CG_Hadoop: Computational Geometry in MapReduce

Ahmed Eldawy [#*] Yuan Li [#] Mohamed F. Mokbel[#$*] Ravi Janardan[#]

[#]Department of Computer Science and Engineering, University of Minnesota, Twin Cities
[$]KACST GIS Technology Innovation Center, Umm Al-Qura University, Makkah, Saudi Arabia
{eldawy,yuan,mokbel,janardan}@cs.umn.edu

ABSTRACT

Hadoop, employing the MapReduce programming paradigm, has been widely accepted as the standard framework for analyzing big data in distributed environments. Unfortunately, this rich framework was not truly exploited towards processing large-scale computational geometry operations. This paper introduces CG_Hadoop; a suite of scalable and efficient MapReduce algorithms for various fundamental computational geometry problems, namely, *polygon union*, *skyline*, *convex hull*, *farthest pair*, and *closest pair*, which present a set of key components for other geometric algorithms. For each computational geometry operation, CG_Hadoop has two versions, one for the Apache Hadoop system and one for the SpatialHadoop system; a Hadoop-based system that is more suited for spatial operations. These proposed algorithms form a nucleus of a comprehensive MapReduce library of computational geometry operations. Extensive experimental results on a cluster of 25 machines of datasets up to 128GB show that CG_Hadoop achieves up to 29x and 260x better performance than traditional algorithms when using Hadoop and SpatialHadoop systems, respectively.

Categories and Subject Descriptors

I.3.5 [**Computational Geometry and Object Modeling**]: Geometric algorithms; H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms

Algorithms

Keywords

Hadoop, MapReduce, Geometric Algorithms

*The work of these authors is supported in part by the National Science Foundation, USA, under Grants IIS-0952977 and IIS-1218168.

SIGSPATIAL'13, November 05 - 08 2013, Orlando, FL, USA
Copyright 2013 ACM 978-1-4503-2521-9/13/11...$15.00.
http://dx.doi.org/10.1145/2525314.2525349

1. INTRODUCTION

Hadoop [17] is a framework designed to efficiently process huge amounts of data in a distributed fashion. It employs the MapReduce programming paradigm [11], which abstracts a parallel program into two functions, *map* and *reduce*. The *map* function maps a single input record to a set of intermediate key value pairs $\langle k, v \rangle$, while the *reduce* function takes all values associated with the same key and produce the final answer. The simplicity and flexibility of the MapReduce paradigm allow Hadoop to be employed in several large-scale applications including machine learning [13], tera-byte sorting [29], and graph processing [14].

In the meantime, there is a recent tremendous increase in devices and applications that generate enormous rates of spatial data. Examples of such devices include smart phones, space telescopes [6], and medical devices [28,35]. Such amount of Big Spatial Data calls for the need to take advantage of the MapReduce programming paradigm [11] to solve various spatial operations. Among the most important spatial operations is the family of Computational Geometry algorithms that are concerned with representing and working with geometric entities in the spatial domain. Examples of such operations include convex hull, skyline, union, and farthest/closest pairs. Although there exist well established computational geometry algorithms for such problems [5, 33], unfortunately, such algorithms do not scale well to handle modern spatial datasets which can contain, for instance, billions of points. For example, computing a convex hull for a data set of 4 billion points using a traditional algorithm may take up to three hours, while computing the union of a data set of 5M polygons takes around one hour and fails with a memory exception for larger data sets.

In this paper, we introduce CG_Hadoop; a suite of scalable and efficient MapReduce algorithms for various fundamental computational geometry problems, namely, *polygon union*, *skyline*, *convex hull*, *farthest pair*, and *closest pair*, which present a set of key components for other geometric algorithms [5, 33]. CG_Hadoop achieves order(s) of magnitude better performance than traditional computational geometry algorithms when dealing with large-scale spatial data. For each computational geometry operation, we introduce two versions of CG_Hadoop. The first version is deployed on the Apache Hadoop system [17]; an open-source MapReduce platform, which is widely used in various MapReduce applications, e.g., see [9,13,14,19,20,29]. The second version of CG_Hadoop is deployed on SpatialHadoop [12]; a Hadoop-based system equipped with spatial indexes and is more suited for spatial operations.

The main idea behind all algorithms in CG_Hadoop is to take advantage of the *divide and conquer* nature of many computational geometry algorithms. The divide and conquer property lends itself to the MapReduce environment, where the bulk of work can be parallelized to work on multiple nodes in a computational machine cluster. Yet, CG_Hadoop has to adapt traditional computational algorithms to work better in the MapReduce environment. For example, unlike traditional algorithms which usually divide the input in half and do multiple rounds, CG_Hadoop divides the input into smaller chunks to ensure that the answer is computed in one MapReduce round, which is preferable for both Hadoop and SpatialHadoop. In addition, we use the distributed spatial indexes provided in SpatialHadoop, whenever possible, to speed up the computations by early pruning input chunks that do not contribute to the answer of the computational geometry operation of interest.

CG_Hadoop, source code available as part of SpatialHadoop at *http://spatialhadoop.cs.umn.edu/*, forms a nucleus of a comprehensive MapReduce library of computational geometry operations. Its open-source nature will act as a research vehicle for other researchers to build more computational geometry algorithms that take advantage of the MapReduce programming paradigm. Extensive experiments on a cluster of 25 machines using both real and generated datasets of sizes up to 128GB show that CG_Hadoop achieves up to 29x and 260x better performance than traditional algorithms when using Hadoop and SpatialHadoop systems, respectively.

The rest of this paper is organized as follows. Section 2 gives a brief necessary background. MapReduce algorithms for the polygon union, skyline, convex hull, farthest pair, and closest pair operations are given in Sections 3 to 7. Section 8 gives an experimental evaluation. Related work is discussed in Section 9, while Section 10 concludes the paper.

2. BACKGROUND

This section gives a background about Hadoop [17] and SpatialHadoop systems as the two platforms used in CG_Hadoop as well as the set of computational geometry operations in CG_Hadoop.

2.1 Hadoop

Hadoop [17] is an open-source framework for data processing on large clusters. A Hadoop cluster consists of one master node and several slave nodes. The master node stores meta information about files (e.g., name and access rights) while slave nodes store the actual data in files (e.g., records). A file is usually uploaded to the Hadoop Distributed File System (HDFS) before it is processed where the file is split into chunks of 64MB (called blocks). The master node keeps track of how the file is split and where each block is stored, while slave nodes store the data blocks. In analogy with a regular file system, the master node stores the file allocation table or INodes, while slave nodes store data in files.

A MapReduce program [11] configures a MapReduce job and submits it to the master node. A MapReduce job contains a set of configuration parameters such as the *map* function and the input file. The master node breaks this job into several *map tasks* and *reduce tasks* and run each one on a slave node. It also breaks the input file into splits and assigns each split to a slave node to be processed as a map task. The map task parses its assigned split using the configured *record reader* and produces a set of key-value pairs $\langle k_1, v_1 \rangle$ which are sent to the *map* function to produce a set of intermediate pairs $\langle k_2, v_2 \rangle$. Intermediate pairs are grouped by k_2 and the *reduce* function collects all intermediate records with the same key and processes them to generate a set of final records $\langle k_3, v_3 \rangle$ which are stored as the job output in HDFS files.

MapReduce and Hadoop have been widely adopted by major industry, e.g., Google [11], Yahoo! [9] Dryad in Microsoft [18], Cassandra in Facebook [19], and Twitter [20]. It has also been widely employed in several large-scale applications including machine learning [13], tera-byte sorting [29], and graph processing [14].

2.2 SpatialHadoop

SpatialHadoop [12] is a comprehensive extension to Hadoop that enables efficient processing of spatial operations. Mainly, it provides a two-layered spatial index in the Hadoop storage layer with implementations of Grid file [26] and R-tree [16] indexing. It also enriches the MapReduce layer with new components that allow using the spatial index structures within MapReduce programs. The built-in indexes in SpatialHadoop help in building efficient algorithms for several spatial operations.

The spatial index in SpatialHadoop is organized as one *global index* and multiple *local indexes*. The global index partitions data across cluster nodes while the local indexes organize data inside each node. The new added components in the MapReduce layer utilize both the global and local indexes to prune file partitions and records, respectively, that do not contribute to the answer. The pruning criteria is determined through a user defined *filter* function which is provided as part of the MapReduce program.

2.3 Computational Geometry Operations

As indicated earlier, CG_Hadoop forms a nucleus of a comprehensive MapReduce library of computational geometry operations. Currently, CG_Hadoop includes five fundamental operations, namely, *Union*, *Skyline*, *Convex Hull*, *Farthest pair*, and *Closest Pair*. Below, we give a brief definition of each operation.

Union. The union of a set S of polygons is the set of all points that lie in at least one of the polygons in S, where only the perimeter of all points is kept while inner segments are removed. Figure 1(a) gives a sample input to the polygon union operation as a set of ZIP code areas, while Figure 1(b) gives the union result.

Skyline. Consider the set of points P in Figure 1(c). Point $p_i \in P$ *dominates* point $p_j \in P$ if each of the coordinates of p_i is greater than or equal to the corresponding coordinate of p_j, with strict inequality in at least one dimension. The *skyline* of P consists of those points of P that are not dominated by any other point of P (e.g., Figure 1(d)). In the computational geometry literature, the skyline points are usually called maximal points [33].

Convex Hull. The convex hull of a set of points P is the smallest convex polygon that contains all points in P, as given in Figure 1(e). The output of the convex hull operation is the points forming the convex hull ordered in a clockwise direction.

Farthest Pair. Given a set of points P, the farthest pair is the pair of points that have the largest Euclidean distance between them. As shown in Figure 1(e), the two points contributing to the farthest pair have to lie on the convex hull.

Closest Pair. Given a set of points P, the closest pair is the pair of points that have the smallest Euclidean distance between them (Figure 1(e)).

3. UNION

A traditional algorithm for the polygon union operation [33] computes the union of two polygons by computing all edges intersections, removing all inner segments, and leaving only segments on the perimeter. For more than two polygons, we start with one polygon, add other polygons to it one by one and compute the union with each polygon added. In PostGIS [32], this operation can be carried out using the following SQL query where the column `geom` stores polygon information of each ZIP code.

```
SELECT ST_Union(zip_codes.geom)
FROM zip_codes;
```

In this section, we introduce two polygon union algorithms for Hadoop and SpatialHadoop. We use input dataset in Figure 1(a) as a clarification example. For ease of illustration and without loss of generality, the example has non-overlapped polygons.

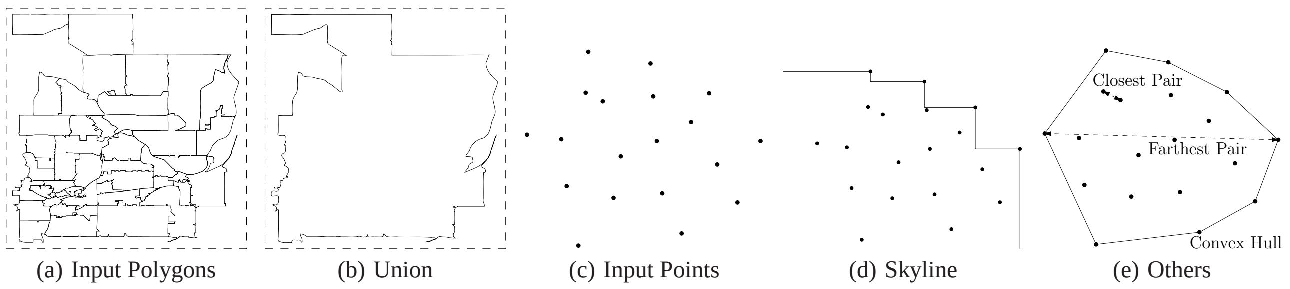


Figure 1: Computational Geometry Operations covered by CG_Hadoop

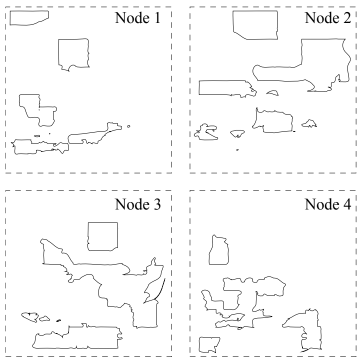


Figure 2: Polygon union in Hadoop

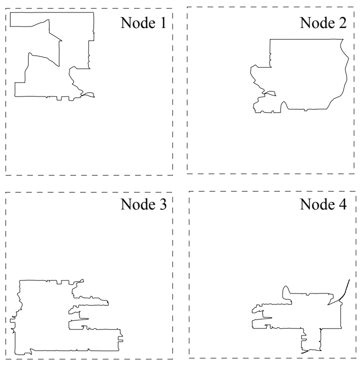


Figure 3: Polygon union in SpatialHadoop

3.1 Union in Hadoop

The main idea of our Hadoop polygon union algorithm is to allow each machine to accumulate a subset of the polygons, and then let a single machine combine the results from all machines and compute the final answer. Our algorithm works in three steps: *partitioning*, *local union*, and *global union*. The *partitioning* step distributes the input polygons into smaller subsets each handled by one machine. This step is performed by the Hadoop `load file` command which splits the file into chunks of 64MB storing each one on a slave node. In the *local union* step, each machine computes the union of its own chunk using a traditional in-memory polygon union algorithm. As each chunk is at most of size 64MB, the in-memory algorithm works fine regardless of the size of the input file. This step is implemented in Hadoop as a *combine* function, which runs locally in each machine. After performing the local union, each machine ends up with a set of polygons that represent the union of all polygons assigned to this machine. The *global union* step is implemented in Hadoop as a *reduce* function, which runs on a single machine to compute the final answer. The reduce function takes the output of all local unions, combines them into one list, and computes their union using the traditional in-memory algorithm. Each machine will end up with only few polygons, making it possible to do the union using the in-memory algorithm.

By taking advantage of a set of parallel machines, rather than performing all the work in a single machine, our proposed algorithm achieves orders of magnitude better performance than that of traditional algorithms. Although there is an overhead in partitioning the data to multiple machines, and then collecting the answer from each machine, such overhead is offset by the cost saving over parallel machines, which can be seen in large-scale spatial data sets. For interested readers, who are familiar with MapReduce programming paradigm, the pseudocode of our Hadoop polygon union algorithm is given in Appendix A.1.

Figure 2 gives the partitioning and local union steps of the input dataset of Figure 1(a) over four cluster computing nodes, where each polygon is assigned to one node. The decision of which node belongs to which partition is completely taken by the Hadoop load file component, where it basically assigns polygons to nodes randomly. As a result, and as can be seen in the figure, some polygons assigned to one node might remain completely disjoint after computing the union. In this case, all these polygons are written to the output. Then, all nodes send their output to a single machine which computes the final answer as given in Figure 1(b)

3.2 Union in SpatialHadoop

Our polygon union algorithm in SpatialHadoop has the same three steps as our algorithm in Hadoop. The only difference is that the partitioning step in SpatialHadoop is done in a spatially-aware manner, as given in Figure 3, where adjacent polygons are assigned to the same machine. The main reason here is that we utilize the underlying index structure in SpatialHadoop to be able to distribute polygons over nodes. In particular, we use the R-tree indexing in SpatialHadoop, where the size of each R-tree node is 64MB, to dump all the entries in each R-tree node to one node cluster. Since, by definition, an R-tree node provides a cluster of adjacent polygons, especially, that all R-trees in SpatialHadoop are bulk loaded, then we guarantee that all polygons in the same node are adjacent.

Although the local and global union steps remain the same, they become much lighter. The local union step mostly produces one output polygon, rather than a set of polygons as in Hadoop, therefore, the global union step processes fewer polygons. In our example, the number of polygons resulting from the local union step drops from 28 polygons in Hadoop to only four polygons in SpatialHadoop making the whole algorithm significantly faster. The pseudocode for the polygon union algorithm in SpatialHadoop is exactly the same as that of Hadoop (Appendix A.1).

4. SKYLINE

A traditional in-memory two-dimensional skyline algorithm [33] uses a divide and conquer approach where all points are initially sorted by their x coordinates and divided into two subsets of equal size separated by a vertical line. Then, the skyline of each half is computed recursively and the two skylines are merged to compute the final skyline. To merge two skylines, the points of the left skyline are scanned in a non-decreasing x order, which implies a non-increasing y order, and each one is compared to the left most point of the right skyline. Once a point on the left skyline is dominated, it is removed along with all subsequent points on the left skyline and the two lists of remaining points from both skylines are concatenated together. The skyline operator is not natively supported in database management systems. Yet, it is of considerable interest in the database literature, where the focus is mainly on disk-based algorithms (e.g., see [7, 31]) with a *non-standard* SQL query.

```
SELECT * FROM points
SKYLINE OF d1 MAX, d2 MAX;
```

In this section, we introduce our two skyline algorithms for Hadoop and SpatialHadoop, while using input dataset in Figure 1(c) as an illustrative example.

4.1 Skyline in Hadoop

Our Hadoop skyline algorithm is a variation of the traditional divide and conquer skyline algorithm [33], where we divide the input into multiple (more than two) partitions, such that each partition can be handled by one machine. This way, the input needs to be divided across machines only once ensuring that the answer is found in one MapReduce iteration. Similar to our Hadoop polygon union algorithm, our Hadoop skyline algorithm works in three steps, *partitioning*, *local skyline*, and *global skyline*. The *partitioning* step divides the input set of points into smaller chunks of 64MB each and distributes them across the machines. In the *local skyline* step, each machine computes the skyline of each partition assigned to it, using the traditional algorithm, and outputs only the non-dominated points. Finally, in the *global skyline* step, a single machine collects all points of local skylines, combines them in one set, and computes the skyline of all of them. Notice that skylines cannot be merged using the technique used in the in-memory algorithm as the local skylines are not separated by a vertical line, and may actually overlap. This is a result of Hadoop partitioning which distributes the points randomly without taking their spatial locations into account. The global skyline step computes the final answer by combining all the points from local skylines into one set, and applying the traditional skyline algorithm. Readers familiar with MapReduce programming can refer to Appendix A.2 for pseudocode.

This algorithm significantly speeds up the skyline computation compared to the traditional algorithm by allowing multiple machines to run independently and in parallel to reduce the input size significantly. For a uniformly distributed dataset of size n points, it is expected that the number of points on the skyline is $O(\log n)$ [4]. In practice, for a partition of size 64MB with around 700K points, the skyline only contains a few tens of points for both real and uniformly generated datasets. Given this small size, it becomes feasible to collect all those points in one single machine that computes the final answer.

4.2 Skyline in SpatialHadoop

Our proposed skyline algorithm in SpatialHadoop is very similar to the Hadoop algorithm described earlier, with two main changes. First, in the *partitioning* phase, we use the SpatialHadoop partitioner when the file is loaded to the cluster. This ensures that the data is partitioned according to an R-tree instead of random partitioning, which means that local skylines from each machine are non overlapping. Second, we apply an extra *filter* step right before the local skyline step. The *filter* step, runs on a master node, takes as input the minimal bounding rectangles (MBRs) of all partitioned R-tree index cells, and prunes those cells that have no chance in contributing any point to the final skyline result.

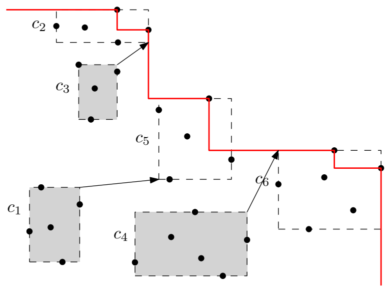


Figure 4: Skyline in SpatialHadoop

The main idea of the new *filter* step is that a cell c_i dominates another cell c_j if there is at least one point in c_i that dominates all points in c_j, in which case c_j is pruned. For example, in Figure 4, cell c_1 is dominated by c_5 because the bottom-left corner of c_5 (i.e., worst point) dominates the top-right corner of c_1 (i.e., best point). The transitivity of the skyline dominance relation implies that any point in c_5 dominates all points in c_1. Similarly, c_4 is dominated by c_6 because the top-left corner of c_6 dominates the top-right corner of c_4. This means that any point along the top edge of c_6 dominates the top-left corner of c_6, and hence, dominates all points in c_4. As the boundaries of a cell are minimal (because of R-tree partitioning), there should be at least one point of P on each edge. We can similarly show that cell c_3 is also dominated by c_2. So, our pruning technique in the *filter* step is done through a nested loop that tests every pair of cells c_i and c_j together. We compare the top-right corner of c_j against three corners of c_i (bottom-left, bottom-right, and top-left). If any of these corners dominates the top-right corner of c_j, we prune c_j out from all our further computations, and do not assign it to any node. Hence, we will not compute its local skyline, nor consider it in the global skyline step.

It is important to note that applying this filtering step in Hadoop will not have much effect, as the partitioning scheme used in Hadoop will not help in having such separated MBRs for different cells. The SpatialHadoop skyline algorithm has much better performance than its corresponding Hadoop algorithm as the *filtering* step prunes out many cells that do not need to be processed. Interested readers can refer to Appendix A.2 for the pseudocode of the filter step.

5. CONVEX HULL

The convex hull shown in Figure 1(e) can be computed as the union of two chains using Andrew's Monotone Chain algorithm [3]. First, it sorts all points by their x coordinates and identifies the left-most and right-most points. Then, the upper chain of the convex hull is computed by examining every three consecutive points p, q, r, in turn, from left to right. If the three points make a non-clockwise turn, then the middle point q is skipped as it cannot be part of the upper chain and the algorithm then considers the points p, r, s, where s is the successor of r; otherwise the algorithm continues by examining the next three consecutive points q, r, s. Once the rightmost point is reached, the algorithm con-

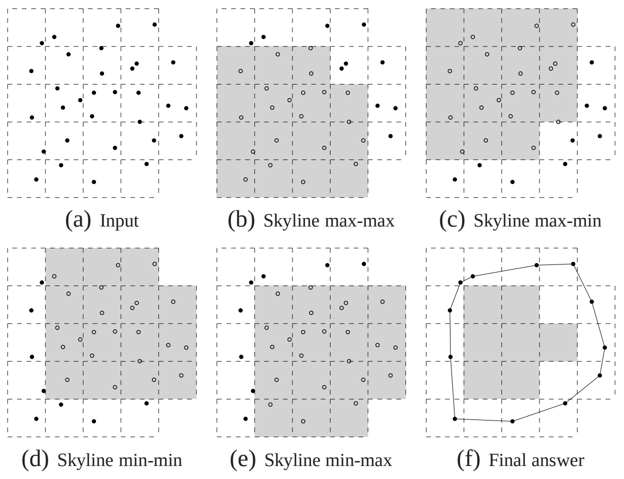


Figure 5: Convex hull in SpatialHadoop

tinues by computing the lower chain in a similar way by checking all points of P from right to left and doing the same check. Using PostGIS [32], the convex hull can be computed by a single SQL query using the function `ST_ConvexHull`. Since this function takes one record as argument, points have to be first combined in one line string using the function `ST_Makeline`.

```
SELECT ST_ConvexHull(ST_Makeline(points.coord))
FROM points;
```

In this section, we introduce two convex hull algorithms for Hadoop and SpatialHadoop, while using input dataset in Figure 1(c) as an illustrative example.

5.1 Convex Hull in Hadoop

Our Hadoop convex hull algorithm is very similar to our Hadoop skyline algorithm, where we start by a *partitioning* phase to distribute the input data into small chunks such that each chunk fits in memory. Then, the *local convex hull* of each subset is calculated using the traditional in-memory algorithm [3] and only those points forming the convex hull are retained. The points from all convex hulls in all machines are combined in a single machine that computes the *global convex hull*, which forms the final answer, using the traditional in-memory convex hull algorithm. Similar to skyline, the number of points on the convex hull is expected to be $O(\log n)$ [10] for uniform data, making this algorithm very efficient in pruning most of the points when computing the local hull and allowing the global hull to be computed in one node.

5.2 Convex Hull in SpatialHadoop

The convex hull algorithm in Hadoop processes more file partitions than necessary. Intuitively, the parts of the file that are towards the center do not contribute to the answer. In SpatialHadoop, we improve the convex hull algorithm by early pruning those partitions that do not contribute to answer. The key idea is that any point on the convex hull must be part of at least one of the four skylines of the dataset (max-max, min-max, max-min, and min-min) [33]. A max/min-max/min skyline considers that maximum/minimum points are preferred in x-y dimensions. This property allows us to reuse the skyline *filtering* step in Section 4.2. As given in Figure 5, we apply the skyline algorithm four times to select the partitions needed for the four skylines and take the union of all these partitions as the ones to process. Clearly, a partition that does not contribute to any of the four skylines will never contribute to the final convex hull. Once the partitions to be processed are selected, the algorithm works similar to the Hadoop algorithm in Section 5.1 by computing the local convex hull of each partition and then combining the local hulls in one machine, which computes the global convex hull. The gain in the SpatialHadoop algorithm comes from the spatially-aware partitioning scheme that allows for the pruning in the *filtering* step, and hence the cost saving in both local and global convex hull computations. For interested readers, the pseudocode of the SpatialHadoop convex hull algorithm is in Appendix A.3.

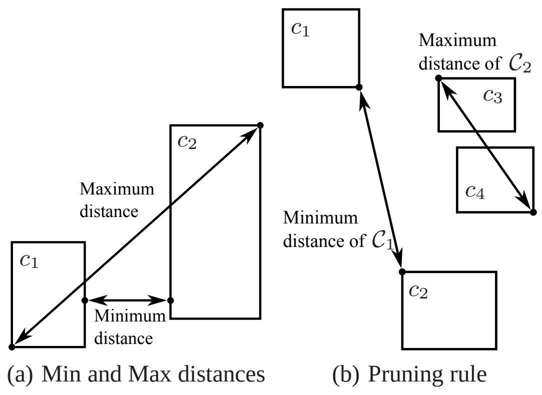

c_1
c_2
Maximum distance
Minimum distance
(a) Min and Max distances
c_1
Maximum distance of $\mathcal{C}_2$
c_3
c_4
Minimum distance of $\mathcal{C}_1$
c_2
(b) Pruning rule

Figure 6: Farthest pair algorithm in SpatialHadoop

6. FARTHEST PAIR

A nice property of the farthest pair (shown in Figure 1(e)) is that the two points forming the pair must lie on the convex hull of all points [34]. This property is used to speed up the processing of the farthest pair operation by first computing the convex hull, then finding the farthest pair of points by scanning around the convex hull using the rotating calipers method [33]. In this section, we introduce our farthest pair algorithms for Hadoop and SpatialHadoop.

6.1 Farthest Pair in Hadoop

A Hadoop algorithm for the rotating calipers method [33] would complete the convex hull first as discussed in Section 6.1. Then, a single-machine will need to scan all the points in the convex hull, which may be a bottleneck based on the number of points in the convex hull. In that case, it may be better to develop a Hadoop algorithm based on parallelizing the brute-force approach of farthest pair algorithm, which calculates the pairwise distances between every possible pair of points and select the maximum. The brute force approach will be expensive for very large input files, yet it may be used if it is not feasible for one machine to calculate the farthest pair from the points in the convex hull as in the rotating calipers method. Overall, both the brute force and rotating calipers methods have their own drawbacks when realized in Hadoop.

6.2 Farthest Pair in SpatialHadoop

Our SpatialHadoop algorithm works similar to our skyline and convex hull algorithms as we have four steps, *partitioning*, *filtering*, *local farthest pair*, and *global farthest pair*. In the *partitioning* step, we mainly use the SpatialHadoop partitioning scheme. In the *filtering* step, we apply a specialized filtering rule for the *farthest pair* operation. The main idea is explained in Figure 6. For each pair of cells, c_i and c_j, we compute the *minimum* (*maximum*) distance between c_i and c_j as the minimum (maximum) possible distance between any two points $p_i \in c_i$ and $p_j \in c_j$ (Figure 6(a)). Then, given two pairs of cells $\mathcal{C}_1 = \langle c_1, c_2\rangle$ and $\mathcal{C}_2 = \langle c_3, c_4\rangle$, we

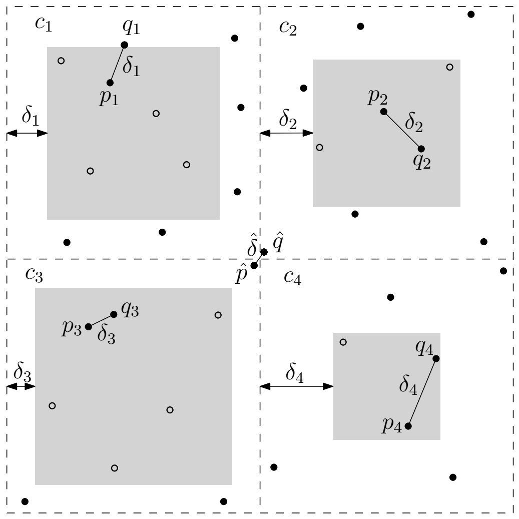


Figure 7: Closest Pair in SpatialHadoop

say that $\mathcal{C}_1$ dominates $\mathcal{C}_2$ if the minimum distance of $\mathcal{C}_1$ is greater than or equal to the maximum distance of $\mathcal{C}_2$. In this case, the pair $\mathcal{C}_2$ can be pruned as it can never contain the farthest pair in the dataset. This is depicted in Figure 6(b), where the farthest pair of $\mathcal{C}_1$ must have a distance greater than the farthest pair of $\mathcal{C}_2$. In this case, the pair of cells $\langle c_3, c_4 \rangle$ will never contribute to the final answer, and hence will not be considered further for any processing.

Once all dominated cell pairs are pruned, the algorithm computes the local farthest pair for each selected pair of cells by finding the *local* convex hull, then applying the rotating calipers algorithm on the result [33]. It is important to note that it is feasible here to use the in-memory algorithms for local convex hull as the size of each pair is bounded by twice the cell size. Finally, the algorithm computes the global farthest pair by collecting all local farthest pairs and selecting the one with largest distance. For interested readers, the pseudocode of the farthest pair algorithms is in Appendix A.4.

7. CLOSEST PAIR

The closest pair (Figure 1(e)) in any dataset can be found using a divide and conquer algorithm [25]. The idea is to sort all points by x coordinates, and then based on the median x coordinate, we partition the points into two subsets, P_1 and P_2, of roughly equal size and recursively compute the closest pair in each subset. Based on the two distances of the two closest pairs found, the algorithm then continues to compute the closest pair of points $p_1 \in P_1$ and $p_2 \in P_2$ such that the distance between them is better (smaller) than the two already found. Finally, the algorithm returns the best pair among the three pairs found. In this section, we introduce our closest pair algorithms for Hadoop and SpatialHadoop.

7.1 Closest Pair in Hadoop

Applying the divide and conquer algorithm described above in Hadoop as-is will be fairly expensive. First, it requires a presort for the whole dataset which requires, by itself, two rounds of MapReduce [29]. Furthermore, the merge step requires random access to the list of sorted points which is a well-known bottleneck in Hadoop file system [21]. On the other hand, using the default Hadoop loader to partition the data and compute the local closest pair in each partition (as in the farthest pair algorithm) may produce incorrect results. This is because data is partitioned randomly, which means that a point in one partition might form a closest pair with a point in another partition. Finally, as we mentioned with the farthest pair problem in Section 5.1, the brute force approach would work but it requires too much computations for large files.

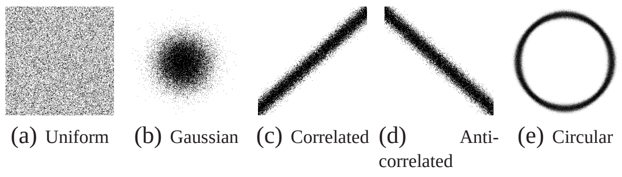


Figure 8: Synthetic data distributions

7.2 Closest Pair in SpatialHadoop

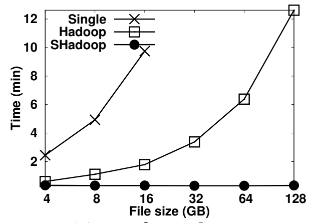
Our closest pair algorithm in SpatialHadoop is an adaptation of the traditional closest pair divide and conquer algorithm [25]. The algorithm works in three steps, *partitioning*, *local closest pair*, and *global closest pair*. In the *partitioning* step, the input dataset is loaded using SpatialHadoop loader which partitions the data into cells as shown in Figure 7. As the size of each partition is only 64MB, the algorithm computes the *local closest pair* in each cell using the traditional divide and conquer algorithm and returns the two points forming the pair. In addition, the algorithm must also return all candidate points that can possibly produce a closer pair when coupled with points from neighboring cells. Looking at Figure 7, let us assume that the closest pair found in c_1 has the distance δ_1. We draw an internal *buffer* with size δ_1 measured from the boundaries of c_1 and return all points inside this buffer (shown as solid points) as the candidate points while all other points are pruned. Notice that the two points forming the closest pair were returned earlier and are not affected by this pruning step. As shown in this example, each cell c_i may have a different buffer size δ_i based on the closest pair found inside this cell. While the minimum of all δ's would be a better and tighter value to compute all buffers, it cannot be used because the MapReduce framework enforces all map tasks to work in isolation which gives the framework more flexibility in scheduling the work. Finally, in the *global closest pair* step, all points returned from all cells are collected in a single machine which computes the global closest pair $\hat{p}, \hat{q}$ by applying the traditional divide and conquer algorithm to the set of all points returned by all machines.

For this algorithm to be correct, the cells must be non-overlapping, which is true for the cells induced by SpatialHadoop partitioning. This ensures that when a point p is pruned, there are no other points in the whole dataset P that are closer than the ones in its same cell. Otherwise, if cells are overlapping, a point p near the overlap area might be actually very close to another point q from another cell, thus none of them can be pruned. For readers familiar with the MapReduce programming, the pseudocode is in Appendix A.5.

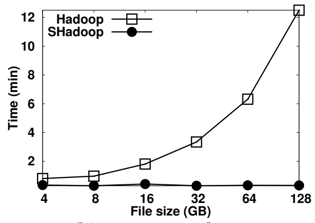
8. EXPERIMENTS

In this section we give an experimental study to show the efficiency and scalability of CG_Hadoop. Both Hadoop and SpatialHadoop clusters are based on Apache Hadoop 1.2.0 and Java 1.6. All experiments were conducted on an internal university cluster of 25 nodes. The machines are heterogeneous with HDD sizes ranging from 50GB to 200GB, memory ranging from 2GB to 8GB and processor speeds ranging from 2.2GHz to 3GHz. Single machine experiments are conducted on a more powerful machine with 2TB HDD, 16GB RAM and an eight core 3.4GHz processor.

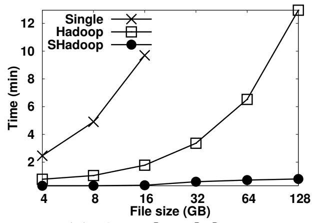
Experiments were run on three datasets: (1) OSM1: A real dataset extracted from OpenStreetMap [30] containing 164M poly-



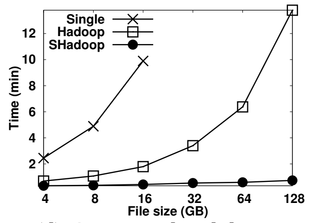
(a) Uniform data



(b) Gaussian data

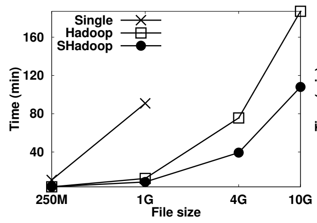


(c) Correlated data

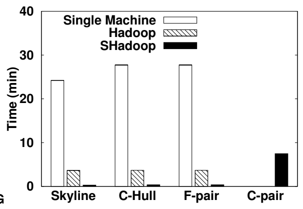


(d) Anti-correlated data

Figure 9: Skyline experiments



(a) Union



(b) Others

Figure 10: Experiments on OSM data

gons from the map (e.g., lakes and parks) with a total size of 80GB. (2) OSM2: A real dataset also extracted from OpenStreetMap and contains 1.7 Billion points from all around the world (e.g., street intersections and points of interest) with a total size of 52GB. (3) SYNTH: A synthetic dataset of points generated randomly in an area of 1M$\times$1M using one of the distributions: *uniform*, *Gaussian*, *correlated*, *anti-correlated*, and *circular* (See Figure 8). *Uniform* and *Gaussian* represent the two most widely used distributions for modeling many real life systems. *Correlated* and *anti-correlated* represent the best and worst cases for skyline. The *circular* data is used specifically for the farthest pair operation to generate the worst case scenario where the convex hull size is very large. The largest dataset generated is of size 128GB containing 3.8 billion points.

We use total execution time as the main performance metric. Sometimes, the results of single machine experiments are omitted if the operation runs out of memory or the numbers are too large that would cause the difference between other algorithms to diminish. Experimental results of the proposed operations running on the real and synthetic datasets are given in Sections 8.1 and 8.2 respectively.

8.1 Real data

This section gives performance results of the proposed operations running on the OSM real datasets. The results of the polygon union algorithm are given separately as it runs on a dataset of polygons while the other four operations run on a dataset of points.

8.1.1 Polygon Union

Figure 10(a) gives the total processing time for the polygon union operation while varying input size. Subsets of different sizes are extracted from the OSM1 dataset by retrieving polygons from regions of different sizes to obtain datasets of sizes 250MB, 1GB, 4GB and 10GB. As shown in Figure 10(a), the single machine polygon union algorithm does not scale and quickly fails for large datasets with an out of memory exception. Although the 4GB dataset fits in memory, the algorithm uses internal data structures that require more memory causing the program to crash. CG_Hadoop algorithms scale better as the workload is distributed in the cluster saving both computation and memory overhead. In addition, CG_Hadoop is much better when it runs in SpatialHadoop due to the spatial partitioning that speeds up local and global union steps. As described in Section 3.2, spatial partitioning helps reducing the size of intermediate data (i.e., local union output) which affects the overall performance of the algorithm as shown.

8.1.2 Other Operations

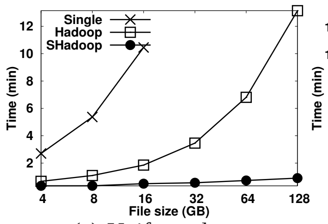
Figure 10(b) shows the results of the different operations running on the OSM2 dataset. The results provide a clear evidence that CG_Hadoop outperforms traditional techniques by orders of magnitude. CG_Hadoop running on SpatialHadoop is represented by a solid bar in the figure but it might be hard to see as the processing times are very small compared to the single machine algorithm. For both skyline and convex hull, CG_Hadoop achieves an average of 8X and 80X speedup when running on Hadoop and SpatialHadoop, respectively. Farthest pair is carried out by computing the convex hull first, then applying the rotating calipers method on the hull which is the preferred method when the size of the convex is small. This causes the running time of the farthest pair and convex hull to be very similar as the rotating calipers algorithm usually takes a fraction of a second for this small convex hull. Later on, we show the results of farthest pair experiments when the convex hull is too large for this method. Finally, for closest pair, only the result of CG_Hadoop on SpatialHadoop is shown as the single machine algorithm failed to run due to an out of memory exception.

8.2 Synthetic Data

In this section we give more detailed results for each operation separately using generated data. Polygon union is not studied using synthetic data as it requires a more sophisticated generator for polygons which goes beyond the scope of this paper. We show the results of the four other operations *skyline*, *convex hull*, *farthest pair*, and *closest pair*. We vary the generated data size from 4 GB to 128GB and the generated data distribution as shown in Figure 8.

8.2.1 Skyline

Figure 9 shows the performance of the skyline operation for both single machine algorithm and CG_Hadoop. The single machine algorithm reads the input points one by one and whenever the physical memory buffer fills up, it runs the skyline algorithm to reduce used buffer size. This allows the algorithm to handle data with arbitrarily large sizes. Although, the single machine algorithm was able to finish all the experiments, some results are omitted from the figures to adjust its scale. When CG_Hadoop is deployed in standard Hadoop, it achieves up to an order of magnitude performance gain due to the parallelization of the computation over the machines in the cluster. The local skyline step is very efficient in pruning most of the points leaving only a little work to be done in the global skyline step. CG_Hadoop can achieve up to two orders of magnitude performance gain when deployed in SpatialHadoop. This performance boost is due to the filter step which prunes partitions that do not contribute to answer minimizing the total number of processed blocks.


(a) Uniform data

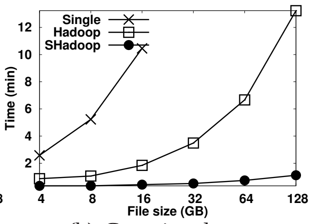
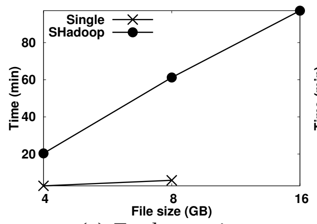

(b) Gaussian data

Figure 11: Convex Hull on SYNTH dataset


(a) Farthest pair

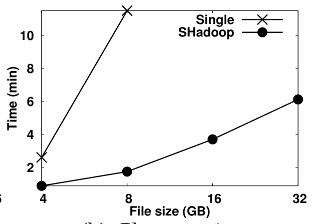

(b) Closest pair

Figure 12: Farthest/Closest pair experiments

8.2.2 Convex Hull

Processing times of convex hull algorithms are shown in Figure 11. The convex hull algorithm reads the input points one by one and whenever the memory buffer is full, it runs an iteration of the convex hull algorithm and keeps only the result which keeps memory usage limited. The convex hull algorithm in CG_Hadoop described in Section 5.1 runs much faster than the single machine algorithm as the hull is computed through distributed processing in the cluster. CG_Hadoop is even more efficient in SpatialHadoop as the filter step allows it to minimize the total processing by early pruning of partitions that do not contribute to answer. Although not shown here for clarity of the figure, CG_Hadoop achieves 260X speedup for the 128GB dataset when deployed in SpatialHadoop compared to the traditional system.

8.2.3 Farthest Pair

There are two techniques to compute the farthest pair in CG_Hadoop. The first one is to compute the convex hull followed by the rotating calipers algorithm [33] which is only applicable when the size of the convex hull is limited. We use this technique for single machine experiments. The second technique is the modified brute force approach described in Section 6.2. Figure 12(a) compares the performance of the two approaches for different input sizes. We use a generated *circular* dataset like the one in Figure 8(e) to get a very large convex hull. As shown, the first technique is more efficient whenever it is applicable as it requires a single scan around the convex hull. However, it fails when the data size exceeds main memory capacity as the convex hull becomes extremely large. On the other hand, the modified brute force approach in CG_Hadoop is less efficient since it requires a lot of distance computations between points to select the pair with maximum distance. However, it has a scalability advantage as it requires a very small memory footprint compared to the single machine algorithm. The recommendation is that the modified brute force should be used only when the rotating calipers method is not applicable.

8.2.4 Closest Pair

Results of the closest pair experiments are shown in Figure 12(b) for different input sizes. The traditional single machine algorithm cannot scale to large files since it has to load the whole dataset in memory first. In the show experiments, the traditional algorithm fails when the input size reaches 16GB. CG_Hadoop achieves much better performance for two reasons. First, the closest pair computation is parallelized on cluster machines which speeds up the whole algorithm. Second, each machine prunes many points that no longer need to be considered for closest pair. As shown, CG_Hadoop is much more scalable and it does not suffer from memory problems because each machine deals with only one partition at a time limiting the required memory usage to a block size.

9. RELATED WORK

The use of MapReduce in the computational geometry field was only discussed from a theoretical perspective [15] to suggest simulating Bulk-Synchronous Parallel (BSP) in MapReduce and use it to solve some computational geometry problems such as convex hull. However, no actual implementations were provided and it was not shown how to implement other algorithms that do not follow the BSP model. To the best of our knowledge, our work in CG_Hadoop is the first to provide detailed MapReduce implementations of various computational geometry problems. In the meantime, there is increasing recent interest in taking advantage of Hadoop to support spatial operations. Existing approaches for supporting spatial data in MapReduce can be classified into two broad categories: (1) Solving a specific spatial operation, and (2) Providing a framework for spatial data.

Specific spatial operations. Existing work in this category has mainly focused on implementing specific spatial operations as MapReduce jobs in Hadoop. Examples of this work has focused on R-tree building [8], range queries over spatial points [38], range queries over trajectory data [24], k-nearest-neighbor (kNN) queries [2, 38], all nearest-neighbor (ANN) queries [36], reverse nearest-neighbor (RNN) queries [2], spatial join [38], exact kNN Join [23] and approximate kNN Join [37].

Unified framework for spatial operations. There exist four recent systems that are built for various spatial operations: (1) Hadoop-GIS [1] is a spatial data warehousing system focusing on processing medical data. (2) Parallel-Secondo [22] is a parallel spatial DBMS which uses Hadoop as a distributed task scheduler, while all storage and spatial query processing are done by spatial DBMS instances running on cluster nodes. (3) $\mathcal{MD}$-HBase [27] extends HBase to support multidimensional indexes which allows for efficient retrieval of points using range and kNN queries. (4) SpatialHadoop [12] extends Hadoop with grid file and R-tree indexes and provides new MapReduce components that allow using the indexes in spatial MapReduce programs.

Our work in this paper, CG_Hadoop, lies in between the above two categories. First, it does not focus on only one specific spatial operation. Instead, it covers five different and fundamental computational geometry spatial operations. Second, it does not provide a new system. Instead, it provides efficient implementations of various computational geometry algorithms within SpatialHadoop, which achieves high performance using the provided spatial indexes. Overall, CG_Hadoop forms a nucleus of a comprehensive MapReduce library of computational geometry operations. Its open-source nature will act as a research vehicle for other researchers to build more computational geometry algorithms that take advantage of the MapReduce paradigm.

10. CONCLUSION

This paper has introduced CG_Hadoop; a suite of scalable and efficient MapReduce algorithms for various fundamental computational geometry operations, namely, *polygon union*, *skyline*, *convex hull*, *farthest pair*, and *closest pair*. For each operation, CG_Hadoop has two versions; one for the Apache Hadoop system and one for the SpatialHadoop system. All algorithms in CG_Hadoop deploys a form of divide-and-conquer method that leverages the distributed parallel environment in both Hadoop and SpatialHadoop, and hence achieves much better performance than their corresponding traditional algorithms. In the meantime, SpatialHadoop algorithms significantly outperform Hadoop algorithms as they take advantage of the spatial indexing and components within SpatialHadoop. Overall, CG_Hadoop forms a nucleus of a comprehensive MapReduce library of computational geometry operations. Extensive experimental results on a cluster of 25 machines of datasets up to 128GB show that CG_Hadoop achieves up to 29x and 260x better performance than traditional algorithms when using Hadoop and SpatialHadoop systems, respectively.

11. REFERENCES

[1] 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.

[2] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-based Geospatial Query Processing with MapReduce. In *CLOUDCOM*, Nov. 2010.

[3] A. M. Andrew. Another Efficient Algorithm for Convex Hulls in Two Dimensions. *Information Processing Letters*, 9(5), 1979.

[4] J. L. Bentley, H. Kung, M. Schkolnick, and C. D. Thompson. On the Average Number of Maxima in a Set of Vectors and Applications. *Journal of the ACM (JACM)*, 25(4), 1978.

[5] M. D. Berg, O. Cheong, M. V. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2008.

[6] K. D. Borne, S. A. Baum, A. Fruchter, and K. S. Long. The Hubble Space Telescope Data Archive. In *Astronomical Data Analysis Software and Systems IV*, volume 77, Sept. 1995.

[7] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, Apr. 2001.

[8] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on Processing Spatial Data with MapReduce. In *SSDBM*, June 2009.

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2), 2008.

[10] K. Dalal. Counting the Onion. *Random Structures & Algorithms*, 24(2), 2004.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of ACM*, 51, 2008.

[12] A. Eldawy and M. F. Mokbel. A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data. In *VLDB*, 2013.

[13] 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.

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

[15] M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, Searching, and Simulation in the MapReduce Framework. In *ISAAC*, Dec. 2011.

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

[17] Apache. Hadoop. `http://hadoop.apache.org/`.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, Mar. 2007.

[19] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *Operating Systems Review*, 44(2), 2010.

[20] G. Lee, J. Lin, C. Liu, A. Lorek, and D. V. Ryaboy. The Unified Logging Infrastructure for Data Analytics at Twitter. *PVLDB*, 5(12), 2012.

[21] H. Liao, J. Han, and J. Fang. Multi-dimensional Index on Hadoop Distributed File System. *ICNAS*, 0, 2010.

[22] J. Lu and R. H. Guting. Parallel Secondo: Boosting Database Engines with Hadoop. In *ICPADS*, Dec. 2012.

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

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

[25] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*, volume 54. Prentice-Hall Englewood Cliffs, 1994.

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

[27] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *MDM*, June 2011.

[28] D. Oliver and D. J. Steinberger. From Geography to Medicine: Exploring Innerspace via Spatial and Temporal Databases. In *SSTD*, 2011.

[29] O. O'Malley. Terabyte Sort on Apache Hadoop. *Yahoo!*, 2008.

[30] OpenStreetMaps. http://www.openstreetmap.org/.

[31] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1), 2005.

[32] PostGIS. Spatial and Geographic Objects for PostgreSQL. `http://postgis.net/`.

[33] F. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[34] M. I. Shamos. *Computational geometry*. PhD thesis, Yale University, 1978.

[35] F. Tauheed, L. Biveinis, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki. Accelerating range queries for brain simulations. In *ICDE*, Apr. 2012.

[36] K. Wang, J. Han, B. Tu, J. D. amd Wei Zhou, and X. Song. Accelerating Spatial Data Processing with MapReduce. In *ICPADS*, Dec. 2010.

[37] C. Zhang, F. Li, and J. Jestes. Efficient Parallel kNN Joins for Large Data in MapReduce. In *EDBT*, Mar. 2012.

[38] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. Spatial Queries Evaluation with MapReduce. In *GCC*, Aug. 2009.

Algorithm 1 Polygon union operation in Hadoop/SpatialHadoop

```
 1: Load the input file using Hadoop/SpatialHadoop file loader
 2: function MAP($P$: Polygon)                    ▷ Identity map function
 3:     output⇐ $\langle 1, p \rangle$
 4: end function
 5: function COMBINE, REDUCE(1, $P$ : Set of polygons)
 6:     Compute the union of the set of polygons $P$
 7:     for Each polygon $p$ in the union do
 8:         output⇐ $\langle 1, p \rangle$
 9:     end for
10: end function
```

APPENDIX

A. MAPREDUCE PSEUDOCODE

This appendix gives the pseudocode for CG_Hadoop algorithms, written in a MapReduce programming paradigm. For more information about the MapReduce programming paradigm, please refer to [11].

A.1 Polygon Union

Algorithm 1 gives the pseudocode of the polygon union operation for both Hadoop and SpatialHadoop. Line 1 loads the input file into the cluster using either Hadoop or SpatialHadoop loader. The map function (lines 2-4) emits each polygon with a constant key. This ensures that all polygons are sent to a single reducer that computes the union of all of them. The local union step is implemented as a combine function (lines 5-10), which computes the union of all polygons locally in each machine. The same function is used as a reduce function that computes the union of all polygons in a

Algorithm 2 Skyline in Hadoop/SpatialHadoop

```
 1: Load the input file using Hadoop/SpatialHadoop file loader
 2: if File is spatially indexed then
 3:     function FILTER($C$: Set of cells)
 4:         Initialize the set $S$ of selected cells to {}
 5:         for each cell $c$ in $C$ do
 6:             if $c$ is not dominated by any cell in $S$ then
 7:                 Add $c$ to $S$
 8:                 Remove all cells $S$ dominated by $c$
 9:             end if
10:         end for
11:     end function
12: end if
13: function MAP($p$: Point)                          ▷ Identity map function
14:     output⇐ $\langle 1, p\rangle$
15: end function
16: function COMBINE, REDUCE(1, $P$: Set of points)
17:     Apply skyline to $P$ to find non-dominated points
18:     for each non-dominated point $p$ do
19:         output⇐ $\langle 1, p\rangle$
20:     end for
21: end function
```

Algorithm 3 Convex hull in Hadoop/SpatialHadoop

```
 1: Load the input file using Hadoop/SpatialHadoop file loader
 2: if File is spatially indexed then
 3:     function FILTER($C$: Set of cells)
 4:         Initialize the set $S$ of selected cells to {}
 5:         for each of the four skylines do
 6:             Apply the skyline filter function to select a subset of $C$
 7:             Add all selected cells to $S$
 8:         end for
 9:     end function
10: end if
11: function MAP($p$: Point)                          ▷ Identity map function
12:     output⇐ $\langle 1, p\rangle$
13: end function
14: function COMBINE, REDUCE(1, $P$: Set of points)
15:     Apply convex hull to $P$ to find points on the convex hull
16:     for each selected point $p$ do
17:         output⇐ $\langle 1, p\rangle$
18:     end for
19: end function
```

Algorithm 4 Farthest pair in SpatialHadoop

```
 1: Load the input file using SpatialHadoop file loader
 2: function FILTER($C_1$, $C_2$: Two sets of cells)
 3:     Initialize the set $S$ of selected cell pairs to {}
 4:     for each pair of cells $k_1 = \langle c_1, c_2\rangle$ in $C_1 \times C_2$ do
 5:         dominated ← false
 6:         for each pair of cells $k_2 = \langle c_3, c_4\rangle$ in $S$ do
 7:             Remove $k_2$ from $S$ if the minimum distance of $k_1 \geq$ the maximum
    distance of $k_2$
 8:             if the minimum distance of $k_2 \geq$ the maximum distance of $k_1$ then
 9:                 Set dominated to true and break the inner loop
10:             end if
11:         end for
12:         Add $k_1$ to $S$ if not dominated
13:     end for
14:     return $S$
15: end function
16: function MAP($P_1$, $P_2$: Two sets of points)
17:     $P \leftarrow P_1 \cup P_2$
18:     Compute the convex hull of $P$ using Andrew's monotone chain algorithm
19:     Compute the farthest pair of points $p_1, p_2$ using rotating calipers method
20:     output ⇐ $\langle 1, \langle p_1, p_2\rangle\rangle$
21: end function
22: function REDUCE(1, $P$: Set of point pairs)
23:     Scan $P$ and return the pair with the largest distance
24: end function
```

Algorithm 5 Closest pair algorithm in SpatialHadoop

```
 1: Load the input file using SpatialHadoop file loader
 2: function MAP($P$: Set of points)
 3:     Compute the closest pair $\langle p, q\rangle$ of $P$ using the divide and conquer algorithm
 4:     output ⇐ $\langle 1, p\rangle$
 5:     output ⇐ $\langle 1, q\rangle$
 6:     Let $\delta$ be the distance between $p$ and $q$
 7:     Draw a buffer with size $\delta$ inside the MBR of $P$ and return all points in the
    buffer
 8: end function
 9: function REDUCE(1, $P$ : Set of points)
10:     Compute and return the closest pair $\hat{p}, \hat{q}$ of $P$ using the divide and conquer
    algorithm
11: end function
```

single reducer. Using one function as both a combine and reduce functions is typical in most MapReduce programs that employ a combine function.

A.2 Skyline

Algorithm 2 gives the pseudocode for the skyline MapReduce algorithm for both Hadoop and SpatialHadoop. Similar to the union algorithm, line 1 loads the data into the cluster using either Hadoop or SpatialHadoop loader. The filter function in lines 3-11 is applied only for SpatialHadoop where it iterates over each cell of the partitioned file and adds it to the list of selected (non-dominated) cells in lines 7 if it is not dominated by any other selected cell. When a cell is added (line 8), all previously selected cells that are dominated by the new added cell c are removed from the set of selected cells because they are no longer non-dominated. The map function in lines 13-15 emits each point with a constant key to ensure they are all reduced by one reducer. The combine function in lines 16-21 computes the local skyline and outputs each selected point. The same function is used as a reduce function to compute the global skyline.

A.3 Convex Hull

Algorithm 3 for computing the convex hull in Hadoop and SpatialHadoop is very similar to the skyline algorithm. The filter function in lines 3-9 applies the skyline filter four times and returns the union of all cells selected. The combine/reduce function in lines 14-19 computes the convex hull of a set of points and returns all points found to be on the hull.

A.4 Farthest Pair

Algorithm 4 gives the pseudocode of the farthest pair algorithm in SpatialHadoop. The file has to be loaded using the spatial file loader in line 1. Then, the filter function in lines 2-15 scans all cell pairs and returns only non-dominated pairs. Notice that unlike previous algorithms where the filter function takes a set of cells, the filter function in this algorithm takes a pair of sets. The reason is that this operation is designed as a binary operation which operates on two files. In this particular case, the two files are just the same file similar to a self join in DBMS. The filter function scans all possible pair of cells in the Cartesian product of C_1 and C_2 and tests the dominance rule devised in Section 6.2 with all previously selected pairs in S. If a previously selected (non-dominated) pair is found to be dominated, it is removed from S. If the currently tested pair is found to be not dominated by any previously selected pair, it is added to the set of selected (non-dominated) pairs. At the end, it returns all selected pairs.

The map function in lines 16-21 is called once for each selected pair of cells. Again, since this is a binary operation, the signature of the map function is a little bit different than previous examples. It takes two sets of points corresponding to all points in the two cells in a selected pair and computes the farthest pair of points in the two sets by combining them together, computing the convex hull and finally applying the rotating calipers method to get the farthest pair of points. This pair is sent to the output as a value with a constant key which ensures that all selected farthest pairs go to a single reducer. Finally, the reduce function in lines 22-24 scans the list of all pairs returned by the map phase to choose the pair with the largest distance.

A.5 Closest Pair

Algorithm 5 gives the pseudocode of the closest pair algorithm in SpatialHadoop. In line 1, the file is initially loaded using SpatialHadoop loader. No filtering is required for this operation because all cells have to be processed. The map function takes a set P of points in one cell and computes their closest pair using a traditional divide and conquer algorithm. The two points forming the closest pair are returned in lines 4 and 5. Then, all points with distance less than δ from the boundaries are also returned by the map function. All these points from all mappers are combined in one reducer to compute their closest pair using a traditional in-memory algorithm.