Efficient Skyline Query Processing in SpatialHadoop

Dimitris Pertesis, Christos Doulkeridis

Department of Digital Systems
School of Information and Communication Technologies
University of Piraeus
18534 Piraeus, Greece

Abstract

This paper studies the problem of computing the skyline of a vast-sized spatial dataset in SpatialHadoop, an extension of Hadoop that supports spatial operations efficiently. The problem is particularly interesting due to advent of Big Spatial Data that are generated by modern applications run on mobile devices, and also because of the importance of the skyline operator for decision-making and supporting business intelligence. To this end, we present a scalable and efficient framework for skyline query processing that operates on top of SpatialHadoop, and can be parameterized by individual techniques related to filtering of candidate points as well as merging of local skyline sets. Then, we introduce two novel algorithms that follow the pattern of the framework and boost the performance of skyline query processing. Our algorithms employ specific optimizations based on effective filtering and efficient merging, the combination of which is responsible for improved efficiency. We compare our solution against the state-of-the-art skyline algorithm in SpatialHadoop. The results show that our techniques are more efficient and outperform the competitor significantly, especially in the case of large skyline output size.

Keywords: Skyline query, spatial data, MapReduce

Email addresses: jimper0@hotmail.com (Dimitris Pertesis), cdoulk@unipi.gr (Christos Doulkeridis)

1. Introduction

With the advent of mobile, GPS-enabled devices, modern applications produce an ever-increasing amount of spatial data at tremendous rates. *Big Spatial Data* is generated at high velocity and often encloses valuable information that can be exploited to deliver new added-value services to the users. Recent reports predict that the value of user-created location-based data will increase rapidly in the coming years, which brings up the significance of data analysis over large-scale spatial data.

Scalable analytics over vast-sized data is best supported today by platforms for parallel data processing. A prominent platform is MapReduce [1] (and its open-source implementation Apache Hadoop), which has become popular due to its salient features that include ease of programming, scalability, and fault-tolerance. However, as also indicated in [2], MapReduce has weaknesses related to efficiency when it needs to be applied to spatial data; a main shortcoming is the lack of any indexing mechanism that would allow selective access to specific regions of spatial data, which would in turn yield more efficient query processing algorithms.

A recent solution to this problem is an extension called *SpatialHadoop* [3], which is a framework that inherently supports spatial indexing on top of Hadoop. In SpatialHadoop, spatial data is deliberately partitioned and distributed to nodes, so that data with spatial proximity is placed in the same partition. In addition, the generated partitions are indexed, thereby enabling the design of efficient query processing algorithms that access only part of the data and still deliver the correct result. As demonstrated in [3], various algorithms are proposed for spatial queries, such as range and k-nearest neighbor queries as well as spatial joins.

However, in the case of advanced query operators, efficiency is not guaranteed even when spatial indexing is used. A notable example is the skyline query [4], which has attracted increased attention for data analytics, mainly because it can be used for decision-making when multiple (and often conflicting) criteria need to be considered. Efficient processing of skyline queries over large-scale spatial datasets is a challenging task, and it is the main topic of this paper.

Essentially, in this paper, we target efficient processing of the skyline query in SpatialHadoop. We propose a query processing framework for skyline algorithms, which is applicable on top of the indexing infrastructure provided by SpatialHadoop, and uses plain techniques that result in improved

performance. In particular, our framework focuses on effective *filtering* of candidate skyline points, as well as on efficient *merging* of local skyline sets.

The proposed framework is generic and can be parameterized by defining different techniques for (a) filtering and (b) merging. In this way, different skyline algorithms can be designed that belong to our framework. In other words, we abstract skyline query processing based on different ways to perform filtering and merging, which yields different skyline algorithms. We demonstrate two algorithms that belong to our framework and provably work much better than a baseline skyline algorithm proposed by the designers of SpatialHadoop [5].

In summary, this paper makes the following contributions:

- We propose a generic framework for skyline processing suitable for MapReduce-style platforms, which focuses on efficient filtering and merging, and apply it in the case of SpatialHadoop (Section 4).
- We present an algorithm that relies on filtering techniques to prune candidate skyline points as early as possible, thereby attaining performance gains (Section 5.1).
- We describe an algorithm that (additionally to filtering techniques) imposes a deliberate order of accessing candidate skyline points, in order to improve the performance of the merging phase (Section 5.2).
- We conduct an experimental study on a cluster of servers, which demonstrates that our algorithms outperform the competitor approach (Section 6).

Moreover, we review existing related work in Section 2, we present preliminary concepts in Section 3, and we summarize our conclusions in Section 7.

2. Related Work

In this section, we provide an overview of related work in two areas highly related to our work. The first is skyline computation in parallel environments, while the second is skyline processing in MapReduce.

2.1. Parallel Skyline Query Processing

Since the first paper on skyline processing in databases appeared [4], a significant number of papers on the topic have emerged; for a recent and brief overview of the current state-of-the-art we refer to [6].

In the case of *distributed data*, as thoroughly reviewed in [7], most approaches follow a three-step process, namely (a) local processing, (b) query routing, and (c) result merging. In this domain, the aim is to minimize the execution time, which (in turn) is the result of minimizing the total processing time, the number of queried nodes, and the network traffic.

Instead, in the case of *parallel data management*, the main phases of skyline query processing are (a) data partitioning, (b) local skyline processing, and (c) merging of local skyline sets. In any parallel skyline processing, data partitioning to servers is deliberately performed for achieving improved performance, therefore skyline algorithms have researched advanced partitioning techniques that are particularly tailored to the properties of the skyline query. Vlachou et al. [8] first showed that a typical grid-partitioning scheme, which is commonly used for the majority of queries over multidimensional data, is deemed insufficient for parallel skyline processing. In their work, a novel angle-based partitioning method is proposed that results in data partitions with salient properties, such as fair work assignment and fast local skyline processing. Later, Köhler et al. [9] proposed a similar partitioning method, where the actual partitioning is performed using hyperplane projections. Afrati et al. [10] study algorithms for parallel skyline queries where the main focus is on the need of one or two synchronization steps, while they also take into account load balancing. In this work, the Massively Parallel (MP) model is assumed for computation, in which the computation is a sequence of global parallel steps. Each step consists of three phases (a) broadcast phase, where some small information is exchanged among servers, (b) communication phase, where some data is exchanged, and (c) computation phase, where local processing is performed.

For efficient skyline processing in *multicore architectures*, the most recent algorithm [11] employs the angle-based partitioning from [8], together with other optimizations, such as parallelization of the partitioning task. Other approaches for multicore skyline processing include PSkyline [12] and ParallelBNL [13]. The PSkyline algorithm, is a divide-and-conquer algorithm optimized for multicore processors. In contrast to most existing divide-and-conquer (D&C) algorithms for skyline computation that divide into partitions based on geometric properties, PSkyline simply divides the dataset linearly

into N equi-sized partitions. The local skyline is then computed for each partition in parallel using the SSkyline algorithm [14], and the local skyline results are subsequently merged. Selke et al. [13] suggest a parallel version of BNL using a shared linked list for the skyline window. In this approach, a sequential algorithm is parallelized without major modifications, and the main challenge is related to the concurrent modification of the shared list. To address this problem, different locking protocols are used, including locking at each access, continuous locking, and lazy locking.

2.2. Skyline Query Processing in MapReduce

For a thorough survey on query processing using MapReduce and its limitations we refer to [2]. Most related to our paper is the work on SpatialHadoop [5], which is an extension of Hadoop that inherently supports spatial data, together with partitioning and indexing mechanisms. In [5], the authors demonstrate how to process a variety of queries (including the skyline query) over SpatialHadoop. The proposed approach described in [5] simply exploits the spatial index to prune cells (corresponding to HDFS blocks) from processing and uses a Combiner to reduce the burden imposed on the Reducer. In this paper, we compare our approach against [5] and show that our techniques achieve significant performance improvements, especially in the case of demanding data distributions where the output size of the skyline query is big. The main contributors to improved performance include (a) the use of specialized filters and (b) efficient determination of the final skyline points by accessing candidate skyline points in a specific order in the Reduce phase.

Other related approaches on skyline query processing in MapReduce include [15, 16, 17]. Recently, in [15], the authors propose algorithms for processing skyline and reverse skyline queries in MapReduce. To prune the search space, histograms are created which enable partitioning based on the regions identified in the histograms. They also use filtering to speed up query processing and the computation of skyline points. In contrast, our techniques focus particularly on spatial data and exploit the index structures provided by SpatialHadoop to prune the search space, without the need for building histograms for this purpose. More importantly, our algorithms require a single MapReduce job to compute the correct skyline set.

In [16], minimal algorithms for MapReduce are studied. An algorithm is deemed minimal when it satisfies the following properties: minimum footprint, bounded traffic, constant round of rounds for termination, and optimal

computation in terms of parallelization. In their work, the authors show how to compute skyline queries for 2-dimensional data, however their approach requires sorted input data or a MapReduce job that sorts the data prior to the job that performs the skyline computation. However, the overhead imposed by sorting a large dataset is non-trivial, and this is also avoided in our work.

A preliminary approach for skyline processing in MapReduce is presented in [17]. In more detail, three algorithms are presented that are adaptations of existing skyline algorithms, namely BNL, SFS and Bitmap, appropriately adjusted to work in MapReduce. However, all resulting algorithms require two cascading MapReduce jobs to produce the skyline set. This imposes a significant overhead in the total execution time, simply due to the extra time required for job scheduling in MapReduce. Moreover, these algorithms do not employ any filtering optimizations as the ones proposed in our work, which would easily speed up processing.

In summary, existing algorithms for skyline processing in MapReduce suffer from at least one of the following two major shortcomings: (a) they do not support specialized indexes to efficiently prune the search space, or (b) they require multiple MapReduce jobs which imposes a substantial overhead.

3. Preliminaries and Background

In this section, we first present the basic concept of the skyline query (Section 3.1), followed by a brief introduction of preliminary concepts of MapReduce (Section 3.2), and an overview of SpatialHadoop (Section 3.3).

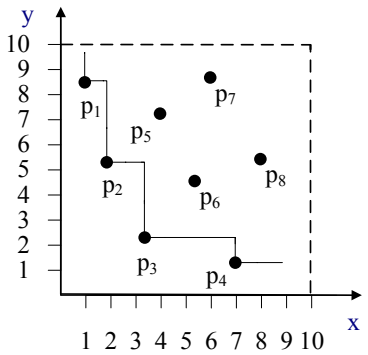

Figure 1: Example of skyline set $SKY = \{p_1, p_2, p_3, p_4\}$.

3.1. Skyline Queries

Consider a spatial database $\mathcal{S}$ that stores points defined in a 2-dimensional space $\mathcal{D}$. A point $p \in S$ can be represented as $p = \{p[1], p[2]\}$ where $p[i]$, is a value on dimension d_i ($i = \{1, 2\}$). The *skyline set* of $\mathcal{S}$ is defined as follows.

Definition 1. ***Skyline:*** *A point $p \in \mathcal{S}$ is said to* dominate *another point $q \in \mathcal{S}$, denoted as $p \prec q$, if (1) on every dimension $d_i \in \mathcal{D}$, $p[i] \leq q[i]$; and (2) on at least one dimension $d_j \in \mathcal{D}$, $p[j] < q[j]$. The* skyline *is a set of points $SKY \subseteq \mathcal{S}$ which are not dominated by any other point. The points in SKY are called skyline points.*

Without loss of generality, in this paper, we assume that skylines are computed with respect to min conditions on all dimensions and that all values are non-negative. Figure 1 shows an example of the skyline of a 2-dimensional dataset.

In a generic distributed setting, where data is horizontally partitioned to machines, the typical method to compute the skyline query of the dataset is to compute the *local skyline* of each partition independently, followed by aggregating the local skyline sets in one machine that eliminates dominated points and computes the *global skyline* (a phase also called merging).

Computing the skyline query in parallel and distributed settings poses significant additional challenges (for a survey of such issues we refer to [7]). In particular, effective *filtering* of local non-skyline points is one technique that prunes points early, thus saves processing cost of local skyline computation and reduces the size of local skyline sets that need to be merged subsequently. In addition, *merging* of local skylines in the global skyline set should also be carefully considered, since it affects the overall cost of reporting the skyline result. In this work, we propose efficient algorithms for skyline computation that are deliberately designed to perform efficient filtering and merging.

3.2. MapReduce

MapReduce [1] is a scalable, flexible and fault-tolerant programming framework for distributed large-scale data analysis. A task to be performed using the MapReduce framework has to be specified as two steps: the *Map* step as specified by a map function takes input (typically from HDFS files), possibly performs some computation on this input, and distributes it to worker nodes, and the *Reduce* step which processes these results as specified by a reduce function. An important aspect of MapReduce is that both

the input and output of the Map step is represented as Key/Value pairs, and that pairs with same key will be processed as one group by the Reducer.

$$map\text{: } (k_1, v_1) \rightarrow list(k_2, v_2)$$
$$reduce\text{: } k_2, list(v_2) \rightarrow list(v_3)$$

In addition to the above basic MapReduce concepts, an interesting optimization that is not obligatory is the use of a *Combiner* function. Its role is to run on the output of the Map phase and perform some filtering or aggregation in order to reduce the number of keys passed to the Reducer. As such, a Combiner takes input the output of Map, and produces output that is essentially the input of Reduce. Typically, the logic of the Combiner is identical to that of the Reducer.

3.3. SpatialHadoop

SpatialHadoop [3] is an extension of the basic Hadoop implementation. It is designed for efficient processing of spatial data, and achieves this by supporting spatial indexing, a feature missing from basic Hadoop. SpatialHadoop utilizes a two-layered spatial index which enables selective access to data by spatial operations. Implemented indexes include R-trees and Grid files. In more detail, SpatialHadoop uses a single *global index* and several *local indexes*. The global index maintains information about the data partitions across cluster nodes. The local indexes organize data stored on single nodes.

Skyline query processing in SpatialHadoop consists of four steps: partitioning, pruning, local skyline processing and global skyline processing.

- In the first step, data is partitioned according to an R-tree, instead of random partitioning (which is the default partitioning used in Hadoop). Each HDFS block corresponds to an R-tree index cell.
- Second, when the query is issued, the master node examines the descriptors of all R-tree index cells (namely their lower left and upper right corners), and prunes those cells that are guaranteed not to include any skyline points.
- Then, local skyline processing is performed on each non-pruned cell (partition) in parallel on different machines.
- Finally, a single machine collects all local skyline points and computes the skyline of the set.

4. A Framework for Skyline Processing in SpatialHadoop

In this section, we describe a framework for efficient skyline query processing that is applicable on top of SpatialHadoop. The proposed framework is a generic MapReduce job that can be parameterized by specific implementations, thus various individual jobs can be created that are members of the framework. In general, the framework is organized similarly to the way skyline processing is performed in SpatialHadoop. However, we introduce optimizations at different phases during skyline processing that improve the performance significantly.

In more detail, our framework employs three advanced techniques for improved performance: (a) a simple filtering technique (*CellsFilter*) that capitalizes on the indexes in SpatialHadoop to prune the search space, (b) the use of a Combiner that drastically reduces the workload of the Reducer, and (c) specialized filters integrated in the Map and Reduce phases for pruning dominated points. CellsFilter corresponds to the second step of SpatialHadoop (as described in Section 3.3), namely pruning of cells. The Combiner performs local skyline processing and corresponds to the third step of SpatialHadoop. Finally, the filters that we employ in the Map and Reduce phases optimize the performance of local and global skyline processing respectively, thus they are related to the third and fourth step of SpatialHadoop. In the following, we describe the functionality of these techniques at high level, and later, in Section 5, we explain the individual methods used for implementing the general functionality.

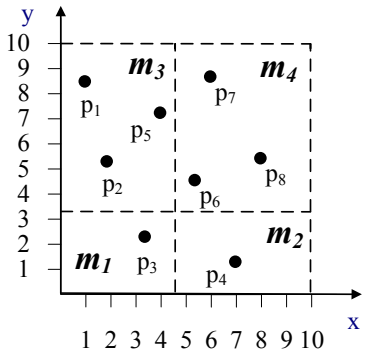

Figure 2: Example of CellsFilter in SpatialHadoop depicting 4 cells: $m_1 \ldots m_4$.

CellsFilter. Before the map phase begins, we exploit the indexes built by SpatialHadoop to prune cells that cannot contribute to the final skyline

set. This is standard practice in skyline algorithms when some form of grid-partitioning is used. The basic idea is that if a cell m_i dominates another cell m_j, then the cell corresponding to m_j can be safely pruned. A cell is said to dominate another cell, if its upper right corner dominates the other cell's lower left corner. In turn, this means that there exists a point in the first cell that dominates all points in the second cell. For instance, consider the example of figure 2, which shows a 2-dimensional dataset partitioned in 4 cells. According to the rule for cell domination, cell m_1 dominates m_4 thus m_4 is pruned for further processing. In fact, m_4 will not be processed at all in the Map phase. As a result, many candidate skyline points are pruned early, and the Map phase has much fewer input records to process.

Combiner. The local skyline processing is essentially implemented by means of a Combiner. The Combiner takes as input the output of the map and computes on different splits the skyline points. We use in the Combiner the same logic as in the Reducer, since both perform skyline processing but on different inputs (different sets of points). The Combiner practically decreases the size of individual map outputs, by eliminating dominated points. In consequence, the Reducer will be much faster because many points will have been already pruned by the Combiner.

Specialized filters. In skyline query processing, a typical technique used to prune non-skyline points is to use some carefully selected points as *filters*. The role of filter points is to eliminate as many as possible candidate points with a small number of comparisons, thus saving processing cost later when the candidates need to be checked to produce the final result. To achieve this goal, filter points need to satisfy the following requirements: (a) the probability $P(f \prec p)$ of a filter point f dominating a random point $p \in \mathcal{S}$ of the dataset should be high, and (b) the number of filter points should be limited. The former requirement determines the effectiveness of the filter points, i.e., how many points will be dominated and eliminated by them. The latter requirement ensures that the processing cost of checking for domination and eliminating points will not be high. In this paper, we employ carefully selected filter points in different phases of the MapReduce job that improve the efficiency of query processing.

4.1. Algorithmic Description

Algorithm 1 presents the proposed framework for skyline processing in SpatialHadoop. It consists of a single MapReduce job, which is preceded by

Algorithm 1 Framework for Skyline Processing in SpatialHadoop

```
1: function CELLSFILTER(C: set of cells)
2: for all cell c ∈ C do
3:     if c is not dominated by any other cell then
4:         Load c in Map function
5:     end if
6: end for
7: end function
8: function MAP(p: point)
9: if p is not dominated by MAPFILTER then
10:     Update MAPFILTER
11:     output ⟨key, p⟩
12: end if
13: end function
14: function COMBINE, REDUCE(key, P: Set of points)
15: for all p ∈ P do
16:     if p is not dominated by REDUCEFILTER then
17:         Update REDUCEFILTER
18:         if p is guaranteed to be a skyline point then
19:             output ⟨null, p⟩
20:         else
21:             Update(CandidateList, p)
22:         end if
23:     end if
24: end for
25: if CandidateList is not empty then
26:     for all p ∈ CandidateList do
27:         output ⟨null, p⟩
28:     end for
29: end if
30: end function
```

a function (CellsFilter) which exploits the indexing mechanisms of SpatialHadoop and prunes data partitions (i.e., HDFS blocks) that are guaranteed not to include skyline points. Essentially, CellsFilter processes the nodes of the global index and prunes dominated nodes (lines 1–7). Notice that this set of nodes (also called cells) is typically orders of magnitude smaller than the size of the dataset, hence this operation incurs low overhead, yet prunes a big part of the dataset.

The Map function aims to keep only a subset of its input (a set of candidate skyline points) that will be processed by the Reduce function. It takes as input a point p and emits it as output, only if it is not eliminated by the filter ($MAPFILTER$) (lines 8–13). More accurately, if p is not dominated, we first check if the filter can be improved by p. For example, this can occur when p has higher pruning power than an existing filter point. In this case, the filter is enhanced based on p. Obviously, the specific enhancement depends on the filter used and is algorithm-specific. We indicate this enhancement as Update $MAPFILTER$ in line 10. The output of the Map function is point p (if it is not dominated by the filter), and it is emitted with a *key* that can be defined by each algorithm as will be demonstrated in

the next section.

The Reduce function (lines 14–30) aims to examine the candidate skyline points and return the final skyline set. It takes as input a set of points P for each *key*. It also employs a filter, called $REDUCEFILTER$, which is independent of the filter used in the Map function. Each point $p \in P$ is checked against the filter for dominance, in order to prune some points early. In case a point is determined to be a skyline point (line 18), it is immediately returned (line 19). It should be clarified that in some occasions (e.g., when P is accessed in a sorted order) it is possible to return skyline points early, before accessing the complete set of points P [7]. Otherwise, if p cannot be guaranteed to be a skyline point, p is placed in a list which is kept updated (i.e., dominated points are eliminated), and this list is later traversed once to output the final skyline set (line 27). We note that we use the same logic of the Reduce function also in the Combiner.

5. Optimizations

The framework for skyline processing described in Section 4 uses filtering as a black-box in the Map and Reduce functions. Also, the *key* used can be defined by the designer of the algorithm, in order to obtain benefits during query processing at the Reduce function. Based on these potential parameters, we propose two individual algorithms that belong to our framework for skyline query processing. The first algorithm, termed *SKY-FLT*, focuses on the use of a specific filtering technique. The second algorithm, termed *SKY-FLT-SORT*, uses a slightly different filtering technique but also uses the *key* deliberately, in order to force a specific order of examining the tuples at the Reduce function.

5.1. Filtering (SKY-FLT)

The first algorithm maintains both in the Map and Reduce functions a list of candidate skyline points (*CandidateList*). This list contains points that are not guaranteed to be skyline points, yet they have not been dominated by any other point thus far; hence they are candidate skyline points. Any new point p that is going to be processed needs to be checked against this set of candidate points. If p is dominated by another point p' ($p' \prec p, p' \in$ CandidateList), then p is not further processed at all. Otherwise, p is added to the CandidateList, and (in addition) any point p' ($p' \in$ CandidateList)

Algorithm 2 *SKY-FLT*

```
1: function CELLSFILTER(C: set of cells)
2: ...
3: end function
4: function MAP(p: point)
5: if p is not dominated by MAPFILTER then
6:     if p is not dominated by any point in CandidateList then
7:         Update MAPFILTER
8:         Update(CandidateList, p)
9:         output ⟨null, p⟩
10:    end if
11: end if
12: end function
13: function COMBINE, REDUCE(null, P: Set of points)
14: for all p ∈ P do
15:    if p is not dominated by REDUCEFILTER then
16:        if p is not dominated by any point in CandidateList then
17:            Update REDUCEFILTER
18:            Update(CandidateList, p)
19:        end if
20:    end if
21: end for
22: if CandidateList is not empty then
23:    for all p ∈ CandidateList do
24:        output ⟨null, p⟩
25:    end for
26: end if
27: end function
```

that is dominated by p ($p \prec p'$) is evicted from the list. This is a quite standard procedure for skyline algorithms in order to produce the result set.

However, checking for dominance for any new point against the CandidateList is computationally intensive, and induces a substantial overhead. To avoid this overhead, the use of a filtering mechanism is advocated. The filtering mechanism consists of some information that can help discarding many points in an efficient way. To this end, we maintain as filter point $f_{mindist}$ the point with minimum sum of coordinates:

$$f_{mindist} = \{p | p \in S' and \nexists p' \in S' : d(O, p') < d(O, p)\}$$

where $S' \subseteq S$ is the subset of S that has been accessed thus far in the algorithm, O denotes the origin of the axes (the point with zero coordinate values), and $d(p, q) = |p[1] - q[1]| + |p[2] - q[2]|$ is the Manhattan distance between two points. Intuitively, the filter point $f_{mindist}$ is a point with high pruning power, i.e., high value of surface for the rectangular area defined by $f_{mindist}$ and the end of the data space. For instance, in Figure 3, three points p_1, p_2, and p_3 that have been accessed are depicted. Among these points, p_2 has the minimum Manhattan distance from the origin of the axes, and the surface of the dominated area is higher than that of the other two points. Hence, p_2 has the highest probability to dominate any new point, assuming a uniform distribution of points in the 2-dimensional space.

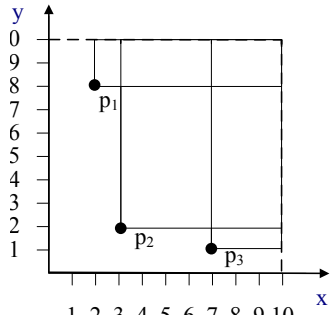

Figure 3: Example of filter point $f_{mindist}$=p_2.

Based on this discussion, we formally define the filters used by the SKY-FLT algorithm as follows. We stress that as more points are being accessed (by either a Map or Reduce function) the filters are updated and their pruning power is improved.

Definition 2. ***MAPFILTER:*** *The filter used in the Map phase of SKY-FLT is the point $f_{mindist}$ among all accessed points by a Map function.*

Definition 3. ***REDUCEFILTER:*** *The filter used in the Reduce phase of SKY-FLT is the point $f_{mindist}$ among all accessed points by the Reduce function.*

Algorithm 2 present the pseudocode of *SKY-FLT*, which uses the $f_{mindist}$ filter both in the Map and Reduce functions, i.e., as $MAPFILTER$ and $REDUCEFILTER$ respectively. For instance, as points are being accessed in some Map task, the $MAPFILTER$ maintains the point (from the ones already accessed) which is closest to the origin of the axes, based on Manhattan distance. The same holds for the $REDUCEFILTER$. If a point p is not dominated by the filter, the CandidateList is updated, and also the filter is updated. Essentially the filter update procedure is that when the Manhattan distance $d(O, p)$ of p is lower than $d(O, f_{mindist})$, then p becomes $f_{mindist}$.

In the following, we provide the proof of correctness of Algorithm 2.

Theorem 1. *(Correctness) Algorithm SKY-FLT returns the correct skyline set.*

Proof. It suffices to show that SKY-FLT does not (a) discard skyline points, and (b) return non-skyline points. For the first part, a point p' is discarded by SKY-FLT only if it is dominated by either $MAPFILTER/REDUCEFILTER$ or by a point in the CandidateList. In any of these cases, the dominating point p is an actual point of the dataset ($p \in P$), and $p \prec p'$, thus p' is not a skyline point. For the second part, the points returned as skyline points are the ones that eventually remained in the CandidateList. We prove that these points are guaranteed to be skyline points by contradiction. Assume a point p' that remained in the CandidateList is not a skyline point. Then, there exists another skyline point p which dominates p', and p also remained in the CandidateList since no other point could have dominated it. But then p' could not be in the CandidateList, since it would have evicted either in line 16 or in line 18 of the Algorithm (according to whether it is accessed after or before p). □

5.2. Sorting (SKY-FLT-SORT)

The previous algorithm essentially introduces a filtering mechanism that improves the performance of query processing, by pruning candidate skyline points early. In this section, we introduce an additional optimization, namely related to final merging of local skyline points in the Reduce function. Recall that *SKY-FLT* needs first to create a CandidateList of points in the Reduce function, and then process this set of points to discover skyline points. The algorithm that we introduce here, termed *SKY-FLT-SORT*, avoids this two-step procedure, by determining a deliberate order of appearance of points in the Reduce function. In this way, skyline points can be reported immediately as soon as they are accessed in the Reduce function, thereby improving the performance of query processing.

The solution to this problem is to make the Reduce function process points in such an order, such that no point can dominate a previously accessed point. This can be achieved by determining a sorted order of examination for any point in the Reduce function. To this end, we use as *key* in the Map function the sum of each point's p coordinates, i.e., $p[1]+p[2]$. The following lemma ensures the correctness of this approach.

Lemma 1. *For any two points $p, q \in S$, if $p[1] + p[2] \leq q[1] + q[2]$, then $q \not\prec p$.*

Proof. By contradiction. Assume $q \prec p$. Then, by Definition 1, either (a) $q[1] < p[1]$ and $q[2] \leq p[2]$, or (b) $q[1] \leq p[1]$ and $q[2] < p[2]$. In both cases,

$q[1] + q[2] < p[1] + p[2]$ which is a contradiction. □

It is well-known that when the key/value pairs sent to a Reduce task belong to multiple keys, these keys are processed in sorted order. By using the sum of a point's coordinates as *key*, we can guarantee that the Reduce function does not need to check whether a new point dominates a previously accessed point. This observation is exploited by *SKY-FLT-SORT* to improve the efficiency of query processing.

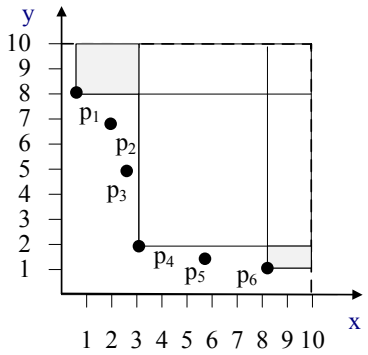

Figure 4: Example of filter points f_{minx}=p_1 and f_{miny}=p_6.

Moreover, the filtering mechanism used by *SKY-FLT-SORT* is enhanced so that we maintain two additional filter points f_{minx}, f_{miny} besides $f_{mindist}$. These points are the ones with minimum coordinate values for each dimension:

$$f_{minx} = \{p | p \in S' and \nexists p' \in S' : p'[1] < p[1]\}$$
$$f_{miny} = \{p | p \in S' and \nexists p' \in S' : p'[2] < p[2]\}$$

Intuitively, these points have extreme minimum values in each dimension respectively, and they can dominate points that may not be dominated by $f_{mindist}$. Therefore, we increase the pruning power of the filtering mechanism, at the cost of maintaining three filter points instead of one. Figure 4 demonstrates the potential benefit of using these additional filter points f_{minx}=p_1 and f_{miny}=p_6. Compared to using only $f_{mindist}$=p_4, the additional benefit is graphically depicted by the grey-shadowed areas. Points in these areas will be dominated and pruned by either f_{minx} or f_{miny}, whereas they could not have been pruned by the filter point $f_{mindist}$.

SKY-FLT-SORT uses only a MAPFILTER and no REDUCEFILTER. This filter is formally presented in Definition 4.

Definition 4. ***MAPFILTER:*** *The filter used in the Map phase of SKY-FLT-SORT consists of three points: $f_{mindist}$, f_{minx} and f_{miny}, among all accessed points by a Map function.*

Algorithm 3 *SKY-FLT-SORT*

```
1: function CELLSFILTER(C: set of cells)
2: ...
3: end function
4: function MAP(p: point)
5: if p is not dominated by MAPFILTER then
6:     Update MAPFILTER
7:     output ⟨p[1] + p[2], p⟩
8: end if
9: end function
10: function COMBINE, REDUCE(key, P: Set of points)
11: for all p ∈ P do
12:     if p[1] < min_x or p[2] < min_y then
13:         output ⟨p[1] + p[2], p⟩
14:         Add(CandidateList, p)
15:         Update min_x, min_y
16:     else if p is not dominated by CandidateList then
17:         output ⟨p[1] + p[2], p⟩
18:         Add(CandidateList, p)
19:     end if
20: end for
21: end function
```

Algorithm 3 describes the pseudocode of *SKY-FLT-SORT*. In the $MAPFILTER$, three filter points are used: $f_{mindist}$, f_{minx}, and f_{miny}. Altogether, these filter points are responsible for pruning new points as early as possible. Moreover, the Map function outputs points that were not eliminated by the filter using the sum of their coordinate values as key. This enables sorted access in the Reduce function, as described above. In the Reduce function, we do not need to use any filtering, because it would not provide significant benefits due to the employed sorting. In contrast, we employ an optimization for avoiding some searches in the CandidateList. Namely, we maintain the minimum x-value min_x and the minimum y-value min_y seen so far, and when a point p has $p[1] < min_x$ or $p[2] < min_y$, we add it to the CandidateList immediately, since no other point in the CandidateList can dominate p. This is a simple trick that saves us the cost of searching the CandidateList in vain in some occasions.

The following Theorem formally proves that the result produced by SKY-FLT-SORT is correct.

Theorem 2. *(Correctness) Algorithm SKY-FLT-SORT returns the correct skyline set.*

Proof. It suffices to show that SKY-FLT-SORT does not (a) discard skyline points, and (b) return non-skyline points. For the first part, a point p' is discarded by SKY-FLT-SORT only if it is dominated by either $MAPFILTER$ or by a point in the CandidateList. In any of these cases, the dominating point p is an actual point of the dataset ($p \in P$), and $p \prec p'$, thus p' is not a skyline point. For the second part, recall that points p are accessed in the Reduce function in increasing order of the sum of their coordinates $p[1]+p[2]$. Then, based on Lemma 1, no unseen point p' can dominate an already seen point p. Thus, when a point p is accessed we only need to check if already seen points dominate p, otherwise p is a skyline point. In Algorithm 3, the points p returned as skyline points must belong to one of the following cases:

- *At least one of p's coordinates is smaller than the minimum coordinate seen so far*: Then, no already seen point can dominate p, since p has at least one coordinate that is minimum among already seen points.
- *p is not dominated by any already seen point*: Then, it is straightforward that p is a skyline point, since it is not dominated by any already seen point.

□

6. Experimental Evaluation

In this section, we present the results of our experimental evaluation. All algorithm are implemented in Java and the experiments were performed on two different clusters of machines with SpatialHadoop installed.

6.1. Experimental Setup

Datasets. For the dataset of 2-dimensional points we employ synthetic data collections of various distributions, in order to be able to vary their size and study the scalability of our approach. The synthetic datasets follow the distributions: (1) uniform (UN), (2) correlated (CO), (3) (moderately) anti-correlated (AC1), and (4) (highly) anti-correlated (AC2). For UN dataset, the values for the two dimensions are generated independently using a uniform distribution. The CO and AC datasets are generated as described in [4]. Additionally, we change the default values of the anti-correlated data generator, in order to increase the anti-correlation and create a dataset with a substantial number of skyline points. This dataset is denoted AC2. To

illustrate the comparative difference of AC1 to AC2, we report the skyline size of AC1 to be approximately 4K points, whereas the same number for AC2 is approximately 160K. As a result, AC2 is a hard setup for any skyline algorithm, due to extremely large number of skyline points.

In addition, we used a real dataset (all_nodes[1]) from OpenStreetMap which contains 1.7 billion points on the planet and is 62.3GB. We used uniform sampling to create three sample datasets of all_nodes of sizes 1, 10 and 20GB.

Platforms. We performed experiments on two cloud platforms, one medium-sized and one small-sized, which were provided to us by external parties. The first experimental platform was provided by the BonFIRE research project [18]. In more detail, we use a cluster of 17 nodes which were installed and configured to run SpatialHadoop. One node is used as NameNode, one as JobTracker, one as Secondary NameNode, and the remaining nodes are DataNodes. The second platform was provided by *Okeanos*[2], an IAAS service for the Greek Research and Academic Community. Due to limited storage availability by the Okeanos platform at the time we were provided access to the service, we configured only four nodes running SpatialHadoop, one master node and three DataNodes. In the following, we denote in the captions of each chart the platform (BonFIRE or Okeanos) on which the respective experiment run.

Algorithms. We implemented and tested our algorithms SKY-FLT and SKY-FLT-SORT that comply with the framework proposed in this paper. For comparative purposes, we also implemented the skyline algorithm (denoted CG-HADOOP in the charts) proposed in [5] by the designers of SpatialHadoop.

Metrics. The main metric used is the running time of each algorithm. We also measured other metrics including map input and output records, combine input and output records, and reduce input and output records. We show the running time in all charts, and we comment on other metrics when necessary, to explain the behavior of each algorithm. Each experiment is executed 10 times and average values are used when reporting results. In addition, we measured the standard deviation and standard error, to ensure that our results are statistically significant. In all cases, the standard error

[1]Available at: `http://spatialhadoop.cs.umn.edu/datasets.html`

[2]`https://okeanos.grnet.gr/home/`

was very small compared to the average value measured.

6.2. Experimental Results on Synthetic Data

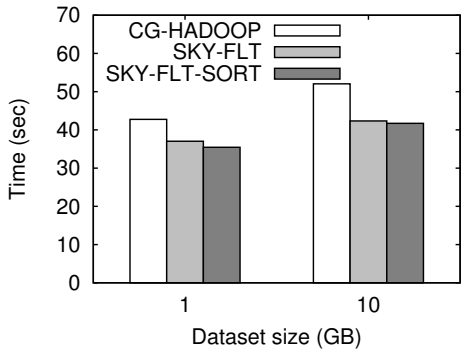

Figure 5: Uniform dataset (BonFIRE platform).

Uniform (UN) dataset. Figure 5 shows the obtained results for the Uniform dataset when increasing the size of the dataset by a factor of 10. Both SKY-FLT and SKY-FLT-SORT perform better than CG-HADOOP and the gain in running time is increased with dataset size. This indicates that our algorithms scale better than the competitor as the size of the data increases. SKY-FLT-SORT is slightly faster than SKY-FLT, due to the more efficient merging employed. We also observed that the CellsFilter prunes around 50% of the Map input records in the small dataset and almost 85% of the Map input records in the large dataset. Recall that CellsFilter is used by all three algorithms. Therefore, for the uniform dataset, simply the use of CellsFilter makes CG-HADOOP quite efficient, however still our algorithms perform better.

Correlated (CO) dataset. Figure 6 presents the results in the case of the Correlated dataset. This data distribution typically produces few skyline points, and is considered an easy setup for any skyline algorithm. In the small dataset, SKY-FLT-SORT performs better than SKY-FLT because the sorting of the Map output records allows better performance in the Reduce phase. In the large dataset, SKY-FLT-SORT performs worse because it has the overhead to sort many more points than in the small dataset. In fact, SKY-FLT outputs fewer than 0.07% of the SKY-FLT-SORT Map output records. Obviously, the sorting improves the performance of the Reducer but when it comes to such a big difference in Map output records (without

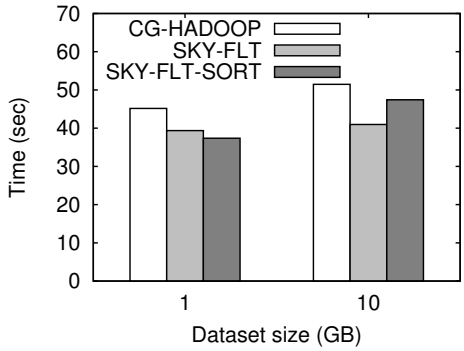

Figure 6: Correlated dataset (BonFIRE platform).

having many global skyline points), the cost of sorting itself is higher than the benefit offered during query processing.

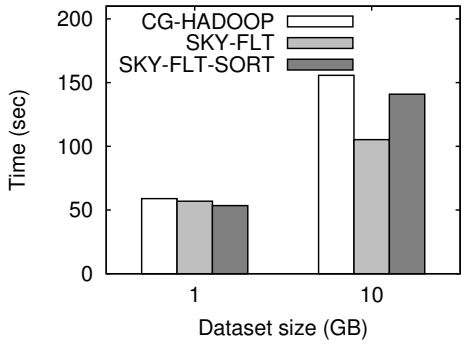

Figure 7: Anti-correlated dataset (BonFIRE platform, medium case: 4K skyline points).

Anti-correlated (AC1) dataset. In Figure 7, the first Anti-correlated dataset is used, which produces around 4K skyline points. In the small dataset, SKY-FLT-SORT is slightly better than SKY-FLT and CG-HADOOP. SKY-FLT-SORT outputs many more points than SKY-FLT, but it sorts them and improves the performance of the Reduce function. SKY-FLT is also better than CG-HADOOP because it prunes many dominated points by using the MAPFILTER and REDUCEFILTER. In the large dataset, both algorithms are still better than CG-HADOOP. However, in this case, SKY-FLT performs better than SKY-FLT-SORT. The reason is that in SKY-FLT-SORT the Combiner receives many input records to process, and this stalls

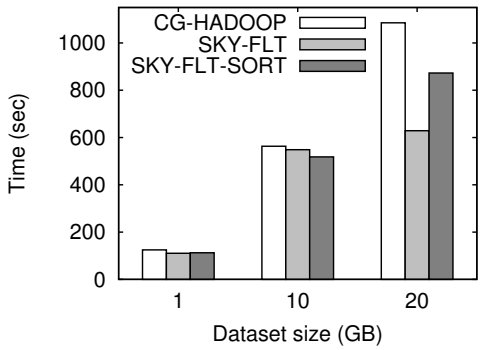

Figure 8: Anti-correlated dataset (Okeanos platform, medium case: 4K skyline points).

the overall completion of the job. Instead, the CandidateList used by SKY-FLT in the Map phase drastically reduces the input size of the Combiner and provides a better overall performance.

In Figure 8, we repeat the same experiment on the Okeanos platform, and we additionally use a larger dataset of 20GB. Again, our algorithms are better than the competitor (CG-HADOOP) in all cases. We also observe a similar trend as in Figure 7, namely that for the larger dataset SKY-FLT outperforms SKY-FLT-SORT, for the same reasons as outlined earlier. Moreover, this experiment demonstrates the scalability of our algorithms as we double the size of the input data, from 10GB to 20GB. In the case of 20GB, both our algorithms require less than twice the time needed for the 10GB dataset.

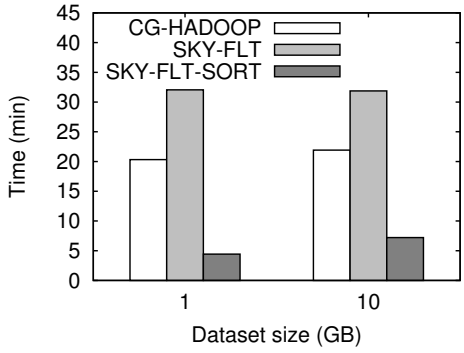

Figure 9: Anti-correlated dataset (BonFIRE platform, hard case: 160K skyline points).

Anti-correlated (AC2) dataset. Figure 9 shows the results in the case

of the Anti-correlated dataset with many skyline points. In this experiment SKY-FLT-SORT is much better than the other two algorithms (notice that the y-axis is in minutes, not seconds). In particular, SKY-FLT-SORT is 4 times better than CG-HADOOP. SKY-FLT-SORT sorts its Map output records and it can perform the Reduce step efficiently to improve performance. Also, its MAPFILTER is not so time-consuming compared to SKY-FLT, which now has to keep a CandidateList with many candidate skyline points which produces a significant overhead. This is the main reason for the low performance of SKY-FLT. However, this experiment clearly demonstrates the value of SKY-FLT-SORT for hard cases of skyline processing, where the output size can be really large.

6.3. Experimental Results on Real Data

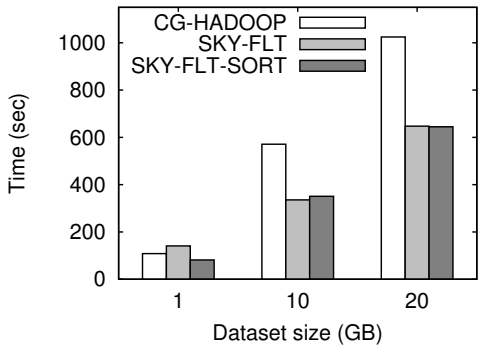

Figure 10: Real dataset (Okeanos platform).

In addition, we performed experiments using a real dataset (all_nodes) from OpenStreetMap on the Okeanos platform. The complete dataset is 62.3GB, and we created three subsets of size 1, 10 and 20GB using random sampling, in order to test the scalability of the proposed algorithms. This experiment aims to verify that the benefits of our algorithms are sustained in the case of non-synthetic data too.

Figure 10 depicts the results acquired using the real dataset. Our algorithms outperform the competitor as the size of the input dataset grows, thus verifying the results obtained from the synthetic datasets. More importantly, we observe that as the size of the dataset increases, the gap between our algorithms and the competitor also increases. For instance, when comparing

CG-HADOOP to SKY-FLT-SORT, the improvement is 25%, 37% and 40% for 1, 10 and 20GB respectively.

7. Conclusions

In this paper, we addressed the challenging problem of computing the skyline of a large spatial dataset in an efficient and scalable way. To this end, we proposed a framework for query processing based on SpatialHadoop, an extension of Hadoop with support for spatial operations. We presented two algorithms that belong to our framework and rely on effective filtering and efficient merging, in order to produce the skyline set. We compared our approach experimentally against the algorithm proposed by the designers of SpatialHadoop. The results show the merits of our approach, especially in the case of demanding datasets with many skyline points, where our approach significantly outperforms the competitor algorithm.

8. Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments that improved the paper. The authors would also like to thank May Chawk for help with running additional experiments needed for the revision of the paper. This publication contains results generated using the BonFIRE multi-site cloud facility. The authors wish to thank the BonFIRE Foundation for their help and support. The authors also thank the Greek Research and Technology Network (GRNET) for providing access to the Okeanos cloud platform.

The research of C. Doulkeridis has been co-financed by ESF and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Aristeia II, Project: ROADRUNNER.

9. References

[1] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: Proc. of OSDI'2004, 2004.

[2] C. Doulkeridis, K. Nørvåg, A survey of analytical query processing in MapReduce, VLDB J.

[3] A. Eldawy, M. F. Mokbel, A demonstration of SpatialHadoop: An efficient MapReduce framework for spatial data, PVLDB 6 (12) (2013) 1230–1233.

[4] S. Börzsönyi, D. Kossmann, K. Stocker, The skyline operator, in: Proceedings of ICDE'2011, 2001.

[5] A. Eldawy, Y. Li, M. F. Mokbel, R. Janardan, CG_Hadoop: Computational geometry in MapReduce, in: SIGSPATIAL/GIS, 2013, pp. 284–293.

[6] J. Chomicki, P. Ciaccia, N. Meneghetti, Skyline queries, front and back, SIGMOD Record 42 (3) (2013) 6–18.

[7] K. Hose, A. Vlachou, A survey of skyline processing in highly distributed environments, VLDB J. 21 (3) (2012) 359–384.

[8] A. Vlachou, C. Doulkeridis, Y. Kotidis, Angle-based space partitioning for efficient parallel skyline computation, in: Proceedings of SIGMOD'2008, 2008.

[9] H. Köhler, J. Yang, X. Zhou, Efficient parallel skyline processing using hyperplane projections, in: Proceedings of SIGMOD'2011, 2011.

[10] F. N. Afrati, P. Koutris, D. Suciu, J. D. Ullman, Parallel skyline queries, in: Proceedings of ICDT'2012, 2012.

[11] S. Liknes, A. Vlachou, C. Doulkeridis, K. Nørvåg, APSkyline: Improved skyline computation for multicore architectures, in: Proceedings of DASFAA'2014, 2014.

[12] S. Park, T. Kim, J. Park, J. Kim, H. Im, Parallel skyline computation on multicore architectures, in: Proceedings of ICDE'2009, 2009.

[13] J. Selke, C. Lofi, W.-T. Balke, Highly scalable multiprocessing algorithms for preference-based database retrieval, in: Proceedings of DASFAA'2010, 2010.

[14] R. Torlone, P. Ciaccia, Finding the best when it's a matter of preference, in: Proceedings of SEBD'2002, 2002.

[15] Y. Park, J.-K. Min, K. Shim, Parallel computation of skyline and reverse skyline queries using MapReduce, PVLDB 6 (14) (2013) 2002–2013.

[16] Y. Tao, W. Lin, X. Xiao, Minimal MapReduce algorithms, in: Proceedings of SIGMOD'2013, 2013, pp. 529–540.

[17] B. Zhang, S. Zhou, J. Guan, Adapting skyline computation to the MapReduce framework: Algorithms and experiments, in: DASFAA Workshops, 2011, pp. 403–414.

[18] K. Kavoussanakis, A. Hume, J. Martrat, C. Ragusa, M. Gienger, K. Campowsky, G. van Seghbroeck, C. Vázquez, C. Velayos, F. Gittler, P. Inglesant, G. Carella, V. Engen, M. Giertych, G. Landi, D. Margery, BonFIRE: the Clouds and Services Testbed, in: 5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2013.