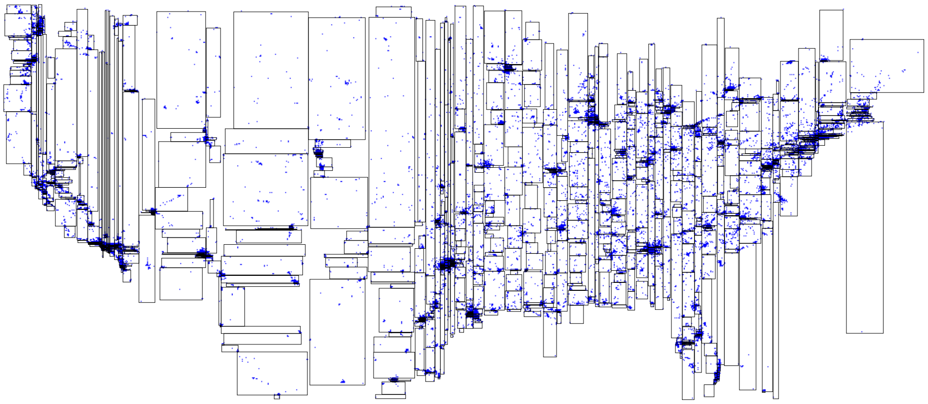


Fig. 2. An R+-tree index in Sphinx built on a table which stores geotagged tweets in the US with a total size of 130 GB

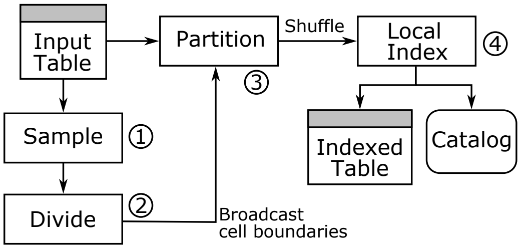


Fig. 3. Indexing plan in Sphinx

5.1 Index Construction in Sphinx

Figure 3 shows the index construction plan in Sphinx which consists of four steps, *sample*, *divide*, *partition*, and *local index*. This method is similar to the index method used in SpatialHadoop [22] and other systems [12, 24]. Implementing the construction plan in Sphinx allows it to be more efficient by supporting the optimized file formats in Impala including RCFile and Parquet. The four steps are described below.

(1) Sample: This step reads a uniform random sample of the table to be indexed. Previous work [22] shows that a 1% sample is enough to produce a high quality index. Distributed random sampling is typically implemented by scanning the entire file and selecting each record in the sample with a probability of 1%. However, due to a limitation in Impala, we cannot apply any random functions in the SQL query. To work around this limitation, we apply a filter function that applies a hash function to the record, uses the hash code to initialize a random number generator, and generates one number from that generator. This way, the function becomes deterministic which satisfies the requirements of Impala.

(2) Divide: The divide step reads the sample and divides the input space into n partitions such that each partition contains roughly the same number of sample points. The number of partitions n is equal to the number of HDFS blocks in the input table. By default, Sphinx applies the STR-based partitioning algorithm [25] which runs in two passes. In the first pass it sorts all points by x and divides the space into $\lceil\sqrt{n}\rceil$ vertical strips each containing roughly the same number of points. In the second pass, it sorts each strip by y and divides it into $\lceil\sqrt{n}\rceil$ horizontal strips with roughly equal number of points. Previous work has shown that this step can be customized to support a wide range of partitioning techniques, e.g., Quad-tree-based and Hilbert-curve-based [22]. This step returns a set of rectangles that represent the boundaries of the partitions.

(3) Partition: In the partition step, the records of the input table are assigned to the index partitions based on their locations. First, the partition boundaries computed in the divide step are broadcast to all cluster nodes. Then, the input table is scanned in parallel and each record is either assigned to one partition or replicated to all overlapping partitioning, depending on the partitioning technique being used. For example, STR-based partitioning assigns a record to exactly one partition while Quad-tree-based partitioning might replicate a record that overlaps multiple partitions. The records from all the machines are then shuffled based on their partition IDs so that all records in each partition are physically colocated in the same machine.

(4) Local Index: In this last step, each partition is locally indexed and then written to HDFS. Each machine processes the partitions, one-by-one, and for each partition is bulk loads the records in main memory and writes the index to the output as a single file. Since each partition fits in one HDFS block, typically 128 MB, it is feasible to construct and write the index using any traditional technique. After all partitions from all machines are written to HDFS, the *catalog serve* in Impala is updated with the index information which includes the MBR of each partition and the corresponding file name.

5.2 Importing SpatialHadoop Indexes

For a better user experience, Sphinx can import the spatial indexes constructed by SpatialHadoop [22]. This allows it to seamlessly work with various indexes based on Quad-tree, K-d tree and others. In this case, the user issues a `CREATE EXTERNAL TABLE` command, and provides the path to the index in HDFS. In addition to creating the table that maps to these files, Sphinx loads the global index into the catalog server.

6 Query Planner

The *query planner* in Sphinx is responsible of generating the *query plan* for a user query, which is later executed by the *query executor*. Sphinx introduces new query plans for both the *range query* and *spatial join* operations. Since Impala does not support any indexes, Sphinx is the first system that shows how to integrate indexes into the query planner of Impala. This can open new research directions of integrating indexes into more complex queries in Impala.

In general, the *query planner* in Impala runs in two phases, namely, *single node planning* and *plan parallelization and fragmentation*. The *single node planning* phase generates a non-executable *logical* plan where each operator is expressed as a node in the plan tree. In the *plan parallelization and fragmentation*, the *logical* plan is translated into a *physical* distributed plan, which can be executed by the cluster nodes. This entire process takes a fraction of a second and runs on a single machine. To conform with this design, Sphinx only utilizes the global index in the query planning phase.

6.1 Range Query Plans

In range query, users want to retrieve all the records that overlap a given query range. Sphinx applies to query plans, a full table scan if the input table is not indexes, and an R-tree search if it is indexed.

Full table scan. In this plan, Sphinx simply scans the entire input table and tests each record according to the query range. This plan is the only one supported by traditional Impala as it does not require indexes and Sphinx applies it if the table is not indexed on the query column. This plan is a direct translation of the following simple SQL query:

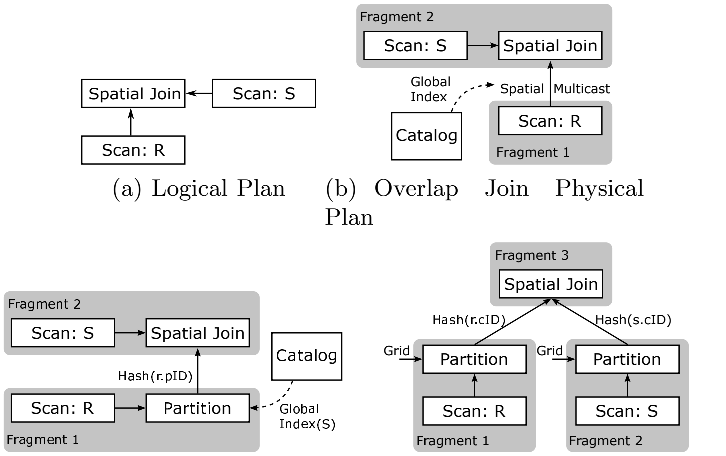
```
SELECT * FROM Points AS P
WHERE Overlap(A, P.x);
```

R-tree search. If the query table is spatially indexed, Sphinx produces this efficient plan that utilizes the index. This plan improves over the full table by employing three new features: (1) *Early pruning*, in which Sphinx utilizes the global index to prune partitions that are completely outside the query range. (2) The *R-tree scanner* replaces the regular scanner in Impala and utilizes the local R-tree indexes to avoid reading records that are outside the query range. (3) The *push down* feature pushes the predicate down to the scanner so that it can utilize the local index. We had to introduce this feature as Impala isolates the query predicate from the scanners. The details of the execution of the *R-tree scanner* are described in Section 7.1.

6.2 Spatial Join Plans

The spatial join query finds all overlapping pairs of records in two input tables, R and S. The only supported plan in Impala is a cross (Cartesian) product followed by a filter which is extremely inefficient for big tables. Instead, Sphinx supports three efficient plans depending on the whether the two input tables have two, one, or zero spatial indexes.

(1) Two Indexes (Overlap Join): If the two input tables are indexed on the join columns, Sphinx provides the *overlap join* algorithm as shown in Fig. 4(b). Without loss of generality, we assume that R is the smaller table. The plan



(a) Logical Plan (b) Overlap Join Physical Plan

(c) Partition Join Physical Plan (d) Co-partition Join Physical Plan

Fig. 4. Spatial Join Query Plans

consists of two fragments assigned to tables R and S. Each machine in fragment 1 reads one partition of the R index while each machine in fragment 2 reads one partition in the S index. Then, each machine in fragment 1 replicates the entire partition to all machines in fragment 2 with an overlapping partition. These links between partitions in the two fragments are based on the partition MBRs extracted from the two global indexes. After that, each machine in fragment 2 performs a spatial join between each partition received from fragment 1 and the assigned partition from S. Sphinx assigns the larger table to fragment 2 to increase the level of parallelism in the join phase as the larger table contains more partition. In order to implement the above plan, Sphinx introduces two new components, the *spatial multicast* communication pattern, and the *spatial join* operator, described below.

Spatial multicast: Communication patterns in Impala control the flow of records from one fragment to another. Sphinx introduces the new *spatial multicast* pattern which can be also helpful to various spatial operations including spatial join. This component first extracts the global indexes for the two tables from the catalog server and uses it to assign an MBR to each partition in the two fragments. Then, it creates a communication link from each partition in fragment 1 to all overlapping partitions in fragment 2.

Spatial join: The *spatial join* operator in Sphinx takes the contents of two partitions and performs a spatial join between the two partitions and generates the answer. The details how the *spatial join* operator is implemented will be described in Section 7.

(2) One Index (Partition Join): Figure 4(c) shows the *partition join* plan which is used when only one table is indexed. This plan resembles the bulk-index-join algorithm [26] and its MapReduce implementation [7]. Without loss of generality, we assume that S is the indexed table. The plan consists of two fragments. Each machine in fragment 1 extracts the global index of S and scans the assigned partition in R. It compares each record $r \in R$ against the partitions of S and outputs the pair $\langle pID, r\rangle$ for each partition pID it overlaps. Then, we use the *hash* communication pattern to send each record r to all overlapping partitions based on the computed attribute pID. Each machine in fragment 2 receives a set of records from R and is assigned a partition in S. It locally computes the spatial join between these two sets of records and output the final answer.

(3) No Indexes (Co-partition Join): If none of the input files is indexed, Sphinx employs the *co-partition join* which is an Impala port of the traditional *partition-based spatial-merge* (PBSM) join algorithm [27], and its MapReduce version [28]. This plan consists of three fragments. In the query planning phase, Sphinx defines a uniform grid based on the MBR of the two files and the number of machines. The two fragments 1&2 scan record in R and S, respectively, and compares each record to the grid. An output record $\langle cID, r\rangle$ or $\langle cID, s$ is written for each overlap between a record and a grid cell. These hash communication pattern is used to group these records by cID and sent to machines in fragment 3. Each machine in fragment 3 is assigned a grid cell and it locally joins all received records from the two tables. It also applies the reference point technique [29] to detect and remove duplicate answers caused by replication.

7 Query Executor

The *query executor* is the component that executes the physical query plan, created by the *query planner*. Sphinx introduces two new components in the query executor, *R-tree scanner* for range queries, and *spatial join* operator. These components are written in C++ for higher performance and less memory overhead.

7.1 R-tree Scanner

The *R-tree scanner* takes as input one table partition P, typically 128 MB, and a rectangular query range A, and returns all records in the partition that overlaps A. The R-tree scanner is implemented as an input scanner which gives it direct access to the disk. To make this possible in Sphinx, we introduced the *predicate push down* feature which pushes the predicate down to the input scanner. This allows the scanner to skip chunks of disk that do not match the query as shown below. A post processing *duplicate avoidance* step might be needed if the input table is indexed with a replicated index, e.g., R+-tree.

The R-tree scanner has three modes of execution depending on the estimated selectivity of the range query. The estimated selectivity is computed as $\sigma = \frac{Area(A\cap P)}{Area(P)}$, where P and A are the MBRs of the partition and the query range, respectively. Depending on the value of σ the three modes of executions are described below. Notice that unlike traditional databases where σ is computed for the entire table, Sphinx computes the value of σ for each partition allowing mixed modes of executions for the same query.

1. Match All ($\sigma = 1.0$): This case applies when the MBR of the partition is completely contained in the query range. In this case, all records in the partition match the query predicate and all of them can be returned without even testing the query predicate.

2. R-tree Search (P is R-tree indexed and $\sigma < \delta$): This case applies under two conditions: (1) the partition P is locally indexed with an R-tree and (2) the estimated selectivity σ is below a threshold δ. In this case, we apply an on-disk R-tree search which can skip big chunks of disk and quickly get the result. Based on our empirical results, we set δ to a default value of 6%.

3. Full Scan (P is not indexed or $\delta \leq \sigma < 1.0$): This case applies if the partition P is not R-tree indexes or the selectivity σ is above the threshold δ. In this case, even if the partition P is indexed, it would be too much overhead to use the R-tree search algorithm as compared to the little saving of disk IO.

Post Processing - Duplicate Avoidance: If the input table is indexed with a replicated index, e.g., R+-tree, this duplicate avoidance step applies the reference point technique to ensure the answer does not contain any duplicates [29].

7.2 Spatial Join Operator

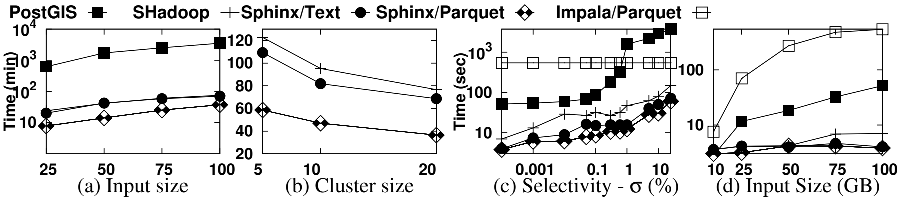
The *spatial join* operator joins two partitions P_1 and P_2 retrieved from the two input files and returns every pair of overlapping records in the two partitions. Similar to the *R-tree scanner*, this operator runs in two steps, *selection* and *duplicate avoidance*. The *selection* step selects pairs of overlapping records in the two partitions. It has three modes of execution depending on the types of local indexes on the two partitions. Unlike the range query, all partitions in one spatial join query follow the same mode of execution as all partitions in one table have the same type of index.

1. R-tree Join: If both partitions are locally indexed with R-trees, i.e., in the *overlap join* algorithm, this operator applies the synchronized traversal algorithm [30] which concurrently traverses both trees while pruning disjoint tree nodes.

2. Bulk Index Join: If only one partition is indexed, i.e., in the *partition join* algorithm, this operator applies the *bulk index join* algorithm [26] which partitions the non-indexed partition according to the R-tree of the indexed one and then joins each partition with the corresponding R-tree node.

3. Plane-sweep Join: If none of the partitions are indexed, i.e., in the *copartition join* algorithm, we apply the traditional planesweep join algorithm [31].

Post Processing - Duplicate Avoidance: If both partitions are indexed with a replicated index, we apply the reference point [29] duplicate avoidance technique to ensure the answer has not duplicates.



(a) Input size (b) Cluster size (c) Selectivity - σ (%) (d) Input Size (GB)

Fig. 5. Index construction and range query performance

8 Experiments

This section provides an extensive experimental study of the initial prototype of Sphinx. The goal is to show the performance gain of Sphinx geared with the spatial indexes and query execution. We run the experiments on four different systems: (1) our proposed prototype of Sphinx, (2) plain-vanilla Impala without any spatial indexes or spatial operators, (3) SpatialHadoop [7] with its spatial indexes and operators, and (4) PostGIS a popular open-source spatial DBMS. Sphinx is built in the code-base of Impala 2.2.0 and is compared to Impala 2.2.0, SpatialHadoop 2.3, and PostGIS 9.3.0. All cluster experiments are conducted on an Amazon EC2 cluster of up-to 20 nodes of type 'm3.xlarge' with quad-core processor and 15GB of RAM. PostGIS is run on a single machine with 64GB of RAM. We use three real datasets extracted from OpenStreetMap and available at [http://spatialhadoop.cs.umn.edu/datasets.html#osm2]. (1) *Nodes* is a set of 2.7 Billion points with a total size of 100GB. (2) *Squares* is derived from the *nodes* dataset by generating $100m^2$ squares centered at each point. (3) *Cities* is a set of 170K polygons extracted from OpenStreetMap with a total size of 500MB. In our experiments, we use the end-to-end query processing time to measure the overall performance.

8.1 Index Construction

Figures 5(a) and (b) show the indexing time in PostGIS, SpatialHadoop, and Sphinx. In general, Sphinx is two orders of magnitude faster than PostGIS and three times faster than SpatialHadoop. We also found that Sphinx running on Parquet is much faster than text files. This finding will urge researchers to consider Parquet format which is not yet the default option in Impala.

Figure 5(a), compares the overall indexing time as the input size increases from 25 to 100 GB. PostGIS is out of the competition with almost two orders of magnitude slowdown as compared to Sphinx on Parquet. SpatialHadoop is on-par with Sphinx on text files due to the similarity of their indexing plans. However, Sphinx on Parquet is three times faster than SpatialHadoop due to the optimizations which are accessible to Sphinx as it is built inside Impala.

Figure 5(b) shows the indexing time for the 100GB dataset as the cluster size is increased from 5 to 20 machines. By taking PostGIS out and using a regular (non-log) scale, we can see that Sphinx on text files is slightly faster than SpatialHadoop due to the more efficient core of Sphinx on Impala.

8.2 Range Query

Figures 5(c) and (d) give the performance of the range query on the *nodes* dataset on Sphinx, Impala, SpatialHadoop, and PostGIS. For each query, we select a random point p from the input and create a query range A as a square centered around p with an area of $\sigma Area(R)$, where $Area(R)$ is the overall area of the input domain. In these experiments, we use `COUNT(*)` to return the total number of results rather than the individual records as this is a typical use-case for querying big spatial data. We rely on the query optimizer of PostGIS to decide whether or not to use the spatial index depending on the selectivity. In general, we found that Sphinx is significantly faster than all other techniques with both text files and Parquet with the latter being the fastest.

In Figure 5(c), we increase the selectivity ratio σ from 0.001% to 25% and measure the query response time. As shown, Sphinx is consistently faster than all other techniques due to the spatial index coupled with the efficient query processing. The R-tree index in PostGIS is only good when the selectivity is low and its performance degrades as the query range increases. The performance of Impala is not affected by the selectivity as it always scans the whole file due to the lack of indexes. While SpatialHadoop uses the same index of Sphinx, the latter is significantly faster even with text files due to the more efficient C++ execution layer in Sphinx.

Figure 5(d) shows the performance of the range query with $\sigma = 0.001\%$ as the input size increases from 10 to 100 GB. While all the techniques scale well with the input size, Sphinx is much faster and the performance gap increases with larger input sizes. It is interesting that the performance of Sphinx and SpatialHadoop remains almost constant as the input size increases as they both utilize the global index to limit the number of processed partitions.

8.3 Spatial Join

Figure 6 gives the performance of the spatial join operation, where we compare the nested loop join in Impala with the overlap join approaches. In Figure 6(a), we compare the performance of the nested loop join in Impala and the spatial join in PostGIS to the *overlap join* approach in SpatialHadoop and Sphinx. Both Impala and Sphinx run on Parquet files to give their best performance. In this experiment, the two inputs are equally sized and obtained using sampling from the *squares* dataset. This experiment clearly shows the superiority of Sphinx over Impala and PostGIS in the spatial join query where it achieves up-to an order of magnitude speedup. As with other experiments, Sphinx is also significantly faster than SpatialHadoop.

In the remaining experiments, we rule out both traditional Impala and PostGIS as they do not support the various spatial join algorithms. In addition, we use two larger datasets as input, *cities* is always used as the first datasets, and samples of the `squares` dataset of different sizes are used. We compare the performance of the three spatial join algorithms in Sphinx, running on both HDFS

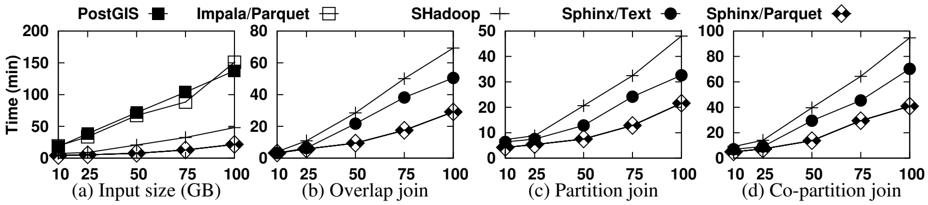
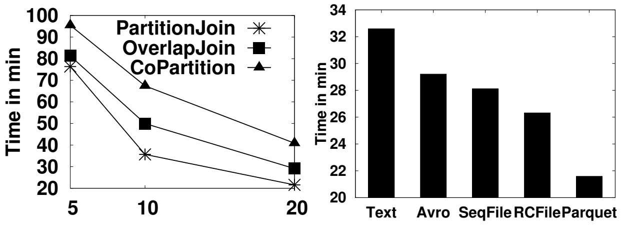


Fig. 6. Spatial Join Performance



(a) Cluster Size (b) Storage Types

Fig. 7. Spatial join tuning in Sphinx

files and Parquet, to the performance of the same algorithm running in SpatialHadoop [7, 28]. Figures 6(b), (c), and (d) show the performance of the *overlap join*, *partition join* and *co-partition* join, respectively, as described in Section 6.2. Sphinx consistently outperforms SpatialHadoop when they both run the same algorithm. Using Parquet in Sphinx gives an additional boost as compared to using text files in HDFS. Overall, Sphinx is up-to 302% and 171% faster than SpatialHadoop when it runs on text files and Parquet, respectively.

Figure 7(a) gives the scalability experiments of the three join algorithms in Sphinx as the cluster size is increased from 5 to 20 machines. The results indicate that all of the three algorithms scale out nicely on the cluster as they parallelize the work over all of the available processing cores in the cluster. Notice that this experiment is not intended to compare the relative performance of the three algorithms as each one runs in a difference input configuration, i.e., two indexes, one index, or no indexes, as described in Section 6.2.

Figure 7(b) shows how the storage type affects the performance of the *overlap join* algorithm in Sphinx. In this experiment, we build two indexes on the *squares* and *cities* datasets and then run the overlap join algorithm on them. First, this figure shows the advantage of building Sphinx, which supports a wide range of storage types, as opposed to other systems, e.g., SpatialHadoop, which only supports text files. Second, it shows that a raw text storage gives the worst performance, even though it is the default option in Impala. On the other hand, Parquet provides the best performance and it is the recommended storage by Cloudera. While other workloads might produce a different behavior, a detailed comparison of these storage types is out of the scope of this paper.

9 Related Work

MapReduce [32] and Hadoop were released as powerful alternatives to traditional DBMS to support Big Data. Hadoop was followed by a series of systems that either build on top of it (e.g., Hive [18] and Pig [33]) or use a renovated system design suitable for other application domains, such as Spark [34] for iterative processing and Impala [15] for running interactive SQL queries. Meanwhile, there has been an ongoing research in supporting spatial queries on these systems to process Big Spatial Data. Such research can be classified into three categories. (1) The *on-top* approach uses an existing system as a black box and provides spatial support through user-defined functions. Techniques in this category are easy to implement but they provide a sub-par performance due to the limitation of the underlying system. Yet, the *on-top* approach was used to express a wide range of spatial queries including, R-tree construction [35], range query [36, 37], k-nearest neighbor (kNN) [36, 38], and spatial join [28, 39]. (2) The *from-scratch* approach which is the other extreme where a system is constructed from scratch to support spatial data. While these systems achieve higher performance, they are very complex to build and maintain. Systems in this category include SciDB [40], an array database for scientific applications, and BRACE [41], an in-memory MapReduce engine for behavioral simulations. (3) The *built-in* approach in which an existing system is extended to support spatial data by injecting spatial data types, primitive spatial functions, spatial indexes, and spatial operations in the core of an existing system to transform it from a spatial-agnostic to a spatial-aware system. Techniques in this category reach a balance between complexity and performance. Systems in this category include HadoopGIS [6]; an extended version of Hive [18] for spatial data, SpatialHadoop [7]; a MapReduce framework for spatial data, $\mathcal{MD}$-HBase [8]; an extension to HBase for supporting spatial data, Parallel Secondo [42]; a parallel spatial DBMS, GeoMesa [10]; a key-value store for spatio-temporal datasets built on Accumulo, ESRI Tools for Hadoop [13], which integrates Hadoop with ArcGIS, GeoSpark [11] a Spark-based system for spatial data, and Simba [12] an in-memory spatial analytics framework built in Spark.

Sphinx belongs to the category of *built-in* approach as it extends the core of Impala with spatial data types, functions, indexes, and operations. Sphinx distinguishes itself from other *built-in* systems as: (1) It adopts a standard SQL interface which makes it easy for existing DBMS users to adopt. (2) It is the only system that constructs spatial indexes inside Impala, which provides up to an order of magnitude speedup compared to other SQL-based systems. (3) It is the only system that uses the runtime code generation feature in Impala to produce an optimized machine code running natively on worker nodes. (4) It executes queries in real-time, where the answer is returned within seconds, by avoiding the huge overhead of the Hadoop runtime environment.

10 Conclusion

In this paper, we introduced Sphinx, the first and only system that extends the core of Impala to provide real-time SQL query processing on big spatial data. Sphinx reuses the spatial HDFS indexing introduced by earlier systems like SpatialHadoop. However, it introduces a completely renovated query processing engine based on the efficient design of Impala. Sphinx introduces primitive spatial data types and functions in the *query parser*. In the *storage/indexing* layer, it either links to SpatialHadoop indexes or construct its own spatial indexes. In the *query planner* layer, Sphinx introduces two new query plans for range query and three new query plans for spatial join. In the *query executor* layer, Sphinx adds two new components for *range query* and *spatial join* which are written in C++. Finally, the comprehensive experimental evaluation showed that Sphinx is much faster than plain-vanilla Impala, SpatialHadoop, and PostGIS in all queries.

References

1. Markram, H.: The Blue Brain Project. Nature Reviews Neuroscience **7**(2) (2006)
2. Pickle, L., et al.: The Crossroads of GIS and Health Information: A Workshop on Developing a Research Agenda to Improve Cancer Control. International Journal of Health Geographics (2006)
3. Auchincloss, A., et al.: A Review of Spatial Methods in Epidemiology: 2000-2010. Annual Review of Public Health **33** (2012)
4. Faghmous, J., Kumar, V.: Spatio-Temporal Data Mining for Climate Data: Advances, Challenges, and Opportunities. Advances in Data Mining, Springer (2013)
5. Sankaranarayanan, J., Samet, H., Teitler, B.E., Sperling, M.D.L.J.: TwitterStand: News in Tweets. In: SIGSPATIAL. (2009)
6. Aji, A., et al.: Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. In: VLDB. (2013)
7. Eldawy, A., Mokbel, M.F.: SpatialHadoop: A MapReduce Framework for Spatial Data. In: ICDE. (2015)
8. Nishimura, S., et al.: $\mathcal{MD}$-HBase: Design and Implementation of an Elastic Data Infrastructure for Cloud-scale Location Services. DAPD **31**(2) (2013) 289–319
9. Nidzwetzki, J.K., et al.: Distributed SECONDO: A Highly Available and Scalable System for Spatial Data Processing. In: SSTD. (2015)
10. Fox, A., et al.: Spatio-temporal Indexing in Non-relational Distributed Databases. In: International Conference on Big Data. (2013)
11. Yu, J., et al.: A Demonstration of GeoSpark: A Cluster Computing Framework for Processing Big Spatial Data. In: ICDE. (2016)
12. Xie, D., et al.: Simba: Efficient In-Memory Spatial Analytics. In: SIGMOD, San Francisco, CA (Jun. 2016)
13. Whitman, R.T., et al.: Spatial Indexing and Analytics on Hadoop. In: SIGSPATIAL. (2014)
14. Eldawy, A., et al.: Sphinx: Distributed Execution of Interactive SQL Queries on Big Spatial Data (Poster). In: SIGSPATIAL. (2015)
15. Kornacker, M., et al.: Impala: A Modern, Open-Source SQL Engine for Hadoop. In: CIDR. (2015)

16. Wanderman-Milne, S., Li, N.: Runtime Code Generation in Cloudera Impala. IEEE Data Engineering Bulletin **37**(1) (2014)
17. Floratou, A., et al.: SQL-on-Hadoop: Full Circle Back to Shared-Nothing Database Architectures. PVLDB (2014)
18. Thusoo, A., et al.: Hive: A Warehousing Solution over a Map-Reduce Framework. PVLDB (2009)
19. Armbrust, M., et al.: Spark SQL: Relational Data Processing in Spark. In: SIGMOD. (2015)
20. Schnitzer, B., Leutenegger, S.T.: Master-client r-trees: A new parallel r-tree architecture. In: SSDBM. (1999)
21. DeWitt, D., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems. CACM (1992)
22. Eldawy, A., Alarabi, L., Mokbel, M.F.: Spatial Partitioning Techniques in SpatialHadoop. In: PVLDB. (2015)
23. : Open Geospatial Consortium `http://www.opengeospatial.org/`.
24. Yu, J., et al.: GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data. In: SIGSPATIAL. (2015)
25. Leutenegger, S., et al.: STR: A Simple and Efficient Algorithm for R-Tree Packing. In: ICDE. (1997)
26. den Bercken, J.V., et al.: The Bulk Index Join: A Generic Approach to Processing Non-Equijoins. In: ICDE. (1999)
27. Patel, J., DeWitt, D.: Partition Based Spatial-Merge Join. In: SIGMOD. (1996) 259–270
28. Zhang, S., et al.: SJMR: Parallelizing spatial join with MapReduce on clusters. In: CLUSTER. (2009) 1–8
29. Dittrich, J.P., Seeger, B.: Data Redundancy and Duplicate Detection in Spatial Join Processing. In: ICDE. (2000)
30. Brinkhoff, T., Kriegel, H., Seeger, B.: Efficient Processing of Spatial Joins Using R-Trees. In: SIGMOD. (1993) 237–246
31. Arge, L., et al.: Scalable Sweeping-Based Spatial Join. In: VLDB. (1998)
32. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of ACM **51** (2008) 107–113
33. Olston, C., et al.: Pig Latin: A Not-so-foreign Language for Data Processing. In: SIGMOD. (2008)
34. Zaharia, M., et al.: Spark: Cluster Computing with Working Sets. In: HotCloud. (2010)
35. Cary, A., Sun, Z., Hristidis, V., Rishe, N.: Experiences on Processing Spatial Data with MapReduce. In: SSDBM. (2009)
36. Zhang, S., et al.: Spatial Queries Evaluation with MapReduce. In: GCC. (2009) 287–292
37. Ma, Q., Yang, B., Qian, W., Zhou, A.: Query Processing of Massive Trajectory Data Based on MapReduce. In: CLOUDDB. (2009)
38. Akdogan, A., et al.: Voronoi-based Geospatial Query Processing with MapReduce. In: CLOUDCOM. (2010)
39. You, S., et al.: Large-Scale Spatial Join Query Processing in Cloud. In: CLOUDDM. (2015)
40. Stonebraker, M., et al.: SciDB: A Database Management System for Applications with Complex Analytics. Computing in Science and Engineering **15**(3) (2013)
41. Wang, G., et al.: Behavioral Simulations in MapReduce. PVLDB (2010)
42. Lu, J., Guting, R.H.: Parallel Secondo: Boosting Database Engines with Hadoop. In: ICPADS. (2012)