The VLDB Journal
https://doi.org/10.1007/s00778-018-0534-5

REGULAR PAPER

Scalable computational geometry in MapReduce

Yuan Li[2] · Ahmed Eldawy[1] · Jie Xue[2] · Nadezda Knorozova[3] · Mohamed F. Mokbel[2] · Ravi Janardan[2]

Received: 12 February 2018 / Revised: 26 August 2018 / Accepted: 7 December 2018

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 has not been exploited for processing large-scale computational geometry operations. This paper introduces CG_Hadoop; a suite of scalable and efficient MapReduce algorithms for various fundamental computational geometry operations, namely *polygon union*, *Voronoi diagram*, *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 the nucleus of a comprehensive MapReduce library of computational geometry operations. Extensive experimental results run on a cluster of 25 machines over datasets of size up to 3.8B records show that CG_Hadoop achieves up to 14x and 115x better performance than traditional algorithms when using Hadoop and SpatialHadoop systems, respectively.

Keywords Computational Geometry · MapReduce · Hadoop · Output-sensitive Algorithms · Distributed Systems

1 Introduction

Hadoop [4] is a framework designed to efficiently process huge amounts of data in a distributed fashion. It employs the MapReduce programming paradigm [14], 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 produces the final answer. The simplicity and flexibility of the MapReduce paradigm allow Hadoop to be employed in several large-scale applications including machine learning [21], tera-byte sorting [41], and graph processing [22].

In recent years, there has been a tremendous increase in devices and applications that generate enormous rates of spatial data. Examples of such devices include smart phones, space telescopes [7], and medical devices [40,49]. The enormous volume of such big spatial data calls points to the need to take advantage of the MapReduce programming paradigm [14] to perform various spatial operations efficiently. 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 Voronoi diagram, convex hull, skyline, polygon union, and farthest/closest pairs. Although there exist well established computational geometry algorithms for such problems [6,46], unfortunately, such algorithms do not scale well to handle modern spatial datasets which can contain, for instance, billions of points. For example, computing a skyline for a dataset of 4B points using a traditional algorithm took up to 90 min, while computing the union of a dataset of 5M polygons took more than 1 h and failed with a memory exception for larger datasets.

✉ Mohamed F. Mokbel
mokbel@umn.edu

Yuan Li
lixx2100@umn.edu

Ahmed Eldawy
eldawy@ucr.edu

Jie Xue
xuexx193@umn.edu

Nadezda Knorozova
knoro002@umn.edu

Ravi Janardan
janardan@umn.edu

1 University of California, Riverside, Riverside, USA

2 University of Minnesota, Twin Cities, Minneapolis, USA

3 Oxford University, Oxford, UK

Published online: 16 January 2019

In this paper, we introduce CG_Hadoop, a suite of scalable and efficient MapReduce algorithms for various fundamental computational geometry operations. CG_Hadoop proposes a generic *skeletal* algorithm that describes how several computational geometry operations can be implemented in MapReduce. After that, it describes how to use this skeleton to implement six computational geometry operations, namely *polygon union*, *Voronoi diagram*, *skyline*, *convex hull*, *farthest pair*, and *closest pair*, which present a set of key components for other geometric algorithms [6,46]. CG_Hadoop achieves orders 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 [4], an open-source MapReduce platform, which is widely used in various MapReduce applications, e.g., see [11,21,22,30,31,41]. The second version of CG_Hadoop is deployed on SpatialHadoop [18], 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 on multiple nodes in a computational machine cluster. However, CG_Hadoop has to adapt traditional computational algorithms to work better in the MapReduce environment through three fundamental changes. (1) Unlike traditional single machine algorithms which usually divide the input in half and do multiple rounds, CG_Hadoop has to use more scalable partitioning techniques [15] and adapt the algorithms correspondingly. (2) While traditional divide-and-conquer algorithms process *all* the data, CG_Hadoop introduces a *pruning* step which early prunes partitions that do not contribute to the final answer. (3) In traditional algorithms, a single machine produces all the output in the final merge step, whereas CG_Hadoop adds a *pruning* step which early flushes parts of the output to keep the final merge step efficient.

CG_Hadoop forms the nucleus of a comprehensive MapReduce library of computational geometry (CG) operations. The source code is available as part of SpatialHadoop at http://spatialhadoop.cs.umn.edu/. Its open-source nature will act as a research vehicle for other researchers to build more CG algorithms that take advantage of the MapReduce programming paradigm. An earlier version of this work [16] had a major limitation since the output had to fit in a single machine which ran the final merge step. We overcome this limitation by adding new *output-sensitive* algorithms that utilize a new *pruning* step to early flush parts of the output in a distributed manner. We use this technique to enhance the *polygon union*, *skyline*, *convex hull* operations and add a novel algorithm for the new *Voronoi diagram* construction operation. In addition, we improve the *farthest pair* operation by adding a more effective pruning step. Extensive experiments on a cluster of 25 machines using both real and generated datasets of sizes up to 3.8 billion records show that CG_Hadoop achieves up to 14x and 115x 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. Section 3 describes the general skeleton of CG_Hadoop algorithms. The MapReduce algorithms for the polygon union, Voronoi diagram, skyline, convex hull, farthest pair, and closest pair operations are given in Sects. 4–9. Section 10 gives an experimental evaluation. Related work is discussed in Sect. 11, while Sect. 12 concludes the paper.

2 Background

This section gives a background about Hadoop [4] and SpatialHadoop [18], which are the two platforms used in CG_Hadoop. It also discusses the set of computational geometry operations incorporated in CG_Hadoop.

2.1 Hadoop

Hadoop [4] 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-data 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, wherein the file is split into chunks of 64 MB (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 the data in files.

A MapReduce program [14] 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 into splits and assigns each split to a slave node for 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 players, e.g., Google [14], Yahoo! [11], Microsoft [27], Facebook [26,30], and Twitter [31]. It has also been employed widely in several large-scale applications including machine learning [21], tera-byte sorting [41], and graph processing [22].

2.2 SpatialHadoop

SpatialHadoop (or SHadoop for simplicity) [17,18] is a comprehensive extension to Hadoop that enables efficient processing of spatial operations. Mainly, it provides a two-layered spatial index in the Hadoop indexing layer with implementations of uniform grid, R-tree [25], Quad-tree [47], K-d tree, and other indexes [15]. 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. Specifically, 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 are determined through a user-defined *filter* function which is provided as part of the MapReduce program.

2.3 Spatial partitioning techniques in HDFS

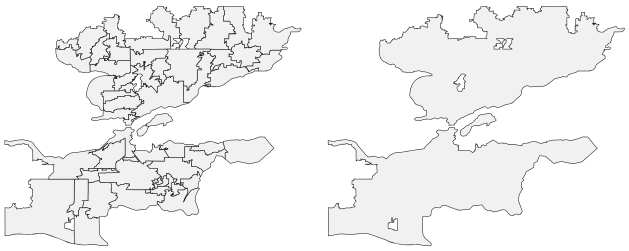
SpatialHadoop and other big spatial data systems support a wide range of spatial partitioning techniques [1,15,19,34,38,52] based on grid, R-tree, R+-tree, Quad-tree, K-d tree, Z-curve, and Hilbert curve. Table 1 summarizes the spatial partitioning techniques that we consider in this paper. It also indicates which indexes produce disjoint partitions as some of the algorithms we describe in this paper only work with disjoint indexes. If the partitioned dataset contains only points, STR and STR+ indexes become similar as no records will need to be replicated to multiple partitions. All the indexes can work with skewed data except for the uniform grid technique. For the six operations described later, we primarily consider the uniform grid index when describing the main algorithm (for simplicity). Each operation is followed by a section that describes how to generalize the algorithm to work with other partitioning techniques.

2.4 Computational geometry operations

As indicated earlier, CG_Hadoop forms the nucleus of a comprehensive MapReduce library of computational geometry operations. Currently, CG_Hadoop includes six fundamental

Table 1 Partitioning techniques in SpatialHadoop

Partitioning	Disjoint
Grid	✓
Quad-tree	✓
STR	Only with points
STR+	✓
K-d tree	✓
Z-curve	
Hilbert curve	



(a) Input Polygons **(b)** Polygon Union

Fig. 1 Union operation in CG_Hadoop

operations, namely *Union*, *Voronoi Diagram*, *Skyline*, *Convex Hull*, *Farthest pair*, and *Closest Pair*. Below, we give a brief definition of each operation.

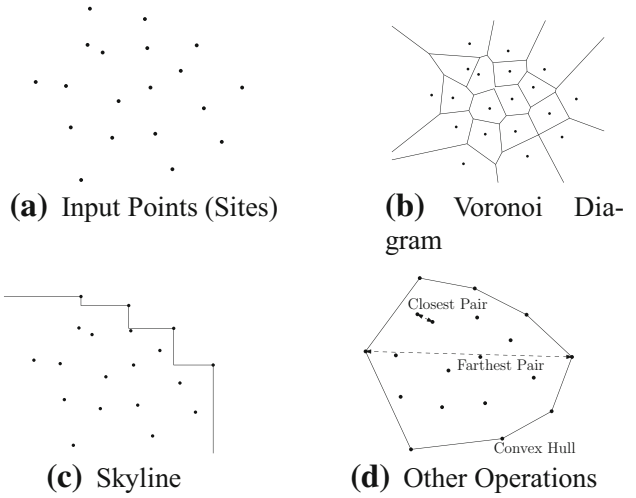
Union The union of a set S of polygons is the set of all such points that lie in at least one of the polygons in S, where only the perimeter of all points is kept and inner segments are removed. Figure 1a gives a sample input to the polygon union operation as a set of ZIP code areas, while Fig. 1b gives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

Voronoi diagram The Voronoi diagram of a set P of points, also called sites, is a tessellation of the space into regions, each associated with a site, such that any point inside each region is closer to the associated site than to any other site. For example, the Voronoi diagram of the sites in Fig. 2a is shown in Fig. 2b.

Skyline Consider the set P of points in Fig. 2a. 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 (see Fig. 2c). In the computational geometry literature, the skyline points are usually called maximal points [46].

Convex hull The convex hull of a set P of points is the smallest convex polygon that contains all the points in P, as shown in Fig. 2d. The output of the convex hull operation is the set of points forming the hull in clockwise order.

Farthest pair Given a set P of points, the farthest pair is a pair of points at the largest Euclidean distance from each other. As shown in Fig. 2d, the two points contributing to the farthest pair have to lie on the convex hull.



(a) Input Points (Sites) **(b)** Voronoi Diagram **(c)** Skyline **(d)** Other Operations

Fig. 2 Computational geometry operations covered by CG_Hadoop

Closest pair Given a set P of points, the closest pair is a pair of points at the smallest L_2-distance from each other (Fig. 2d).

3 Generic MapReduce computational geometry framework

This section describes the general idea that CG_Hadoop uses to implement all the scalable MapReduce algorithms for the above computational geometry operations in MapReduce. The next sections will describe how this skeletal framework is used to implement each of the computational geometry operations supported by CG_Hadoop. The basic idea behind the generic framework is the divide-and-conquer (D&C) approach which partitions the input data into several partitions, processes each one independently, and then combines the results to produce the final answer. The D&C approach lends itself to the MapReduce programming paradigm as *mappers* process partitions in parallel while *reducers* merge the results of the mappers. Unfortunately, the standard divide-and-conquer algorithms would perform poorly if applied as-is in MapReduce due to the following limitations.

(1) Standard D&C algorithms typically run in $\log n$ iterations to minimize the computation time. A straightforward MapReduce implementation would run in $\log n$ rounds, i.e., MapReduce jobs, which is extremely inefficient. In MapReduce, there is a significant overhead in starting each job and the goal is to minimize the number of rounds to achieve the best performance [23]. We overcome this challenge by employing efficient spatial partitioning techniques [15] which allow the algorithms to run in *only one* MapReduce round.
(2) Traditional algorithms process *all* the input data under the assumption that all the data are already in main memory. In MapReduce, data is partitioned in blocks among multiple machines and there is an overhead of accessing each partition. Applying the traditional algorithm as-is would *read* every partition which adds a significant overhead. CG_Hadoop overcomes this limitation by employing an appropriate *filtering* step for some algorithms which can early eliminate partitions that do not contribute to the final answer.
(3) In the traditional D&C algorithms, the final output is *all* produced by a single process. In MapReduce, this means that all the output is generated by a single machine that runs the final merge step. If the output is too large, that last reducer will always fail. For example, the Voronoi Diagram and Delaunay Triangulation algorithms *always* produce an output that is several times larger than the input. CG_Hadoop resolves this issue by introducing a *pruning* step that allows the processes to early flush parts of the output leaving only a small amount of data to be processed in the final step. This part is one of the new contributions introduced in this paper as compared to our earlier work [16].

Based on the three key ideas above, we propose the following five-step framework to handle scalable computational geometry operations in MapReduce. In the rest of this section, we describe the high-level idea of the *generic* idea. In the following sections, we show how this generic idea is applied to the six computational geometry operations. The framework has five steps, *partitioning*, *filtering*, *local processing*, *pruning*, and *merging*, which are described briefly below.

1. *Partitioning* In this step, the input is partitioned into fixed-size blocks with a default size of 64 MB each. The goal is to allow each machine to process these blocks in parallel and ensure that each one will fit in the main memory during processing. The challenge in this step is to find the best partitioning technique that minimizes the processing time. We consider both non-spatial partitioning that is supported by default in Hadoop and several spatial partitioning techniques that are introduced in Spatial-Hadoop [15].
2. *Filtering* This *optional* step early filters out some partitions that do not contribute to the final answer. This step is applicable only when a spatial partitioning technique is used as the *filtering logic* relies on the partition boundaries. In addition, it is only applicable to some algorithms that can eliminate chunks of the input without affecting the final answer, e.g., skyline and convex hull.
3. *Local processing* In this step, the mappers run in parallel to process all the blocks that were produced by the *partitioning* step and not eliminated by the *pruning* step. As implied by its name, this step requires only *local* processing without any communication across different

Table 2 Summary of the six operations supported by CG_Hadoop w.r.t. the five steps discussed in Sect. 3

	Partitioning	Filtering	Local process	Pruning	Merging
Hadoop union	Any	–	Polygon union	–	Polygon union
SHadoop union	Any spatial				
Enhanced union	Disjoint spatial			Prune segments outside partition MBR	–
Hadoop VD [2]	On x-axis	–	VD	–	VD merge
SHadoop VD	Disjoint spatial			Flush safe VD regions	
Hadoop skyline	Any	–	Skyline	–	Skyline
SHadoop skyline	Any spatial	Filter partitions that are dominated by other partitions			
Enhanced skyline	Disjoint spatial			Prune points that are dominated by the global SKY	–
Hadoop CH	Any	–	Convex hull	–	Convex hull
SHadoop CH	Any spatial	Filter partitions that are dominated by the four possible skylines			
Enhanced CH	Disjoint spatial			Prune partitions based on Theorem 3	
Closest pair	Disjoint spatial	–	Closest pair	Prune all points within a buffer of size δ	Closest pair
Farthest pair	Any spatial	Prune any pair of partitions with a maximum distance less than the minimum distance of another pair	Farthest pair	–	Select max

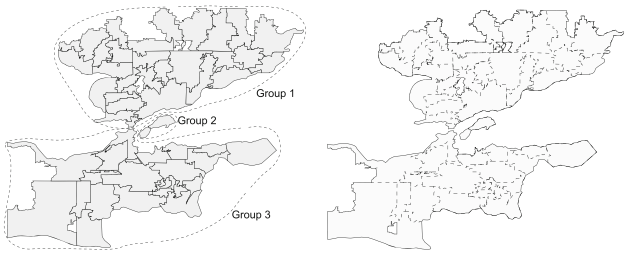
machines, which allows this step to scale out perfectly on the available processing nodes.

4. *Pruning* In this *optional* step, each machine identifies parts of the output of the *local processing* step that are not needed for the next *merging* step and prunes them. The pruned parts are either written to disk as part of the output or are removed if they are no longer needed. The goal of this step is to minimize the size of the data that will be processed in the final *merging* step which usually runs on a single machine.
5. *Merging* This final step runs in the *reduce* function and takes the output of the previous step, which is produced by several machines, and combines them together to produce the final answer. If the output of all the machines is small enough to be processed by a single machine, this step can run on one machine. Otherwise, this step runs in several rounds, where each round runs on several machines with the goal of reducing the data size until it can fit on a single machine. In our experiments, we found that two rounds are enough in practice to process all the inputs we were testing. This allows us to run the first round in parallel in the *reduce* function and the second round as a *post-processing* step on a single machine.

Table 2 summarizes the operations introduced in CG_Hadoop and shows how they are all implemented in terms of the five functions described above. The rest of this paper will describe each of these operations in more details.

Application to different partitioning techniques This part describes how the proposed idea applies to different types of partitions. There are three types of partitioning techniques that we consider in this paper, namely non-spatial partitioning, overlapping spatial partitioning, and spatial disjoint partitioning. (1) Non-spatial partitioning is the default one used in Hadoop and it is partitioning the data without considering the spatial location, e.g., hash partitioning. (2) The overlapping spatial partitioning assigns each record to one partition based on its spatial attribute but the partitions might overlap, especially when partitioning polygons, e.g., STR and Z-curve-based partitioning. (3) The disjoint spatial partitioning produces non-overlapping partitions at the cost of replicating some records to multiple partitions, e.g., quad-tree-based partitioning. Table 1 indicates the spatial partitioning techniques supported by SpatialHadoop [15] and shows which ones are disjoint.

Some of the algorithms we propose in this paper require a specific type of partitioning technique in order to work. The second column in Table 2 indicates the type of partitioning technique that is required by each algorithm. We can classify the algorithms based on their applicability to different partitioning techniques into three classes. (1) The first class of algorithms can work with the three types of partitioning techniques described above. These algorithms are denoted in the second column as *any*. (2) The second class of algorithms requires the use of a spatial partitioning technique (either overlapping or disjoint) to work, and these are denoted as *any*



(a) Grouping step **(b)** Merging step

Fig. 3 Polygon union on a single machine

spatial in the table. These algorithms typically apply a filtering step and use the minimum bounding rectangle (MBR) of the partitions in the filtering step. (3) The third class of algorithms requires a disjoint spatial partitioning technique to work, and they are denoted as *disjoint spatial*. These algorithms typically apply a pruning step to early produce a partial result and require disjoint partitions to ensure that the partial result will not be affected by other overlapping partitions.

4 Union

A traditional algorithm for the polygon union operation [6] 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 [45], 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 four polygon union algorithms as one for a single machine, one for Hadoop and two for SpatialHadoop. We use the input dataset in Fig. 1a as a running example. For ease of illustration and without loss of generality, the example has non-overlapping polygons.

4.1 Union in a single machine

In this section, we describe a simple single machine algorithm for computing the union of a set of polygons. This algorithm acts as a baseline in our experiments. It is also used as a building block in subsequent Hadoop and SpatialHadoop algorithms. The algorithm applies two simple heuristics that improve its performance with real data. The two heuristics are applied in two steps of the algorithm, namely *grouping* and *merging*.

In the *grouping* step, polygons are split into groups of overlapping polygons such that there is no overlap between two polygons in two different groups. This step is illustrated in Fig. 3a where the input polygons are split into three groups. This step breaks down the input set into smaller subsets where the union of each group can be computed independently. It also allows the use of multi-core CPUs which gains further speed-ups. To perform this grouping, we start with a forest of sets where each set includes a single polygon. Then, we carry out a *self-spatial-join* operation to find all pairs of overlapping polygons. For each overlapping pair, we union the two sets in which they are contained. We use the *disjoint-set* data structure [12] which merges two sets in almost a constant time. At the end of this step, each resulting set will contain all overlapping polygons in one group.

In the *merging* step, the polygon union of each group is computed separately. We use the popular Java Topology Suite (JTS) [28] which recommends the use of the *buffer* operation to compute the union. All polygons in one group are combined into a multi-polygon, and the buffer operation is applied to that multi-polygon to keep only the outer boundary of the union result and remove all internal segments. Figure 3b illustrates the merging step where internal line segments, which are removed, are marked in dotted lines.

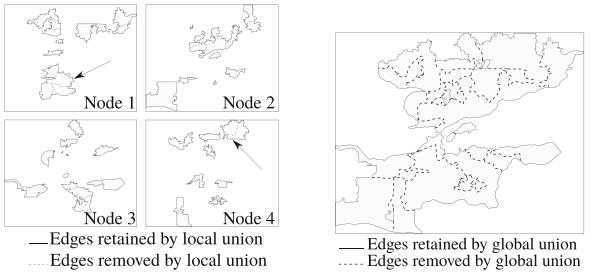
4.2 Union in Hadoop

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 *merging*, as detailed below.

The *partitioning* step distributes the input polygons into smaller subsets, each handled by a machine. This step is performed by the Hadoop `load file` command which splits the file into chunks of 64 MB, each stored on a slave node.

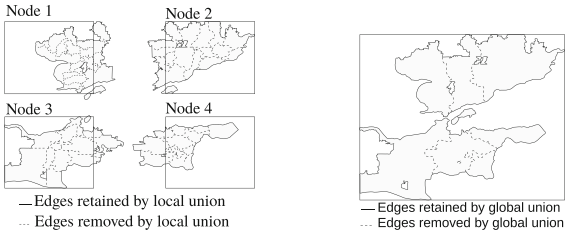
In the *local union* step, each machine computes the union of its own chunk using a traditional in-memory polygon union algorithm where it retains the line segments at the boundaries and removes internal line segments (see Fig. 4a). As each chunk is at most of size 64 MB, the in-memory algorithm works fine regardless of the size of the input file. This step is implemented in Hadoop as part of the mapper which runs locally in each machine. After this step is done, each machine ends up with a set of polygons that represent the union of all polygons assigned to it.

The final *merging* 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 ends up with only few polygons, making it possible to do the union using the in-memory algorithm.



(a) Hadoop partitioning and local union **(b)** Merging

Fig. 4 Polygon union in Hadoop



(a) SHadoop partitioning and local union **(b)** Global union

Fig. 5 Polygon union in SpatialHadoop

By taking advantage of a set of parallel machines, rather than performing all the work in a single machine, our proposed algorithm achieves significant performance improvement over that of traditional single machine 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 time saved over parallel machines, which can be seen in large-scale spatial datasets. As shown in Fig. 4a, the local union step removes some line segments which reduces the work needed at the global union step. The pseudo-code of the polygon union algorithm in Hadoop is described in Sect. 4.3.

Figure 4a gives the partitioning and local union steps of the input dataset of Fig. 1a over four cluster computing nodes, where each polygon is assigned to one of the four nodes. The decision of which node belongs to which partition is made by the default HDFS partitioner, 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, the local union algorithm combines them in one *multi-polygon* record and writes it to the output. Then, all nodes send their output to a single machine which runs the *merging* step as shown in Fig. 4b which combines all the multi-polygons generated by the local union step and computes the union of all of them.

4.3 Union in SpatialHadoop

Our first-cut 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 uses a spatial partitioning rather than the default non-spatial partitioning as depicted in Fig. 5, where adjacent polygons are assigned to the same machine. The main advantage here is that when adjacent polygons are processed by the same node, there is a higher chance of removing *interior* edges which produces simpler and smaller polygons as a result of the local union step. Any spatial partitioning technique can be used such as the STR technique [15], which produces near equi-sized partitions of 64 MB each.

In Fig. 5a, the partitioning step uses SpatialHadoop partitioning which assigns a set of nearby records to each node. This allows the local union step to remove more internal edges as shown in the figure. As a result, the global union step has only a little work to do as most of the internal line segments have already been removed by the local union step. In this particular example, the number of polygons resulting from the local union step drops from 32 polygons in Hadoop to only seven polygons in SpatialHadoop, thus making the whole algorithm significantly faster.

Algorithm 1 gives the pseudo-code 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 *local union* step is implemented as a map function (lines 2–7), which computes the union of a set of polygons and, for each polygon p in the result, it emits an intermediate pair $\langle 1, p \rangle$. Using a constant key $k = 1$ ensures that all polygons are sent to a single reducer that computes the union of all of them. The *global union* step is implemented as a reduce function (lines 8–13), which is very similar to the map function except that it writes the resulting polygons directly to the final output. It is clear from the pseudo-code that, if the local union step running as a combiner function does not reduce the size of the input, the global union step running as a reduce function will end up processing the whole input on a single machine.

4.4 Enhanced union in SpatialHadoop

Although the union algorithm mentioned above overcomes some of the limitations of the Hadoop implementation, the algorithm has a severe bottleneck in the *merging* step as it runs in the main memory of a single machine. The memory and processing overhead on that single machine can greatly limit the overall performance of the algorithm. It can even cause the algorithm to fail if the final output size is too large to fit in the main memory of a single machine.

The *enhanced union* algorithm in SpatialHadoop overcomes the above limitation by employing a novel union computation algorithm which is completely distributed and eliminates the final merging step. It adds a *pruning* step that

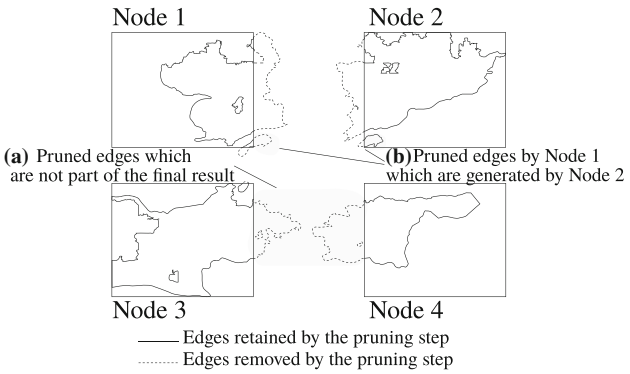


Fig. 6 Pruning step in the enhanced union algorithm in SHadoop

early detects the line segments that would be eliminated by the final *merging* step and clips them without actually running that expensive final step.

Algorithm 1 Union operation in Hadoop/SpatialHadoop

```
1: Load the input file using Hadoop/SpatialHadoop file loader
2: function MAP(k, P: Set of polygon)        ▷ The key k is not used
3:     Compute the union of the set of polygons P
4:     for Each polygon p in the union result do
5:         Emit an intermediate pair ⟨1, p⟩
6:     end for
7: end function
8: function REDUCE(1, P : Set of polygons)
9:     Compute the union of the set of polygons P
10:    for Each polygon p in the union do
11:        Write p to the output
12:    end for
13: end function
```

Figure 6 illustrates the main idea of the enhanced union algorithm which runs in three steps, *partitioning*, *local union*, and *pruning*. The first two steps are exactly the same as the regular SpatialHadoop union algorithm described in Sect. 4.3. The third step replaces the *merging* step and runs in a distributed manner as part of the map function. Thus, the enhanced algorithm does not require a reduce function.

In the *pruning* step, the result of the *local union* step is refined by clipping useless parts of the result as shown in Fig. 6. The clipped portions, shown in dotted lines, are defined as all line segments that lie outside the partition boundaries. If only a part of a line segment is outside the partition boundaries, that segment is broken at the boundaries and only the portion that is inside the boundaries is retained. The line segments that are removed by the pruning step are either (a) not part of the final result, or (b) are part of the final result but are generated by another machine. For example, in Fig. 6, the clipped parts by nodes 3 and 4 are removed as they are not part of the final answer. Referring back to Fig. 5b, if the *merging* step were applied, it would remove these parts. Also in Fig. 6, although some lines, which belong to the final answer, are clipped by Node 1, they are regenerated by Node 2 to ensure correctness. The pruning step does not identify the removed line segments as either case (a) or (b), but it applies one rule, which removes all parts that are outside the partition boundaries.

The pruning step is crucial to the enhanced union algorithm as it has two main objectives. First, it reduces the output size by removing all unnecessary line segments from the result of the union step. Second, it allows the *enhanced union* algorithm to skip the merging step as it ensures that each line segment in the answer is produced by exactly one node. While we could run a post-processing phase which stitches all line segments into one big polygon, it is not necessary and can be inapplicable if the size of the resulting polygon is too large to fit in a single machine.

Algorithm 2 gives the pseudo-code of the enhanced union algorithm in SpatialHadoop. Unlike Algorithm 1, we only use the SpatialHadoop file loader as this algorithm is not applicable to the default Hadoop loader. The two remaining steps are implemented in the *map* function, and no *reduce* function is required. In line 3, the *local union* step computes the union of all polygons $\mathcal{P}$ in one partition. For each polygon p in the union result, the pruning step in line 5 prunes p to the boundaries of the partition which is given to the map function as the key k. The result of the pruning step is directly written to the final output.

5 Voronoi diagram

Given a set S of distinct points, also called sites or generators, in the Euclidean plane, we associate all locations in that space with the closest member $s \in S$ of the point set with respect to the Euclidean distance. The result is a tessellation of the plane into a set of regions associated with members of the point set. This tessellation is called a planar Voronoi diagram generated by the point set, and the regions constituting the Voronoi diagram are called the Voronoi regions.

A traditional in-memory Voronoi diagram algorithm uses a divide-and-conquer approach [24,46]. The divide step partitions the set S into two smaller subsets S_1 and S_2 based on a vertical line and computes Voronoi diagrams for both subsets. The merge step merges the two Voronoi diagrams into a final Voronoi diagram by suitably editing Voronoi regions that are close to the cut line.

Algorithm 2 Enhanced polygon union in SpatialHadoop

```
1: Load the input file using SpatialHadoop file loader
2: function MAP(k: Rectangle, P: Set of polygon)
3:     Compute the union of the set of polygons P
4:     for Each polygon p in the union do
5:         q ← Prune p to the boundaries of the partition k
6:         Write q to the final output
7:     end for
8: end function
```

Traditional Voronoi diagram algorithms that work on a single machine fall short in processing very large datasets due to two fundamental challenges, *memory usage* and *limited processing*. The standard representation of a Voronoi diagram is approximately 29 times bigger than the original input [46] which renders all traditional algorithms limited in constructing a Voronoi diagram for terabytes of data. In addition, the limited processing capability of a single machine makes it very inefficient for building a Voronoi diagram for very large datasets.

5.1 Voronoi diagram in Hadoop

The parallel processing power of Hadoop is used to construct the Voronoi diagram (VD) more efficiently using a MapReduce job where the map phase implements the divide step over multiple machines, while the reduce phase carries out the merge step on a single machine [2]. That algorithm fits with our skeletal algorithm proposed in Sect. 3 where it runs in three steps, partitioning, local processing, and merging. The partitioning step partitions the data into vertical strips based on the x-coordinate, assuming that the input is sorted. The local processing step computes the VD for each vertical strip. Finally, the merging step combines all the local VDs into one final VD using the traditional divide-and-conquer merge strategy [24,46]. Although this algorithm can speed up the construction process to some limit, it has a bottle neck in the merging step which always runs on a single machine. Since the size of the generated Voronoi diagram is larger in size as compared to the input, the algorithm becomes inapplicable for very large input sizes.

5.2 Voronoi diagram in SpatialHadoop

In this section, we propose a novel algorithm for constructing the Voronoi diagram that overcomes the limitation of the state-of-the-art algorithm in Hadoop [2]. First, the proposed algorithm utilizes the spatial partitioning techniques in SpatialHadoop which makes the Voronoi diagram construction more efficient than one-dimensional splitting [46]. Second, we add a *pruning* step which saves the memory consumption by early flushing final parts of the partial Voronoi diagrams to the output. In other words, the algorithm early detects parts of the Voronoi diagram that will not be affected by any future merge step and writes them directly to the output. Only the portions of the Voronoi diagrams that may be needed in the merge step are transferred from mappers to reducers which greatly reduces the network overhead between mappers and reducers, as well as the memory consumption and computational overhead of the merge step. This allows our algorithm to operate efficiently on very large datasets.

Figure 7 illustrates the *pruning* technique employed in our Voronoi diagram algorithm. This figure shows four partial

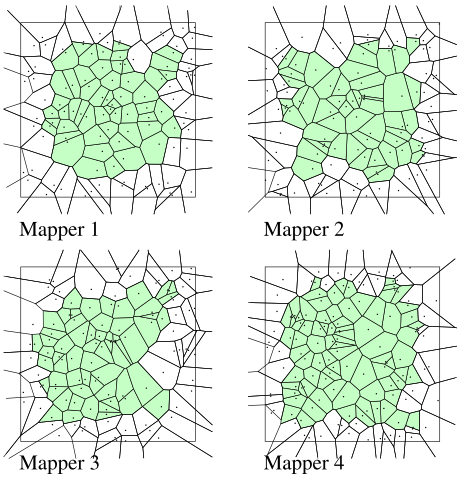


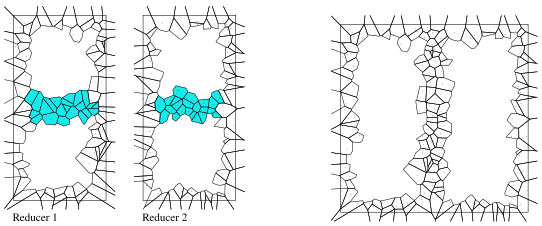
Fig. 7 Pruning safe Voronoi regions (shaded) in the local VD step

Voronoi diagrams which are constructed for four partitions of the input file. These four partitions should be merged to produce the final answer as shown in Fig. 8c. Instead of transferring the four complete partial Voronoi diagrams to one machine to merge them, each node detects the *final regions*, i.e., the ones that will not be modified by the subsequent merge operation, prunes those regions and flushes them to the final output. This means the subsequent merge steps, shown in Fig. 8a, b, process only a small fraction of the Voronoi regions.

The Voronoi diagram construction algorithm runs in four steps, namely *partitioning*, *local VD*, *pruning*, and *merging*. The *partitioning* step uses SpatialHadoop partitioner to partition the input dataset into 64 MB blocks each defined by a boundary rectangle (MBR) and is assigned all points contained in that rectangle.

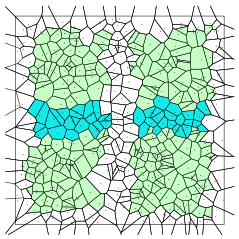
The *local VD* step constructs the VD for each partition separately using any traditional divide-and-conquer in-memory algorithm [24,46]. This step runs in the *map* phase where each mapper processes one partition of at most 64 MB, as shown in Fig. 7.

The *pruning* step applies the VD pruning rule, described later, to classify each Voronoi region as either a *final* or a *non-final* region. Final regions are removed from the VD and are directly flushed to the final output. The remaining non-final regions are transferred to the next *merging* step. This step is crucial as it allows us to reduce the amount of data transferred to the final merging step. Typically, the data structure required to represent local VDs is several times bigger than the original input. Thus, without our pruning technique, for a large dataset, it will be impossible to merge all of the sub-diagrams into the final diagram on one machine. In reality, however, for two sub-diagrams being merged together, only a small fraction of Voronoi regions, those near the partition boundaries, will actually be involved in the merge, and most



(a) Pruning safe Voronoi regions (shaded) in the vertical merge step

(b) In the horizontal merge step, all Voronoi regions are final as there is no subsequent merge step.



(c) The final answer of the Voronoi diagram operation. Green, blue and black regions are generated by the local VD, V-merge, and H-merge.

Fig. 8 Steps of computing the Voronoi diagram in SpatialHadoop

Algorithm 3 Voronoi diagram (VD) in SpatialHadoop

```
1: Load the input file using SpatialHadoop file loader
2: function MAP(k: Rectangle, S: Set of points)
3:     Compute the V(S) using the in-memory D&C algorithm [24]
4:     Use the pruning rule to find all safe regions in V(S)
5:     Remove safe regions from V(S) and write them to the output
6:     c_i ← the column number that contains the partition k
7:     Emit the intermediate key ⟨c_i, V(S)⟩
8: end function
9: function REDUCE(c_i, V: Set of partial VDs)        ▷ Vertical Merge
10:    Sort V by y-coordinate
11:    Merge partial diagram in V one by one into one VD V_{c_i}
12:    Use the pruning rule to find and output the safe regions in V_{c_i}
13:    Write V_{c_i} to the intermediate output
14: end function
15: function COMMITJOB                                ▷ Horizontal Merge
16:    Read back all written Voronoi diagrams
17:    Sort Voronoi diagrams by x and merge them into the final VD
18:    Write the resulting VD to the final output
19: end function
```

of the regions will remain unmodified. Thus, the pruning step detects the final regions that are safe from the merging step and flushes them to the output.

The *merging* step uses the regular divide-and-conquer merging technique to combine the partial VDs resulting from the pruning step to produce the final result. Although the merging step processes only small portions of the original VDs, as a result of the pruning step, the final answer will still be correct because our proposed pruning rule removes only the regions that do not affect the merging step. If there are only a few partitions, e.g., tens of partitions, the merging step can be carried our by a single machine. Otherwise, if there are hundreds or thousands of partitions, the merging step runs in two rounds, vertical merge (V-merge) and horizontal merge (H-merge.) The V-merge step merges partitions vertically, as shown in Fig. 8a and applies the pruning rule again to early flush final regions which are not affected by the next H-merge step. The H-merge step, shown in Fig. 8a, merges the vertical strips horizontally to produce the final answer. No pruning step is needed after the H-merge step because all the regions are final and there are no subsequent merge steps.

Figure 8c gives the final output of the VD operation as the combination of all final regions produced in the three steps, *local VD*, *V-merge*, and *H-merge*. In the figure, regions are color coded in either green, blue, or black according to whether they are generated by the *local VD*, *V-merge*, *H-merge*, respectively.

Algorithm 3 gives the pseudo-code of the Voronoi diagram operation in CG_Hadoop. Line 1 loads the file using SpatialHadoop loader. The *local VD* step is implemented as a map function in lines 2–8. Line 3 applies the traditional single machine divide-and-conquer algorithm to compute VD for all points in one partition. Line 4 applies the pruning rule to the computed Voronoi diagram which prunes all final Voronoi regions and writes them to the final output. Line 7 emits the remaining regions in the VD with a key c_i, which represents the id of the vertical strip that contains that partition (e.g., the column number in the grid index).

The V-merge step is implemented as a reduce function in lines 9–14. Line 10 sorts Voronoi regions in the column c_i along the y-axis according to the MBRs of the VDs. Since the MBRs are disjoint, sorting by any point in them will result in the same consistent final order. Line 11 merges the local VDs one by one by their sort order to produce one Voronoi diagram V_{c_i} associated with that column. Line 12 applies the pruning rule which detects final Voronoi regions and writes them to the final output. The remaining part of V_{c_i} is written to an intermediate output to be merged by the *H-merge* step.

The H-merge step is implemented as a CommitJob function, in lines 15–19, which the MapReduce framework calls after all reducers are done. Line 16 reads back all intermediate VDs written by the V-merge step. Line 17 sorts them by x and merges them into one final VD in sorted order. Line 18 writes the resulting VD to the final output without applying the pruning rule as there are no subsequent merge steps.

The pruning rule The Voronoi diagram pruning rule tests a *region* in the Voronoi diagram against the boundaries of the corresponding partition to determine whether the region is *safe* or not. A *safe region* will never be altered by future merge steps. The pruning rule is derived from the basic definition of the Voronoi diagram, i.e., each region covers all locations that are closer to one site than any other site.

First of all, any Voronoi region that does not lie completely inside the partition P is non-safe. This can be easily realized as another site in a neighboring partition can lie inside the

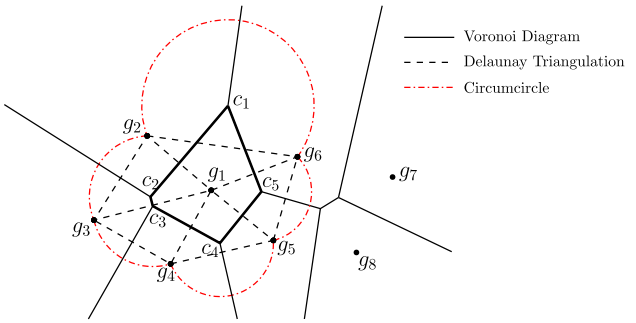


Fig. 9 Voronoi diagram pruning rule

region and the region should be modified accordingly. This, by definition, includes all open regions that are on the boundaries of the Voronoi diagram. In the next part, we are only concerned with closed regions that lie completely inside the partition boundaries.

Figure 9 shows an example of a closed Voronoi region associated with generator (site) g_1. We define a *dangerous zone* as the union of the circles that are centered at every vertex of the region (i.e., $c_1..c_5$), passing through the two neighboring sites. The pruning rule basically tests if the dangerous zone falls completely inside the boundaries of the partition P. If it lies completely inside P, then the Voronoi region is safe/final; otherwise, it is non-safe/non-final. The following theorem proves the correctness.

Theorem 1 *A closed Voronoi region is safe and will remain unchanged if and only if there are no new sites added inside the dangerous zone.*

Proof We will prove the theorem by looking at its dual version, i.e., Delaunay Triangulation. Let v be any vertex on the closed Voronoi cell (region) c of some generator g, and let g_α and g_β be the two endpoints of the arc centered at v. By the duality of Voronoi Diagram and Delaunay Triangulation, g, g_α, and g_β must form a Delaunay triangle, and the generator g must be on the circumcircle of g, g_α, and g_β. By the property of Delaunay Triangulation, there are no sites inside it. On the other hand, if there is any generator inside the circle, then g, g_α, and g_β will not form a Delaunay triangle, and consequently, the corresponding Voronoi cell c will change.

If there is a new site added inside the dangerous zone, then that site will lie inside some circumcircle, and thus, cell c must be affected; otherwise, all the involved Delaunay triangles will remain unchanged, and consequently, cell c will not change also. □

Based on Theorem 1, we have the following corollary.

Corollary 1 *A Voronoi region c associated with generator g is safe if its dangerous zone falls completely inside the boundaries of partition P where* $g \in P$.

Proof Since the partitions are disjoint and the dangerous zone falls completely inside P, no new sites can be added inside the dangerous zone and c can be declared safe. □

Notice that the condition in Corollary 1 is a necessary but not a sufficient condition for the safety of a region c. In other words, the dangerous zone might cross the boundaries of the partition P, but there might not be other sites inside the dangerous zone, and thus, c is still safe. However, the proposed algorithm uses the weaker condition because it allows each machine to work completely independently without having to check the sites stored in neighboring partitions. It is also much faster as it tests against a fixed rectangle (i.e., partition boundaries) instead of a large set of sites in neighboring partitions.

Efficient application of the pruning rule This part describes two optimization techniques that CG_Hadoop employs to apply the pruning rule more efficiently. A straightforward implementation of the pruning step is to apply the pruning rule on *every* region in the Voronoi diagram. However, this would be very inefficient given the complexity of computing the dangerous zone and the large number of Voronoi regions in a diagram. From Fig. 7, one sees that all non-safe regions are very close to partition boundaries. This means that we can significantly speed up the pruning step by searching only close to the partition boundaries. We formalize this by stating the following two simple observations:

1. All Voronoi regions overlapping the partition boundaries are non-safe.
2. Any non-safe region has to be adjacent to another non-safe region.

It is easy to prove the correctness of the first observation. The proof of the correctness of the second observation is given in Appendix A. These two observations imply that all non-safe regions form one contiguous block that intersects with the partition boundary. Thus, we can find all non-safe regions by traversing the regions, e.g., using a breadth-first search, starting from the regions that overlap the partition boundaries, and expanding the traversal only to neighboring *non-safe* boundaries. All non-visited regions automatically become safe. The list of regions to be visited is initialized with all boundary regions, i.e., the regions that overlap with the MBR of the partition. Then, for each region, the pruning rule is applied to find whether the region is safe or not. If it is safe, no further action is taken for that region. Otherwise, if the region is non-safe, all *adjacent* regions are added to the list of regions to be visited. Two regions are adjacent if they share at least one edge. As a result of applying this technique, the pruning step takes 50 msec on a diagram of 1.4M regions, where the rule is applied on only 7K regions.

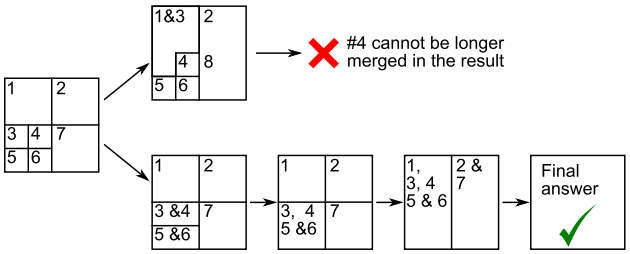


Fig. 10 Merging in Quad-tree

5.3 Application to different partitioning techniques

The proposed Voronoi diagram algorithm requires a disjoint partitioning technique for the pruning step to work under the assumption that no additional points can be within the boundaries of the partition. Furthermore, the V-merge and H-merge steps require the two merged partitions to be separable by a straight line. This can be easily enforced in both grid partitioning and STR partitioning by merging partitions vertically and then horizontally. However, other partitioning techniques require careful considerations to merge as illustrated by the two examples below.

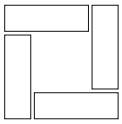
Figure 10 illustrates an example where the input contains seven partitions based on a Quad-tree. If partition 1 is merged with partition 3, for example, the result can no longer be merged with partition 4 as they are not separated by a straight line. Preferably, we should follow the merge order of the Quad-tree as shown in the figure. Simply put, two partitions are merged together only if they are siblings in the Quad-tree, e.g., partitions 3, 4, 5 and 6 are all siblings. After those four partitions are merged together, the resulting partition is a sibling of partitions 1, 2, and 7, and they can all be merged together. The same idea can be followed in a K-d tree where two partitions are only merged if they are siblings in the K-d tree.

Figure 11 gives another example of a valid R-tree-based partitioning where the partitions are disjoint. This situation is impossible to merge because merging any two partitions would result in a bigger partition that overlaps one of the two existing partitions and the merge step would fail afterward.

6 Skyline

A traditional in-memory two-dimensional skyline algorithm [46] 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

Fig. 11 A valid R-tree partition where the VD merge process fails



y order, and each one is compared to the leftmost 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 [8,43]) 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 the input dataset in Fig. 2a as an illustrative example.

6.1 Skyline in Hadoop

Our Hadoop skyline algorithm is a variation of the traditional divide-and-conquer skyline algorithm [46], 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 64 MB 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 their skyline. 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.

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, the expected number of points on the skyline is $O(\log n)$ [5]. In practice, for a partition of size 64 MB 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 a single machine that computes the final answer.

Algorithm 4 Skyline in Hadoop/SpatialHadoop

```
1: Load the input file using Hadoop/SpatialHadoop file loader
2: if File is spatially partitioned 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⇐ ⟨1, p⟩
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⇐ ⟨1, p⟩
20:     end for
21: end function
```

6.2 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 are 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 *filtering* step right before the local skyline step. The *filtering* step, which runs on the master node, takes as input the minimal bounding rectangles (MBRs) of all partitioned R-tree index cells, and filters out those cells that have no chance of contributing any point to the final skyline result.

The main idea of the new *filtering* step is that a cell c_i dominates another cell c_j if there is at least *one* (data) point in c_i that dominates *all* (data) points in c_j, in which case c_j is pruned. For example, in Fig. 12, cell c_1 is dominated by c_5 because the bottom-left corner of c_5 dominates the top-right corner of c_1. 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

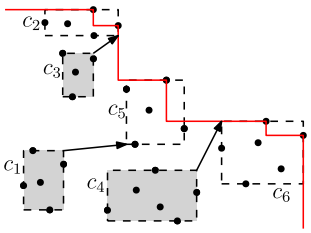


Fig. 12 Skyline in SpatialHadoop

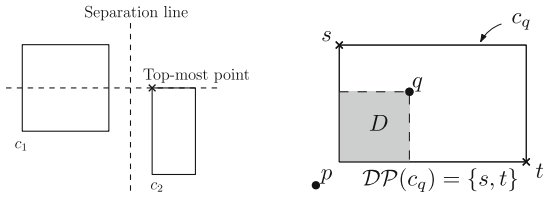
pair of cells c_i and c_j. We compare the top-right corner of c_j against three corners of c_i (bottom-left, bottom-right, and top-left). If any of these corners dominate the top-right corner of c_j, we prune c_j out from all our further computations and do not assign it to any node. Hence, we will not 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 necessarily yield 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.

Algorithm 4 gives the pseudo-code 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 filtering step 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 cells. When a cell is added (line 8), all previously selected cells that are dominated by the newly 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.

6.3 Output-sensitive skyline in SpatialHadoop

The algorithm described in Sect. 6.2 has a major limitation of producing all the output in a single machine which carries out the final merging step. The algorithm will fail if the output is too large to fit in a single machine. In this section, we introduce an *output-sensitive* skyline algorithm in SpatialHadoop that scales well even if the output is too large to fit in one machine. Basically, it relies on the spatial partitioning of the input data to locally identify the points that belong to the final output in a completely distributed setting without having to combine the partial answers in a single machine. Below, we describe a key idea, called the *dominance power* rule, of our enhanced algorithm. Then, we give a detailed



(a) Example of dominance power rule **(b)** Example of Lemma 1

Fig. 13 Illustrating the concept of dominance power

description of the algorithm. Finally, we provide a formal proof of the *dominance power* rule and the runtime analysis.

The dominance power rule Figure 13a illustrates the *dominance power* rule which is the key idea behind the output-sensitive skyline algorithm. It relies on the *disjoint* partitioning of the input set of points using one of the spatial partitioning techniques supported by SpatialHadoop. Since the partitions are enclosed in orthogonal disjoint rectangles, we can always find an orthogonal line, either horizontal or vertical, that separates any two partitions. Figure 13a shows an example of two partitions separated by a vertical line. This means that any point in c_2 dominates all points in c_1 along the x- axis, i.e., any point in c_2 has a higher value on the x-axis. Consequently, this means that the point in c_2 with the highest y value, i.e., any point along the top edge, has the highest *dominance power* over c_1 among all the points in c_2. If the MBR of c_2 is known, its top-left corner can be used as the point with the highest dominance power. This means that we need to compare the points in c_1 only to the top-left corner of MBR(c_2) to find all dominated points. We can similarly show that if the two partitions are separated by a horizontal line, the bottom-right corner of the MBR of c_2 will have the highest dominance power. This key finding means that we can abstract each partition into two points only, the top-left and right-bottom corners of the MBR, and use only those two points from all partitions to *prune* the points that are not on the final skyline. We call these two points for a cell c the dominance power set $\mathcal{DP}(c)$. We also call the skyline of all $\mathcal{DP}$ sets, the global dominance power set (SKY), i.e., $\text{SKY} = sky(\bigcup \mathcal{DP}(c_i))$.

Algorithm details Based on the dominance power rule, we propose our output-sensitive skyline algorithm which runs as a single map-only job in four steps, namely partitioning, filtering, local processing, and pruning.

In the *partitioning* step, a disjoint spatial partitioning is used to partition the data into blocks where the MBRs of the blocks are disjoint. In addition, the global dominance power set (SKY) is computed as the skyline of the top-left and bottom-right corners of the MBRs of all partitions which is then broadcast to all nodes to be used in the filtering step.

In the filtering step, the skyline filtering rule described in Sect. 6.2 is applied. Notice that in a worst-case scenario, where all input points are part of the skyline, the filtering step will not be able to filter out any partitions.

As in the SpatialHadoop algorithm described in Sect. 6.2, the *local processing* step runs in the map function and computes the local skyline of each partition using any single machine algorithm. Since the size of each partition is bounded by the HDFS block capacity, any main-memory single machine algorithm is guaranteed to work.

The *pruning* step is introduced in this output-sensitive algorithm, and it allows each machine to write a part of the final skyline without the need of an additional merge step. This step compares the points on the local skyline, computed in the local processing step, against the global dominance power set (SKY) received from the master node in the partitioning step. If any of the local points is dominated by any point in SKY, the local point is pruned as it cannot be part of the final answer. Otherwise, if the local point is not dominated by any point in SKY, the local point is directly written to the output as part of the final answer.

No *merging* step is required here as the output of the pruning step in all machines comprises the final output as shown in the following proof.

Proof of the dominance power rule First, let us start off with some definitions.

- Let p be a point, and let c_p denote the cell containing it.
- Given two points p and q, we use notion $p \prec q$ to denote that q dominates p.
- Let sky$(\cdot)$ be the skyline of a set of points. With some abuse of notation, let sky(c) be the local skyline in a cell c.
- Let $\mathcal{DP}(c)$ be the set with the highest dominance power in a cell c. It consists of the top-left and bottom-right corners of its minimum bounding rectangle (MBR). For simplicity, we write $p \prec \mathcal{DP}(c)$ if point p is dominated by at least a point in $\mathcal{DP}(c)$.

We then have the following important lemma.

Lemma 1 *If p and q are two points such that $p \prec q$ and $p \notin c_q$, then $p \prec \mathcal{DP}(c_q)$.*

Proof Let dom(p) be the dominance region of some point p, namely $\text{dom}(p) = \{p' \mid p' \prec p\}$. For a contradiction, assume that p is not dominated by any point in $\mathcal{DP}(c_q)$. Then, we must have $p \in D$, where $D = \text{dom}(q) \setminus \bigcup_{s \in \mathcal{DP}(c_q)} \text{dom}(s)$. It is clear that $D \subseteq c_q$, which implies that $p \in c_q$, contradicting the fact that $p \notin c_q$. See Fig. 13b for an example. □

Let G consists all the cells stored in SpatialHadoop, and define $\text{SKY} = \text{sky}\left(\bigcup_{c \in G} \mathcal{DP}(c)\right)$. We then have the following sufficient and necessary condition for a point to be final/non-final for outputting.

Theorem 2 *A point p is* **not** *on the final skyline of the dataset iff p is dominated by at least a point on SKY or sky(c_p).*

Proof ($\Rightarrow$) If p is dominated by a point of sky(c_p), then we are done. Otherwise, p belongs to sky(c_p), but does not belong to the final skyline. Then, p must be dominated by some other point q that is in a different cell. Since all the cells are guaranteed to be disjoint in SpatialHadoop, by Lemma 1, we have $p \prec \mathcal{DP}(c_q)$. Since both points in $\mathcal{DP}(c_q)$ are elements of SKY, or are dominated by other point(s) on SKY, p is dominated by point(s) on SKY.

($\Leftarrow$) This direction is trivial. If p is dominated by some point, it definitely cannot be on the final skyline. □

Communication cost The communication cost of the algorithm described above is $O(|G|^2)$, where $|G|$ is the total number of partitions. This can be easily proven because, in worst case, the SKY dataset can contain $2|G|$ points and the dataset is replicated to each machine and there can be up to $|G|$ machines. In Appendix B, we provide an optimization that reduces the communication cost to only $O(|G|)$ which can be a significant improvement for very large files.

7 Convex hull

The convex hull shown in Fig. 2d 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 leftmost and rightmost points. Then, the upper chain of the convex hull is computed by examining every three consecutive points p, q, r, successively, 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 continues 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 [45], 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, using the input dataset in Fig. 2a as an illustrative example.

7.1 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*, using the traditional in-memory convex hull algorithm. Similar to skyline, the number of points on the convex hull is expected to be $O(\log n)$ [13] 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.

7.2 Convex hull in SpatialHadoop

The convex hull algorithm in Hadoop processes more file partitions than necessary. Intuitively, the parts of the file that are toward 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) [46]. Here a max–min skyline is one where the maximum (resp. minimum) points in the x (resp. y) dimension are preferred. Similarly for the other skylines, this property allows us to reuse the skyline *filtering* step in Sect. 6.2. As given in Fig. 14, 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 Sect. 7.1 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.

Algorithm 5 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.

7.3 A more efficient algorithm in SpatialHadoop

In this part, we propose a novel and more efficient way to compute the convex hull in SpatialHadoop. This method fully exploits the convex hull property to prune as many points as

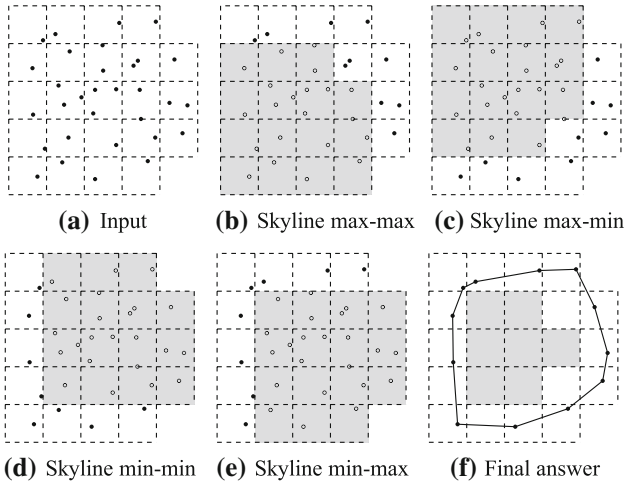


Fig. 14 Convex hull in SpatialHadoop

possible and thus has a better scalability. Our algorithm runs as a single map-reduce job in three steps, i.e., partitioning, local-pruning (map), and merging (reduce).

Algorithm 5 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⇐ ⟨1, p⟩
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⇐ ⟨1, p⟩
18:     end for
19: end function
```

In the *partitioning* step, the dataset S is partitioned into disjoint blocks, $S_1, \ldots, S_m$, where we assume that node i has only the access to S_i. In addition, let $B_1, \ldots, B_m$ be the *MBR*s for $S_1, \ldots, S_m$. We broadcast them to all nodes to be used in the next step.

In the *local-pruning* step, each node i locally computes $\mathcal{CH}(S_i)$. Let V_i be the set of vertices of $\mathcal{CH}(S_i)$. Then, each node prunes V_i by removing the points which definitely *cannot* be a vertex of the final hull $\mathcal{CH}(S)$, using the information of $\mathcal{CH}(S_i)$ and $B_1, \ldots, B_m$. (We will explain this part shortly.) Finally, node i writes the resulting set V_i' to the disk with a unique key w.r.t. the master node.

In the *merging* step, the master node reads in all V_i', computes the convex hull of $\bigcup_{i=1}^{m} V_i'$, and writes the final result into the disk.

The crucial part is the local-pruning step. With $\mathcal{CH}(S_i)$ and $B_1, \ldots, B_m$ in hand, how can we determine whether a vertex of $\mathcal{CH}(S_i)$ is definitely not a vertex of $\mathcal{CH}(S)$? We need the following property. (Here, for convenience, we assume all points in S are in general position. This assumption can be easily removed in practice.)

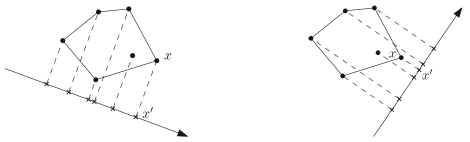
Theorem 3 *Let X be a set of points in $\mathbb{R}^d$ and $x \in X$ be a point. Then, x is a vertex of $\mathcal{CH}(X)$ if and only if there is a d-dim unit vector (or, a direction) v satisfying that $\langle x, v\rangle > \langle y, v\rangle$ for any $y \neq x$ in X. Here, $\langle \cdot, \cdot \rangle$ denotes the inner product.*

Visually, Theorem 3 is equivalent of saying that vertex x lies on the convex hull if and only if there exists a direction such that the projection of x on this direction is ahead of the projections of the remaining vertices; see Fig. 15a. However, such a direction cannot be chosen arbitrarily due to the existence of other points; see Fig. 15b for an example. We can compute the union of all infeasible directions won by some other point and consider its complement. If the complement is not empty, $x \in \mathcal{CH}(X)$; otherwise, $x \notin \mathcal{CH}(X)$. Let $I_x = \{v : \langle v, x\rangle \leq \langle v, y\rangle \text{ for some } y \in X\}$ denote the *infeasible directions* for vertex x.

Fix some node i. Assume the local convex hull $\mathcal{CH}(S_i)$ and $B_1, \ldots, B_m$ are available. Let t be a vertex of $\mathcal{CH}(S_i)$. We will show how to (approximately) compute I_t. For each $j \neq i$, by using the information of B_j, we investigate the 2-dim unit vectors v which *definitely* satisfy the condition $\langle t, v\rangle \leq \langle r, v\rangle$ for some $r \in S_j$. These vectors can be found by first finding the visible region from vertex t to the box B_j and then identifying the two directions that are perpendicular to the boundaries. We use $U_j(t)$ to denote all the directions in between these two, inclusively. As such, $U_j(t)$ is connected and can be represented as an angle; see Fig. 16a. Clearly, $U_j(t)$ can be computed in $O(1)$ time since $|B_j| = 4$.

It is worth noting that the way we construct $U_j(t)$ does not exactly follow the definition since node i has no information at all about the local hull $\mathcal{CH}(S_j)$ computed by node j. If $\mathcal{CH}(S_j)$ is available, an accurate (and usually wider) $U_j(t)$ can be found, as shown in Fig. 16b. But this way, $U_j(t)$ can no longer be computed in $O(1)$ time as $|\mathcal{CH}(S_j)|$ is generally not a constant. Furthermore, an all-to-all broadcast of all the local hulls must be done, which is obviously too expensive. Thus, we have to make certain trade-offs. Approximating $U_j(t)$ using the bounding box B_j could allow some points that are not on the final hull to survive the local-pruning step, but it should be faster and more scalable in practice.

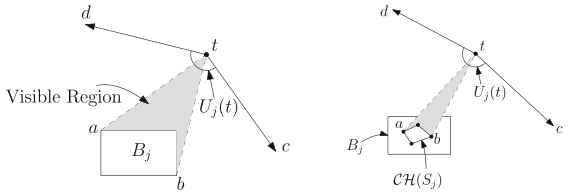
Previously, $U_j(t)$ is defined only for $j \neq i$. We now look at the case when $i = j$ and consider the points in S_i. With $\mathcal{CH}(S_i)$ in hand, we can *precisely* determine all the infeasible



(a) x is a point on the CH and its projection x' (w.r.t. the selected direction) is the leader among all remaining projections.

(b) An example illustrating an invalid direction for vertex x.

Fig. 15 Figures illustrating Theorem 3



(a) Illustrating how to compute U_j w.r.t. t and B_j. Here, we have $at \perp ct$ and $bt \perp dt$.

(b) A wider U_j can be achieved if we have $\mathcal{CH}(S_j)$ in hand.

Fig. 16 Illustrating the computation of the infeasible regions

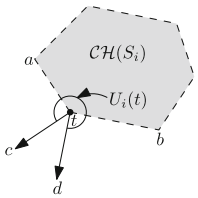


Fig. 17 Illustrating how to compute U_i w.r.t. t and $\mathcal{CH}(S_i)$. Again, we have $at \perp ct$ and $bt \perp dt$

directions for t w.r.t. $S_i \setminus \{t\}$. We define $U_i(t) = \{v : \langle t, v\rangle \leq \langle r, v\rangle$ for some $r \in S_i\}$, which is also connected and can be represented as an angle; see Fig. 17. We can compute $U_i(t)$ in constant time by giving the two vertices of $\mathcal{CH}(S_i)$ that are adjacent to t.

Finally, we set $I_t = \bigcup_{i=1}^{m} U_i(t)$, where the union can be efficiently computed in $O(m \log m)$ time via a polar-angle sort. If the complement of I_t is empty, it is safe to discard vertex t; otherwise, collect t into V_i'. We perform the above procedure for all points in $\mathcal{CH}(S_i)$ and then write V_i' into the disk. This completes all the details required by the local-pruning step.

8 Farthest pair

A nice property of the farthest pair (shown in Fig. 2d) is that the two points forming the pair must lie on the convex hull of all points [46]. 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 [46]. In this section, we introduce our farthest pair algorithms for Hadoop and SpatialHadoop.

8.1 Farthest pair in Hadoop

A Hadoop algorithm for the rotating calipers method [46] would complete the convex hull first as discussed in Sect. 8.1. Then, a single machine would 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 the 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.

8.2 Farthest pair in SpatialHadoop

Our SpatialHadoop algorithm works similar to our skyline and convex hull algorithms in that 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 of the farthest pair pruning rule is that two partitions that are very close to each other cannot possibly produce the farthest pair. In other words, that farthest pair has to come from two partitions that are far away of each other. To formalize the pruning rule, we define *upper* and *lower* bounds for the farthest pair that is produced from two partitions c_1 and c_2.

Figure 18a shows the upper and lower bounds of the farthest pair for two partitions. The *upper bound* indicates the maximum possible distance between two points in the two partitions. Since all points in the two partitions have to be contained in the two MBRs, we calculate the *upper bound* as the maximum distance between the corners of the two MBRs. On the other hand, the *lower bound* is a value for which we are guaranteed to find a pair between the partitions whose distance is at least that value. In an earlier version of this algorithm [16], we used the *minimum* distance between the two partitions as a lower bound; however, we can obtain a tighter bound by using the fact that the two MBRs are in fact *minimal*. This means that there should be at least one point on each of its four sides. Therefore, there has to be at least one pair of points on the two farthest *horizontal* sides, and, similarly, a pair of points on the two farthest *vertical* sides. Figure 18a is an example of these two distances, where d_1 and d_2 are the maximum horizontal and vertical distances, respectively. The higher of these two values is used as the *lower* bound for the farthest pair distance between c_1 and c_2.

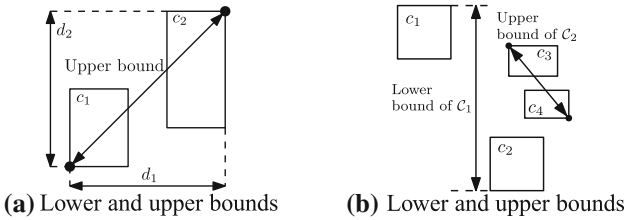


Fig. 18 Farthest pair algorithm in SpatialHadoop

By using these two definitions, we can devise a simple pruning rule for pairs of partitions. Figure 18b gives an example of two pairs of cells $\mathcal{C}_1 = \langle c_1, c_2 \rangle$ and $\mathcal{C}_2 = \langle c_3, c_4 \rangle$. We say that $\mathcal{C}_1$ dominates $\mathcal{C}_2$ if the *lower* bound of $\mathcal{C}_1$ is greater than the *upper bound* of $\mathcal{C}_2$. In this case, the pair $\mathcal{C}_2$ can be pruned as it can never produce a farther pair than the one produced by $\mathcal{C}_1$. This is depicted in Fig. 18b, 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.

The pruning rule can be easily applied in two passes. The first pass iterates over all partition pairs and computes the *greatest lower bound* as the maximum of all lower bounds of all pairs. This value indicates the smallest possible distance between the two points in the final answer. This means that any pair of partitions that cannot produce a higher value that can be pruned. The second pass iterates again over all pairs of partitions and compares the *upper bound* of each pair, to the *greatest lower bound* computed in the first round. If the *upper bound* is less that the *greatest lower bound*, that pair is pruned.

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 [46]. 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 HDFS block capacity, e.g., 128 MB. Finally, the algorithm computes the global farthest pair by collecting all local farthest pairs and selecting the one with largest distance.

Algorithm 6 gives the pseudo-code 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 returns a set of cells, the filter function in this algorithm returns a set of *pairs* of cells as the *map* function processes a pair of cells. The filter function runs two passes where the first pass, in lines 4–8, computes the greatest lower bound (GLB) on the farthest pair, and the second pass, in line 10–13, selects all pairs that can possible produce an answer greater than or equal to GLB.

Algorithm 6 Farthest pair in SpatialHadoop

```
1: Load the input file using SpatialHadoop file loader
2: function FILTER(C: Sets of cells (partitions))
3:     // Pass 1: Compute a lower bound (LB) of the answer
4:     Initialize the greatest lower bound GLB to 0
5:     for each pair of cells ⟨c1, c2⟩ ∈ C × C do
6:         LB ← the lower bound between c1 and c2
7:         Update GLB if LB > GLB
8:     end for
9:     // Pass 2: Select all pairs that can produce a pair with distance > GLB
10:    Initialize the set S of selected cell pairs to {}
11:    for each pair of cells ⟨c1, c2⟩ ∈ C × C do
12:        Add ⟨c1, c2⟩ to S if the upper bound of c1 and c2 ≥ GLB
13:    end for
14:    return S
15: end function
16: function MAP(P1, P2: Two sets of points)
17:    P ← P1 ∪ P2
18:    Compute the convex hull of P using Andrew's monotone chain algorithm
19:    Compute the farthest pair of points p1, p2 using rotating calipers method
20:    Emit the intermediate pair ⟨key = 1, value = ⟨p1, p2⟩⟩
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
```

The map function in lines 16–21 is called once for each selected pair of cells. The map function takes as input 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. It uses a traditional technique which combines all points together (line 17, computes their convex hull (line 18) and finally applies the rotating calipers method to get the farthest pair of points (line 19). Line 20 emits this pair of points to the intermediate output with a constant key=1, which ensures that all selected pairs of points 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.

9 Closest pair

The closest pair (Fig. 2d) in any dataset can be found using a divide-and-conquer algorithm [46]. 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 $p_2 \in P_2$. Finally, the algorithm returns the closest pair among the three pairs found. In this section, we introduce our closest pair algorithms for Hadoop and SpatialHadoop.

9.1 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 [41]. Furthermore, the merge step requires random access to the list of sorted points which is a well-known bottleneck in HDFS [32]. 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 are 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 Sect. 7.1, the brute-force approach would work but it requires too much computation for large files.

9.2 Closest pair in SpatialHadoop

Our closest pair algorithm in SpatialHadoop is an adaptation of the traditional closest pair divide-and-conquer algorithm [46]. 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 Fig. 19. As the size of each partition is only 64 MB, 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 Fig. 19, 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 $\langle \hat{p}, \hat{q} \rangle$, by 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

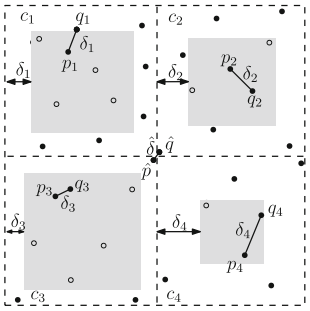


Fig. 19 Closest pair in SpatialHadoop

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, and thus, none of them can be pruned.

Algorithm 7 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 ⟨p, q⟩ of P using the divide-and-
       conquer algorithm
4:     output ⇐ ⟨1, p⟩
5:     output ⇐ ⟨1, q⟩
6:     Let δ be the distance between p and q
7:     Draw a buffer with size δ 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 ⟨p̂, q̂⟩ of P using the divide-
       and-conquer algorithm
11: end function
```

Algorithm 7 gives the pseudo-code for the closest pair algorithm in SpatialHadoop. In line 1, the file is initially loaded using the 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.

10 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 1.2.1 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 50 to 200 GB, memory

ranging from 2 to 8 GB, and processor speeds ranging from 2.2 to 3 GHz. Single machine experiments are conducted on a more powerful machine with 1TB RAM and an eight-core 3.4 GHz processor. Although all datasets we use fit in 1TB of memory, some algorithms fail due to the limitation of the array size in Java to 2^{31} entries. While we could use multiple arrays and treat them as one longer array, it would have complicated the algorithms and it would not change the results in this paper.

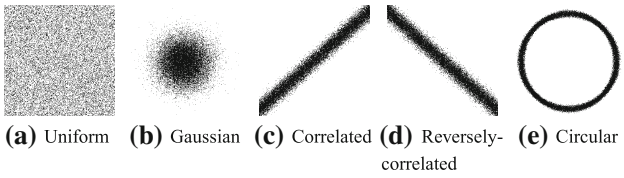
Experiments were run on three datasets: (1) OSM1: A real dataset extracted from OpenStreetMap [42] containing up to 164M polygons from the map (e.g., lakes and parks) with a total size of 80 GB. (2) OSM2: A real dataset also extracted from OpenStreetMap and containing up to 2.7B points from all around the world (e.g., street intersections and points of interest) with a total size of 92 GB. (3) SYNTH: A synthetic dataset of points generated randomly in an area of 1M×1M using one of the distributions: *uniform*, *Gaussian*, *correlated*, *reversely correlated*, and *circular* (see Fig. 20). *Uniform* and *Gaussian* represent the two most widely used distributions for modeling many real life systems. *Correlated* and *reversely correlated* represent the best and worst cases for skyline. The *circular* data are 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 128 GB and contains 3.8B points.

We use total execution time as the our main metric. Sometimes, the results of single machine experiments are omitted if the operation runs out of memory or the numbers are so large that they would hide the performance difference between other algorithms. Experimental results for the six proposed operations are given in Sects. 10.1–10.5.

10.1 Polygon union

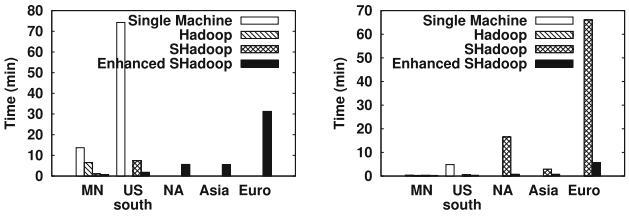
Figure 21 gives the total processing time for the polygon union operation on the OSM real dataset while varying input size. We obtain five subsets from the OSM dataset corresponding to Minnesota (MN), US south region, North America (NA), Asia, and Europe, which contain 0.35, 4.31, 20.18, 20.24, and 135 million polygons, respectively. We process each dataset using four algorithms: a single machine in-memory algorithm, Hadoop algorithm, SpatialHadoop algorithm, and enhanced SpatialHadoop algorithm, as described in Sect. 4. We further obtain two datasets, *complex* and *simple*. The *complex* dataset contains the original polygons with all their vertices, while the *simple* dataset contains a simplified version by taking the convex hull of each polygon.

Figure 21a gives the running time for the four algorithms on the *complex* OSM dataset. The single machine polygon union algorithm does not scale and quickly fails for large datasets, while CG_Hadoop scales to very large datasets with



(a) Uniform **(b)** Gaussian **(c)** Correlated **(d)** Reversely-correlated **(e)** Circular

Fig. 20 Synthetic data distributions



(a) Complex polygons **(b)** Simple polygons

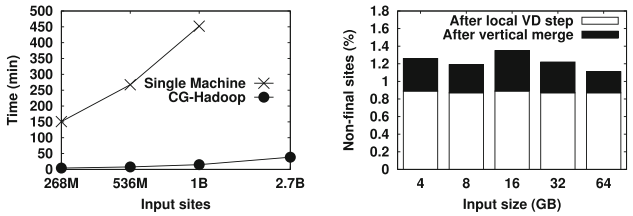
Fig. 21 Polygon union experiments

an order of magnitude speedup. The Hadoop union algorithm does not scale to large datasets as it uses the default Hadoop partitioner which partitions records randomly. As the input size increases, there is only a small chance that two adjacent polygons are assigned to the same partition which reduces the amount of line segments being removed the local union step. In fact, it fails with the 4.3M polygons dataset while the single machine with 1TB of memory can process it. The SpatialHadoop union algorithm makes use of the constructed index to remove more line segments in the local union step as more adjacent polygons are assigned to the same partition. However, it still fails with the 20M polygons dataset due to the overhead of the final global union step. The enhanced SpatialHadoop union algorithm is very scalable compared to all other algorithms as it runs completely distributed without the need for a final merge step which runs on a single machine. This allows it to process very large datasets which other algorithms fail to process.

Figure 21b shows the performance of the union algorithm with the *simple* dataset. The single machine and Hadoop algorithms still fail with the 20M polygons dataset and higher. However, the SpatialHadoop union algorithm is able to process all the sizes of the dataset where it failed with their *complex* versions. The enhanced union algorithm is still much faster with up to 20x speedup.

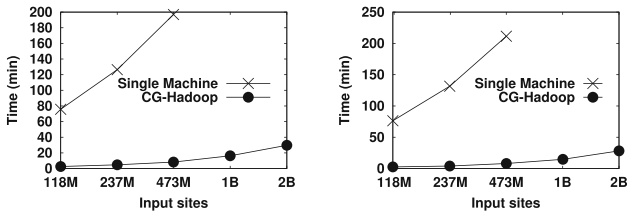
10.2 Voronoi diagram

Figure 22 shows the performance of the Voronoi diagram algorithm running on the OSM real dataset. We use random sampling to obtain subsets of sizes 270M, 536M, 1B, and 2.7B points. Due to the huge size of the Voronoi diagram, the single machine algorithm runs out of memory with the 2.7 billion datasets even though it has 1TB of memory. On the other hand, the CG_Hadoop algorithm is much more scalable



(a) OSM data **(b)** Pruning technique

Fig. 22 Voronoi diagram experiments on OSM dataset



(a) Uniform data **(b)** Gaussian data

Fig. 23 Voronoi diagram experiments on SYNTH dataset

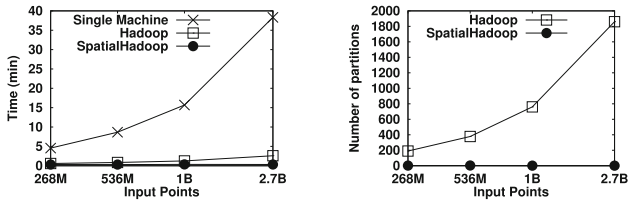
as it (1) computes local VDs for partitions in parallel, (2) early prunes final Voronoi regions in local VDs, and (3) merges the local VDs in parallel to compute the final VD. This results in up to 30x speedup in running time and allows CG_Hadoop to process the datasets which the single machine with 1TB memory has failed to process.

Figure 22b further illustrates the power of the pruning rule by showing the percentage of sites that remain after the *local VD* step and after the *vertical merge* step. The local VD step is able to prune almost 99% of the sites which leave only 1% of the sites to process in the merge step. Furthermore, the vertical merge step, which merges the partitions vertically, leaves less than 0.5% of the input sites to be merged in the final horizontal merge step.

Figure 23a, b shows the performance of the Voronoi diagram algorithm on synthetic data of uniform and Gaussian distributions, respectively. In Fig. 23a, we vary the input size from 4 to 64 GB of uniformly generated data and measure the runtime. We observe similar trends to the real dataset where the single machine fails with the 1B dataset while CG_Hadoop scales very nicely with an order of magnitude speedup over the single machine algorithm.

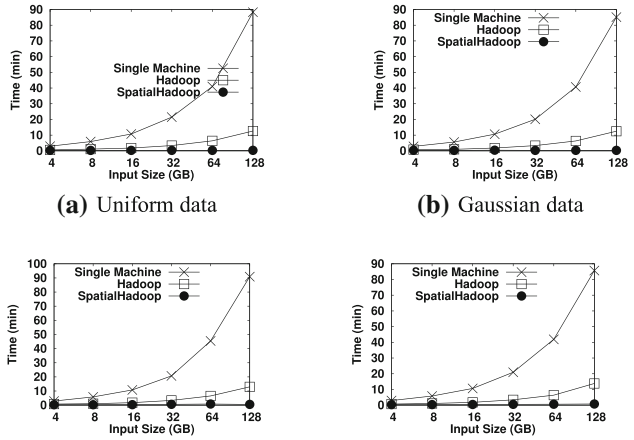
10.3 Skyline

Figure 24 compares the performance of the skyline algorithms on a single machine and CG_Hadoop with real OSM dataset. This single machine algorithm is able to scale to input sizes beyond the Java array limit of 2^{31} by computing the skyline as soon as the array fills up. This allows it to cut down the array size and keeps it below the limit. As the input size increases from 268M points to 2.7B points,



(a) Running time **(b)** Pruning technique

Fig. 24 Skyline on OSM dataset



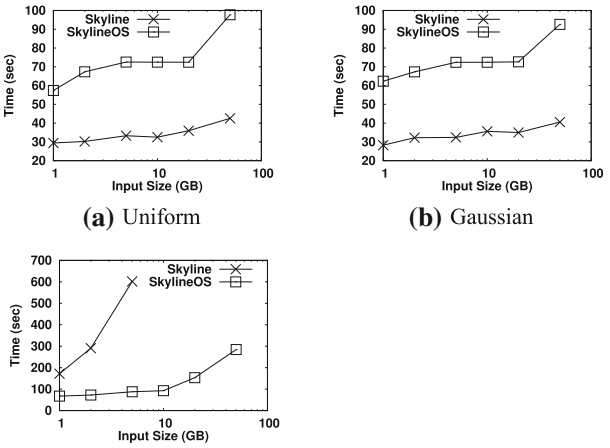
(a) Uniform data **(b)** Gaussian data

(c) Correlated data **(d)** Reversely-correlated data

Fig. 25 Skyline experiments on SYNTH dataset

the performance of the single machine algorithm degrades and fails the largest dataset while both CG_Hadoop algorithms are much more scalable with 14x and 115x speedup when running on Hadoop and SpatialHadoop, respectively. When running on Hadoop, CG_Hadoop achieves one order of mangitude speedup over the single machine algorithm as it parallelizes the work over the machines increasing both the loading time of the input file and the processing time. On SpatialHadoop, CG_Hadoop achieves up to 115x speedup over the single machine algorithm as it prunes unnecessary partitions in addition to the parallelized loading and processing. Figure 24b further clarifies the speedup of SpatialHadoop over Hadoop by showing the number of partitions that each technique processes. The number, which the skyline Hadoop algorithm processes, increases with the input size as it has to process the whole input file. On the other hand, the skyline SpatialHadoop algorithm applies the pruning function which utilizes the index to prune unnecessary partitions and processes at most 3 partitions in this experiment.

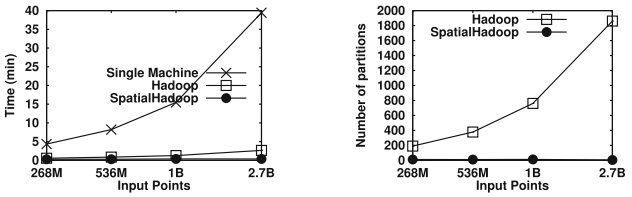
Figure 25 gives the performance of the skyline operation on generated dataset of four different distributions, namely uniform, Gaussian, correlated, and reversely correlated. When CG_Hadoop is deployed in standard Hadoop, it consistently achieves an order of magnitude performance gain, on all data distributions, due to the parallelization of the computation over the cluster machines. The local skyline



(a) Uniform **(b)** Gaussian

(c) Worst-case reversely-correlated

Fig. 26 Output-sensitive skyline algorithm (SkylineOS)



(a) Running time **(b)** Pruning function

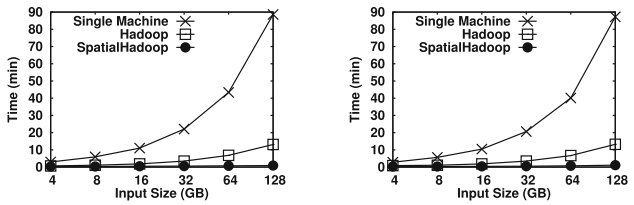
Fig. 27 Convex Hull on OSM dataset

step is very efficient in pruning most of the points leaving only a little work for 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 the final output, and thus minimizes the total number of processed blocks.

Finally, Fig. 26 compares performance between the regular skyline and the output-sensitive skyline algorithms in SpatialHadoop with *disjoint* indexes. The output-sensitive algorithm is slightly worse under uniform and Gaussian data because it runs two phases, while the regular algorithm runs one phase and can work fine. However, for the worst-case dataset (i.e., reversely correlated), the regular algorithm takes a long time as it cannot prune any data locally, and it fails with very big datasets that cannot fit into the memory of a single machine. The output-sensitive algorithm, on the other hand, scales very well in this case.

10.4 Convex hull

Figure 27 shows the running time of the convex hull algorithms on a single machine and in CG_Hadoop on the real dataset OSM. Similar to skyline, the single machine convex hull algorithm is able to scale beyond the Java array limit by computing the convex hull as soon as the array fills up



(a) Uniform data **(b)** Gaussian data

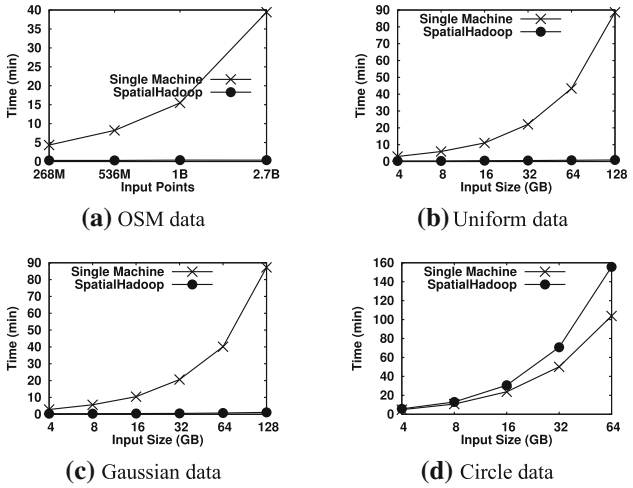
Fig. 28 Convex Hull on SYNTH dataset

which reduces the array size. The performance of the single machine algorithm quickly degrades due to the limited capabilities of a single machine including loading time from disk and processing the whole data in the main memory. CG_Hadoop achieves up to 14x and 109x speedup when running on Hadoop and SpatialHadoop, respectively. Figure 27b shows the power of the pruning function in reducing number of partitions that the SpatialHadoop convex hull algorithm processes. While the number of partitions quickly increases for the Hadoop algorithm, which processes the whole file, the number of partitions processed by SpatialHadoop is no more than 12, even for the 2.7B points file.

Figure 28 gives the running times of the convex hull operation on generated datasets of up to 128 GB with 3.8B points. The convex hull algorithm in CG_Hadoop, described in Sect. 7.1, 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 the answer. Although not shown here for clarity of the figure, CG_Hadoop, deployed in SpatialHadoop, achieves 260x speedup for the 128 GB dataset compared to the traditional system.

10.5 Farthest pair

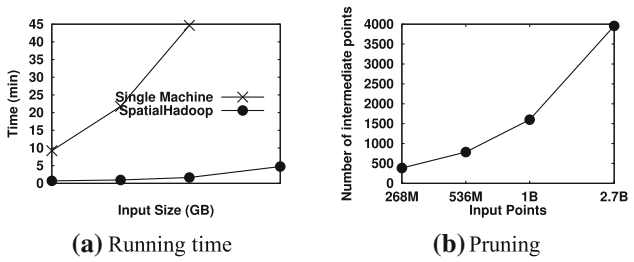
Figure 29 gives the performance of the farthest pair algorithm. Since the farthest pair of points have to be on the convex hull, the farthest pair operations compute the convex hull as a preprocessing step to reduce the size of the data. In Fig. 29a–c, we show the running times on OSM real dataset, uniformly distributed and Gaussian generated datasets, respectively. In these three cases, the size of the convex hull is very small which causes the running time to be dominated by the convex hull step. Figure 29d shows the worst- case scenario, when the farthest pair runs on the circular dataset, where the size of the convex hull is maximized. In this case, we apply the algorithm described in Sect. 8.2 which prunes pairs of partitions that do not contribute to the answer and calculate pairwise distances in remaining pairs. Unlike other operations, in which CG_Hadoop performs faster than



(a) OSM data **(b)** Uniform data

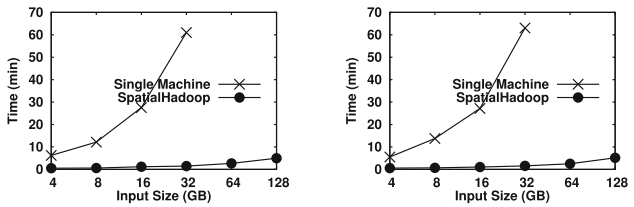
(c) Gaussian data **(d)** Circle data

Fig. 29 Farthest pair experiments



(a) Running time **(b)** Pruning

Fig. 30 Closest pair on OSM dataset



(a) Uniform data **(b)** Gaussian data

Fig. 31 Closest pair on SYNTH dataset

the single machine, this operation performs slightly better on a single machine than it does on CG_Hadoop. The reason is that the single machine algorithm performs the rotating calipers algorithm on the array of points in the main memory which is very efficient as the points are stored in a sorted array. One must keep in mind, though, that the single machine has 1TB of memory, while SpatialHadoop runs on a cluster of machines with much less memory capacities. (Even the collective memory of all machines in the cluster is less than 1TB.) This makes the SpatialHadoop algorithm widely applicable to larger datasets on commodity machines.

10.6 Closest pair

Figure 30 gives the running time of the closest pair algorithm on the OSM real dataset. Due to the huge overhead of the Hadoop closest pair algorithm, as described in Sect. 9.1, we only apply the single machine and SpatialHadoop algorithms. The performance of the single machine algorithm quickly degrades due to the limited capabilities of a single machine. However, CG_Hadoop nicely scales out with at least one order of magnitude speedup. Although the closest pair algorithm in CG_Hadoop has to process the whole dataset, as it does not apply an early pruning step, it can still achieve up to 27x performance speedup due to the parallelization of the work over a cluster of machines. Furthermore, it manages to process the 2.7B points dataset while it fails on a single machine with 1TB of memory.

Figure 30b shows the power of the pruning technique in SpatialHadoop closest pair algorithm. As the input size increases, this figure shows the number of intermediate points produced by the *local closest pair* step and processed by the *global closest pair* step. It shows that only about one-millionth of the input points make it to the global closest pair step, which runs on a single machine, while all other points are early pruned by the local closest pair step, which runs in parallel on all machines. This explains the order of magnitude speedup in SpatialHadoop algorithm as compared to the single machine algorithm.

Results of the closest pair experiments on synthetic data are shown in Fig. 31 for different input sizes. Similar to the real dataset, the traditional single machine algorithm cannot scale to large files as it has to load the whole dataset in memory first. In the experiments shown, the traditional algorithm fails when the input size reaches 64 GB. 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, thus limiting the required memory usage to a block size.

11 Related work

There has been significant recent interest in taking advantage of Hadoop and similar big data systems to support spatial query processing. Existing work can be broadly classified into three categories: (1) solving specific spatial operations, (2) providing a framework for spatial data, and (3) solving computational geometry operations.

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 have focused on R-tree construction [9], range queries over spatial points [55], range queries over trajectory

data [36], k-nearest-neighbor (kNN) queries [2,20,39,55], all nearest-neighbor (ANN) queries [50], reverse nearest-neighbor (RNN) queries [2], spatial join [55], exact kNN Join [35], approximate kNN Join [53], and optimal location selection/searching algorithms [10,48]. In all these algorithms, the underlying Hadoop system is used as-is, and the spatial query processing is provided by implementing the spatial query processing as map and reduce functions.

Unified framework for spatial operations In this approach, spatial query processing is provided by injecting spatial query awareness inside an existing system. There exist six systems that are closely related to CG_Hadoop. (1) Hadoop-GIS [1] is a spatial data warehousing system, which extends Hive with a grid-based index, and uses it to support range query and self-spatial-join. (2) Parallel-Secondo [33] 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 [37] extends HBase to support multidimensional indexes, based on Z-Curve ordering, which allows for efficient retrieval of points using range and kNN queries. (4) SpatialHadoop [17,18] extends Hadoop with spatial indexes [15], based on grid, R-tree, R+-tree, and Quad-tree [47], and provides new MapReduce components that allow using the indexes in spatial MapReduce programs. (5) ESRI Tools for Hadoop [52] implements the PMR-Quad-tree in Hadoop and uses it to support range query and kNN queries. (6) GeoMesa [19] extends a key-value store, called Accumulo, with geohash-based index, and provides efficient interactive queries on top of it such as range kNN queries. It is also interesting to notice that [51] builds a CPU-GPU Hybrid system to accelerate pathology image data cross-comparison.

Computational geometry operations The use of MapReduce in the computational geometry field was discussed from a theoretical perspective [23] to suggest simulating the 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. Only two computational geometry operations have been implemented in MapReduce, skyline, and Voronoi diagram (VD) construction. The skyline operation [54] was first implemented in MapReduce by following an angle-based partitioning from the origin which allows efficient pruning of points. It was then studied in [29] using hyperplane projections, and finally, both skyline and reverse skyline queries were investigated again in [44] using MapReduce. The Voronoi diagram is constructed in MapReduce [2] by directly mapping the traditional divide-and-conquer algorithm to map and reduce functions. However, it does not scale due to the bottleneck in the merge operation.

Our work in this paper, CG_Hadoop, lies in between the second and third categories above. First, it does not focus on one computational geometry operation, rather it covers a set of six different and fundamental computational geometry operations and describes efficient implementations for each of them using various spatial indexes. Second, it does not provide a new system. Instead, it utilizes the extensibility of SpatialHadoop and the efficient spatial indexes in it to implement the six computational geometry operations efficiently. In an earlier version of this work [16], we proposed five fundamental computational geometry operations. In this paper, we further show its applicability to computational geometry operations by providing five new algorithms: four improved algorithms for *polygon union*, skyline, convex hull, and *farthest pair* and a novel algorithm for the new *Voronoi diagram* construction operation. The open-source nature of CG_Hadoop will act as a research vehicle for other researchers to build more computational geometry algorithms that take advantage of the MapReduce paradigm.

12 Conclusion and future work

This paper has introduced CG_Hadoop, a suite of scalable and efficient MapReduce algorithms for various fundamental computational geometry operations, namely *polygon union*, *Voronoi diagram*, *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 deploy a form of divide-and-conquer that leverages the distributed parallel environment in both Hadoop and SpatialHadoop and hence achieves much better performance than their corresponding traditional algorithms. Moreover, SpatialHadoop algorithms significantly outperform Hadoop algorithms as they take advantage of the spatial indexing and components within SpatialHadoop. Overall, CG_Hadoop forms the nucleus of a comprehensive MapReduce library of computational geometry operations. Extensive experimental results on a cluster of 25 machines with datasets up to 128 GB show that CG_Hadoop achieves up to 29x and 260x better performance than traditional algorithms when using Hadoop and SpatialHadoop systems, respectively.

CG_Hadoop, in this paper, mainly focuses on two-dimensional data, which already cover many everyday applications. For future work, it would be interesting to see whether it is extensible to big data in higher dimensions. For instance, kNN queries are often used in machine learning to compute the similarity between two (high-dimensional) vectors. However, the complexities of most operations in CG_Hadoop increase drastically with respect to the dimensionality, so new ideas must be further explored to design efficient and scalable solutions. Potential directions here include building more efficient spatial indexing/partitioning methods and attempting approximation algorithms.

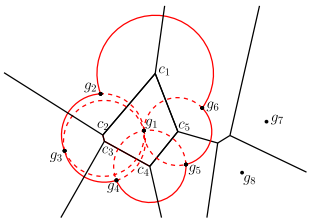


Fig. 32 Example of shared disks in dangerous zones

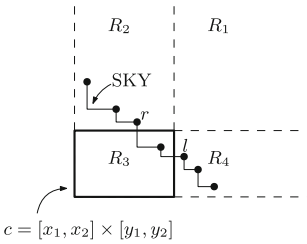


Fig. 33 An example illustrating Theorem 4

A Voronoi diagram proofs

This appendix lays out the proofs related to the Voronoi diagram construction algorithm (Sect. 5.2).

Proof of contiguity of non-safe Voronoi regions

Lemma 2 *Let c be a closed cell and denote all its direct neighbor cells as $NN(c)$. Also, define $\mathcal{Z}(\cdot)$ be the dangerous zone of c. Then, the dangerous zone of c is a subset of the union of the dangerous zones of its neighbors, that is, $\mathcal{Z}(c) \subseteq \bigcup_{c' \in NN(c)} \mathcal{Z}(c')$.*

Proof The dangerous zone $\mathcal{Z}(c)$ essentially consists of the union of several disks centered at the vertices of c. One just notices that each of these disks is also part of the dangerous zone of (at least one of) its neighbor cells. Therefore, it simply follows that $\mathcal{Z}(c) \subseteq \bigcup_{c' \in NN(c)} \mathcal{Z}(c')$. (Fig. 32 gives an example where the disk centered at c_5 is shared among the Voronoi regions of g_1, g_5, and g_6.) □

Lemma 2 naturally implies that the following corollary.

Corollary 2 *Any non-safe cell must be adjacent to another non-safe cell if there is more than one such cell.*

Proof For a contradiction, assume a cell c is surrounded by all safe cells, but c itself is non-safe. Then, there must exist a new site, say g, introduced inside $\mathcal{Z}(c)$ that causes c to be non-safe. By Lemma 2, g must be in the dangerous zone of at least one of c's direct neighbors, making that cell non-safe, which is a contradiction. □

B Reducing communication cost in the output-sensitive Skyline

The output-sensitive skyline algorithm described in Sect. 6.3 has a communication cost of $O(|G|^2)$ where $|G|$ is the number of partitions. This communication cost is due to broadcasting the set SKY of all points with the highest domination power to each node. Here, we show an improved algorithm that can reduce the communication cost to $O(|G|)$ by applying a more efficient technique that can selectively send a fixed number of points to each node. Below, we show how to select this subset and prove that it has the same domination power as the whole set SKY. Recall that $\text{sky}(c)$ is the local skyline in a cell c.

Theorem 4 *For any cell $c \in G$ and a point set S, define $sky(c) \Cap S = \{p \in sky(c) \mid \nexists q \in S \; s.t. \; p \prec q\}$. Then there exists a subset, $SKY(c)$, of SKY satisfying (i) $|SKY(c)| \leq 4$ and (ii) $sky(c) \Cap SKY = sky(c) \Cap SKY(c)$, that is, $SKY(c)$ has the same dominance power as SKY.*

Proof Let c denote the region $[x_1, x_2] \times [y_1, y_2]$. Then, it is clear that only the points in $[x_1, \infty) \times [y_1, \infty)$ can affect $\text{sky}(c)$. We divided it into four parts as illustrated in Fig. 33: $R_1 = (x_2, \infty) \times (y_2, \infty)$, $R_2 = [x_1, x_2] \times (y_2, \infty)$, $R_3 = [x_1, x_2] \times [y_1, y_2]$, and $R_4 = (x_2, \infty) \times [y_1, y_2]$.

If $R_1 \cap \text{SKY} \neq \emptyset$, then the entire local skyline in c will be dominated. Adding any point of $R_1 \cap \text{SKY}$ to $\text{SKY}(c)$ will satisfy both conditions, i.e., $|\text{SKY}(c)| = 1$ and that point will guarantee that $\text{sky}(c) \Cap \text{SKY} = \text{sky}(c) \Cap \text{SKY}(c) = \emptyset$. It is worth mentioning that this case is equivalent to our prefiltering step.

When $R_1 \cap \text{SKY} = \emptyset$, we construct $\text{SKY}(c) = (R_3 \cap \text{SKY}) \cup \{l, r\}$, where l (resp., r) is the leftmost (resp., rightmost) point in $R_4 \cap \text{SKY}$ (resp., $R_2 \cap \text{SKY}$). And if $R_4 \cap \text{SKY} = \emptyset$ (resp., $R_2 \cap \text{SKY} = \emptyset$), we simply ignore l (resp., r). The reason for selecting only the extreme points r and l in the second and fourth quadrant, respectively, is straightforward, i.e., they dominate, respectively, the most area in the interior of c compared to other points. This suffices to prove the second condition. To show condition (i), one just notices that $|R_3 \cap \text{SKY}| \leq 2$ since c can only send the master machine at most two points for computing SKY. Therefore, $\text{SKY}(c) \leq 2 + 2 = 4$. □

References

1. Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., Saltz, J.: Hadoop-GIS: a high performance spatial data warehousing system over MapReduce. In: VLDB (2013)
2. Akdogan, A., Demiryurek, U., Banaei-Kashani, F., Shahabi, C.: Voronoi-based geospatial query processing with MapReduce. In: CLOUDCOM (2010)
3. Andrew, A.M.: Another efficient algorithm for convex hulls in two dimensions. Inf. Process. Lett. **9**(5), 216–219 (1979)
4. Apache. Hadoop. http://hadoop.apache.org
5. Bentley, J.L., Kung, H., Schkolnick, M., Thompson, C.D.: On the average number of maxima in a set of vectors and applications. J. ACM: JACM **25**(4), 536–543 (1978)

6. Berg, M.D., Cheong, O., Kreveld, M.V., Overmars, M.: Computational Geometry: Algorithms and Applications. Springer, Berlin (2008)
7. Borne, K.D., Baum, S.A., Fruchter, A., Long, K.S.: The hubble space telescope data archive. In: Astronomical Data Analysis Software and Systems IV, vol. 77 (1995)
8. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: ICDE (2001)
9. Cary, A., Sun, Z., Hristidis, V., Rishe, N.: Experiences on processing spatial data with MapReduce. In: SSDBM, pp. 302–319. New Orleans, Louisiana (2009)
10. Choudhury, F.M., Culpepper, J.S., Bao, Z., Sellis, T.: Finding the optimal location and keywords in obstructed and unobstructed space. VLDB J. **27**, 445–470 (2018)
11. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. PVLDB **1**(2), 1277–1288 (2008)
12. Cormen, T.H., Leisorson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2009)
13. Dalal, K.: Counting the onion. Random Struct. Algorithms **24**(2), 155–165 (2004)
14. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**, 107–113 (2008)
15. Eldawy, A., Alarabi, L., Mokbel, M.F.: Spatial partitioning techniques in SpatialHadoop. In: PVLDB, pp. 1602–1605. Kohala Coast, HI (2015)
16. Eldawy, A., Li, Y., Mokbel, M.F., Janardan, R.: CG_Hadoop: computational geometry in MapReduce. In: SIGSPATIAL, pp. 284–293. Orlando, FL (2013)
17. Eldawy, A., Mokbel, M.F.: A demonstration of SpatialHadoop: an efficient MapReduce framework for spatial data. In: VLDB (2013)
18. Eldawy, A., Mokbel, M.F.: SpatialHadoop: a MapReduce framework for spatial data. In: ICDE (2015) **(to appear)**
19. Fox, A., Eichelberger, C., Hughes, J., Lyon, S.: Spatio-temporal indexing in non-relational distributed databases. In: BigData, pp. 291–299. Santa Clara, CA (2013)
20. García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M., Manolopoulos, Y.: Enhancing SpatialHadoop with closest pair queries. In: East European Conference on Advances in Databases and Information Systems, pp. 212–225. Springer, Berlin (2016)
21. Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., Sindhwani, V., Tatikonda, S., Tian, Y., Vaithyanathan, S.: SystemML: declarative machine learning on MapReduce. In: ICDE (2011)
22. Giraph. http://giraph.apache.org/
23. Goodrich, M.T., Sitchinava, N., Zhang, Q.: Sorting, searching, and simulation in the MapReduce framework. In: ISAAC (2011)
24. Guibas, L.J., Stolfi, J.: Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. In: STOC, pp. 221–234. Boston, MA (1983)
25. Guttman, A.: R-Trees: A dynamic index structure for spatial searching. In: SIGMOD (1984)
26. Huai, Y., Chauhan, A., Gates, A., Hagleitner, G., Hanson, E.N., O'Malley, O., Pandey, J., Yuan, Y., Lee, R., Zhang, X.: Major technical advancements in apache hive. In: ACM SIGMOD, pp. 1235–1246 (2014)
27. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: EuroSys (2007)
28. Java Topology Suite. http://hadoop.apache.org/
29. Köhler, H., Yang, J., Zhou, X.: Efficient parallel skyline processing using hyperplane projections. In: ACM SIGMOD, pp. 85–96. ACM (2011)
30. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. Oper. Syst. Rev. **44**(2), 35–40 (2010)
31. Lee, G., Lin, J., Liu, C., Lorek, A., Ryaboy, D.V.: The unified logging infrastructure for data analytics at Twitter. PVLDB **5**(12), 1771–1780 (2012)
32. Liao, H., Han, J., Fang, J.: Multi-dimensional index on Hadoop distributed file system. In: ICNAS, pp. 240–249 (2010)
33. Lu, J., Guting, R.H.: Parallel secondo: boosting database engines with Hadoop. In: ICPADS (2012)
34. Lu, P., Chen, G., Ooi, B.C., Vo, H.T., Wu, S.: ScalaGiST: scalable generalized search trees for MapReduce systems. PVLDB **7**(14), 1797–1808 (2014)
35. Lu, W., Shen, Y., Chen, S., Ooi, B.C.: Efficient processing of k nearest neighbor joins using MapReduce. PVLDB **5**, 1016–1027 (2012)
36. Ma, Q., Yang, B., Qian, W., Zhou, A.: Query processing of massive trajectory data based on MapReduce. In: CLOUDDB (2009)
37. Nishimura, S., Das, S., Agrawal, D., Abbadi, A.E.: MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In: MDM (2011)
38. Nishimura, S., Das, S., Agrawal, D., El Abbadi, A.: $\mathcal{MD}$: design and implementation of an elastic data infrastructure for cloud-scale location services. DAPD **31**(2), 289–319 (2013)
39. Nutanong, S., Zhang, R., Tanin, E., Kulik, L.: The v*-diagram: a query-dependent approach to moving knn queries. Proc. VLDB Endow. **1**(1), 1095–1106 (2008)
40. Oliver, D., Steinberger, D.J.: From geography to medicine: exploring innerspace via spatial and temporal databases. In: SSTD (2011)
41. O'Malley, O.: Terabyte sort on Apache Hadoop. Yahoo! (2008)
42. OpenStreetMap. http://www.openstreetmap.org/
43. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. TODS **30**(1), 41–82 (2005)
44. Park, Y., Min, J., Shim, K.: Parallel computation of skyline and reverse skyline queries using mapreduce. Proc. VLDB Endow. **6**(14), 2002–2013 (2013)
45. PostGIS. Spatial and Geographic Objects for PostgreSQL. http://postgis.net/
46. Preparata, F., Shamos, M.I.: Computational Geometry: An Introduction. Springer, Berlin (1985)
47. Samet, H.: The Quadtree and related hierarchical data structures. ACMCS **16**(2), 187–260 (1984)
48. Sun, Y., Qi, J., Zhang, R., Chen, Y., Du, X.: Mapreduce based location selection algorithm for utility maximization with capacity constraints. Computing **97**(4), 403–423 (2015)
49. Tauheed, F., Biveinis, L., Heinis, T., Schürmann, F., Markram, H., Ailamaki, A.: Accelerating range queries for brain simulations. In: ICDE (2012)
50. Wang, K., Han, J., Tu, B., Dai, J., Zhou, W., Song, X.: Accelerating spatial data processing with MapReduce. In: ICPADS, pp. 229–236. Shanghai, China (2010)
51. Wang, K., Huai, Y., Lee, R., Wang, F., Zhang, X., Saltz, J.: Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. Proc. VLDB Endow. **5**(11), 1543–1554 (2012)
52. Whitman, R.T., Park, M.B., Ambrose, S.A., Hoel, E.G.: Spatial indexing and analytics on Hadoop. In: SIGSPATIAL. Dallas, TX (2014)
53. Zhang, C., Li, F., Jestes, J.: Efficient parallel kNN joins for large data in MapReduce. In: EDBT (2012)
54. Zhang, J., Jiang, X., Ku, W.S., Qin, X.: Efficient parallel skyline evaluation using MapReduce. TPDS **PP**(99), 1–14 (2015)
55. Zhang, S., Han, J., Liu, Z., Wang, K., Feng, S.: Spatial queries evaluation with MapReduce. In: GCC (2009)