The Ecosystem of SpatialHadoop

Ahmed Eldawy, Mohamed F. Mokbel
Department of Computer Science and Engineering, University of Minnesota, USA

Abstract

There is a recent outbreak in the amounts of spatial data generated by different sources, e.g., smart phones, space telescopes, and medical devices, which urged researchers to exploit the existing distributed systems to process such amounts of spatial data. However, as these systems are not designed for spatial data, they cannot fully utilize its spatial properties to achieve high performance. In this paper, we describe SpatialHadoop, a full-fledged MapReduce framework which extends Hadoop to support spatial data efficiently. SpatialHadoop consists of four main layers, namely, language*,* indexing*,* query processing*, and* visualization*. The* language *layer provides a high level language with standard spatial data types and operations to make the system accessible to non-technical users. The* indexing *layer supports standard spatial indexes, such as grid, R-tree and R+-tree, inside Hadoop file system in order to speed up spatial operations. The* query processing *layer encapsulates the spatial operations supported by SpatialHadoop such as range query,* k *nearest neighbor, spatial join and computational geometry operations. Finally, the* visualization *layer allows users to produce images that describe very large datasets to make it easier to explore and understand big spatial data. SpatialHadoop is already used as a main component in several real systems such as MNTG, TAREEG, TAGHREED, and SHAHED.*

1 Introduction

With the recent explosion in the amounts of spatial data, many researchers are trying to process these data efficiently using the distributed systems that run on hundreds of machines such as Hadoop and Hive. Unfortunately, most of these systems are designed for general data processing and this generality comes with the price of a sub-par performance with spatial data. Therefore, there are active research projects which try to extend these system to well support spatial data. Most notably, ESRI released a suit of GIS tools for Hadoop [15] which integrates Hadoop with their flagship ArcGIS product. Hadoop-GIS [2] extends Hive with a grid index and efficient implementation of range and self-join queries. Similarly, $\mathcal{MD}$-HBase [12] extends HBase with Quad tree and K-d tree indexes for point datasets and support range and kNN queries.

In this work, we describe the recent work in SpatialHadoop [6], a full-fledged system for spatial data which extends Hadoop in its core to efficiently support spatial data. SpatialHadoop is available as an open source software at http://spatialhadoop.cs.umn.edu/ and has been already downloaded around 80,000 times. SpatialHadoop consists of four main layers, namely, *language*, *indexing*, *query processing*, and *visualization*. In the *language* layer, SpatialHadoop provides a high level language, termed Pigeon [5], which provides standard spatial data types and query processing for easy access to non-technical users. The *indexing* layer provides efficient spatial indexes, such as grid, R-tree, and R+-tree, which organize the data nicely in the distributed file system. The indexes are organized in two levels, one *global* index that partitions the data across machines, and multiple *local* indexes that organize records in each machine. The *query processing* layer encapsulates a set of spatial operations that ship with SpatialHadoop including basic spatial operations, join operations and computational

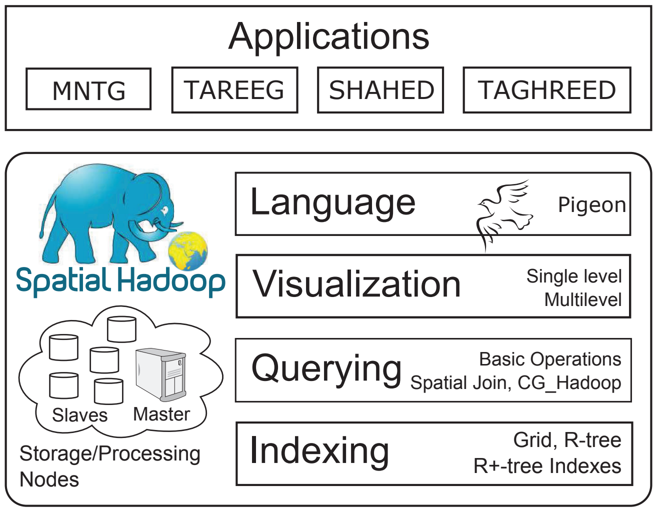


Figure 1: Overview of SpatialHadoop

geometry operations. The *visualization* layer allows users to explore big spatial data by generating images that provide bird's-eye view on the data. SpatialHadoop is already used in several real systems, such as SHAHED [7], TAREEG [3], MNTG [11], and TAGHREED [10].

2 Overview of SpatialHadoop

Figure 1 gives an overview of SpatialHadoop. SpatialHadoop runs on a cluster containing one master node, that breaks a MapReduce job into smaller tasks, and multiple slave nodes that carry out these tasks. The core of SpatialHadoop consists of four main layers, namely, *language*, *indexing*, *query processing*, and *visualization*, described briefly below.

(1) The **Language** layer contains **Pigeon** [5], a high level language with OGC-compliant spatial data types and functions. Pigeon is discussed in Section 3. (2) The **Indexing** layer provides standard spatial indexes, such as grid, R-tree, and R+-tree, which are used to store the data in an efficient way in the Hadoop Distributed File System (HDFS). Indexes are organized in two-layers, one global index that partitions data across nodes, and multiple local indexes to organize records inside each node. These indexes are made available to the MapReduce programs through two new components, namely, SpatialFileSplitter and SpatialRecordReader. The spatial indexing layer is described in Section 4. (3) The **Query Processing** layer encapsulates the spatial operations supported by SpatialHadoop. This includes *basic operations*, *join operations*, and *CG_Hadoop* [4] which is a suite of fundamental computational geometry operations. Developers and researchers can enrich this layer by implementing more advanced spatial operations. The supported operations are discussed in Section 5. (4) The **Visualization** layer provides efficient algorithms to visualize big spatial data by generating images that give a bird's-eye view to the data. SpatialHadoop supports *single level* images, which are generated at a fixed resolution, and multilevel images, which are generated at multiple resolutions to allow users to zoom in. The details of the visualization layer is provided in Section 6.

The core of SpatialHadoop is designed to serve as a backbone for applications that deal with large scale data processing. In Section 7, we describe SHAHED [7] as a case study of a real system which uses SpatialHadoop to analyze and visualize large scale satellite data.

3 Language Layer: Pigeon

Most MapReduce-based systems require huge coding efforts, therefore, they provide easy high level languages that make them usable by non-technical users, such as, HiveQL [14] for Hive and Pig Latin [13] for Hadoop. SpatialHadoop does not provide a completely new language, instead, it provides, Pigeon [5], which extends Pig Latin language [13] by adding spatial data types, functions, and operations that conform to the Open Geospatial Consortium (OGC) standard [1]. In particular, we add the following:
1. OGC-compliant spatial **data types** including, `Point`, `LineString`, and `Polygon`. Since Pig Latin does not allow defining new data types, Pigeon overrides the `bytearray` data type to define spatial data types. Conversion between `bytearray` and `geometry`, back and forth, is done automatically on the fly which makes it transparent to end users.
2. Basic spatial functions which are used to extract useful information from a single shape; e.g., `Area` calculates the area of a polygonal shape.
3. OGC-standard **spatial predicates** which return a Boolean value based on a test on the input polygon(s). For example, `IsClosed` tests if a linestring is closed while `Touches` checks if two geometries touch each other.
4. Spatial analysis functions which perform some spatial transformations on input objects such as calculating the `Centroid` or `Intersection`. These functions are usually used to performs a series of transformations on input records to produce final answer.
5. Spatial aggregate functions which take a set of spatial objects and return a single value which summarizes all input objects; e.g., the `ConvexHull` returns one polygon that represents the minimal convex polygon that contains all input objects.

In addition to the functions in Pigeon, we do the following changes to the language.
1. KNN Keyword. A new keyword `KNN` is added to perform a k-nearest neighbor query.
2. FILTER. To support a range query, we override the Pig Latin selection statement, `FILTER`, to accept a spatial predicate as an input and calls the corresponding procedure for range queries.
3. JOIN. To support spatial joins, we override the Pig Latin join statement `JOIN` to take two spatial files as input. The processing of the `JOIN` statement is then forwarded to the corresponding spatial join procedure.

4 Spatial Indexing

Traditional Hadoop stores data files in the Hadoop Distributed File System (HDFS) as heap files. This means that the data is partitioned into HDFS blocks, of 64 MB each, without taking the values of the records into consideration. While this is acceptable for traditional queries and applications, it results in a poor performance for spatial queries. There exist traditional spatial indexes, such as the R-tree [8], however, they are designed for the local file system and traditional *procedural* programming, hence, they are not directly applicable to Hadoop which uses HDFS and MapReduce *functional* programming. HDFS is inherently limited as files can be only written in sequential manner and, once written, cannot be modified.

To overcome the limitations of traditional spatial indexes, SpatialHadoop proposes a two-layer spatial index structure which consists of one *global* index and multiple *local* indexes. The global index partitions data into HDFS blocks and distributes them among cluster nodes, while local indexes organize records inside each block. The separation of global and local indexes lends itself to the MapReduce programming paradigm where the global index is used while preparing the MapReduce job while the local indexes are used for processing the map tasks. In addition, breaking the file into smaller partitions allows each partition to be indexed separately in memory and dumping it to a file in a sequential manner. SpatialHadoop uses this two-level design to build a grid index, R-tree and R+-tree.

Figure 2 shows an example of an R-tree index built in SpatialHadoop for a 400 GB dataset of all map objects in the world extracted from OpenStreetMap. Blue lines represent data while black rectangles represent partition

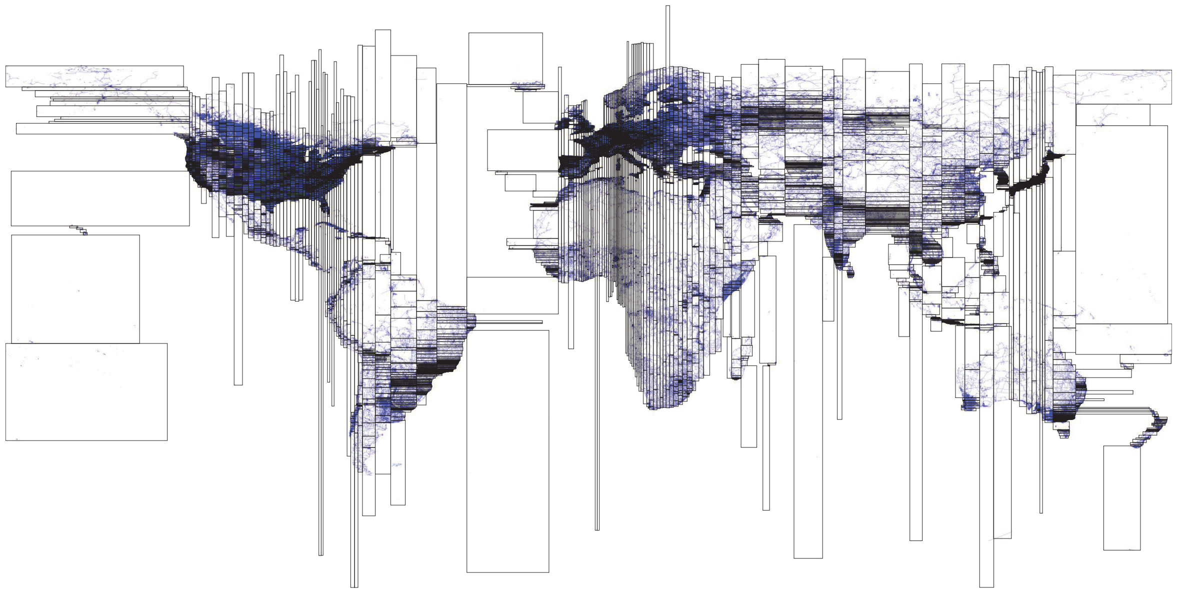
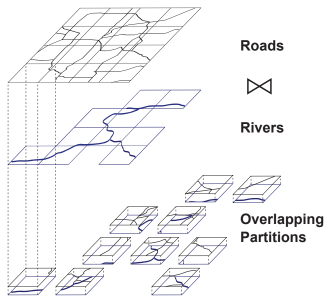


Figure 2: R-tree index of a 400 GB OpenStreetMap dataset representing all map objects (Best viewed in color)

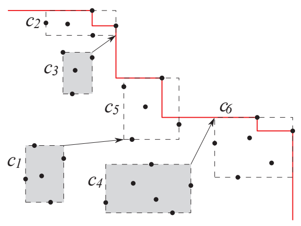
boundaries of the global index. As shown in this example, SpatialHadoop adjusts the size of each partition based on data distribution such that the total size of the contents of each partition is 64MB which ensures load balancing. Records in each partition are stored together as one HDFS block in one machine.

The index is constructed in one MapReduce job that runs in three phases. (1) The *partitioning* phase divides the space into n rectangles, then, it partitions the data by assigning each record to overlapping rectangles. The challenge in this step is how to adjust these rectangles such that the contents of each partition is around 64 MB of data to fit in one HDFS block. To overcome this challenge, we first calculate the desired number of partitions by dividing the input file size $|S|$ by the HDFS block capacity B, i.e., $n = |S|/B$. Then, for the grid index, we partition the space using a uniform grid of size $\sqrt{n} \times \sqrt{n}$ assuming uniformly distributed data. For R-tree and R+-tree, we draw a random sample from the input file, and bulk load this sample into an in-memory R-tree of n leaf nodes using the STR algorithm [9]. Then, the boundaries of the leaf nodes are used to partition the file assuming that the random sample is representative for data distribution. (2) In the *local indexing* phase, each partition is processed separately on a single machine and a local index is constructed in memory before it is dumped to disk. Since the partitioning phase adjusts the size of each partition to be of a single HDFS block, it becomes possible for each machine to completely load it into memory, build the index, and write it to disk in a sequential manner. (3) The final *global indexing* phase constructs a global index on the master node which indexes all HDFS blocks in the file using their MBRs as indexing key. The global index is kept in the main memory of the master node and it provides an efficient way to select file blocks in a specific range.

Once the data is stored efficiently in the file system as indexes, we need to add new components that allow MapReduce programs to use them. Without these new components, the traditional MapReduce components shipped with Hadoop will not be able to make use of these indexes and will treat them as heap files. Therefore, SpatialHadoop adds two new components, namely, *SpatialFileSplitter* and *SpatialRecordReader*. The SpatialFileSplitter takes a spatially indexed input file and a user-defined *filter function* and it exploits the global index in the input file to prune partitions that do not contribute to answer. The SpatialRecordReader takes a locally indexed partition returned by the filter function and exploits its local index to retrieve the records that match the user query. These two components allow developers to implement many spatial operations efficiently as shown in the next section.



(a) Spatial Join



(b) Pruning in Skyline

Figure 3: Spatial Queries in SpatialHadoop

5 Query Processing

The efficient indexes and the new MapReduce components introduced in the indexing layer give the core of SpatialHadoop that enables the possibility of efficient realization of many spatial operations. In this section, we show a few case studies of three categories of operations, namely, *basic operations*, *join operations* and *computational geometry* operations. Developers can follow similar techniques to add more operations such as kNN join or reverse nearest neighbor operations.

5.1 Basic Operations

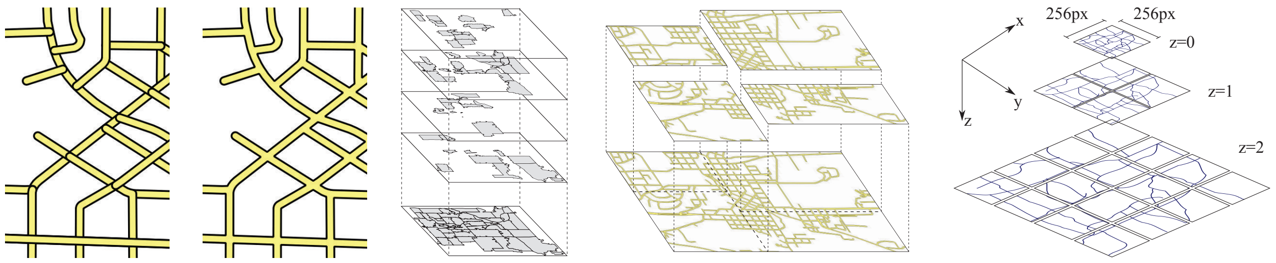
SpatialHadoop contains a number of basic spatial operations such as range query and k-nearest neighbor query. A range query takes a set of spatial records R and a query area A as input, and returns the records that overlap with A. SpatialHadoop exploits the global index with the SpatialFileSplitter to select only the partitions that overlap the query range A. Then, it uses the SpatialRecordReader to process the local indexes in matching partitions and find matching records. Finally, a duplicate avoidance step filters out duplicate results caused by replication in the index. Although this algorithm is efficient as it quickly prunes non-relevant partitions, it takes considerable time for very small ranges due to the overhead imposed by Hadoop for starting any MapReduce job. Therefore, if the query range is very small, i.e., matches only a few partitions, the algorithm can be implemented on a single machine without starting a MapReduce job, which provides an interactive response [7, 10].

5.2 Join Operations

Join operations are usually more complex as they deal with more than one file. In a spatial join query, the input consists of two sets of spatial records R and S and a spatial join predicate θ, e.g., `overlaps`, and the output is the set of all pairs $\langle r, s\rangle$ where $r \in R$, $s \in S$, and the join predicate θ is true for $\langle r, s\rangle$. SpatialHadoop proposes a MapReduce-based algorithm where the SpatialFileSplitter exploits the two global indexes to find overlapping pair of partitions as illustrated in Figure 3(a). The map function uses the SpatialRecordReader to exploit the two local indexes in each pair to find matching records. Finally, a duplicate avoidance step eliminates duplicate pairs in the answer caused by replication in the index.

5.3 CG_Hadoop

CG_Hadoop [4] is a suite of computational geometry operations for MapReduce. It supports five fundamental computational geometry operations, namely, polygon union, skyline, convex hull, farthest pair, and closest pair, all implemented as MapReduce algorithms. We show the skyline algorithm as an example while interesting readers can refer to [4] for further details.



(a) Not smoothed (b) Smoothed (c) Overlay (d) Stitch (e) Multilevel Image

Figure 4: Visualization

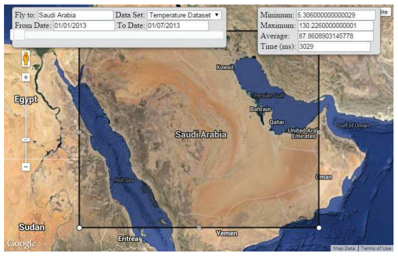
In the skyline operation, the input is a set of points P and the output is the set of *non-dominated* points. A point p dominates a point q if p is greater than q in all dimensions. CG_Hadoop adapts the existing divide-and-conquer skyline algorithm to Hadoop as a MapReduce program. Furthermore, CG_Hadoop utilizes the spatial index constructed using SpatialHadoop to prune partitions that are outside the query range. A partition c_i is pruned if *all* points in this partition are dominated by at least one point in another partition c_j, in which case we say that c_j dominates c_i. For example in Figure 3(b), c_1 is dominated by c_5 because the top-right corner of c_1 (i.e., best point) is dominated by the bottom-left corner of c_5 (i.e., worst point). The transitivity of the skyline domination rule implies that *any* point in c_5 dominates *all* points in c_1. In addition, the partition c_4 is dominated by c_6 because the top-right corner of c_4 is dominated by the top-left corner of c_6 which means that any point along the top edge of c_6 dominates all points in c_4. Since the boundaries of each partition are tight, there has to be at least one point along each edge.

6 Visualization

The visualization process involves creating an image that describes an input dataset. This is a natural way to explore spatial datasets as it allows users to find interesting patterns in the input which are otherwise hard to spot. Traditional visualization techniques rely on a single machine to load and process the data which makes them unable to handle big spatial data. SpatialHadoop provides a visualization layer which generates two types of images, namely, *single level* image and *multilevel* images, as described below.

6.1 Single Level Image Visualization

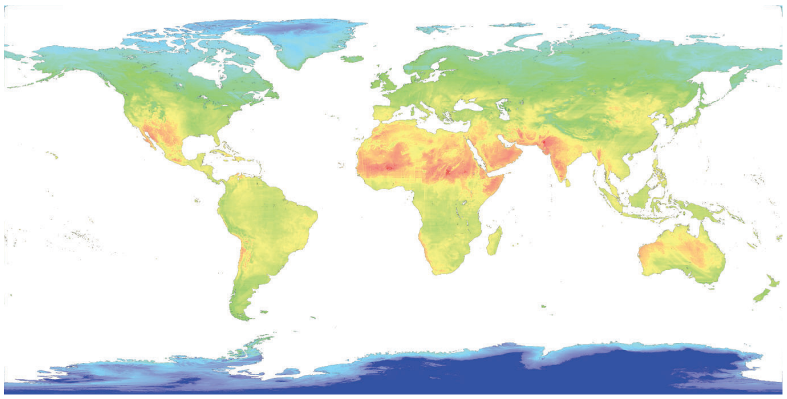
In single level image visualization, the input dataset is visualized as a single image of a user-specified image size ($width \times height$) in pixels. SpatialHadoop generates a single level image in three phases. (1) The *partitioning* phase partitions the data using either the default non-spatial Hadoop partitioner or using the spatial partitioner in SpatialHadoop depending on whether the data needs to be *smoothed* or not. Figure 4(a) shows an example of visualizing a road network without smoothing where intersecting road segments are overlapping each other, while Figure 4(b) shows the correct and desired image where intersecting road segments are merged (i.e., smoothed). If a smooth function is needed, we have to use a spatial partitioner to ensure that intersecting road segments are processed by the same machine and can be merged. (2) In the *rasterize* phase, the machines in the cluster process the partitions in parallel and generate a partial image for each partition. If the default Hadoop partitioner is used, each partial image has the same size of the final desired image because the partition contains data from all over the input space. On the other hand, if a spatial partitioner is used, each partial image would be of a small size according to the region covered by the associated partition. (3) In the *merging* phase, the partial images are combined together to produce the final image. If a non-spatial partitioner is used, partial images are *overlaid* as



Fly to: Saudi Arabia Data Set: Temperature Dataset
From Date: 01/01/2013 To Date: 01/07/2013

Minimum: 5.306000000000029
Maximum: 130.2260000000001
Average: 87.8608903145778
Time (ms): 3029

(a) Screen shot of SHAHED



(b) World heat map generated by SHAHED

Figure 5: Analyzing and Visualizing Satellite Data using SHAHED

they all have the size of the final image as shown in Figure 4(c). On the other hand, if a spatial partitioner is used, the merging phase *stitches* partial images together as shown in Figure 4(d).

6.2 Multilevel Image Visualization

The quality of a single level image is limited by its resolution which means users cannot zoom in to see more details. Therefore, SpatialHadoop also supports multilevel images which consist of small tiles produced at different zoom levels as shown in Figure 4(e). The input to this algorithm is a dataset and a range of zoom levels $[z_{min}, z_{max}]$ and the output is all image tiles in the specified range of levels. A naïve approach is to use the single level image algorithm to generate each tile independently but this approach is infeasible due to the excessive number of MapReduce jobs to run. For example, at zoom level 10, there will be more than one million images which would require running one million MapReduce jobs. Alternatively, SpatialHadoop provides a more efficient algorithm that runs in two phases only, *partition* and *rasterize*. (1) The partition phase scans all input records and replicates each record r to all overlapping tiles in the image according to the MBR of r and the MBR of each tile. This phase produces one partition per tile in the desired image. (2) The *rasterize* phase processes all generated partitions and generates a single image out of each partition. Since the size of each image tile is small, a single machine can generate that tile efficiently. This technique is used in [7] to produce temperature heat maps for NASA satellite data.

7 Case Study: SHAHED

The core of SpatialHadoop is used in several real applications that deal with big spatial data including MNTG [11], a web-based traffic generator; TAREEG [3], a MapReduce extractor for OpenStreetMap data; TAGHREED [10], a system for querying and visualizing twitter data, and SHAHED [7], a MapReduce system for analyzing and visualizing satellite data which is further discussed in this section. SHAHED is a tool for analyzing and exploring remote sensing data publicly available by NASA in a 500 TB archive. It provides a web interface (Figure 5(a)) where users navigate through the map and the system displays satellite data for the selected area.

SHAHED uses the indexing layer in SpatialHadoop to organize satellite data in a uniform grid index as the data is uniformly distributed. Furthermore, it builds an aggregate-quad-tree local index inside each grid cell to speed up both selection and aggregate queries. On top of the spatial index, it provides a multi-resolution temporal index which organizes data in days, months and years. For example, in the *daily* level, it builds a separate spatial index for each day, while in the *months* level, it builds one index for each month. The goal is to provide efficient query processing for both small and large temporal ranges.

In the *query processing* layer, it provides both selection and aggregate spatio-temporal queries where the input is a data set, e.g., temperature, a spatial range represented as a rectangular region on the map and a temporal range provided as a date range on the calendar (see Figure 5(a)). In selection queries, all values in the chosen dataset and spatio-temporal range are either returned to the user as a file to download, or further processed to produce an image as shown below. In aggregate queries, only the minimum, maximum and average values are returned.

SHAHED also makes use of the *visualization* layer to visualize satellite data. The results of the selection query are visualized as a satellite heat map. For example, it is used to generate a temperate heat map for the whole world, as shown in Figure 5(b), which consists of a total of 500 Million points. If a date range is selected instead of a single date, an animating video is generated which shows the change of temperature over the selected time [1]. SHAHED also uses the multilevel image visualization technique to precompute heatmaps for different datasets over the whole world and allow users to navigate these datasets on a web interface by overlaying the generated images over the world map and updating them as the user navigates.

References

[1] `http://www.opengeospatial.org/`.

[2] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. In *VLDB*, 2013.

[3] L. Alarabi, A. Eldawy, R. Alghamdi, and M. F. Mokbel. TAREEG: A MapReduce-Based Web Service for Extracting Spatial Data from OpenStreetMap. In *SIGMOD*, 2014.

[4] A. Eldawy, Y. Li, M. F. Mokbel, and R. Janardan. CG_Hadoop: Computational Geometry in MapReduce. In *SIGSPATIAL*, 2013.

[5] A. Eldawy and M. F. Mokbel. Pigeon: A Spatial MapReduce Language. In *ICDE*, 2014.

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

[7] A. Eldawy, M. F. Mokbel, S. Alharthi, A. Alzaidy, K. Tarek, and S. Ghani. SHAHED: A MapReduce-based System for Querying and Visualizing Spatio-temporal Satellite Data. In *ICDE*, 2015.

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

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

[10] A. Magdy, L. Alarabi, S. Al-Harthi, M. Musleh, T. Ghanem, S. Ghani, and M. F. Mokbel. Taghreed: A System for Querying, Analyzing, and Visualizing Geotagged Microblogs. In *SIGSPATIAL*, Nov. 2014.

[11] M. F. Mokbel, L. Alarabi, J. Bao, A. Eldawy, A. Magdy, M. Sarwat, E. Waytas, and S. Yackel. MNTG: An Extensible Web-based Traffic Generator. In *SSTD*, 2013.

[12] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi. $\mathcal{MD}$-HBase: Design and Implementation of an Elastic Data Infrastructure for Cloud-scale Location Services. *DAPD*, 31(2):289–319, 2013.

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

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

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

[1] Please refer to an example at http://youtu.be/hHrOSVAaak8