# A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data[*]

Ahmed Eldawy
Department of Computer Science and Engineering
University of Minnesota
eldawy@cs.umn.edu

Mohamed F. Mokbel
Department of Computer Science and Engineering
University of Minnesota
mokbel@cs.umn.edu

## ABSTRACT

This demo presents SpatialHadoop as the first full-fledged MapReduce framework with native support for spatial data. SpatialHadoop is a comprehensive extension to Hadoop that pushes spatial data inside the core functionality of Hadoop. SpatialHadoop runs existing Hadoop programs as is, yet, it achieves order(s) of magnitude better performance than Hadoop when dealing with spatial data. SpatialHadoop employs a simple spatial high level language, a two-level spatial index structure, basic spatial components built inside the MapReduce layer, and three basic spatial operations: range queries, $k$-NN queries, and spatial join. Other spatial operations can be similarly deployed in SpatialHadoop. We demonstrate a real system prototype of SpatialHadoop running on an Amazon EC2 cluster against two sets of real spatial data obtained from Tiger Files and OpenStreetMap with sizes 60GB and 300GB, respectively.

## 1. INTRODUCTION

MapReduce-like systems, e.g., Hadoop, have been around for years and was proven to be an efficient framework for big data analysis for many applications, e.g., machine learning [3], tera-byte sorting [9], and graph processing [1]. In the mean time, there is a recent explosion of spatial datasets generated by different sources, e.g., smart phones, medical devices, and space telescopes. Unfortunately, Hadoop is ill-equipped for supporting spatial data as its core framework is unaware of spatial data properties. Existing attempts to process spatial data on Hadoop focus mainly on specific data types and operations, e.g., range queries on trajectories [6] and kNN join on points [5, 13]. Also, the efficiency of such operations is limited as the internal Hadoop system is unaware of spatial data.

In this demo, we present SpatialHadoop as the first full-fledged MapReduce framework with native support for spatial data; available as open-source at *http://spatialhadoop.cs.umn.edu/*. SpatialHadoop is a comprehensive extension to Hadoop (around 12,000 lines of code inside Hadoop) that pushes spatial constructs and the awareness of spatial data inside Hadoop code base. As a result, SpatialHadoop works in a similar way to Hadoop where programs are written in terms of *map* and *reduce* functions, and hence existing Hadoop programs can run as is on SpatialHadoop. Yet, if the programs deal with spatial data, SpatialHadoop will have order(s) of magnitude better performance than Hadoop. For example, Figures 1(a) and 1(b) show how to express a spatial range query in Hadoop and SpatialHadoop, respectively. For 70M spatial objects on a 20-nodes cluster, Hadoop executes this query in 200 seconds, while SpatialHadoop executes the same query in 2 seconds.

```
Objects  =  LOAD 'points' AS (id:int, x:int, y:int);
Result   =  FILTER Objects BY   x < x2 AND x > x1
                                AND y < y2 AND y > y1;
```
(a) Range query in Hadoop

```
Objects  =  LOAD 'points' AS (id:int, Location:POINT);
Result   =  FILTER Objects BY
              Overlaps (Location, Rectangle(x1, y1, x2, y2));
```
(b) Range query in SpatialHadoop

**Figure 1: Range query in Hadoop vs. SpatialHadoop**

SpatialHadoop pushes its spatial constructs in all layers of Hadoop, namely, *language*, *storage*, *MapReduce* and *operations* layers. In the language layer, a simple high level language is provided to simplify spatial data analysis for non-technical users. In the storage layer, a two-layered spatial index structure is provided where the *global* index partitions data across nodes while the *local* index organizes data in each node. This structure is used to build a grid index [7], an R-tree [4] or an R+-tree [11]. In the MapReduce layer, two new components are added to allow MapReduce programs to access indexed files as input, namely, SpatialFileSplitter and SpatialRecordReader. The SpatialFileSplitter exploits the global index by pruning partitions that do not contribute to the query answer, while the SpatialRecordReader exploits the local index to efficiently access records within each partition. The operations layer contains a number of spatial operations (range query, kNN and spatial join), implemented using the indexes and new components in the MapReduce layer. Other spatial operations can be added in a similar way.

SpatialHadoop is distributed as an open source, which allows contributors in the research community to further extend its functionality. The basic components shipped in the core of SpatialHadoop help in implementing more spatial operations in different applications efficiently. As case studies, SpatialHadoop already has three spatial operations, range queries, $k$-nearest-neighbor queries, and spatial join. We envision that SpatialHadoop will act as a research vehicle where various researchers will contribute their spa-

[*]This work is supported in part by the National Science Foundation under Grants IIS-0952977 and IIS-1218168

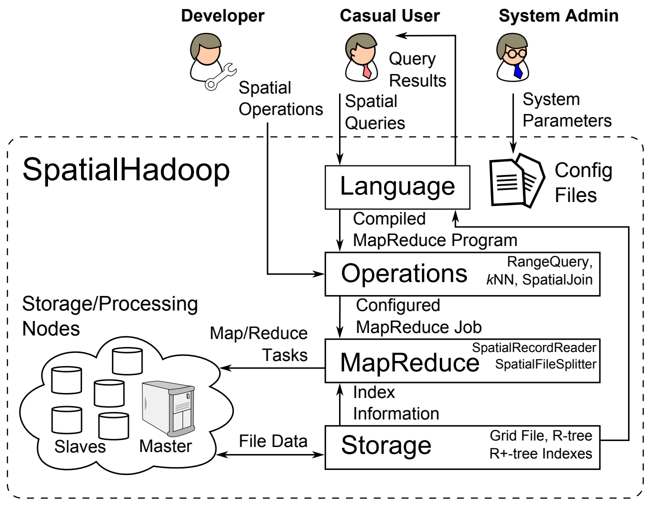


**Figure 2: SpatialHadoop architecture**

tial operations and analysis techniques inside, providing a rich system to be widely used by developers, practitioners, and researchers.

We will demonstrate SpatialHadoop with its real system prototype running on an Amazon EC2 cluster against two sets of real spatial data obtained from Tiger Files [12] and OpenStreetMap [10]. Tiger files include 70 Million spatial objects (size of 60GB) of road segments, water features, and other geographic information in USA. OpenStreetMap includes map information from the whole world including road segments, points of interest, and buildings boundaries with a total size of 300GB.

## 2. SpatialHadoop ARCHITECTURE

Figure 2 depicts the system architecture of SpatialHadoop. A SpatialHadoop cluster contains one master node that accepts a user query, breaks it into smaller tasks, and carries out the tasks on multiple slave nodes. There are three types of users who interact with SpatialHadoop, *casual users*, *developers* and *administrators*. Casual users are non-technical users who access SpatialHadoop through the provided language to process their datasets. Developers are more advanced users who have deeper understanding of the system and can implement new spatial operations, which could be specific to some applications. Administrators are able to tune up the system through adjusting system parameters in the configuration files provided with SpatialHadoop installation.

SpatialHadoop adopts a layered design of four main layers, namely, *language*, *storage*, *MapReduce*, and *operations* layers. The *language* layer provides a simple high level SQL-like language that supports spatial data types and operations. The *storage* layer employs a two-level index structure of *global* and *local* spatial index structures. The *global* index partitions data across computation nodes while the *local* index organizes data inside each node. The *MapReduce* layer has two new components, namely, *SpatialFileSplitter* and *SpatialRecordReader* that exploits the global and local indexes, respectively, to prune data that do not contribute to the query answer. The *operations* layer encapsulates the implementation of various spatial operations that take advantage of the spatial indexes and the new components in the MapReduce layer. SpatialHadoop is initially equipped with an efficient implementation of three basic spatial operations, namely, range query, *k*NN, and spatial join. Other spatial operations can be added to the operations layer using a similar approach of the implementation of basic spatial operations.

## 3. LANGUAGE LAYER

SpatialHadoop provides a simple high level language that simplifies the interaction with the system for non-technical users. This language provides a built-in support for spatial data types, spatial primitive functions, and spatial operations. Spatial data types (`Point`, `Rectangle`, and `Polygon`) are used to define the schema of an input file upon its loading process. The spatial primitive functions `Distance`, `Overlaps`, and `MBR` are applied to spatial attributes to calculate the distance between the centroid of two shapes, find whether two shapes overlap or not, and compute the minimal bounding rectangle of a polygon, respectively. The spatial operations range query, $k$-nearest neighbor, and spatial join are applied to input files with spatial attributes and produce the results in another output file.

Rather than creating a new spatial language from scratch, SpatialHadoop extends Pig Latin [8], a high level language for Hadoop by adding new spatial constructs while preserving the original functionality. In particular, SpatialHadoop language overrides the keywords `FILTER` and `JOIN`, when their parameters have spatial predicate(s), to perform range query and spatial join, respectively. For example, when the `FILTER` keyword is used with the `Overlaps` predicate, SpatialHadoop reroutes its processing to the range query operation. For $k$ nearest neighbor queries, a new keyword `KNN` is introduced. Following is an example that calculates the 100 nearest houses to the query point `query_loc`.

```
houses = LOAD 'houses' AS (id:int, loc:point);
nearest_houses = KNN houses WITH_K=100
  USING Distance(loc, query_loc);
```

## 4. STORAGE LAYER

In the storage layer, SpatialHadoop adds new spatial indexes that are well adapted for the MapReduce environment. These indexes overcome a limitation in Hadoop, which supports only non-indexed heap files. There are two challenges that prevent traditional spatial indexes to be used as-is in Hadoop. First, traditional indexes are designed for the procedural programming paradigm while SpatialHadoop uses the MapReduce programming paradigm. Second, traditional indexes are designed for local file systems while SpatialHadoop uses the Hadoop Distributed File System (HDFS), which is inherently limited as files can be written in an append only manner, and once written, they cannot be modified. To overcome these challenges, SpatialHadoop organizes its index in two levels, *global* and *local* indexing. The *global* index partitions data across nodes in the cluster while the *local* index organizes data efficiently within each node. The separation of global and local indexes lends itself to the MapReduce programming paradigm where the global index is used for preparing the MapReduce job while the local indexes are used for processing map tasks. Breaking the file into smaller partitions allows indexing each partition separately in memory and writing it to a file in a sequential manner.

The global index is kept in the main memory of the master node while each local index is stored as one file block (typically 64MB) in a slave node. SpatialHadoop supports grid file [7], R-tree [4] and R+-tree [11] indexes. An index is constructed for an existing file by issuing the new file system command `writeSpatialFile` introduced in SpatialHadoop, where the user specifies the input file, column to index, and index type to construct.

An index is constructed in SpatialHadoop through a MapReduce job that runs in three phases, namely, *partitioning*, *local indexing*, and *global indexing*. In the partitioning phase, a file is spatially partitioned such that each partition is contained in a rectangle while its contents fits in one file block (64MB). A grid index partitions the space using a uniform grid while R-tree and R+-tree use a distribution-aware R-tree partitioning which reads a random sample from the input file, bulk loads this sample into a temporary in-memory R-tree, and uses the boundaries of the leaf nodes of that R-tree for partitioning the whole file. Notice that in grid and R+-tree, some records may be replicated if they overlap multiple partitions while in R-tree each record is written to the best fitting partition [4]. Replicated records are handled later in the query processing to avoid producing duplicate results. In the local indexing phase and according to the type of index being constructed, a local index is created for each partition separately and flushed to a file with one HDFS block which is annotated by the MBR of the partition. Since each partition has a fixed size (64MB), the local index is constructed in memory before it is written to disk in one shot. The third and final phase is global indexing where the files containing local indexes are concatenated into one big file and a global index is constructed to index all partitions using their MBRs as keys and stored in main memory of the master node. In case of system failure, the global index is reconstructed from block MBRs only when needed.

## 5. MAPREDUCE LAYER

The MapReduce layer in traditional Hadoop is designed to work with non-indexed heap files as input. However, spatial operations in SpatialHadoop take spatially indexed files as input, which requires different handling. Moreover, some spatial operations, e.g., spatial join, are binary operations that take two input files as input. To be able to handle spatially indexed files, SpatialHadoop introduces two new components in the MapReduce layer, namely, *SpatialFileSplitter* and *SpatialRecordReader*, that exploits the global and local indexes, respectively, for efficient data access.

The *SpatialFileSplitter* takes as input one or two spatially indexed files in addition to a user provided filter function. Then, it uses the global index to prune file blocks that do not contribute to the query answer (e.g., outside query range), based on their minimal bounding rectangles which were assigned when the index was created. In the case of binary operations where there are two input files, the *SpatialFileSplitter* uses two global indexes, as one per file, to select pairs of file blocks that need to be processed together (e.g., overlapping blocks in spatial join). The *SpatialRecordReader* utilizes the local index by allowing records in one block to be accessed through the local index instead of iterating over all records one-by-one. It reads the local index from the assigned block and passes a pointer to this index to the map function which utilizes the index to select the processed records without the need to iterate over all records. Together, *SpatialFileSplitter* and *SpatialRecordReader* help developers writing many spatial operations as MapReduce programs.

## 6. OPERATIONS LAYER

The spatial indexes introduced in the storage layer along with the new components in the MapReduce layer give the possibility of realizing many spatial operations efficiently in SpatialHadoop. In this demonstration, we show the implementations of range query, kNN, and spatial join as three case studies of how to exploit the new storage and MapReduce layers in SpatialHadoop. Other spatial operations such as kNN join and shortest path can be also implemented following a similar approach.

In range queries, the *SpatialFileSplitter* uses the global index to select only the partitions that overlap the query range. Each of the selected partitions goes through a *SpatialRecordReader* which extracts the local index in that partition and executes a traditional range query on that index to find matching records. As some records are replicated during indexing, the reference point duplicate avoidance technique [2] is employed on the matching records to ensure that each answer record is reported exactly once.

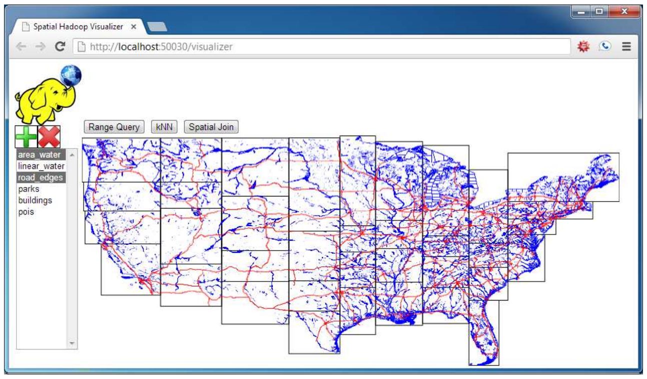


**Figure 3: SpatialHadoop front end**

The $k$-nearest-neighbor operation is carried out in two iterations. In the first iteration, the *SpatialFileSplitter* uses the global index to select the partition that contains the query point. The local index in that partition is extracted by the *SpatialRecordReader* and used to find the $k$NN in that partition. To test whether the answer is correct or not, a test circle is drawn with the query point as the center and the distance to the $k^{th}$ neighbor as radius. If the test circle fits completely in the processed partition, the answer is correct. If it overlaps with other partitions, a second iteration is carried out to process those overlapping partitions.

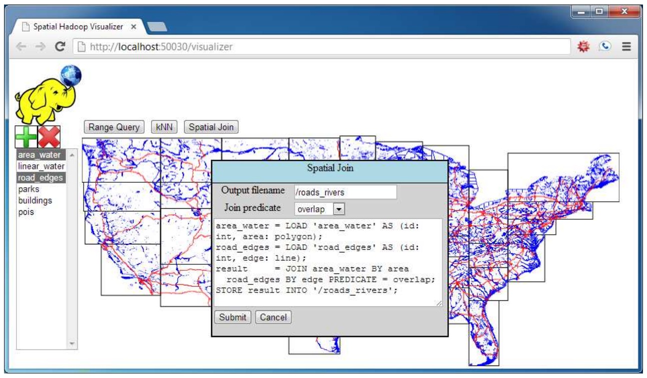
For spatial join, the *SpatialFileSplitter* uses the two global indexes in both files to find all pairs of overlapping partitions. Each pair of overlapping partitions is processed by a *SpatialRecordReader*, which uses the local indexes in both files to find overlapping records.

## 7. DEMONSTRATION SCENARIO

We deploy the real system prototype of SpatialHadoop (downloadable at *http://spatialhadoop.cs.umn.edu/*) on an Amazon EC2 cluster of 20 nodes. The cluster is loaded with two real datasets obtained from Tiger files [12] and OpenStreetMap [10]. For Tiger files, we extract three files that represent road segments, rivers, and lakes in the US. From OpenStreetMap, we extract files that represent road segments, points of interest, parks, and buildings boundaries for the whole world. The attendee can access the Amazon EC2 cluster through a front end machine (i.e., a laptop) while all the processing is done by the cluster in the backend.

### 7.1 Front End

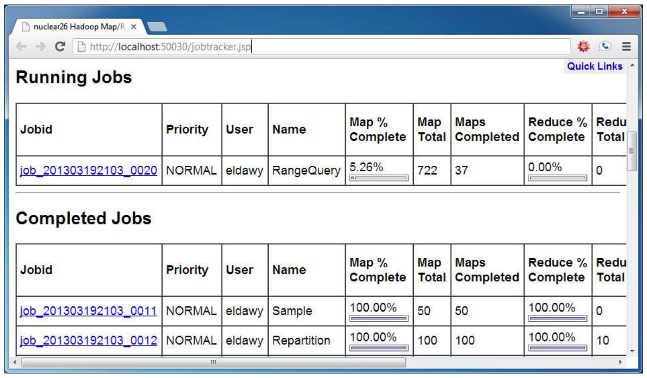
Figure 3 depicts the system front end, which is a querying and visualization tool designed to help users and administrators to interact with SpatialHadoop. On the left, there is a selection pane that lists all files loaded in the system. Users can upload new files using the add button, or remove existing files using the delete button. When a file is selected, the contents of this file are visualized on the screen as shown in the figure. When more than one file is selected, they are all visualized with different colors to help contrasting two or more files. In the figure, blue and red lines present water areas (rivers and lakes) and primary roads in US, respectively. The user can then use the toolbar on top to perform a query (range query, kNN or spatial join) with the selected file(s). The front end displays the progress of the query as it is running and once finished, the results can be visualized.



Output filename: /roads_rivers
Join predicate: overlap

```
area_water = LOAD 'area_water' AS (id:
int, area: polygon);
road_edges = LOAD 'road_edges' AS (id:
int, edge: line);
result     = JOIN area_water BY area
  road_edges BY edge PREDICATE = overlap;
STORE result INTO '/roads_rivers';
```

Figure 4: Joining roads with rivers



Running Jobs

| Jobid | Priority | User | Name | Map % Complete | Map Total | Maps Completed | Reduce % Complete | Redu Total |
|---|---|---|---|---|---|---|---|---|
| job_201303192103_0020 | NORMAL | eldawy | RangeQuery | 5.26% | 722 | 37 | 0.00% | 0 |

Completed Jobs

| Jobid | Priority | User | Name | Map % Complete | Map Total | Maps Completed | Reduce % Complete | Redu Total |
|---|---|---|---|---|---|---|---|---|
| job_201303192103_0011 | NORMAL | eldawy | Sample | 100.00% | 50 | 50 | 100.00% | 0 |
| job_201303192103_0012 | NORMAL | eldawy | Repartition | 100.00% | 100 | 100 | 100.00% | 10 |

Figure 5: Progress of running jobs

## 7.2 Operations

First, attendee select one of the files by clicking it to visualize its contents on the screen. The data is visualized by running a MapReduce job that generates an image out of the data in the selected file. The generated image only includes the spatial attributes of the file and draws the records according to their type (point, rectangle, or polygon). As depicted in Figure 3, the visualization can also display the boundaries of the global index used to index the displayed file. This allows the attendee to compare the grid index to R-tree and realize that the grid index is more suitable for uniformly distributed datasets while R-tree works better with skewed data. The boundaries given in the figure are clearly obtained from an R-tree as they are not uniform. Showing the index boundaries is optional and is used to only show the system internals.

Once a file is selected, an attendee can issue a query on that file by selecting an operation from the toolbar on the top. The available operations are range query, kNN and spatial join. Only the spatial join requires two files to be selected as it is a binary operation. As shown in Figure 4, when an operation is selected, a dialog box is displayed to the attendee to provide query parameters and output file name. For the range query, the attendee provides the query range as a rectangle defined by its two corner points. For the kNN, the parameters are the query point and number of neighbors ($k$). For spatial join, only the join predicate needs to be provided where the default is the overlap predicate. An example of an interesting query is to join parks with lakes to find all parks that have lakes in them and visualize results on the screen. As the parameters of the query are set, the front end displays the query written in the spatial language of SpatialHadoop as shown in Figure 4. Once a query is submitted to the system, the front end sends the query to the back end for processing. As shown in Figure 5, the attendee can see the progress of the query as it is running in the system. This administration interface lists the progress of all running jobs as well as all completed job. Subsequent queries can be submitted to the system and they will run concurrently in the back end. Once a query ends successfully, attendee can visualize the results on screen.

## 7.3 Comparison with Hadoop

To contrast SpatialHadoop with Hadoop, we also set up a separate cluster of 20 nodes running traditional Hadoop. An attendee can run the same query on both clusters (i.e., Hadoop and SpatialHadoop clusters) and watch the progress of both of them side-by-side. As the functionality of traditional Hadoop is preserved in SpatialHadoop, non-spatial queries can still run in SpatialHadoop without any overhead. This can be tested by the attendees by submitting a non-spatial query to both clusters and comparing the performance of both clusters.

## 7.4 Installation and Configuration

SpatialHadoop is open source and is publicly available on the web. During the demonstration, we will show a quick setup guide on how to install and run SpatialHadoop on a single machine in a few minutes. To install SpatialHadoop, the first step is to download the binaries as a compressed file and decompress it to the local disk. Then, the installation is configured by editing some configuration files. After that, the SpatialHadoop server is started and some example operations are executed to show the interaction with the server. These steps are available on the official web page of SpatialHadoop (*http://spatialhadoop.cs.umn.edu/*) and the attendee will be able to see them in action.

## 8. REFERENCES

[1] Giraph. `http://giraph.apache.org/`.

[2] J.-P. Dittrich and B. Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE*, pages 535–546, Mar. 2000.

[3] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, Apr. 2011.

[4] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, June 1984.

[5] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient Processing of k Nearest Neighbor Joins using MapReduce. *PVLDB*, 5:1016–1027, 2012.

[6] Q. Ma, B. Yang, W. Qian, and A. Zhou. Query Processing of Massive Trajectory Data Based on MapReduce. In *CLOUDDB*, pages 9–16, Oct. 2009.

[7] J. Nievergelt, H. Hinterberger, and K. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *TODS*, 9(1):38–71, 1984.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *SIGMOD*, June 2008.

[9] O. O'Malley. Terabyte Sort on Apache Hadoop. 2008.

[10] OpenStreetMap. `http://www.openstreetmap.org/`.

[11] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB*, 1987.

[12] TIGER files. `http://www.census.gov/geo/www/tiger/`.

[13] C. Zhang, F. Li, and J. Jestes. Efficient Parallel kNN Joins for Large Data in MapReduce. In *EDBT*, Mar.