Efficient Skyline Computation in MapReduce

Kasper Mullesgaard[†], Jens Laurits Pedersen[†], Hua Lu[†], Yongluan Zhou[‡]

[†]*Department of Computer Science, Aalborg University, Denmark*
jmzebub@gmail.com, thekaninos@gmail.com, luhua@cs.aau.dk

[‡]*Department of Mathematics and Computer Science, University of Southern Denmark, Denmark*
zhou@imada.sdu.dk

ABSTRACT

Skyline queries are useful for finding interesting tuples from a large data set according to multiple criteria. The sizes of data sets are constantly increasing and the architecture of back-ends are switching from single-node environments to non-conventional paradigms like MapReduce. Despite the usefulness of skyline queries, existing works on skyline computation in MapReduce do not take full advantage of parallelism but still run significant parts serially. In this paper, we propose a novel approach to compute skylines efficiently in MapReduce. We design a grid partitioning scheme to divide the data space into partitions, and employ a bitstring to represent the partitions. The bitstring is efficiently obtained in MapReduce, and it clearly helps prune partitions (and tuples) that cannot have skyline tuples. Based on the grid partitioning, we propose two MapReduce algorithms to compute skylines. Both algorithms utilize the bitstring and distribute the original tuples to multiple mappers and make use of them to compute local skylines in parallel. In particular, MapReduce Grid Partitioning based Single-Reducer Skyline Computation (MR-GPSRS) employs a single reducer to assemble the local skylines appropriately to compute the global skyline. In contrast, MapReduce Grid Partitioning based Multiple Reducer Skyline Computation (MR-GPMRS) further divides local skylines and distributes them to multiple reducers that compute the global skyline in an independent and parallel manner. The proposed algorithms are evaluated through extensive experiments, and the results show that MR-GPMRS significantly outperforms the alternatives in various settings.

1. INTRODUCTION

Given a set R of multi-dimensional tuples, a skyline query [4] returns a subset S_R that contains all the tuples in R that are not dominated by any others in R. The dominance relationship between two tuples is defined as follows.

DEFINITION 1. *(**Tuple Dominance**) Tuple r_i dominates tuple r_j if and only if r_i is not worse than r_j for all dimensions and r_i is better than r_j for at least one dimension.*

We call this conventional definition *tuple dominance* to distinguish it from the *partition dominance* that is to be defined in Section 3.1 for partitions of the data space. In the tuple dominance, whether a dimensional value is better or worse than another for one dimension is determined by the semantics of the dimension and/or the configuration of the skyline query. Typically, a value v_1 has to be either larger or smaller than another value v_2 for v_1 to be better than v_2. Without the loss of generality, this paper assumes that a smaller value is better. We use $r_i \prec r_j$ to denote that r_i dominates r_j.

Skyline queries have a wide variety of applications that are characterized by multi-criteria decision as the core problem. Processing skyline queries, also known as skyline computation, is computationally costly. To decide whether a tuple is in the skyline or not, many tuple dominance checks may be needed and each check may involve all d dimensions. Skyline computation is both IO-consuming and CPU-intensive in the centralized settings. Therefore, for the sake of overall efficiency, it is interesting to compute skylines in the distributed and/or parallel settings.

According to a recent survey [10], a considerable number of approaches have been proposed for skyline processing in distributed and/or parallel environments. However, very few [6, 20] have studied skyline computation in the MapReduce platform, although it is being increasingly used to process massive data for its scalability and fault-tolerance. The availability of scalable MapReduce systems, such as Hadoop [2], makes it desirable to leverage such systems for large-scale parallel skyline computation. It is noteworthy that the existing techniques for distributed and parallel skyline computation follow a paradigm radically different from MapReduce, as they require arbitrary inter-node communication and coordination.

On the other hand, the existing MapReduce skyline algorithms [6, 20] are not well designed in that they do not take full advantage of parallelism but still run significant parts serially. The main difficulty lies in that global skyline tuples cannot be decided solely based on local information of each individual data partition and hence the lack of inter-mapper and inter-reducer communication in MapReduce limits the parallelism of the computation.

In this paper, we propose a novel approach to compute skylines efficiently in MapReduce. We design a grid partitioning scheme to divide the data space into partitions, and use a compact bitstring to represent the partitions. The bitstring is efficiently obtained in MapReduce, and it not

10.5441/002/edbt.2014.05

only gives all nodes an overview of the input data but also helps prune data partitions that cannot have skyline tuples. Based on the grid partitioning, we propose two MapReduce algorithms to compute skylines. Both algorithms utilize the bitstring and distribute the original tuples to multiple mappers and make use of them to compute local skylines in parallel. In particular, MapReduce Grid Partitioning based Single-Reducer Skyline Computation (MR-GPSRS) employs a single reducer to assemble the local skylines appropriately to get the global skyline, whereas MapReduce Grid Partitioning based Multiple Reducer Skyline Computation (MR-GPMRS) further divides local skylines and distributes them to multiple reducers that compute the independent parts of the global skyline in parallel.

We make the following contributions in this paper.

- We design a compact bitstring representation for a grid based data space partitioning scheme.
- We propose two MapReduce skyline algorithms by utilizing the bitstring to prune data and parallelize processing.
- We give a model to estimate the costs of the proposed skyline algorithms in MapReduce.
- We evaluate the effectiveness and efficiency of the proposals through extensive experiments.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 details the grid partitioning scheme and the bitstring. Section 4 presents the MapReduce Grid Partitioning based Single-Reducer Skyline Computation (MR-GPSRS) algorithm. Section 5 elaborates on the MapReduce Grid Partitioning based Multiple Reducer Skyline Computation (MR-GPMRS) algorithm. Section 6 gives a model to estimate the costs for the algorithms. Section 7 reports on the experimental studies. Section 8 concludes the paper and discusses future work.

2. RELATED WORK

In this section, we briefly review MapReduce and the existing works on skyline computation in MapReduce.

2.1 The MapReduce Framework

MapReduce is a framework for distributed computing. As illustrated in Figure 1, it is based on a Map and a Reduce function [9]. The Map function is invoked for each record in the input file and it produces a list of key-value pairs, i.e., $Map(k1, v1) \rightarrow list(k2, v2)$. The Reduce function is then invoked once for each unique key and the associated list of values, and produces key-value pairs that are the part result of the MapReduce job, i.e., $Reduce(k2, list(v2)) \rightarrow list(k3, v3)$. All lists of $(k3, v3)$ from all calls of Reduce constitute the complete result for the entire input to MapReduce. Several MapReduce jobs can be chained together, later phases being able to refine and/or use the results from earlier phases.

A distributed file system is used to store the data processed and produced by the MapReduce job. An input file is split up, stored, and possibly replicated on the different nodes in the cluster where MapReduce is running. The nodes are then able to access their local splits when processing data. When the data from the Map function has been processed by the different nodes, the results are shuffled between the nodes so the required data can be accessed locally

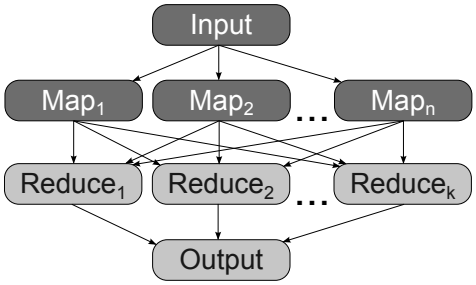


Figure 1: MapReduce process

when the Reduce function is invoked. The operations of data split, storage, and replication are automatically done by the MapReduce infrastructure without user intervention.

It can be necessary to replicate data across nodes. In Hadoop [2], the implementation of MapReduce we use, the Distributed Cache can be used for this purpose. When a MapReduce job starts, data written to the Distributed Cache is transferred to all nodes, making it accessible in the Map and Reduce functions. This paper assumes that the Distributed Cache, or something similar, is available.

2.2 Skyline Computation in MapReduce

Considerable amounts of works [3, 5, 8, 11, 13, 16, 18, 19, 21] have been proposed for skyline processing in parallel and distributed computing environments. A non-exhaustive review can be found in a survey [10]. However, such works are not suitable for MapReduce. The skyline algorithms in such works rely heavily on flexible inter-node communications to coordinate distributed and/or parallel processing among nodes, whereas MapReduce does not support inter-mapper or inter-reducer communications and mapper/reducer communication is strictly constrained by the key-value form. A recent work [1] also applies grid-based partitioning for parallel skyline queries; however, it uses computational models that are substantially different from MapReduce.

Zhang et al. [20] adapt three centralized skyline algorithms to the MapReduce framework. The MapReduce - Block Nested Loop (MR-BNL) algorithm partitions each data dimension into two halves, distribute the resulting data partitions to mappers, and compute local skyline on each mapper using the Block Nested Loop (BNL) [4] skyline algorithm. Finally, all local skylines are sent to a single reducer to compute the global skyline. The MapReduce - Sort Filter Skyline (MR-SFS) algorithm has the same overall process as MR-BNL but it applies the presorting technique [7] to compute local skylines. In addition, the MR-Bitmap algorithm uses the bitmap algorithm [14] to determine dominance in skyline computation on each node. Although MR-Bitmap is able to use multiple reducers for global skyline computing, it can only handle data dimensions with limited number of distinct values.

Chen et al. [6] adapt the angular partitioning technique [17] and propose the MapReduce - Angle (MR-Angle) skyline algorithm. Angular partitioning divides the data space using angles, motivated by the observation that skyline tuples are located near the origin. In MR-Angle, angle based data partitions are distributed to mappers for local skyline computation, and a single reducer is used to find the global skyline.

This work distinguishes from existing proposals [6, 20] by

three important points. First, this work employs a generalized grid partitioning schema to partition the data space, and designs a bitstring structure to represent the grid partitioning. It is noteworthy that the bits in our bitstring indicate empty partitions and dominated partitions, whereas the bit flags used in [20] are merely codes for data partitions but not for data contents. As a result, our design allows early and much more aggressive pruning of unpromising data partitions before all partitions are sent to mappers for local skyline computation. Second, this work makes use of multiple reducers to parallelize the global skyline computing. Third, unlike MR-Bitmap, this work is able to handle data dimensions with arbitrary number of distinct values. In the experimental studies, we compare our algorithms to MR-BNL and MR-Angle. We skip MR-SFS as it is less efficient than MR-BNL; we skip MR-Bitmap because it cannot apply to the continuous numeric data domains that we work on in this research.

Park et al. [12] propose another MapReduce skyline algorithm SKY-MR. Before starting MapReduce, SKY-MR obtains a random sample of the entire data set and builds a quadtree for the sample to identify dominated sampled regions. In contrast, the bitstring used in this work does not require sampling, and it is built in parallel by MapReduce.

3. GRID PARTITIONING AND BITSTRING

To compute skylines efficiently in MapReduce, it is critical to utilize the parallelism provided by the paradigm. For that purpose, we use a $n \times n$ grid to partition the data space such that the skyline computation is parallelized on different partitions. The partitioning also allows us to prune unpromising partitions that cannot contain any skyline tuples, which further speeds up the skyline computation.

In this section, we detail the grid partitioning scheme and its bitstring representation. Section 3.1 presents the partitioning scheme, Section 3.2 describes the bitstring and a MapReduce algorithm for generating the bitstring, and Section 3.3 discusses how to decide the value of n in the grid partitioning. Table 1 lists the notations used in this paper.

Table 1: Frequent Notations

Symbol	Interpretation
R	A set of tuples
S_R	The skyline of R
r, r_i, r_j, t	A tuple in R
n	Partitions per dimension (PPD)
d	Dimensionality of R
p, p_i, p_j	A partition of the data space
P	A set of partitions
m	The number of mappers
BS	A bitstring for the grid partitioning
IG	A group of independent partitions

3.1 Grid Partitioning of Data Space

We employ an $n \times n$ grid to partition the data space. As a result, each dimension is divided into n parts and there are n^d partitions in total for a d-dimensional space. We refer to n as *partitions per dimension* (PPD), and use P to denote the set of all n^d partitions. A naive method computes the local skylines for all the n^d partitions and merges them correctly to get the global skyline as the final result.

As a matter of fact, it is not necessary to compute all the n^d local skylines. Similar to the dominance relationship between two tuples, dominance relationship can also be defined for two partitions. The main difference here is that the dominance relationship between two partitions is based on their corner points because each of them may have many points. Given a partition p, we use $p.max$ and $p.min$ to denote its maximum corner and minimum corner respectively. The maximum corner of a partition is defined as the partition's corner that has the highest (worst) values on all dimensions. Likewise, the minimum corner of a partition is defined as the corner that has the lowest (best) values on all dimensions. In the context of skyline computation, the dominance relationship between the partitions $p_1, p_2, \ldots, p_{n^d} \in P$ can be exploited to exclude unpromising partitions that cannot contain skyline tuples. In the following, we give the relevant definitions for partitions.

Definition 2. *(**Partition Dominance**) A partition p_i dominates another partition p_j, denoted by $p_i \prec p_j$, if and only if $p_i.max$ dominates $p_j.min$. In other words, $p_i \prec p_j \Leftrightarrow p_i.max \prec p_j.min$.*

Accordingly, we have the following lemma. Its correctness can be easily proved according to the transitivity property [4] of tuple dominance. This lemma allows us to prune an entire partition like p_j without computing its local skyline.

Lemma 1. *Given two partitions p_i and p_j, if $p_i \prec p_j$, then any tuple in p_i dominates all tuples in p_j, i.e., $\forall r_i \in p_i$ and $\forall r_j \in p_j$ we have $r_i \prec r_j$.*

We also adapt the definitions of dominating region [11] and anti-dominating regions [15] for partitions in our setting.

Definition 3. *(**Dominating Region**) A partition p_i's dominating region, denoted as $p_i.DR$, contains all partitions that are dominated by p_i, i.e., $p_i.DR = \{p_j \mid p_j \in P \wedge p_i \prec p_j\}$.*

Definition 4. *(**Anti-dominating Region**) A partition p_i's anti-dominating region, denoted as $p_i.ADR$, contains all partitions that may have tuples that dominate $p_i.max$, i.e., $p_i.ADR = \{p_j \mid p_j \in P \wedge p_j.min \prec p_i.max\}$.*

An example is shown in Figure 2 where the data space is partitioned by a 3×3 grid. For partition p_4, its dominating region is $\{p_8\}$ and its anti-dominating region is $\{p_0, p_1, p_3\}$. If we know that partition p_4 is not empty, i.e., it contains at least one tuple in R, partition p_8 can be safely pruned without local skyline computation. On the other hand, we only need to compare p_4's local skyline to those of p_0, p_1, and p_3 in order to merge it into the global skyline.

Therefore, it is important to discover whether a partition is empty or not. We proceed to design an effective and efficient mechanism to capture such information in MapReduce.

3.2 Bitstring Representation and Generation

In the grid partitioning, the only partitions of interest are those that are not empty with respect to the given tuple set R. We represent the $n \times n$ partitioning scheme as a bitstring $BS_R(0, 1, 2, \ldots, n^d - 1)$ where for $0 \leq i \leq n^d - 1$

$$BS_R[i] = \begin{cases} 1, & \text{if } p_i \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

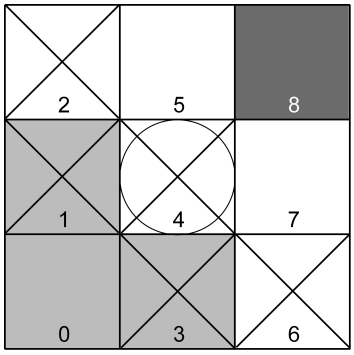


Figure 2: An example of grid partitioning

The resulting bitstring can be in either row-major or column-major order. The only difference is how the index (i) of a partition in the bitstring is calculated. The column-major order is used in this paper. We refer to the running example shown in Figure 2, where non-empty partitions are marked with crosses and partition indexes are given in integers from 0 to 8. As a result, the bitstring is 011110100.

The next problem is how to generate such a bitstring BS_R efficiently for a given tuple set R. In MapReduce, we can parallelize the bitstring generation through multiple mappers. The idea is as follows. First, the original tuple set R is divided into disjoint subsets $R_1, R_2, \ldots, R_m$ and each of them is sent to a corresponding mapper. Subsequently, a mapper receiving R_i $(1 \leq i \leq m)$ obtains a local bitstring BS_{R_i} according to the grid partitioning scheme and Equation 1. Further, all such local bitstrings are sent to a single reducer where the bitwise or operator is applied to all of them to obtain BS_R, i.e., $BS_R = BS_{R_1} \vee BS_{R_2} \vee \ldots \vee BS_{R_m}$.

After the global bitstring BS_R is obtained, it is possible to prune unpromising partitions by traversing BS_R. This can be done according to the dominating relationships for partitions defined in Section 3.1. In particular, if $p_j \prec p_i$ for partitions $p_i, p_j \in P$, we reset $BS_R[i]$ to 0 to exclude partition p_i from further consideration in the skyline computation. This way can result in fewer partitions and data tuples to be involved in the skyline computation. Finally, the reducer generates a bitstring BS_R where for $0 \leq i \leq n^d - 1$

$$BS_R[i] = \begin{cases} 0, & \text{if } p_i = \emptyset \text{ or } \exists p_j (p_i \in p_j.DR); \\ 1, & \text{otherwise.} \end{cases} \quad (2)$$

The flow of the bitstring generation in MapReduce is illustrated in Figure 3. Algorithms 1 and 2 are for the mappers and the reducer respectively.

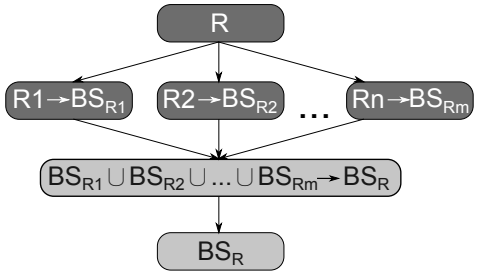


Figure 3: The flow of bitstring generation in MapReduce

Algorithm 1 *Mapper of the bitstring generation*

Input: A subset R_i of the data set R, the dimensionality of the data set d, and the PPD n.
Output: A bitstring BS_{R_i} for all the n^d partitions with respect to R_i.

```
1: Initialize a bitstring BS_{R_i} with all n^d bits set to 0
2: for each tuple t ∈ R_i do
3:     Decide the partition p_j that t belongs to
4:     BS_{R_i}[j] ← 1
5: Output(null, BS_{R_i})
```

Algorithm 2 *Reducer of the bitstring generation*

Input: The set of all local bistrings BSs, the dimensionality of the data set d, and the PPD n.
Output: A bitstring BS_R for all the n^d partitions with respect to R.

```
1: Initialize a bitstring BS_R with all n^d bits set to 0
2: for each bitstring BS_{R_i} ∈ BSs do
3:     BS_R ← BS_R ∨ BS_{R_i}
4: for each i from 0 to n^d − 1 do
5:     if BS_R[i] = 1 then
6:         for each partition p_j ∈ p_i.DR do
7:             BS_R[j] ← 0
8: Output(null, BS_R)
```

3.3 Choosing the Number of Partitions per Dimension

The Partitions per Dimension (PPD) is an important parameter for the proposed MapReduce skyline algorithms because PPD determines the number of tuples per partition (TPP) and thus the workloads for mappers and reducers in skyline computation. If TPP is too small, comparing grid partitions to check partition dominance is not worthwhile compared to checking the tuple dominance within each of those partitions. Conversely, if TPP is too high, the grid partitioning is too rough and checking partition dominance cannot prune many partitions. It is not a trivial task to set an appropriate PPD for grid partitioning.

Therefore, we propose a heuristic here for choosing the number n for PPD. Let c be the cardinality of tuple set R, d be the dimensionality, and TPP is the desired number of tuples per partition. Assuming an independent distribution of all tuples in the data space, we have the following equation

$$\frac{c}{n^d} = \text{TPP} \quad (3)$$

From this, n is resolved as:

$$n = \sqrt[d]{\frac{c}{\text{TPP}}} \quad (4)$$

To derive n according to Equation 4, we need the ideal value for TPP, which however depends on various factors including data characteristics (cardinality, dimensionality, distribution, etc.) and mapper/reducer capacities. Therefore, we turn to an indirect way of deciding n that makes estimates for TPP.

Specifically, we extend the MapReduce steps for grid partitioning (Section 3.2 and Figure 3) as follows. As the reducer cannot trigger the mappers to redo the grid partitioning, we make the mappers do a series of partitioning and

generate a series of local bitstrings accordingly, using different PPD values from 2 to $n_m = \sqrt[d]{c}$. The series of local bitstrings, denoted as $BS^2_{R_i}, BS^3_{R_i}, \ldots, BS^{n_m}_{R_i}$ on the i-th mapper, are all sent to the reducer. The reducer merges the local bitstrings that result from the same PPD value and gets $n_m - 1$ global bitstrings $BS^2_R, BS^3_R, \ldots, BS^{n_m}_R$. In particular, $BS^j_R = BS^j_{R_1} \vee BS^j_{R_2} \vee \ldots \vee BS^j_{R_m}$, where $2 \leq j \leq n_m$. For each BS^j_R, the reducer counts the non-empty partitions as ρ and make an estimate for TPP as $\text{TPP}_e = c/\rho$. On the other hand, we have TPP$= c/j^d$ according to Equation 3. For all global bitstrings BS^j_Rs, we check $|\text{TPP}_e - \text{TPP}| = |c/\rho - c/j^d|$ and find the one $BS^{\widetilde{n}}_R$ that results in the minimal absolute difference. This global bitstring (and the grid partitioning scheme indicated by it) is to be used in the subsequent MapReduce skyline computation.

4. GRID PARTITIONING BASED SINGLE-REDUCER SKYLINE COMPUTATION IN MAPREDUCE

In this section, we present a MapReduce algorithm MR-GPSRS: Grid Partitioning based Single Reducer Skyline computation algorithm. It accepts a tuple set R and a corresponding bitstring BS_R as input, employing multiple mappers and a single reducer to compute the skyline S_R for R. During the computation, it makes use of the bitstring BS_R for pruning. The overall flow of MR-GPSRS is illustrated in Figure 4.

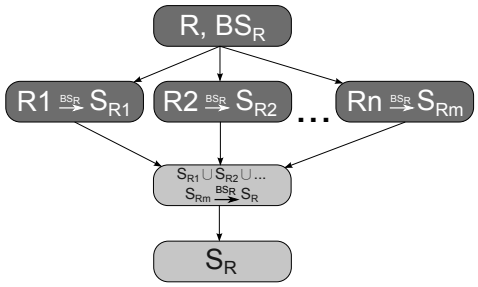


Figure 4: The flow of MR-GPSRS

The Map step of MR-GPSRS is shown in Algorithm 3. It takes a subset R_i of R and the global bitstring BS_R as input, and processes each tuple t in R_i as follows. First, it finds the partition p_j that contains t (line 3). Tuple t is further processed only if p_j's corresponding bit in the bitstring is 1 (line 4). In particular, p_j's corresponding local skyline S_{p_j} is either initialized as $\{t\}$ (lines 5–6) or updated with respect to t (line 8). The update is done by Algorithm 4 that works like the BNL algorithm [4]. It adds t to the local skyline if t is not dominated by current local skyline tuples; it also removes those local skyline tuples that turn out to be dominated by t.

After all tuples in R_i are processed, variable $\mathcal{S}$ contains a series of local skylines (S_ps) each of which corresponds to an unpruned partition p that contains tuple(s) in R_i. Given two involved partitions p_i and p_j, it is possible that p_i's corresponding local skyline S_{p_i} have some tuples that dominate those in p_j's corresponding local skyline S_{p_j}, and vice versa. In order to remove such false positives[1], we call another function ComparePartitions (Algorithm 5) for each S_p in $\mathcal{S}$ (lines 9–10). Finally, the local skylines organized as $\mathcal{S}$ is output (line 11).

Algorithm 5 takes a local skyline S_p for partition p and a set $\mathcal{S}$ of local skylines for other partitions as input. For each local skyline S_{p_i} in $\mathcal{S}$ (line 1), the algorithm checks if the partition p_i is in $p.ADR$ (line 2). If positive, S_{p_i} may have tuples that dominate those in S_p and those dominated tuples are removed from S_p (line 3). After the whole $\mathcal{S}$ is checked, the algorithm returns S_p without any false positives with respect to $\mathcal{S}$.

Algorithm 3 *Map of MR-GPSRS*

Input: A subset R_i of R, and the bitstring BS_R.
Output: R_i's local skyline S_{R_i} (in the form of a set of local skylines S_{p_j}s for non-empty partitions p_js).

```
1: S ← ∅
2: for each tuple t ∈ R_i do
3:     Decide the partition p_j that t belongs to
4:     if BS_R[j] = 1 then
5:         if S_{p_j} = ∅ then
6:             S_{p_j} ← {t}; Add S_{p_j} to S
7:         else
8:             S_{p_j} ← InsertTuple(t, S_{p_j})
9: for each local skyline S_p ∈ S do
10:    S_p ← ComparePartitions(S_p, S \ {S_p})
11: Output(null, S)
```

Algorithm 4 *InsertTuple*

Input: A tuple t, and a set s of local skyline tuples
Output: The updated local skyline, i.e., $S_{s\cup\{t\}}$.

```
1: check = true
2: for each tuple t' ∈ s do
3:     if t' ≺ t then
4:         check = false
5:         BREAK
6:     if t ≺ t' then
7:         remove t' from s
8: if check then
9:     add t to s
10: return s
```

Algorithm 5 *ComparePartitions*

Input: A local skyline S_p for partition p, and a set of local skylines $\mathcal{S}$ for other partitions
Output: A reduced S_p such that all its tuples dominated by a local skyline tuple from $\mathcal{S}$ are removed.

```
1: for each local skyline S_{p_i} ∈ S do
2:     if partition p_i ∈ p.ADR then
3:         remove from S_p all those tuples that are dominated by tuples in S_{p_i}
4: return S_p
```

The Reduce step of MR-GPSRS is shown in Algorithm 6. It receives the local skylines $\mathcal{S}_1$, $\mathcal{S}_2$, …, $\mathcal{S}_m$ from all map-

[1]False positives refer to those local skyline tuples that are excluded from the global skyline.

pers and merges them into the correct global skyline. For the sake of simple presentation, each $\mathcal{S}_k$ $(1 \leq k \leq m)$ can be regarded as an array $(S^k_{p_0}, S^k_{p_1}, \ldots, S^k_{p_{n^d-1}})$. Each element $S^k_{p_i}$ $(0 \leq i \leq n^d - 1)$ is part of R_k's local skyline corresponding to partition p_i. Note that some of those elements in such an array can be empty. The first part of Algorithm 6 (lines 1–6) effectively merges all such k arrays into one $(S_{p_0}, S_{p_1}, \ldots, S_{p_{n^d-1}})$ where S_{p_i} is the skyline of $\bigcup_{1 \leq k \leq m} S^k_{p_i}$.

In a similar way to lines 9–10 in Algorithm 3, the second part of Algorithm 6 (lines 7–8) calls function ComparePartitions for each S_{p_i}, which removes the false positives from S_{p_i} with respect to all other S_{p_j}s $(0 \leq j \leq n^d - 1$ and $j \neq i)$. Finally, the global skyline is obtained and output (line 9).

Algorithm 6 *Reduce of MR-GPSRS*

Input: The local skylines $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$ from all mappers.
Output: The global skyline S_R.

```
1: for each partition p ∈ P do
2:     S_p ← ∅
3:     for each local skyline S_k from S_1 to S_m do
4:         Get the local skyline S_p^k for partition p from S_k
5:         for each tuple t ∈ S_p^k do
6:             S_p ← InsertTuple(t, S_p)
7: for each partition p_i ∈ P do
8:     ComparePartitions(S_{p_i}, {S_{p_0}, ..., S_{p_{i-1}}, S_{p_{i+1}}, ..., S_{p_{n^d-1}}})
9: Output(null, ⋃_{p∈P} S_p)
```

5. GRID PARTITIONING BASED MULTIPLE-REDUCER SKYLINE COMPUTATION IN MAPREDUCE

MR-GPSRS relies on a single reducer for computing the global skyline, which becomes a bottleneck when the skyline is larger. The problem can be alleviated by utilizing the grid partitioning technique to identify subsets of partitions for which the global skyline can be computed independently. As a result, multiple reducers can be used, each responsible for computing an independent part of the global skyline.

In this section, we propose MR-GPMRS: Grid Partitioning based Multiple-Reducer Skyline computation algorithm.

5.1 Overall Idea

The grid partitioning scheme allows for identifying *independent groups*: Sets of partitions that can be processed independently to obtain the skyline of those partitions.

DEFINITION 5. ***(Independent Partition Group)*** *A set of partitions P_I is an independent partition group if and only if the following holds: $\forall p \in P_I \Rightarrow p.ADR \subseteq P_I$.*

Given a tuple set R and a set of partitions P_x, we use R_{P_x} to denote the subset of all tuples that fall in the partitions in P_x. Independent groups guarantee a very useful property as described in the following lemma.

LEMMA 2. *If P_I is an independent partition group, then R_{P_I}'s local skyline must be a part of the global skyline, i.e., $S_{R_{P_I}} \subseteq S_R$.*

PROOF. $\forall t \in S_{R_{P_I}}$, we need to prove $t \in S_R$. We prove it by contradiction. Suppose $\exists t \in S_{R_{P_I}}$ and $t \notin S_R$. In S_R, there must be at least one tuple t' that dominates t. We have $t' \notin R_{P_I}$; otherwise, we would have $t \notin R_{P_I}$.

Let the partition containing t be p, and the one containing t' be p'. According to Definition 4, $p' \in p.ADR$. As $t \in S_{R_{P_I}}$, we have $t \in R_{P_I}$ and $p \in P_I$ according to the meaning of R_{P_I}. As P_I is an independent partition group, we have $p' \in P_I$. Again, by the meaning of R_{P_I}, we have $t' \in R_{P_I}$. Thus, contradiction is reached and the lemma is proved. □

This lemma allows us to use multiple reducers to compute local skylines in parallel. Specifically, we group non-empty grid partitions in P into g independent partition groups $P_{I_1}, P_{I_2}, \ldots, P_{I_g}$. Accordingly, each mapper divides its local skyline S_{R_i} into g subsets $S_{R_i,1}, S_{R_i,2}, \ldots, S_{R_i,g}$, and sends each of the g subsets to a corresponding reducer. To ease the presentation, we look at the j-th reducer. It receives local skyline subsets $S_{R_1,j}, S_{R_2,j}, \ldots, S_{R_m,j}$, each from a corresponding mapper. We use $\widehat{S}_{R,j}$ to denote the union $S_{R_1,j} \cup S_{R_2,j} \cup \ldots \cup S_{R_m,j}$. Note that $\widehat{S}_{R,j} \subseteq R_{P_{I_j}}$ and $\widehat{S}_{R,j} \supseteq S_{R_{P_{I_j}}}$ because local skyline computations on mappers cannot have false negatives[2] but false positives. As a result, the local skyline of $\widehat{S}_{R,j}$ is $S_{R_{P_{I_j}}}$ and thus a part of the global skyline. Upon receiving $\widehat{S}_{R,j}$ $(1 \leq j \leq g)$, the j-th reducer computes and outputs the local skyline $S_{R_{P_{I_j}}}$ independently as it is a part of the global skyline due to the reasoning above. In this way, the computations of reducers are parallelized without involving a final single reducer for eliminating false positives. The overall flow of MR-GPMRS is illustrated in Figure 5, where $S_{R_{P_{I_j}}}$ is referred to as S_j $(1 \leq j \leq g)$ for simplicity.

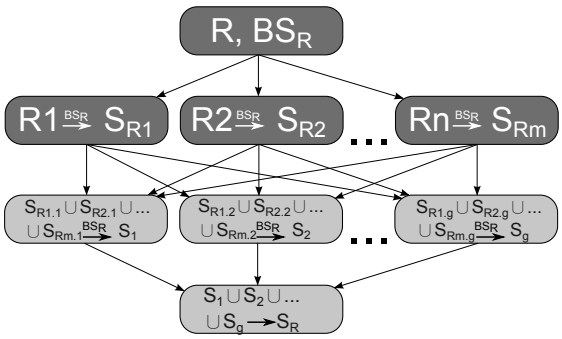


Figure 5: The flow of MR-GPMRS

5.2 Generation of Independent Partition Groups

In this section, we discuss how to generate independent partition groups for a given set of grid partitions P. We need the following definition.

DEFINITION 6. ***(Maximum Partition)*** *A non-empty partition $p_m \in P$ is a maximum partition if and only if the following holds: $\forall p \in P \Rightarrow p_m \notin p.ADR$.*

[2]False negatives refer to those global skyline tuples that are excluded in local skylines.

Maximum partitions help the generation of independent groups as follows. Starting from the partition in P with the highest index, we look for maximum partitions. When a maximum partition p_m is encountered, we use it as a seed and get all partitions that may have tuples dominating those in p_m, i.e., $p_m.ADR$. We thus get the union of $\{p_m\}$ and $p_m.ADR$ as one independent group. We then set the bits for all used partitions to 0, and start to look for the next maximum partition and so on so forth. The generation is formalized in Algorithm 7. The correctness of this algorithm can be proved by the properties of Definitions 4, 6, and 6. We omit the proof here due to the page limit.

Algorithm 7 *GenerateIndependentGroups*

Input: The set P of grid partitions, and the bitstring BS_R.
Output: A set of independent partition groups IG.

```
1: IG ← ∅
2: while BS_R ≠ 0 do
3:     Get the partition p_m with the largest index m from P
4:     ig ← {p_m} ∪ p_m.ADR
5:     for each partition p_i ∈ ig do
6:         BS_R[i] ← 0
7:     Add ig to IG
8: return IG
```

An example is shown in Figure 6 where non-empty partitions are shaded. Partition p_6 is the first maximum partition encountered by the generation algorithm as it is not in he anti-dominating region of any other partitions. Thus, the independent group from p_6 and $p_6.ADR = \{p_3\}$ is $IG_1 = \{p_3, p_6\}$. Next, maximum partition p_4 is encountered and independent group $IG_2 = \{p_1, p_3, p_4\}$ is generated. Finally, maximum partition p_2 results in the last independent group $IG_3 = \{p_1, p_2\}$. It may be necessary to replicate some partitions, e.g., partitions p_1 and p_3 in Figure 6, among the independent groups as they lie in the anti-dominating regions of partitions in multiple groups. However, independent groups cannot be subsets of each other.

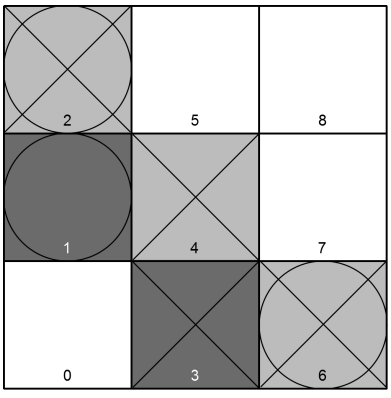


Figure 6: An example of generation of independent groups

5.3 The MR-GPMRS Algorithm

Like the MR-GPSRS algorithm, the MR-GPMRS algorithm also accepts a tuple set R and a corresponding bitstring BS_R as input. Its Map step is described in Algorithm 8, and its Reduce step is in Algorithm 9.

The first part (lines 1–10) of Algorithm 8 is the same as the counterpart of Algorithm 3, computing the local skylines for each involved grid partition. After that, the bitstring BS_R is used to generate the independent groups (line 11). Note that this step is the same on all mappers, so that the independent groups are generated consistently by all of them. Otherwise, inconsistency of independent groups across mappers would cause wrong skyline results on reducers. The last part (lines 12–19) of Algorithm 8 distributes the local skylines to corresponding reducers according to the independent groups. The description given in Section 5.1 has the simplifying assumption that the number of independent groups equals that of reducers. Algorithm 8 lifts the assumption and sends independent groups to reducers in a round-robin way.

Algorithm 8 *Map of MR-GPMRS*

Input: A subset R_i of R, the bitstring BS_R, and the number of reducers r.
Output: R_i's local skyline S_{R_i} (in the form of a set of local skylines S_{p_j}s for non-empty partitions p_js).

```
1: S ← ∅
2: for each tuple t ∈ R_i do
3:     Decide the partition p_j that t belongs to
4:     if BS_R[j] = 1 then
5:         if S_{p_j} = ∅ then
6:             S_{p_j} ← {t}; Add S_{p_j} to S
7:         else
8:             S_{p_j} ← InsertTuple(t, S_{p_j})
9: for each local skyline S_p ∈ S do
10:    S_p ← ComparePartitions(S_p, S \ {S_p})
11: IG ← GenerateIndependentGroups(S, BS_R)
12: i = 0
13: for each independent partition group ig ∈ IG do
14:    S_i ← ∅
15:    for each partition p ∈ ig do
16:        Get local skyline S_p from S
17:        S_i ← S_i ∪ S_p
18:    Output(i % r + 1, (S_i, ig))
19:    i++
```

The Reduce step of MR-GPMRS in Algorithm 9 is overall similar to that of MR-GPSRS in Algorithm 6. Nevertheless, unlike Algorithm 6 that organizes local skylines with respect to all partitions in set P, Algorithm 9 only involves partitions in the input independent group ig. Thus, the workload of the single reducer is distributed to multiple reducers independently. This independency and parallelism significantly improves the overall skyline computation efficiency.

5.4 Implementation Issues

In this section, we discuss two issues that need consideration in the implementation of MR-GPMRS.

5.4.1 Merging Independent Groups

Problems may arise when there are more independent partition groups than reducers. Extra communication cost may incur as some partitions and their local skylines are sent multiple times from a mapper to reducers. Also, reducers may have imbalanced computation load due to the partitions they receive. The round-robin way of distributing the groups to the reducers, as described in Section 5.3, does not

Algorithm 9 *Reduce of MR-GPMRS*

Input: The local skyline parts $S_{R_1,j}, S_{R_2,j}, \ldots, S_{R_m,j}$ from all mappers, and the independent partition group ig.
Output: The skyline of $\bigcup_{1 \le i \le m} S_{R_i,j}$.

```
1: S ← ∅
2: for each partition p ∈ ig do
3:     S_p ← ∅
4:     for each i from 1 to m do
5:         Get the local skyline S_p^i for partition p from S_{R_i,j}
6:         for each tuple t ∈ S_p^i do
7:             S_p ← InsertTuple(t, S_p)
8:     Add S_p to S
9: for each each partition p ∈ ig do
10:    ComparePartitions(S_p, S \ {S_p})
11: Output(null, ⋃_{p∈ig} S_p)
```

resolve these two problems well.

In our implementation, we merge independent groups when there are more of them than reducers. One option of merging is based on optimizing the communication cost. Specifically, independent groups that have the most partitions in common are merged. This method, however, does not guarantee the load balance among the reducers as this can make some reducers receive more different partitions than others.

An alternative is to merge independent groups based on the estimated computation cost. Given an independent group $IG_i = \{p_m\} \cup p_m.ADR$, we estimate the computation cost as the number of partitions in $p_m.ADR$, i.e., $|p_m.ADR|$. The intuition behind it is that function ComparePartitions(.) (Algorithm 5) in the Reduce step and the size of $p_m.ADR$ is a critical factor for the execution time of the function.

We conducted preliminary tests to compare the two merging options. The computation cost based merging results in more balanced loads among reducers and better overall efficiency. Therefore, we use this option in our experimental studies.

5.4.2 Elimination of Duplicates in Skyline Outputs

When grid partitions, and their corresponding local skylines as well, are replicated in different (merged) independent groups sent to different reducers, the local skylines for those partitions are computed and reported in duplicates by those reducers. In order to eliminate the duplicates in skyline outputs, it is necessary to control which reducers output the local skylines for the replicated partitions.

In the implementation, we choose a particular independent group as the *responsible* group for a partition that has replicates. We send a designation notification to the reducer when the responsible group is sent out together with the output from a mapper. Consequently, a reducer only computes and outputs the local skyline for a replicated partition if it receives the designation notification.

The responsible group is chosen based on the estimated computation cost. Following the same line of reasoning in Section 5.4.1, we choose the independent group $IG_i = \{p_m\} \cup p_m.ADR$ with the minimal $|p_m.ADR|$. This is intended to not further burden reducers that already have higher computation costs.

6. COST ESTIMATION

In this section, we discuss the cost estimation for the proposed algorithms. In particular, we are interested in the number of partition-wise comparisons performed between different grid partitions in the MapReduce skyline computation, i.e., how many times the critical operation of function ComparePartitions(.) (line 3 in Algorithm 5) is executed. We focus on this function because it is called by both map and reduce steps and it constitutes the most critical part in the grid partitioning based skyline algorithms. The notations used in the cost estimation are shown in Table 2.

To ease the cost estimation, we make two assumptions. First, we assume that each grid partition generated in each mapper is non-empty. Second, we assume that calling function ComparePartitions(.) by a mapper does not cause empty partitions. That is to say, comparing different grid partitions does not prune partitions but only part of the tuples in them. As a matter of fact, these assumptions stipulate a worst-case scenario and thus the relevant estimate is an upper bound of the real cost.

We consider the partitions that survive the bitstring based pruning. A d-dimensional grid has a number of $d-1$-dimensional surfaces equal to $d \times 2$. After the partition pruning using the bitstring (from Equation 1 to Equation 2 in Section 3.2), half of these surfaces, i.e. d surfaces, are filled with remaining partitions. The other d surfaces, as well as the rest of the partitions, are dominated. An example is shown in Figure 6. In this 3×3 2-dimensional grid, there are $2 \times 2 = 4$ 1-dimensional surfaces. In terms of partitions, the four surfaces are: $surf_1 = \{p_2, p_1, p_0\}$, $surf_2 = \{p_0, p_3, p_6\}$, $surf_3 = \{p_6, p_7, p_8\}$, and $surf_4 = \{p_8, p_5, p_2\}$. If each partition was non-empty, then partitions p_4, p_5, p_7, and p_8 would be dominated and pruned by using the bitstring. This would leave $d = 2$ intact surfaces, namely $surf_1$ and $surf_2$. The overlap of remaining surfaces must be considered. In this example, the overlap between the remaining surfaces $surf_1$ and $surf_2$ is partition p_0.

We use $p_{rem}(n, d)$ to denote the number of remaining partitions after the bitstring based pruning. It can be calculated by subtracting the number of partitions of a $n-1 \times n-1$ grid from that of a $n \times n$ grid, because the former captures the number of pruned partitions in a $n \times n$ grid partitioning scheme. Therefore, we have

$$\rho_{rem}(n, d) = n^d - (n-1)^d \tag{5}$$

In the running example, the pruned partitions, namely p_4, p_5, p_7, and p_8, can be regarded as forming a 2×2 grid. So the number of remaining partitions after pruning for the 3×3 grid is $3^2 - 2^2 = 5$.

Furthermore, the partition-wise comparisons to be done for a single partition p depends on its anti-dominating region, as p is compared against another partition p_j only if $p_j \in p.ADR$. We use $p_{dom}(n, d)$ to denote the number of such partition-wise comparisons for a partition p. This number is equal to the product of p's coordinates[3] in the grid minus one:

$$\rho_{dom}(n, d) = p.d_1 \times p.d_2 \times \ldots \times p.d_d - 1 \tag{6}$$

In the running example shown in Figure 6, partition p_2 has coordinates (1, 3) in the grid. The number of partition-wise

[3]On each dimension, the coordinates start with 1 from the origin.

Table 2: Notations for Cost Estimation

Symbol	Interpretation
$\rho_{rem}(n,d)$	The number of remaining partitions in a grid after bitstring based partition pruning
$\rho_{dom}(n,d)$	The number of partition-wise comparisons for a single partition
$\kappa(n,d)$	The number of partition-wise comparisons for a single surface in a grid
$\kappa_{mapper}(n,d)$	The number of partition-wise comparisons for a single mapper
$\kappa_{reducer}(n,d)$	The number of partition-wise comparisons for the reducer with the highest workload

comparisons for p_2 is thus $1 \times 3 - 1 = 2$. This is consistent with the fact that we need to compare p_2 with p_1 and p_0 in function ComparePartitions(.).

Summing up for all partitions in a surface using Equation 6, we calculate the total number of partition-wise comparisons in $\kappa(n,d)$:

$$\kappa(n,d) = \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} \dots \sum_{i_d=1}^{n} (i_1 \times i_2 \times \dots \times i_d - 1) \quad (7)$$

However, to get the number of partition-wise comparisons for all surfaces, the overlap between surfaces must be considered. We calculate the sum as follows. For the first surface, we calculate the number of partition-wise comparisons according to Equation 7. For the second surface, we calculate the number by subtracting the overlap between it and the first from the sum according to Equation 7. For the third surface, we have to subtract the overlap between it and the first, as well as the overlap between it and the second. In this way, we calculate the number for the d surfaces as follows.

$$\kappa_1(n,d) = \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} \dots \sum_{i_{d-1}=1}^{n} (i_1 \times i_2 \times \dots \times i_d - 1)$$

$$\kappa_2(n,d) = \sum_{i_1=2}^{n} \sum_{i_2=1}^{n} \dots \sum_{i_{d-1}=1}^{n} (i_1 \times i_2 \times \dots \times i_d - 1)$$

$$\kappa_3(n,d) = \sum_{i_1=2}^{n} \sum_{i_2=2}^{n} \dots \sum_{i_{d-1}=1}^{n} (i_1 \times i_2 \times \dots \times i_d - 1)$$

$\dots$

$$\kappa_d(n,d) = \sum_{i_1=2}^{n} \sum_{i_2=2}^{n} \dots \sum_{i_{d-1}=2}^{n} (i_1 \times i_2 \times \dots \times i_d - 1)$$

As a result, the number of partition-wise comparisons on a single mapper is estimated as

$$\kappa_{mapper}(n,d) = \sum_{i=1}^{d} s_i(n,d) \quad (8)$$

Note that the estimation above applies to the Map step of both MR-GPSRS and MR-GPMRS. For the single reducer in MR-GPSRS, its number of partition-wise comparisons is estimated in the same way due to our two assumptions.

For a reducer in MR-GPMRS, only a single surface has to be considered. This is because each surface is an independent partition group that is processed independently by a corresponding reducer. The reducer with the most partition-wise comparisons is the one that has the biggest surface, the one for which no overlap is considered. Therefore, the number of partition-wise comparisons on a single reducer in MR-GPMRS is estimated as

$$\kappa_{reducer}(n,d) = s_1(n,d) \quad (9)$$

In Section 7.5, we experimentally evaluate the cost estimations presented in this section.

7. EXPERIMENTAL STUDIES

In this section, we report the results of experimental studies on the proposed MapReduce skyline algorithms.

7.1 Experimental Settings

We compare the proposed algorithms MR-GPSRS and MR-GPSRS with two existing MapReduce skyline algorithms, namely MR-BNL [20] and MR-Angle [6]. All algorithms are implemented in Java.

We use a cluster of thirteen commodity machines to run the experiments. Twelve of the machines have an Intel Pentium D 2.8 GHz Core2 processor. Three of them have 1 GB RAM, four of them have 2 GB RAM, and five of them have 3 GB RAM. The last machine has an Intel Pentium D 2.13 GHz Core2 processor and 2 GB RAM. The machines are connected by a 100 Mbit/s LAN. We use operating system Ubuntu 12.04 to run the machines, and Hadoop 1.1.0 to build the MapReduce environment on the cluster. By default, MR-GPMRS uses one reducer per node in the cluster.

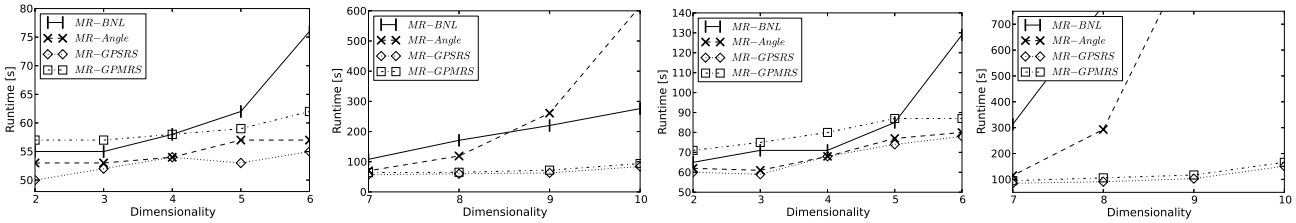
For the tuple set R, we use synthetic data sets of independent and anti-correlated distributions. The data are generated according to the existing methods [4]. The cardinalities of used R are 1×10^5, 5×10^5, 1×10^6, 2×10^6, and 3×10^6. The dimensionality of R is in the range of [2..10].

In most of the experiments, we measure the skyline computation runtime, i.e., the elapsed time from the moment the computation starts to the moment the global skyline is fully output. For MR-GPSRS and MR-GPSRS algorithms, we include the time cost of the bitstring generation in the runtime.

7.2 Effect of Dimensionality

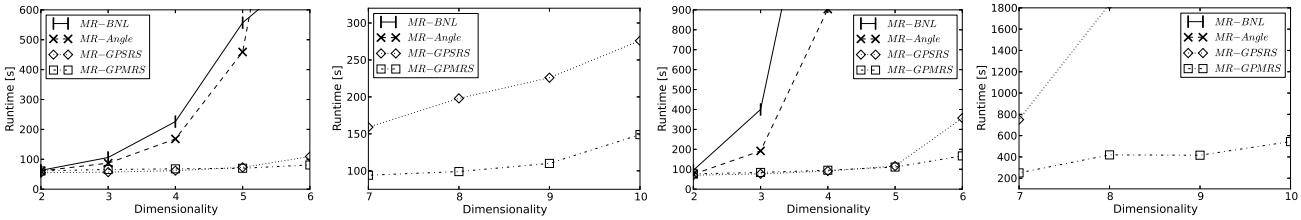
We first investigate the effect of data dimensionality on the MapReduce skyline algorithms. For either data distribution, we use two cardinalities (1×10^5 and 2×10^6), and vary the dimensionality from 2 to 10. The results of runtime of all algorithms are shown in Figures 7 and 8 for independent and anti-correlated distributions respectively.

For independent data distribution, MR-GPSRS performs the best according to the results shown in Figure 7. When the dimensionality is low (from 2 to 5), MR-GPMRS performs slightly worse than the alternatives, as shown in Figures 7(a) and (c). When the dimensionality increases, MR-GPMRS performs very steadily, whereas MR-BNL and MR-Angle deteriorate almost exponentially. In particular, when the dimensionality is from 7 to 10, MR-GPMRS and MR-GPSRS perform comparably and both are significantly



(a) dim. 2-6, card. 1×10^5 (b) dim. 7-10, card. 1×10^5 (c) dim. 2-6, card. 2×10^6 (d) dim. 7-10, card. 2×10^6

Figure 7: Effect of dimensionality on independent data



(a) dim. 2-6, card. 1×10^5 (b) dim. 7-10, card. 1×10^5 (c) dim. 2-6, card. 2×10^6 (d) dim. 7-10, card. 2×10^6

Figure 8: Effect of dimensionality on anti-correlated data

better than the two alternatives, as shown in Figures 7(b) and (d). These observations indicate that the proposed grid partitioning scheme is effective in pruning partitions and data in skyline computation.

The mediocrity of MR-GPMRS in the settings of low dimensionality and cardinality is attributed to the small skyline sizes in those settings. MR-GPMRS has overheads on communication and data loading for the parallelism of using multiple reducers, which do not pay off when the local skylines and the global skyline only occupy a small fraction of the independent data set. In contrast, when the dimensionality increases a larger fraction of tuples enter the skylines and MR-GPMRS is able to compute the skyline more efficiently by parallel reducers, and thus it starts to outperform MR-BNL and MR-Angle.

For the anti-correlated data, as shown in Figure 8, MR-GPMRS is the best in almost all tested settings except that MR-GPSRS is slightly better when the dimensionality is less than 5 (see Figures 8(a) and (c)). For anti-correlated data, a large fraction of the tuples are in the skyline and the fraction becomes much higher when the dimensionality is larger than 5. As a result, the reducer parallelism in MR-GPMRS has an advantage in processing much more skyline tuples. It is clear that MR-GPMRS scales well as the dimensionality increases for both low and high cardinalities.

In contrast, MR-Angle and MR-BNL cannot terminate in a reasonable period of time for higher dimensionalities, and therefore they are excluded in Figures 8(b) and (d). Furthermore, MR-GPSRS performs significantly slower than MR-GPMRS for the low cardinality and high dimensionality data sets (see Figure 8(b)). For the higher cardinality, as shown in Figure 8(d), MR-GPSRS's inferiority is more apparent and MR-GPSRS does not terminate in a reasonable period of time for the highest dimensionality from 8 to 10. The single reducer in MR-GPSRS is not able to efficiently handle the many skyline tuples in higher dimensional anti-correlated data sets.

In summary, when the dimensionality increases a higher fraction of the tuples enter the skyline, and thus MR-GPMRS's ability to find the global skyline tuples in parallel outweighs its increased overhead. For higher dimensionalities and cardinalities, this advantage is even more evident. In other settings where the skyline tuples only occupy a small fraction of the entire data set, MR-GPSRS performs better than MR-GPMRS and using multiple reducers is not worth the extra overhead in MR-GPMRS.

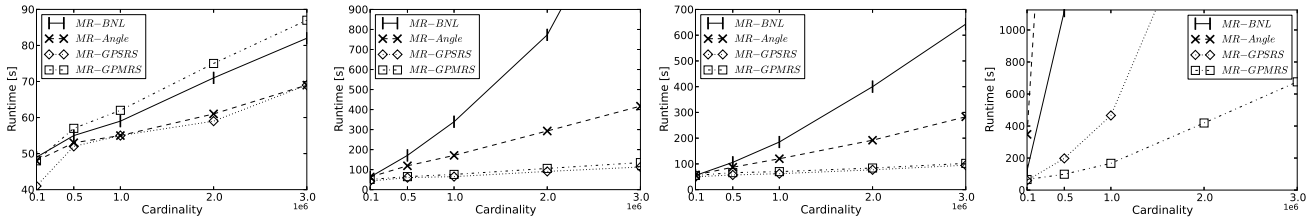
7.3 Effect of Cardinality

In this part of experiments, we investigate the effect of data cardinality on the MapReduce skyline algorithms. We run experiments on 3-dimensional and 8-dimensional data sets of both distributions. We vary the cardinality from 1×10^5 to 3×10^6. The results are reported in Figure 9.

The results obtained from independent data are shown in Figures 9(a) and (b). For the dimensionality 3, as shown in Figure 9(a), MR-GPMRS is the slowest for all cardinalities. This disadvantage is again due to the relative small skyline size in the independent distribution. On the other hand, MR-GPSRS has the best runtime for all cardinalities, whereas MR-Angle ties with it for the cardinalities 1×10^6 and 3×10^6.

For the dimensionality of 8, referring to Figure 9(b), MR-GPMRS and MR-GPSRS run fastest, with MR-GPSRS being slightly faster. The difference is due to the small skyline fraction of the independent data where the multiple reducers do not pay off. The superiority of MR-GPMRS and MR-GPSRS over the alternatives again show that our design of the grid partitioning scheme and the bitstring based partition pruning is effective for skyline computation in MapReduce.

The results obtained from anti-correlated data are shown in Figures 9(c) and (d). Here, MR-GPMRS and MR-GPSRS are superior for covered settings. For the lower dimensionality of 3, MR-GPSRS is marginally better than MR-GPMRS. However, for the higher dimensionality of 8, MR-GPSRS is increasingly worse than MR-GPMRS and fails to terminate



(a) 3-d independent (b) 8-d independent (c) 3-d anti-correlated (d) 8-d anti-correlated

Figure 9: Effect of cardinality

in a reasonable period of time for the highest cardinalities. The superiority of MR-GPMRS is again attributed to the increasing fractions of skyline tuples in the data set.

7.4 Effect of The Number of Reducers

We also investigate the effect of the number of reducers used in MR-GPMRS. For either data distribution, we use a 8-dimensional data set with the cardinality of 2×10^6. We vary the number of reducers from 1 (using MR-GPSRS) to 17. Note that Hadoop allows utilizing the multiple cores in the nodes to implement multiple reducers on the same node. The experimental results are shown in Figure 10.

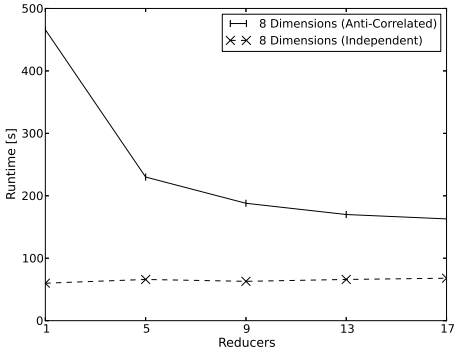


Figure 10: Effect of the number of reducers in MR-GPMRS

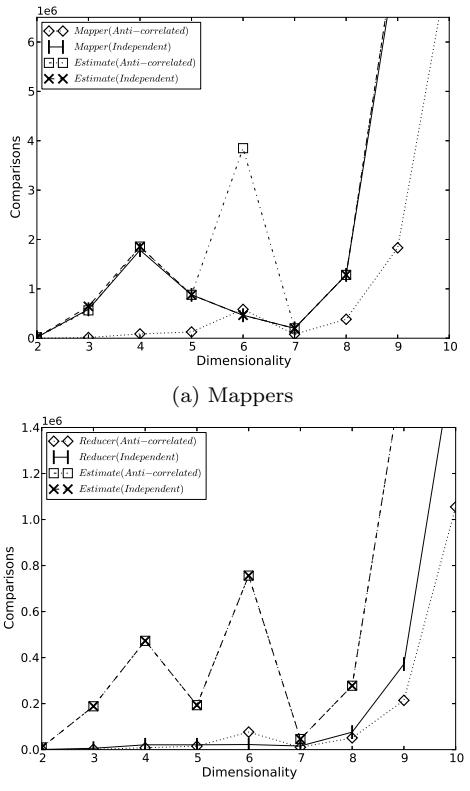
For the independent data set, increasing reducers does not improve the skyline computation runtime. The runtime almost does not change when the number of reducers is increased. There is actually a small increase when the number of reducers is increased from 1 to 5, which is caused by the additional overhead of using multiple reducers.

In contrast, more reducers clearly shortens the runtime for computing skyline on the anti-correlated data set. The largest improvement occurs when the number of reducers is increased from 1 to 5, i.e., switching from MR-GPSRS to MR-GPMRS, whereas further more reducers help in a moderate way. Also, the runtime decreases even when the number of reducers is higher than the number of nodes. In particular, the performance of MR-GPMRS is the best when the highest number of reducers, 17 in our setting, are used.

The overall runtime difference on the two distributions is related to the different skyline sizes of them. The fraction of skyline tuples in the data set needs to be high enough for the extra overhead to be offset by the parallelism of multiple reducers in MR-GPMRS. The independent data set used has a much smaller skyline than the anti-correlated one, which renders the use of multiple reducers on the former not so beneficial as on the latter.

7.5 Evaluation of The Cost Estimation

Finally, we evaluate the cost estimation described in Section 6. Specifically, we run MR-GPMRS on a series of data sets with a cardinality of 1×10^6, record the real numbers of partition-wise dominance comparisons in the executions, and compare them with the numbers suggested by the estimates derived in Section 6. The results are reported in Figure 11. The numbers from the real executions are recorded for the mapper and the reducer that have the highest number of comparisons.



(a) Mappers

(b) Reducers

Figure 11: Results on cost estimation

The results about mapper costs are shown in Figure 11(a). For independent data, the estimated costs for mappers closely match their counterparts from the real execution. This is not surprising since the cost estimation described in Section 6 assumes that data are of independent distribution.

In contrast, the cost estimation is not that precise for anti-correlated data since the assumption does not hold. Nevertheless, it is noteworthy that the estimated cost is higher than the real cost in every case, which means that the estimates can be still be used as an upper bound of the worst-case costs even for anti-correlated data sets.

Referring to Figure 11(b), the results show that the cost estimation is more inaccurate for the reducers on both data distributions. The reason for the higher inaccuracy is that it is not mathematically feasible to capture how independent partition groups are generated in the cost estimation. Nevertheless, the estimated costs can still work as the upper bound of the worst-case costs for reducers, as suggested by the results in the figure.

8. CONCLUSION AND FUTURE WORK

In this paper, we propose two novel algorithms, namely MR-GPSRS and MR-GPMRS, for efficient skyline computation in MapReduce. The main feature of the algorithms is that they allow decision making across mappers and reducers, which is a valuable achievement in MapReduce where mappers/reducers are stateless and inter-node communication is not supported. The feature is accomplished by a grid partitioning scheme for the data space, as well as a bitstring describing the partitions empty and non-empty states. The bitstring allows to prune data in the unit of data partitions, and it enables the mappers and reducers to involve only the relevant partitions when comparing tuples for dominance checks. Specifically, MR-GPSRS employs a single reducer to find the final global skyline, whereas MR-GPMRS avoid the bottleneck by utilizing the bitstring to partition the final skyline computation among multiple reducers.

The results of extensive experimental studies show that MR-GPSRS and MR-GPMRS consistently outperform existing MapReduce skyline algorithms in terms of efficiency and scalability. The experimental results also discover the best scenarios for each proposed algorithms. In particular, MR-GPMRS performs significantly better when a large fraction of the tuples are in the skyline, while MR-GPSRS performs marginally better when the skyline fraction is small.

Several directions exist for future work. Multiple reducers in MR-GPMRS do not give the best performance when the skyline fraction is low in the input data set. To obtain optimal performance on arbitrary inputs, a hybrid method can be developed by combining MR-GPSRS and MR-GPMRS. Such a method should be able to switch between the two algorithms automatically, and intelligently decide how many reducers to use.

This work studies how to make good use of the parallelism of MapReduce for skyline computation, whereas the local skyline computation on a single node is not the research focus. It is still interesting to optimize the local skyline computations and explore how such optimizations would affect the overall performance in the context of MapReduce.

When merging independent groups for less reducers in MR-GPMRS, the method used in this paper prefers minimizing computational cost to minimizing communication cost. It is interesting to develop a merging method that balances the two different costs.

9. REFERENCES

[1] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman. Parallel skyline queries. In *ICDT*, pp. 274–284, 2012.

[2] Apache. Welcome to Apache™Hadoop®! http://hadoop.apache.org/.

[3] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *EDBT*, pp. 256–273, 2004.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pp. 421–430, 2001.

[5] L. Chen, B. Cui, and H. Lu. Constrained skyline query processing against distributed data sites. *TKDE*, 23(2):204–217, 2011.

[6] L. Chen, K. Hwang, and J. Wu. MapReduce skyline query processing with a new angular partitioning approach. In *IPDPS Workshops & PhD Forum*, pp. 2262– 2270, 2012.

[7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pp. 717–719, 2003.

[8] B. Cui, L. Chen, L. Xu, H. Lu, G. Song, and Q. Xu. Efficient skyline computation in structured peer-to-peer systems. *TKDE*, 21(7):1059–1072, 2009.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pp. 107–113, 2004.

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

[11] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in manets. In *ICDE*, pp. 66, 2006.

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

[13] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Efficient execution plans for distributed skyline query processing. In *EDBT*, pp. 271–282, 2011.

[14] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pp. 301–310, 2001.

[15] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *ICDE*, page 65, 2006.

[16] G. Valkanas and A. N. Papadopoulos. Efficient and adaptive distributed skyline computation. In *SSDBM*, pp. 24–41, 2010.

[17] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *SIGMOD*, pp. 227–238, 2008.

[18] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis. Skypeer: Efficient subspace skyline computation over distributed data. In *ICDE*, pp. 416–425, 2007.

[19] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. El Abbadi. Parallelizing skyline queries for scalable distribution. In *EDBT*, pp. 112–130, 2006.

[20] B. Zhang, S. Zhou, and J. Guan. Adapting skyline computation to the MapReduce framework: Algorithms and experiments. In *DASFAA Workshops*, 2011.

[21] L. Zhu, Y. Tao, and S. Zhou. Distributed skyline retrieval with low bandwidth consumption. *TKDE*, 21(3):384–400, 2009.