Sphinx: Distributed Execution of Interactive SQL Queries on Big Spatial Data

Ahmed Eldawy[1*] Mostafa Elganainy[2$] Ammar Bakeer[3$]
Ahmed Abdelmotaleb[4$] Mohamed Mokbel[5*$]
[*]University of Minnesota [$]GIS Technology Innovation Center
{[1]eldawy,[5]mokbel}@cs.umn.edu {[2]melganainy,[3]abakeer,[4]aothman}@gistic.org

ABSTRACT

This paper presents Sphinx, a full-fledged distributed system which uses a standard SQL interface to process big spatial data. Sphinx adds spatial data types, indexes and query processing, inside the code-base of Cloudera Impala for efficient processing of spatial data. In particular, Sphinx is composed of four main components, 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 traditional Impala.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS

Keywords

Spatial, Impala, SQL, Range Query, Spatial Join, Sphinx

1. INTRODUCTION

The recent explosion in the amounts of spatial data generated by many applications, such as satellite images, GPS tracks, medical images, and geotagged tweets, urged researchers and developers to extend big data systems to efficiently support spatial data. This includes Hadoop-GIS [1], SpatialHadoop [4], and ESRI tools for Hadoop [12], among others. Unfortunately, all these systems suffer from the following two limitations. (1) Despite SQL-like languages, such as HiveQL, they lack an ANSI-standard SQL interface which is much preferred by existing DBMS users, and (2) they inherit the limitations of the underlying systems, such as significant startup time, and materializing intermediate data to disk, which impede these system from reaching the full potential of the underlying hardware.

In this paper, we introduce Sphinx; a full-fledged system for distributed execution of interactive SQL queries on Big Spatial Data. We have chosen to build Sphinx inside Impala [7] rather than any other open-source distributed big data system (e.g., Hadoop and Spark) as Impala has several advantages which include: (1) it adopts the ANSI-standard SQL interface, (2) employs query optimization, (3) C++ runtime code generation, and (4) low-level direct disk access. With these features, Impala achieves an order of magnitude speedup [5, 7, 11] on standard TPC-H and TPC-DS queries compared to other popular SQL-on-Hadoop systems such as Hive [10] and Spark-SQL.

```
SELECT   COUNT(*)
FROM     OSM_Points
WHERE    x ≥ x1 AND x < x2 AND
         y ≥ y1 AND y < y2;
```

(a) Range query in Impala

```
SELECT   COUNT(*)
FROM     OSM_Points
WHERE    Contains(Rectangle(x1, y1, x2, y2),
         OSM_Points.coords);
```

(b) Range query in Sphinx

Figure 1: Range query in Impala vs. Sphinx

Figure 1 shows a range query example that gives the essence of Sphinx. Figure 1(a) shows a range query expressed in Impala using primitive data types and operations, and it takes 21 minutes on a table of 2.7 Billion points with a cluster of 10 cores. As Impala does not understand the properties of this spaital query, it has to perform a full-table scan with a very little room of optimization to do. In Sphinx, the same query is expressed as shown in Figure 1(b). In addition to the expressive language, this query runs in one second on Sphinx, giving three orders of magnitude speedup over plain-vanilla Impala. The main reason behind this performance boost is the spatial indexes that we add in Sphinx and the spatial query processing that is injected in the core of the query planner and executor.

Figure 2 gives an overview of Sphinx which consists of four components, all implemented inside the core of Impala. (1) The *query parser* (Section 2) enriches the SQL interface with spatial data types (e.g., Point and Polygon), spatial functions (e.g., Overlap and Touch), and new commands to construct and import spatial indexes. (2) The *indexer* (Section 3) constructs spatial indexes based

*This work is supported in part by the GIS Technology Innovation Center under project GISTIC-13-05

SIGSPATIAL'15, November 03-06, 2015, Bellevue, WA, USA
©2015 ACM. ISBN 978-1-4503-3967-4/15/11$15.00
DOI: http://dx.doi.org/10.1145/2820783.2820869.

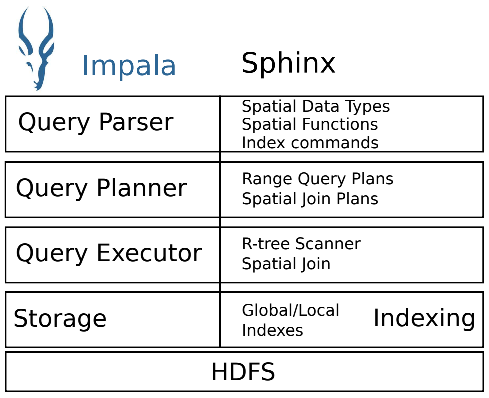


Figure 2: Overview of Sphinx

on grid, R-tree or Quad-tree, and organized in two layers as one global index and multiple local index. (3) The *query planner* (Section 4) utilizes the spatial indexes to introduce new efficient query plans for the *range query* and spatial join operations. (4) The *query executor* (Section 5) introduces the *R-tree scanner* and *spatial join* operators, which use C++ runtime code generation to efficiently execute *spatial range* and *spatial join* queries, respectively. We conduct an experimental evaluation on the proposed system prototype using a publicly available real dataset of 2.7 Billion points extracted from OpenStreetMap. We show that Sphinx outperforms traditional Impala by up-to three orders of magnitude with both range query and spatial join queries. In addition, we show that Sphinx scales well with both the input size and the cluster size.

2. QUERY PARSER

To make it user-friendly and easy to use, Sphinx extends the *query parser* of Impala to introduce spatial data types, functions, and new commands to construct and import spatial indexes.

Spatial Data Types. Sphinx adds the `Geometry` datatype as an abstraction for all standard spatial datatypes, such as `Point`, `Linestring` and `Polygon`, as defined by the Open Geospatial Consortium (OGC). We adopt the standard Well-Known Text (WKT) format to be able to import text files from other systems such as PostGIS and Oracle Spatial.

Spatial Functions Sphinx adds OGC-compliant spatial functions which are implemented as either user-defined functions (UDF) or 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. These functions include *basic functions*, e.g., `MakePoint`, *spatial predicates*, e.g., `Touch`, *spatial analysis* functions, e.g., `Union`, and *spatial aggregate* functions, e.g., `Envelope` which computes the minimum-bounding rectangle (MBR) of a set of objects.

Spatial Indexing Sphinx also adds new commands to constructs spatial indexes, and import existing indexes from SpatialHadoop.

```
/* Create R-tree index on the coords attribute */
CREATE INDEX PointsIndex
  ON Points USING RTREE (coords);
/* Import an index built on the coords attribute */
CREATE EXTERNAL TABLE OSM_Points
  (... /* Schema definition */)
  INDEXED ON coords AS RTREE
  LOCATION('/osm_points');
```

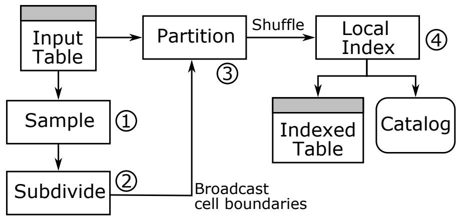


Figure 3: Indexing plan in Sphinx

3. SPATIAL INDEXER

In this section, we describe how Sphinx constructs spatial indexes on HDFS-resident tables. The main goal is to store the records in a spatial-aware way by grouping nearby records and storing them physically together in the same HDFS block. While traditional Impala already provides a *partitioned table* feature, where a table is hierarchically partitioned based on a sequence of columns, Sphinx employs spatial indexes which overcome the following three limitations in partitioned table: (1) While Impala assigns one value per partition, e.g., dpartment ID, Sphinx assigns a region, i.e., a rectangle, to each partition which is more suitable to spatial data. (2) While Impala assigns each record to exactly one partition, Sphinx can replicate a record to multiple partitions which can be used to index polygons which span multiple partitions. (3) Impala puts the burden of choosing partition boundaries on the user, which is not suitable for skewed spatial data. Sphinx provides a one-statement index command which takes care of defining partition boundaries based on the data distribution. In the rest of this section, we first describe how the index is stored in Sphinx, and then we explain how Sphinx builds this spatial index efficiently.

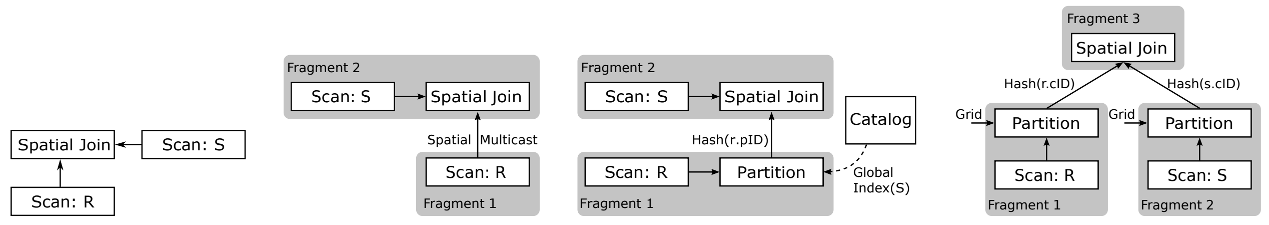
3.1 Index Layout

Sphinx employs a two-layered design for spatial indexes in HDFS [1,4,12], where the *global index* is stored in the master node and defines how records are partitioned across machines, while *local indexes* are stored inside slave nodes and define how records are internally organized inside that node. This design well fits with the architecture of Impala and Sphinx where the global index is stored in the *catalog server* on the master and the local indexes are stored in HDFS data nodes, and are processed by the `impalad` processes running on the slaves. This also allows Sphinx to easily import an index which was built in SpatialHadoop by simply importing the global index into the catalog server.

3.2 Index Construction in Sphinx

Sphinx provides an efficient algorithm for constructing a spatial index on a user-selected attribute in a table. We focus on the construction of R-tree and R+-tree as examples of *non-replicated* and *replicated* indexes, respectively. Figure 3 illustrates the index construction plan in Sphinx. When the user issues a `CREATE INDEX` statement, Sphinx creates this query plan which is executed in parallel using its query execution engine. The indexing algorithm consists of four phases, namely, *sampling*, *subdivision*, *partitioning*, and *local indexing*, described below.

The Sampling Phase. The job of this phase is to summarize the data so that it fits on a single machine while preserving its distribution, to some level. This summary will be used in the next step to decide how to partition the space across machines while balancing the load. To summarize the data, this phase scans the input, in parallel, and reads a sample of 1% of the records. Each record is converted to a point by taking the centroid of the index key. Finally, all sample points are grouped in one machine for the next phase.



(a) Logical Plan (b) Overlap Join Physical Plan (c) Partition Join Physical Plan (d) Co-partition Join Physical Plan

Figure 4: Spatial Join Query Plans

The Subdivision Phase. This phase runs on a single machine, and it subdivides the space into n cells, which will be used to partition the input records as one partition per cell. The objective is to balance the load across partitions while fitting each partition in a single HDFS block, typically 128MB. Sphinx adjusts number of cells n to number of HDFS blocks in the indexed dataset which ensures that the average partition size is equal to HDFS block capacity. Then, it subdivides the space into n cells, each containing roughly the same number of sample points, by bulk loading the sample into an R-tree using the sort-tile-recursive (STR) algorithm [8]. The leaf nodes of the STR-tree are used as cell boundaries, which are broadcast to all nodes to be used in the next phase.

The Partitioning Phase. This phase scans the input table and assigns each record to overlapping cell(s). The main challenge is to handle boundary objects which overlap more than one cell. Sphinx employs either a *distribution* or *replication* strategy, for R-tree and R+-tree, respectively. The *distribution* strategy assigns a record to exactly one cell and expands the cell boundaries to fully contain the record. The *replication* strategy replicates a record to all overlapping cells, and the query executor will have to handle the replication to ensure a correct answer as described in Section 5. This step is implemented as a *broadcast join* where the smaller table (*cell boundaries*) is replicated to all machines and the join predicate uses either the *distribution* or *replication* strategies.

The Local Indexing Phase. In the final phase, records are shuffled across machines and grouped by the *CellID* column. The contents of each cell are processed separately where they are bulk loaded into an in-memory local index, e.g., R-tree, and the index is written to HDFS as one file. Since the size of each partition is expected to be within the HDFS block capacity, this step can handle arbitrarily large files. If one partition goes beyond the HDFS block capacity, it is split into chunks, each fits in one HDFS block.

4. QUERY PLANNER

The *query planner* in Sphinx is responsible on generating a *query plan* for a user query, which is later executed by the *query executor*. In general, the *query planner* first generates a single-machine *logical plan*, which is never executed. Then, it translates it into a distributed *physical plan* which is executed in parallel. Sphinx introduces new query plans for both the *range query* and *spatial join* operations, as described below.

4.1 Range Query Plans

In range query, the input is a query range A and a spatial attribute x in a table R, while the output is all records $r \in R$ where $r.x$ overlaps A. Traditional Impala supports only one plan for range query which employs a *full table scan* and compares each record to the query area. If the input table R is indexed on the search column x, Sphinx utilizes the spatial index to build a more efficient *R-tree search* plan. This plan improves over the *full scan* plan by two new features. (1) The *early pruning* feature which utilizes the *global index* to prune partitions that are outside the query area. (2) It uses *R-tree scanners* which utilize the *local indexes* in selected partitions to quickly select matching records. The details of the R-tree scanner will be described in Section 5.

4.2 Spatial Join Plans

In spatial join, the input is composed of two tables, R and S, with designated geometric columns, $R.x$ and $S.y$, and a spatial predicate θ, such as *touch* or *overlap*. The output is a table T that contains all records $t = \langle r, s\rangle$, where the predicate θ is true for $\langle r.x, s.y\rangle$, $r \in R$, and $s \in S$. Figure 4(a) shows the logical query plan of spatial join. Traditional Impala can translate this plan into one physical plan that uses the naive spatial join algorithm which computes the cross join $R \times S$, followed by a spatial filter on the predicate θ. Sphinx improves on this approach by introducing three alternative physical plans based on whether the two tables are indexed, one table is indexed, or none of them are indexed, all described below.

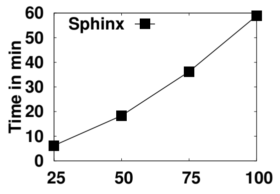
(1) **Overlap Join:** This plan, shown in Figure 4(b), is used if the two input tables are indexed on the join columns. The basic idea is to find pairs of overlapping partitions and perform a single-machine spatial join between every pair of partitions. To realize this plan, Sphinx introduces the novel *spatial multicast* connection patterns which creates a communication stream between every pair of machines which are assigned overlapping partitions. To join a pair of partitions, Sphinx uses the *spatial join* operator which will be described in Section 5.

(2) **Partition Join:** This plan, shown in Figure 4(c), is employed if only one input table is indexed. In this case, the non-indexed table, say R, is partitioned to match the other (indexed) table. Once the table R is partitioned, there will be a one-to-one correspondence between the partitions of R and S where each pair of partitions are joined using the *spatial join* operator.

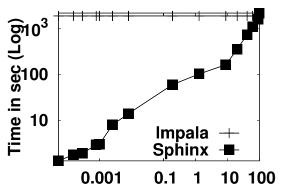
(3) **Co-partition Join:** If none of the input files are indexed, *sphinx* employs the *co-partition join* which is a port of the traditional *partition-based spatial-merge* (PBSM) join algorithm [9]. In this plan, shown in Figure 4(d), both input files are partitioned using a common uniform grid, and the contents of each grid cell from the two files are spatially joined.

5. QUERY EXECUTOR

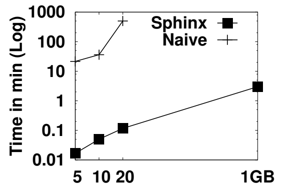
The *query executor* is the component that executes the physical query plans, created by the *query planner*, in the distributed environment. Sphinx introduces two new components in the query executor, namely, *R-tree scanner* for range queries, and *spatial join* operator. These components are completely written in C++ and make use of the Impala runtime code generation [11], which gives a higher performance compared to other big data systems written in Java.



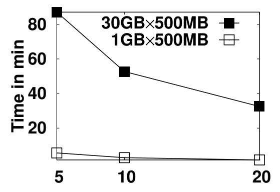
(a) Indexing



(b) Range Query



(c) Spatial Join (Input Size)



(d) Spatial Join (Cluster Size)

Figure 5: Experimental results of Impala and Sphinx

5.1 R-tree Scanner

The *R-tree scanner* takes as input one locally-indexed partition P and a rectangular query range A, and returns all records in P that overlap A. The R-tree scanner starts by computing the estimated selectivity σ using the equation $\sigma = Area(A \cap P)/Area(P)$, where A and P are the MBRs of the query range and processed partition, respectively. The MBR of P is available as part of the global index which is stored in the main memory of the master node. Based on the selectivity, the R-tree scanner has three modes of operation.

(1) **Match All ($\sigma = 1.0$):** If the P is completely contained in A, all records are added to the answer without testing them against the query A. (2) **Full Scan ($\delta < \sigma < 1.0$):** If the selectivity is larger than a threshold δ, the R-tree is known to impose a significant overhead on the search query. Thus, the R-tree scanner skips the index and compares each record against the query range A. (3) **R-tree search ($\sigma \leq \delta$):** If the selectivity is lower than δ, the R-tree index is utilized to quickly retrieve matching records.

If the index is replicated, e.g., R+-tree, a final *duplicate avoidance* step is carried out to remove duplicate answers.

5.2 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. This operator also has three modes of execution based on the local indexes in the two joined partitions.

(1) **R-tree Join:** If both partitions are locally indexed using R-tree, this execution mode the synchronized traversal algorithm [2,6] to concurrently traverse both trees while pruning disjoint tree nodes. (2) **Bulk Index Join:** If only one partition is locally indexed, this execution mode uses the *bulk index join* algorithm [3,6] which partitions the non-indexed partition according to the R-tree of the indexed partition, and then joins each pair of corresponding partitions. (3) **Plane-sweep Join:** If none of the partitions are indexed, the spatial join operator performs a plane-sweep join algorithm [6] which works efficiently with non-indexed data.

Similar to the R-tree scanner, the spatial join operator applies a *duplicate avoidance* step if the input partitions are indexed using a replicated index.

6. EXPERIMENTS

Sphinx is implemented inside Impala 1.2.1 and deployed on an Amazon EC2 cluster of 20 single-core nodes. Figure 5(a) shows a nice linear scale up of the index construction time as the input table increases from 25GB to 100GB. Figure 5(b) shows three orders of magnitude speedup of Sphinx over Impala when running a range query. Even with high selectivity ratios, Sphinx gives almost the same performance as Impala which indicates a low index overhead.

Figure 5(c) compares the performance of the spatial join query in Impala and Sphinx, where the two input tables are of the same size. In this experiments, we only use the *overlap join* algorithm in Sphinx which is executed when the two input files are indexed. As shown in the figure, the naive algorithm in Impala is not scalable at all as it quickly fails, even for small input sizes. Figure 5(d) shows the performance of the spatial join in Sphinx with much larger files while increasing the cluster size from 5 to 20 nodes. This experiments shows the great performance and scalability of the spatial join operation in Sphinx.

7. CONCLUSION

In this paper, we introduced Sphinx, the first and only system that extends the core of Impala to provide real-time query processing of SQL queries on spatial data. Sphinx modifies the *query planner* by injecting standard spatial data types, spatial functions as well as a new command to construct spatial indexes. It adopts a two-layered approach to build a spatial index which consists of one global index, stored in a *catalog*, which partitions records across machines, and multiple local indexes, stored in HDFS blocks, that organize records contained in each partition. We also extend the *query planner* by building efficient query plans for range and spatial join queries. These plans utilize the global index to prune unwanted partitions from the input. We also implemented two new components in the *query executor*, namely, R-tree scanner and spatial join operator, which use runtime code generation to generate optimized machine code that runs natively on the system. Finally, we provided an experimental study on large scale real data that show the efficiency and scalability of Sphinx as it compares to Impala.

8. REFERENCES

[1] A. Aji and *et al*. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. In *VLDB*, 2013.

[2] T. Brinkhoff. *et al* Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD*, pages 237–246, 1993.

[3] J. V. den Bercken. *et al* The Bulk Index Join: A Generic Approach to Processing Non-Equijoins. In *ICDE*, page 257, 1999.

[4] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce Framework for Spatial Data. In *ICDE*, 2015.

[5] A. Floratou. *et al* SQL-on-Hadoop: Full Circle Back to Shared-Nothing Database Architectures. *PVLDB*, 7(12), 2014.

[6] E. H. Jacox and H. Samet. Spatial join techniques. *TODS*, 32(1):7, 2007.

[7] M. Kornacker and *et al*. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

[8] S. Leutenegger, M. Lopez, and J. Edgington. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*, 1997.

[9] J. Patel and D. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD*, 1996.

[10] A. Thusoo. *et al* Hive: A Warehousing Solution over a Map-Reduce Framework. *PVLDB*, 2009.

[11] S. Wanderman-Milne. *et al* Runtime Code Generation in Cloudera Impala. *IEEE Data Engineering Bulletin*, 37(1):31–37, 2014.

[12] R. T. Whitman, M. B. Park, S. A. Ambrose, and E. G. Hoel. Spatial Indexing and Analytics on Hadoop. In *SIGSPATIAL*, 2014.