Parallel Computation of Skyline and Reverse Skyline Queries Using MapReduce

Yoonjae Park
Seoul National University
Seoul, Korea
yjpark@kdd.snu.ac.kr

Jun-Ki Min
Korea Univ. of Tech. & Edu.
CheonAn, Korea
jkmin@kut.ac.kr

Kyuseok Shim
Seoul National University
Seoul, Korea
shim@ee.snu.ac.kr

ABSTRACT

The skyline operator and its variants such as dynamic skyline and reverse skyline operators have attracted considerable attention recently due to their broad applications. However, computations of such operators are challenging today since there is an increasing trend of applications to deal with big data. For such data-intensive applications, the MapReduce framework has been widely used recently.

In this paper, we propose efficient parallel algorithms for processing the skyline and its variants using MapReduce. We first build histograms to effectively prune out non-skyline (non-reverse skyline) points in advance. We next partition data based on the regions divided by the histograms and compute candidate (reverse) skyline points for each region independently using MapReduce. Finally, we check whether each candidate point is actually a (reverse) skyline point in every region independently. Our performance study confirms the effectiveness and scalability of the proposed algorithms.

1. INTRODUCTION

The skyline operator [4] and its variants such as dynamic skyline [22] and reverse skyline [11] operators have recently attracted considerable attention due to their broad applications including product or restaurant recommendations [18, 19], review evaluations with user ratings [17], querying wireless sensor networks [29] and graph analysis [33].

The skyline is a set of all points that are not dominated by any other point. A point p_i is said to dominate another point p_j if p_i is not greater than p_j in all dimensions and p_i is smaller than p_j in at least a single dimension. Consider a laptop database D with *price* and *weight* attributes in Figure 1(a). A user who wants to buy a laptop can consider the laptops in the skyline $\{p_1, p_3, p_5, p_7\}$ only, since there always exists a better laptop in the skyline for any laptop which is not in the skyline.

The dynamic skyline is a set of all points that are not dominated by any other point with respect to (*wrt*) the distances to a given query point. When a user wants to find a laptop whose price and weight are close to 50 and 25 respectively, the dynamic skyline of D wrt a query point $\langle 50, 25\rangle$ can be useful candidate laptops to be purchased.

Suppose that each point p_i represents a user who purchased a laptop in Figure 1(a) and a company wants to estimate the sales of a laptop to be manufactured whose price and weight will be 50 and 25 respectively. If $q = \langle 50, 25\rangle$ belongs to the dynamic skyline of D wrt a point p_i, we can assume that the user p_i finds the laptop q interesting. The reverse skyline of D wrt q is defined as a set of every point $p_i \in D$ such that q belongs to the dynamic skyline of D wrt p_i. Thus, the reverse skyline of D wrt q is a set of all customers who will be interested in q.

Computing the skyline or its variants is challenging today since there is an increasing trend of applications expected to deal with big data. For example, Wal-Mart has a 4PB (that's 4×10^{15} bytes) data warehouse of purchase records with dozens of attributes [5] where skyline and its variant operators are frequently used as primitive operators to determine pricing and marketing strategies. For such data-intensive applications, the MapReduce[10] framework has recently attracted a lot of attention. MapReduce is a programming model that allows easy development of scalable parallel applications to process big data on large clusters of commodity machines. Google's MapReduce or its open-source equivalent Hadoop [2] is a powerful tool for building such applications. Recently, a variant of Hadoop was also developed to support online query processing in [9], which enables MapReduce to be utilized for such applications including event monitoring and stream data processing.

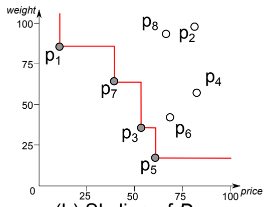
Most of existing serial algorithms[4, 7, 15, 22, 25] for (reverse) skyline computations rely on some centralized indexing structures such as B$^+$-trees [8] or R*-trees [3]. However, in the MapReduce framework, there is no functionality provided for building and accessing such spatial indexes because it is difficult to provide efficient and scalable distributed indexes in several thousands of machines. Thus, it is hard to extend such existing algorithms into the MapReduce framework. A preliminary work to adapt skyline processing to the MapReduce framework was recently presented in [30]. The proposed algorithms are simple extensions of previous serial skyline algorithms in [4, 7, 25]. Furthermore, dynamic skyline and reverse skyline operators were not addressed in [30].

In this paper, we propose efficient parallel algorithms, called *SKY-MR* and *RSKY-MR*, which compute the skylines and reverse skylines using MapReduce respectively. In the first phase, we build new histograms, called the *sky-quadtree* and *rsky-quadtree*, which are extensions of quadtrees [12]

Proceedings of the VLDB Endowment, Vol. 6, No. 14
Copyright 2013 VLDB Endowment 2150-8097/13/14... \$ 10.00.

ID	Price	Weight
p_1	15	85
p_2	85	95
p_3	55	35
p_4	80	55
p_5	60	15
p_6	70	40
p_7	40	60
p_8	65	90

(a) Data set D



(b) Skyline of D

Figure 1: An example of a skyline

to effectively prune out non-skyline and non-reverse skyline points respectively in advance. The new histograms are also used for load balancing of computations in the MapReduce framework. In the second phase, we partition data based on the regions divided by our proposed histograms and compute candidate (reverse) skyline points for each region independently using MapReduce. Finally, we check whether each candidate point is actually a (reverse) skyline point in every region independently by another MapReduce phase. Extensive performance study shows that our algorithms are very efficient and significantly better than the state-of-the-art algorithms [1, 14, 30]. Although our proposed algorithms are devised for the MapReduce framework, they can be also applied to other frameworks such as MPI [16] and multi-cores. Experimental results confirm the effectiveness and scalability of our proposed algorithms in such other frameworks as well.

2. PRELIMINARIES

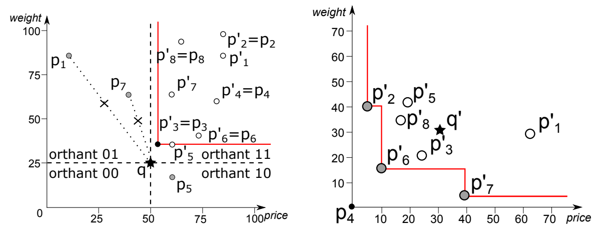
2.1 Skyline and Its Variants

Consider a d-dimensional data set $D = \{p_1, p_2, \ldots, p_{|D|}\}$. A point p_i is represented by $\langle p_i(1), p_i(2), \cdots, p_i(d)\rangle$ where $p_i(k)$ is the k-th coordinate of p_i. A point p_i *dominates* another point p_j, denoted as $p_i \prec p_j$, if the two conditions hold: (1) for every k with $1 \le k \le d$, we have $p_i(k) \le p_j(k)$ and (2) there exists k with $1 \le k \le d$ such that $p_i(k) < p_j(k)$ holds. The *skyline* of D, represented by $SL(D)$, is a subset of D where every point in $SL(D)$ is not dominated by every other point in D. In other words, $SL(D) = \{p_i \in D \mid \nexists\, p_j (\neq p_i) \in D \text{ s.t. } p_j \prec p_i\}$.

Given a query point q, we say that a point p_i *dynamically dominates* another point p_j with respect to (*wrt*) q, denoted as $p_i \prec_q p_j$, if and only if (1) $|p_i(k) - q(k)| \le |p_j(k) - q(k)|$ for all k with $1 \le k \le d$ and (2) there exists k with $1 \le k \le d$ s.t. $|p_i(k) - q(k)| < |p_j(k) - q(k)|$. The *dynamic skyline* [22] is represented by $DSL(q, D)$ such that $DSL(q, D) = \{p_i \in D \mid \nexists\, p_j (\neq p_i) \in D \text{ s.t. } p_j \prec_q p_i\}$. To compute $DSL(q, D)$, each point p_i in D is converted to a point p'_i with $p'_i(k) = |p_i(k) - q(k)|$ for all $k = 1, 2, \cdots, d$. Then, $DSL(q, D)$ is obtained by computing the skyline among the converted points.

Based on the definition of the dynamic skyline, the notion of the *reverse skyline* is proposed in [11]. Given a d-dimensional data set D and a query point q, the *reverse skyline*, represented by $RSL(q, D)$, is the set of every point p_i in D satisfying $q \in DSL(p_i, D \cup \{q\} - \{p_i\})$ (i.e., the query point q is contained in the dynamic skyline wrt p_i).

Example 2.1: *Consider the data D in Figure 1(a). In Figure 1(b), we plot every point in D into a 2-dimensional space where each point in $SL(D)$ is represented by a shaded circle. Since p_1 is not dominated by any other point in D, $p_1 \in SL(D)$. Figure 2(a) shows the converted points to the new space whose origin is a query point $q = \langle 50, 25\rangle$. For instance, $p_1 = \langle 15, 85\rangle$ is converted to $p'_1 = \langle |15 - 50|,$*



(a) Dynamic skyline wrt q (b) Dynamic skyline wrt p_4

Figure 2: An example of dynamic skylines

$|85 - 25|\rangle = \langle 35, 60\rangle$. We represent $DSL(q, D) = \{p_3\}$ with a bold circle in Figure 2(a). Figure 2(b) illustrates the dynamic skyline wrt p_4. Since q does not belong to the dynamic skyline wrt p_4, p_4 is not a reverse skyline point (i.e., $p_4 \notin RSL(q, D)$). ∎

2.2 The MapReduce Framework

In the MapReduce framework, data is represented as (key, value) pairs and a distribute file system (DFS) initially partitions data in multiple machines. Computation is carried out by using two user defined functions: map and reduce functions. A *map* function takes a key-value pair as input and may output several key-value pairs. After the key-value pairs emitted by all map functions are grouped by keys in the *shuffling phase*, a *reduce* function is invoked with each distinct key and the list of all values sharing the key, and the reduce function may output key-value pairs. Presentation of a MapReduce algorithm consists of three functions which are map, reduce and main functions.

3. RELATED WORK

After skyline processing was introduced in [4], several techniques were introduced in [7, 15, 22, 25]. In [22], the progressive I/O optimal algorithm BBS was proposed and the dynamic skyline was also introduced. Later on, the reverse skyline was introduced in [11], and the algorithms, called BBRS and RSSA, are proposed for reverse skyline processing. Many existing algorithms utilize R*-trees to check whether a point belongs to the (reverse) skyline or not. Since there is no functionality provided for building and accessing distributed R*-trees in the MapReduce framework, such algorithms are not suitable to be parallelized using MapReduce. In addition, the variants of the skyline queries such as top-k frequent skyline [6], spatial skyline [24], probabilistic (reverse) skyline for uncertain data [20, 23], continuous skyline for stream data [26, 31], and stochastic skyline [21] have been introduced.

Recently, several techniques for processing skyline and reverse skyline queries in distributed environments such as MANET [13], sensor networks [29] and other distributed systems [1, 14, 28, 32] have been proposed. Even though such techniques are not proposed for the MapReduce framework, the work in [1, 14] can be processed with MapReduce.

In [1], Afrati et al. investigated skyline processing using other parallel models. They split the space into $\lceil t^{1/(d-1)}\rceil^d$ (using GMP-model) or t^d (using MP-model) grid partitions holding similar number of points, where t is the number of machines to be used and d is the number of dimensions, and prune the partitions without any skyline point by using dominance relationships of relaxed skylines. Then, the global skyline is calculated in parallel in every unpruned partition. The proposed 1-step and 2-step algorithms in [1] utilize GMP-model and MP-model respectively. Even though the 2-step algorithm may be able to prune more points by utilizing detailed smaller partitions, pruning partitions takes significant amount of time than the 1-step algorithm due to its quadratic nature of time complexity with a large number of partitions. Note that the number of partitions split by each algorithm is fixed according to the number of machines used. In contrast, by utilizing a *sky-quadtree* with a split threshold ρ, our *SKY-MR* varies the number of partitions adaptively for load balancing and prunes a set of partitions quickly at once by taking advantage of our node id representations to identify the dominance relationship between a leaf node and an internal node representing all partitions represented by its descendant nodes in the *sky-quadtree*. *SKY-MR* also performs effective skyline computation in each machine to process points in unpruned remaining partitions.

Köhler et al. proposed the algorithm PPPS for multi-core machines in [14]. PPPS first samples a small set B and generates an initial partition with every point in D which is not dominated by any point in B. Then, it repeatedly selects a partition and splits it into two partitions until the number of the partitions becomes the desired number of cores c. The local skyline is next computed for every partition in parallel using a multi-core machine. Finally, merging the local skylines of all partitions are performed in $O(\log(c))$ iterations. In each iteration, pairs of partitions are merged in parallel using multi-cores to compute the local skylines of merged partitions until there remains a single partition only. Since the number of partitions decreases in half by each iteration, the i-th iteration can utilize $c/2^i$ cores only. However, our *SKY-MR* computes the global skyline by considering every partition independently and simultaneously once. To do so, some local skyline points in each partition is sent to other partition where those points are required to check whether each local skyline point is actually a global skyline point or not, by checking dominance relationship with node id representations.

The most relevant work to ours is the algorithm *MR-BNL* in [30]. The d-dimensional data space is first partitioned into 2^d subspaces according to the median of each dimension and the local skyline of every subspace is computed in parallel. The global skyline is next calculated by *MR-BNL* in a single machine from all the local skylines. Note that identifying the median of every dimension is very expensive and only up to 2^d machines can be utilized for parallelization. In addition, since a single machine computes the global skyline, it is not scalable with a large number of local skyline points. Furthermore, processing of dynamic skyline and reverse skyline was not addressed in [30].

4. SKYLINE PROCESSING USING MAPREDUCE

The parallel algorithm *SKY-MR* to discover the skyline $SL(D)$ in a given data set D consists of the following three phases. The pseudocode of *SKY-MR* is shown in Figure 3.

(1) **Sky-quadtree building phase:** To filter out non-skyline points effectively earlier, we propose a new histogram, called the *sky-quadtree*. To speed up, we build a *sky-quadtree* with a sample of D where each leaf node with non-skyline sample points only is marked as "pruned".

(2) **Local skyline phase:** We partition the data D based on the regions divided by the *sky-quadtree* and compute the local skyline for the region of every unpruned leaf node independently using MapReduce by calling *L-SKY-MR*.

```
Function SKY-MR( $D$, $\rho$, $d$, $\delta$ )
$D$: a data set, $\rho$: the split threshold,
$d$: the dimension, $\delta$ : local skyline threshold
begin
1.  $sample$ = ReservoirSampling( $D$ );
2.  $sky\text{-}quadtree$ = SKY-QTREE( $sample$, $\rho$, $d$ );
3.  Broadcast $sky\text{-}quadtree$;
4.  $Local\text{-}SL$ = RunMapReduce(L-SKY-MR);
5.  if $Local\text{-}SL$.size $\geq \delta$ then
6.      Broadcast non-empty leaf node ids;
7.      $SL$ = RunMapReduce(G-SKY-MR);
8.  else $SL$ = G-SKY( $Local\text{-}SL$ );
9.  return $SL$;
end
```

Figure 3: The SKY-MR

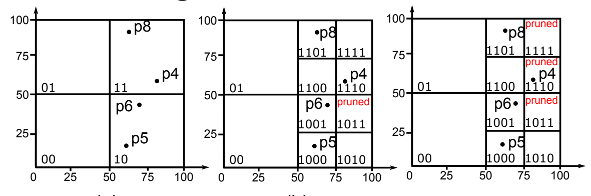


Figure 4: An example of sky-quadtree building

(3) **Global skyline phase:** We calculate the global skyline using MapReduce from the local skyline points in every unpruned leaf node by calling *G-SKY-MR*. When the number of local skyline points is small, we run the serial algorithm *G-SKY* in a single machine to speed up.

4.1 SKY-QTREE: The Sky-Quadtree Building Algorithm

A *sky-quadtree* is an extension of quadtrees [12] which subdivide the d-dimensional space recursively into sub-regions. In a *sky-quadtree*, internal nodes have exactly 2^d children and each leaf node has at most a predefined number of points ρ called the *split threshold*. We denote the region of a node n as $region(n)$. An id is assigned to each node based on its location in *sky-quadtrees*. In a d-dimensional space, the id of a node n with depth k is represented by $id(n) = a_1a_2 \cdots a_{k\cdot d}$ which consists of the first $(k-1)\cdot d$ bits coming from its parent node and the remaining d bits $a_{(k-1)\cdot d+1}a_{(k-1)\cdot d+2} \cdots a_{k\cdot d}$ where $a_{(k-1)\cdot d+i} = 0$ (or $a_{(k-1)\cdot d+i} = 1$) if the i-th dimensional range of the $region(n)$ is the first half (or the second half) of its parent's i-th dimensional range. Similarly, we let $node(id)$ represent the node with an id id. We can decompose $id(n)$ into d number of bit strings $sub(id(n), i)$s (for $1 \leq i \leq d$) s.t. $sub(id(n), i) = a_ia_{(i+d)}a_{(i+2\cdot d)} \cdots a_{(i+(k-1)\cdot d)}$.

Given a pair of bit strings $a = a_1a_2 \cdots a_p$ and $b = b_1b_2 \cdots b_q$, we say that $a = b$ if $a_i = b_i$ for all $i = 1, 2, \cdots, min(p, q)$, and $a < b$ if there exists an integer j, with $1 \leq j \leq min(p, q)$, s.t. $a_i = b_i$ for all $i = 1, 2, \cdots, j-1$ and $a_j < b_j$. Similarly, we write $a > b$ if there exists an integer j, with $1 \leq j \leq min(p, q)$, s.t. $a_i = b_i$ for $i = 1, 2, \cdots, j-1$ and $a_j > b_j$.

Definition 4.1.: *Given a pair of leaf nodes n_i and n_j in a sky-quadtree, if every point in $region(n_i)$ dominates all points in $region(n_j)$,* n_i dominates n_j *and we represent it by $n_i \prec n_j$. If every point in $region(n_i)$ does not dominate all points in $region(n_j)$,* n_i does not dominate n_j *and we denote it by $n_i \not\prec n_j$.*

Dominance relationships by node ids: Based on the following proposition, we can efficiently identify the nodes dominated by another non-empty leaf node in a *sky-quadtree* by utilizing node ids.

Proposition 4.2.: *Given a pair of nodes n_i and n_j in a sky-quadtree, n_i dominates n_j if $sub(id(n_i), k) < sub(id(n_j)$,* *k*) *for all* k=1, 2, ⋯ , d. *Similarly,* n_i *does not dominate* n_j (*i.e.,* $n_i \not\prec n_j$) *if there exists* k *such that* $sub(id(n_i), k) > sub(id(n_j), k)$ *with* k=1, 2, ⋯ , d.

Proof: Let $region(n)$=$\langle[n(1)^-, n(1)^+),\cdots,[n(d)^-, n(d)^+)\rangle$ where $[n(k)^-, n(k)^+)$ is the range of the k-th dimension of n's covering region. If $sub(id(n_i), k) < sub(id(n_j), k)$, both $[n_i(k)^-, n_i(k)^+)$ and $[n_j(k)^-, n_j(k)^+)$ are disjoint and $n_i(k)^+ \le n_j(k)^-$. Thus, for a pair of points p_i in $region(n_i)$ and p_j in $region(n_j)$, we have $p_i(k) < n_i(k)^+ \le n_j(k)^- \le p_j(k)$. If $sub(id(n_i), k) < sub(id(n_j), k)$ holds for all k=1, 2, ⋯ , d, we have $p_i \prec p_j$ and $n_i \prec n_j$. Similarly, if there exists k such that $sub(id(n_i), k) > sub(id(n_j), k)$, we have $p_i(k) > p_j(k)$ and $n_i \not\prec n_j$ ▮

Building a sky-quadtree: In order to quickly build a *sky-quadtree*, we utilize a random sample obtained from D by reservoir sampling [27]. Since we use a sample only, we may prune fewer non-skyline points than using D. However, the use of sampling does not affect the correctness of our skyline computation algorithm *SKY-MR* because all skyline points exist in unpruned leaf nodes.

The procedure *SKY-QTREE* (in line 2 of Figure 3) builds a *sky-quadtree* by inserting a sample into the root node and recursively splits each node n to 2^d child nodes whenever the number of points in n exceeds the split threshold ρ. When splitting a node n, we insert each point p in $region(n)$ into its child node n_i into which p is inserted. If the last d-bit string of n_i's id is $00\cdots0$ (i.e., the first half in every dimension), we mark n_j whose last d-bit string of its id is $11\cdots1$ (i.e., the second half in every dimension) as "pruned" and skip all remaining points belonging to n_j. After all points are inserted into child nodes, we recursively split each unpruned child node. When we cannot split any more, starting from the root node, we traverse the *sky-quadtree* to mark every node dominated by a non-empty leaf node as "pruned".

Example 4.3.: *Consider the data* D *in Figure 1(a) and the split threshold* ρ=1. *Suppose that a sample* $\{p_4, p_6, p_7, p_8\}$ *is inserted into the root node. In Figure 4(a), the root node is subdivided since it has more than* ρ *points. The id of the root node's child node in the top-left corner is 01 since the region covers the first and second halves of the root node's first and second dimensions respectively. The node id* 1011 *can be decomposed into* $sub(10\mathbf{11},1)$ = 11 *and* $sub(1\mathbf{0}1\mathbf{1},2)$ = 01. *Because* $sub(10\mathbf{00},1) < sub(10\mathbf{11},1)$ *and* $sub(1\mathbf{0}0\mathbf{0},2) < sub(1\mathbf{0}1\mathbf{1},2)$, *we have node(1000)* $\prec$ *node(1011). Thus, we mark node*(1011) *as "pruned" in Figure 4(b). In addition, since node*(1000) $\prec$ *node*(1110) *and node*(1000) $\prec$ *node*(1111) *hold, both nodes are marked as "pruned". The final sky-quadtree obtained is presented in Figure 4(c).* ▮

4.2 L-SKY-MR: The Local Skyline Computation Algorithm

We next present the parallel algorithm *L-SKY-MR* that calculates the local skyline independently for every unpruned leaf node in the *sky-quadtree*. The *sky-quadtree* Q is first broadcast to all map functions. Each map function is next called with a point p in D. If the point p is in the region of an unpruned leaf node n_p of Q, we output the key-value pair $\langle n_p, p\rangle$. Otherwise, we do nothing.

In the shuffling phase, the key-value pairs emitted by all map functions are grouped by each distinct leaf node, and a reduce function is called with each node n and its point list L. Each reduce function computes the local skyline in L (i.e., $SL(L)$) and outputs $\langle n,p\rangle$ for every local skyline point p. It also produces an artificial d-dimensional point referred to as the *virtual max point* of the node n which is denoted by vp_n where $vp_n(k) = max_{p\in SL(L)}p(k)$ with $1 \le k \le d$. Every virtual max point of each unpruned leaf node is output to the file VIRTUAL in the Hadoop distributed file system(HDFS). The virtual max point will be used to reduce the number of checking dominance relationships by the following proposition.

Proposition 4.4: *If a point* p *does not dominate the virtual max point of a leaf node* n (*i.e.,* vp_n) *in a sky-quadtree,* p *does not dominate every local skyline point in* $region(n)$.

Proof: We will prove the contrapositive: if p dominates a local skyline point in $region(n)$, we have $p \prec vp_n$. Since the point p dominates a local skyline point p_l in $region(n)$, we have $p(k) \le p_l(k)$ for every k with $1 \le k \le d$ and there exists k such that $p(k) < p_l(k)$. By the definition of the virtual max point, $p_l(k) \le vp_n(k)$ holds for every k. Thus, we also have $p(k) \le vp_n(k)$ for every k and there exists k such that $p(k) < vp_n(k)$. In other words, $p \prec vp_n$. ▮

In addition, each reduce function selects a single local skyline point, called a *sky-filter point*, for each dimension which has the minimum value on the dimension. The local skyline points dominated by such selected sky-filter points will be filtered out in the next global skyline phase. All sky-filter points are stored to the file called SKY-FILTER in HDFS.

Example 4.5: *Consider the sky-quadtree in Example 4.3. Figures 5(a)–(d) show the data flow in the local skyline phase of* SKY-MR. *After the sky-quadtree is broadcast to all map functions, each map function is invoked with a point* p *in* D *as illustrated in Figure 5(a). For instance,* $\langle 10, p_1\rangle$ *is emitted since* p_1 *is contained in the unpruned leaf node, node*(10). *In Figure 5(b), the key-value pairs emitted from all map functions are shown. The key-value pairs grouped by each distinct key are provided in Figure 5(c). Each reduce function finally outputs the local skyline of a node and the virtual max point as well as sky-filter points. Consider node*(10) *whose skyline points are* $\{\langle 15, 85\rangle, \langle 40, 60\rangle\}$. *The reduce function with node*(10) *outputs* $\langle 15, 85\rangle$ *and* $\langle 40, 60\rangle$ *as sky-filter points. It also outputs* $\langle 40, 85\rangle$ *as a virtual max point. The points output by all reduce functions are illustrated in Figure 5(d).* ▮

Discussion: We can utilize R*-trees instead of our *sky-quadtrees*. However, since R*-trees are optimized to reduce the amount of "dead space" (empty area) covered by their nodes, a large portion of uncovered space tends to be generated in R*-trees. Furthermore, generating an R*-tree from a sample increases uncovered space even more. Since every point belonging to the uncovered space in an R*-tree cannot be pruned, using an R*-tree instead of a *sky-quadtree* produces a lot of unpruned points resulting in a significant increase of execution times in the next phase. In addition, it is difficult to compute local skyline and global skyline in each node of an R*-tree independently because the regions represented by nodes in an R*-tree are overlapped with each other.

4.3 G-SKY-MR: The Global Skyline Computation Algorithm

The procedure *G-SKY-MR* computes the global skyline in every non-empty unpruned leaf node independently using MapReduce. In the map function called with each local

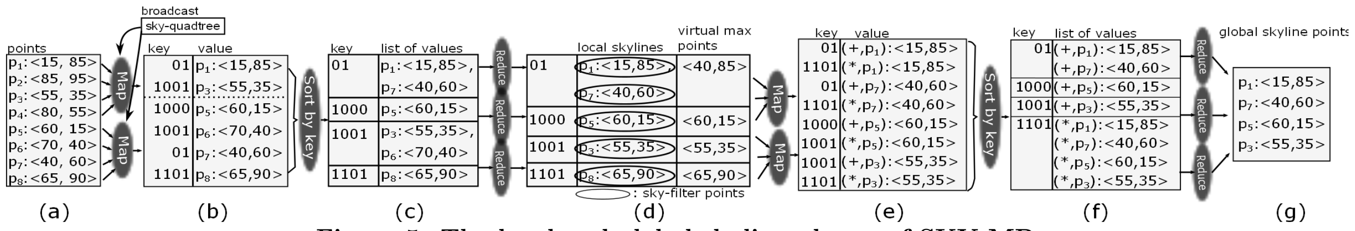


Figure 5: The local and global skyline phases of SKY-MR

```
Function G-SKY-MR.map( $n_i$, $p$ )
$n_i$: a node id, $p$: a point belongs to the node with id = $key$
begin
1.  $nodes$ = LoadNonEmptyNodes();
2.  if DominatedByFilterPoints( $p$ ) then return;
3.  output( $n_i$, $(+, p)$ );
4.  for each node id $n_j$ ($\neq n_i$) in $nodes$ do
5.      if IsNeeded($n_i$, $n_j$) then
6.          if $p \prec vp_{n_j}$ then output( $n_j$, $(*, p)$ );
end
```

Figure 6: The G-SKY-MR.map

point, the point is emitted to every other unpruned leaf node in which it may dominate at least a point in the node. Since it is straightforward to implement the serial algorithm *G-SKY*, we omit the details of *G-SKY* here.

The pseudocode of *G-SKY-MR.map* is shown in Figure 6. In *G-SKY-MR*, a map function with each local skyline point p discards the point p if p is dominated by a sky-filter point chosen in the previous phase. Otherwise, the pair $\langle n_i, (+, p)\rangle$ is emitted where n_i is the leaf node containing p and the symbol '+' represents that p is in $region(n_i)$ (in lines 1-3 of *G-SKY-MR.map*).

If a local skyline point p_j of node n_j is dominated by at least a local skyline point of the other nodes, p_j cannot be a global skyline point. However, if every point p is sent to all other nodes except n_i, the communication overhead is very expensive. By Definition 4.1, when $n_i \not\prec n_j$, every point in n_i cannot dominate the points in n_j. The procedure *IsNeeded*(n_i, n_j) easily identifies such case (i.e., $n_i \not\prec n_j$) using ids of two nodes based on Proposition 4.2. If $n_i \not\prec n_j$, *IsNeeded*(n_i, n_j) returns false. Otherwise, *IsNeeded*(n_i, n_j) returns true and we output the pair $\langle n_j, (*, p)\rangle$ where '*' indicates that the point p is not in $region(n_j)$ but p may dominate at least a point in $region(n_j)$. However, if p does not dominate vp_{n_j}, we do not emit the pair $\langle n_j, (*, p)\rangle$ due to Proposition 4.4 (in lines 4-6).

Each reduce function called with a node n_i next computes the global skyline points by checking whether each of n_i's local skyline points annotated with '+' is dominated by a local skyline point associated with '*' which comes from the other nodes.

EXAMPLE 4.6**:** *The behavior of* G-SKY-MR *is illustrated in Figures 5(d)–(g). Every map function is called with each local skyline point. For example, the map function with* p_1 *emits* $\langle 01,(+,p_1)\rangle$ *since the point* p_1 *is in* $region(node(01))$*. In addition,* $\langle 1101,(*,p_1)\rangle$ *is emitted since* p_1 *dominates the virtual max point of* $node(1101)$*. However, in the map function invoked with* p_5*,* $\langle 1101,(*,p_5)\rangle$ *is not emitted because* p_5 *does not dominate the virtual max point* $\langle 65,90\rangle$ *in* $node(1101)$*. Figure 5(e) shows the key-value pairs emitted by all map functions. The key-value pairs after the shuffling phase are shown in Figure 5(f). Each reduce function computes the global skyline of its associated node. After all reduce functions are finished, we obtain the skyline in Figure 5(g).* ∎

Extending to dynamic skylines: We first convert each point p_i in D to a point p'_i using a query point q where $p'_i(k)=|p_i(k) - q(k)|$ for $k=1,\cdots,d$, as presented in Section 2.1. Then, we calculate the dynamic skyline wrt q by computing skyline points among the converted points. Extending *SKY-MR* to handle the dynamic skylines is straightforward since at the first and second phases, each point in D can be easily transformed into a new space whose origin is the query point q. Due to lack of space, we do not present the details of dynamic skyline processing using MapReduce.

Extending to multi-cores: We can develop the multi-thread procedure *SKY-MC* to compute skylines with the key idea of *SKY-MR* by using multiple threads instead of MapReduce. After data are divided into partitions based on a *sky-quadtree*, threads calculate the local skylines in parallel independently for every unpruned leaf node. To compute the global skyline in parallel by utilizing sky-filter and virtual max points, we invoke multiple threads again.

Extending to MPI: Message Passing Interface (MPI) [16] is a language-independent communication protocol used to develop parallel programs on a cluster of machines. We can also develop the procedure *SKY-MP* which computes skylines using MPI with the key idea of *SKY-MR*.

5. REVERSE SKYLINE PROCESSING USING MAPREDUCE

To filter out non-reverse skyline points efficiently, we divide the data D into 2^d orthants wrt a query point q as illustrated in Figure 2(a). The set of all data points located in an orthant o is denoted as D_o. For each orthant o represented by the region $\langle [o(1)^-, o(1)^+], \cdots, [o(d)^-, o(d)^+]\rangle$, the id, denoted by $a_1 a_2 \cdots a_d$, is assigned where $a_i = 0$ if $[o(i)^-, o(i)^+] = [-\infty, q(i)]$ and $a_i = 1$ if $[o(i)^-, o(i)^+] = [q(i), \infty]$.

LEMMA 5.1**:** *For* p_i, $p_j \in D$*, if* p_j *is located at an orthant* o *and* p_j *dynamically dominates a query point* q *wrt* p_i *(i.e.* $p_j \prec_{p_i} q$ *), then* p_i *is also in the same orthant* o*.*

Proof: When $p_j \prec_{p_i} q$, we have $|q(k) - p_i(k)| \geq |p_j(k) - p_i(k)|$ for all $k = 1, \cdots, d$. Squaring both sides and rearranging terms, the above condition becomes equivalent to $0 \geq (p_j(k) - p_i(k))^2 - (q(k) - p_i(k))^2$. Then, we get $0 \geq (p_j(k) + q(k) - 2p_i(k)) \cdot (p_j(k) - q(k)) = -2 \cdot (p_i(k) - q(k))(p_j(k) - q(k)) + (p_j(k) - q(k))^2$. Since $2 \cdot (p_i(k) - q(k)) \cdot (p_j(k) - q(k)) \geq (p_j(k) - q(k))^2 \geq 0$ for all $k=1,\cdots,d$, $(p_i(k) - q(k))$ and $(p_j(k) - q(k))$ have the same sign. Thus, p_i and p_j are in the same orthant. ∎

Note that $p_i \notin RSL(q, D)$ if there exists a point $p_j \in D$ such that $p_j \prec_{p_i} q$. Since every point dynamically dominating q wrt p_i is always located in the same orthants in which p_i is located by Lemma 5.1, our brute-force algorithm *BR-RSKY-MR* calculates the reverse skyline of each orthant independently and merges all reverse skylines.

```
**Function** RSKY-MR($D$, $q$, $\rho$, $d$, $\delta$)
$D$: a dataset, $q$: a query point, $\rho$: the split threshold,
$d$: the dimension, $\delta$: strong reverse skyline threshold
**begin**
1. $sample$ = ReservoirSampling( $D$ );
2. $rsky\text{-}quadtrees$=RSKY-QTREE($sample$, $\rho$, $d$);
3. Broadcast $q$ and $rsky\text{-}quadtrees$;
4. $Local\text{-}RSL$ = RunMapReduce(L-RSKY-MR);
5. **if** $Local\text{-}RSL$.size $\geq t$
6. **then** Broadcast $q$, $non\text{-}empty\ leaf\ node\ ids$;
7.     $RSL$ = RunMapReduce(G-RSKY-MR);
8. **else** $RSL$ = G-RSKY($Local\text{-}RSL$);
9. return $RSL$;
**end**
```

Figure 7: The RSKY-MR

To compute $RSL(q, D)$ efficiently, we next devise the algorithm *RSKY-MR* with the following three phases. *RSKY-MR* utilizes *rsky-quadtrees* which are a variant of *sky-quadtrees*. The pseudocode of *RSKY-MR* is presented in Figure 7.

(1) **Rsky-quadtree building phase:** By running *RSKY-QTREE*, we build an *rsky-quadtree* associated with each orthant from a sample obtained by reservoir sampling [27].

(2) **Local reverse skyline phase:** For each unpruned leaf node of every *rsky-quadtree*, we compute candidate reverse skyline points in parallel by invoking *L-RSKY-MR*. In addition, the local dynamic skyline of the midpoints between every point p ($\in D$) and q is selected to prune non-reverse skyline points in the next phase.

(3) **Global reverse skyline phase:** We check in parallel whether each candidate reverse skyline point is actually a global reverse skyline point. Similar to *SKY-MR*, depending on the number of candidate reverse skyline points produced in the previous phase, the global reverse skyline is computed on a single machine by calling *G-RSKY* or on multiple machines by invoking *G-RSKY-MR*.

5.1 RSKY-QTREE: The Rsky-Quadtree Building Algorithm

For effective pruning with *rsky-quadtrees*, we adopt the idea of midpoints introduced in [11, 29]. The midpoint between a point p and a query point q is defined as $mid(p, q) = \langle (p(1)+q(1))/2, \cdots, (p(d)+q(d))/2 \rangle$. Since $|(p(i)+q(i))/2 - q(i)| \leq |p(i) - q(i)|$ holds for each dimension i, the following is trivially true.

Proposition 5.2.: *The midpoint $mid(p, q)$ always dynamically dominates p wrt q.*

We develop the following lemmas to identify efficiently whether a point in D is a global reverse skyline point.

Lemma 5.3.: *Given an orthant o and a query point q, $p_i \in D_o$ is not in the reverse skyline of D_o wrt q, if and only if there exists another point $p_j \in D_o$ s.t. $mid(p_j, q) \prec_q p_i$.*

Proof: ($\Rightarrow$:) When $p_i \notin RSL(q, D_o)$, $q \notin DSL(p_i, D_o)$ holds and there exists a point $p_j (\in D_o)$ s.t. $p_j \prec_{p_i} q$. Since $|p_j(k) - p_i(k)| \leq |q(k) - p_i(k)|$ for all k=$1, \cdots, d$, we can derive $(p_j(k) - q(k))^2 \leq 2 \cdot (p_i(k) - q(k))(p_j(k) - q(k))$, as shown in the proof of Lemma 5.1. Since p_i and p_j are in the same orthant by Lemma 5.1, $|p_j(k) - q(k)|/2 = |(p_j(k) + q(k))/2 - q(k)| \leq |p_i(k) - q(k)|$. Similarly, we can derive $|(p_j(k)+q(k))/2 - q(k)| < |p_i(k) - q(k)|$ for at least a single dimension k. Thus, by the definition of the midpoints, there exists $p_j \in D_o$ such that $mid(p_j, q) \prec_q p_i$.

($\Leftarrow$:) We have $|(p_j(k)+q(k))/2 - q(k)| = |p_j(k) - q(k)|/2 \leq |p_i(k) - q(k)|$ for all k=$1, \cdots, d$, when $mid(p_j, q) \prec_q p_i$. By multiplying $2(p_j(k) - q(k))$ to both sides, we get

$$(p_j(k) - q(k))^2 \leq 2(p_j(k) - q(j))(p_i(k) - q(k)). \quad (1)$$

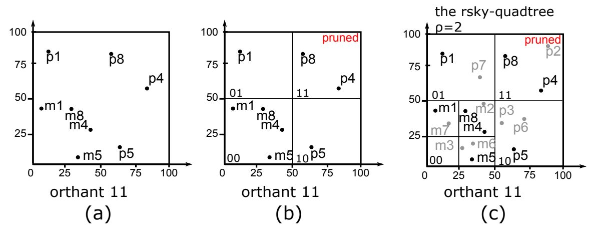


Figure 8: An example of rsky-quadtree building

Since $|p_j(k) - p_i(k)| = |p_j(k) - q(k) - p_i(k) + q(k)|$, we have $(p_j(k) - p_i(k))^2 = (p_j(k) - q(k))^2 + (p_i(k) - q(k))^2 - 2(p_j(k) - q(k))(p_i(k) - q(k))$. By replacing $(p_j(k) - q(k))^2$ with $2(p_j(k) - q(j))(p_i(k) - q(k))$ and using the above inequality (1), we obtain $(p_j(k) - p_i(k))^2 \leq (p_i(k) - q(k))^2$ and thus $|p_j(k) - p_i(k))| \leq |p_i(k) - q(k)|$ holds for every dimension k. Similarly, we can show that $|p_j(k) - p_i(k))| < |p_i(k) - q(k)|$ for at least a dimension k. Consequently, $p_j \prec_{p_i} q$ and q cannot be a dynamic skyline point wrt p_i. In other words, if $mid(p_j, q) \prec_q p_i$, $p_i \notin RSL(q, D_o)$. ∎

Lemma 5.4.: *For two points p_i, $p_j \in D_o$, if $p_j \prec_q p_i$, we have $p_i \notin RSL(q, D_o)$.*

Proof: Since $mid(p_j, q) \prec_q p_j$ by Proposition 5.2, $mid(p_j, q) \prec_q p_i$ holds. Thus, $p_i \notin RSL(q, D_o)$ due to Lemma 5.3. ∎

We develop the procedure *RSKY-QTREE* to build *rsky-quadtrees*. The main differences from *SKY-QTREE* presented in Section 4.1 are as follows:

(1) Given a query point q and a data set D, an *rsky-quadtree* associated with each orthant o is built by inserting sample points $p \in D_o (\subset D)$ and their midpoints. (2) In an *rsky-quadtree*, every node n is marked as "*pruned*" if there exists a point $p \in D_o$ dynamically dominating the node n since all points belonging to the node n cannot be in the reverse skyline. (3) In an *rsky-quadtree*, every node n is also marked as "*pruned*" if there exist at least two points p_i, $p_j \in D_o$ whose $mid(p_i, q)$ and $mid(p_j, q)$ dynamically dominate the node n. Since $mid(p, q)$ always dynamically dominates p according to Proposition 5.2, we need at least two midpoints to prune a node of an *rsky-quadtree*.

Example 5.5.: *Consider the data D in Figure 1(a) with a query point q=$\langle 0, 0 \rangle$ and the split threshold ρ=2. Dividing D into 4 orthants wrt q results in a non-empty orthant o with id=11 only. Assume $\{p_1, p_4, p_5, p_8\}$ is a sample of D. All sample points and their midpoints are inserted into the root node as shown in Figure 8(a) where m_i represents $mid(p_i, q)$. We recursively subdivide the data space starting from the root node until the number of points and midpoints in each unpruned leaf node of the rsky-quadtree is at most ρ. Since there are multiple midpoints dynamically dominating the node with id=11 (i.e., m_1, m_4, m_5 and m_8), it is marked as "pruned" as illustrated in Figure 8(b). The rsky-quadtree constructed from the sample is shown in Figure 8(c).* ∎

5.2 Computations of Reverse Skylines using Rsky-Quadtrees

To illustrate how to compute the reverse skylines using *rsky-quadtrees*, we utilize the following definitions.

Definition 5.6.: *For a leaf node n, let $L_p(n) = \{p \in D|\ p$ is located in region$(n)\}$, $L_m(n) = \{mid(p, q)|p \in D$ s.t. $mid(p, q)$ is located in region$(n)\}$ and $L(n) = L_p(n) \cup L_m(n)$. The* strong reverse skyline *$SRSL(q, L(n))$ of $L(n)$ wrt q is $\{p_j \in L_p(n) \mid p_j \in RSL(q, L_p(n))$ and $\nexists m (\neq mid(p_i, q)) \in L_m(n)$ s.t. $m \prec_q p_j\}$.*

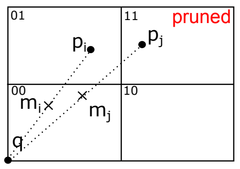


Figure 9: Points and their midpoints in an orthant

A reverse skyline point p is a strong reverse skyline point of the node containing p since p is not dominated by the midpoints of all other points in D according to Lemma 5.3. Thus, if we can eliminate all non-reverse skyline points from $SRSL(q, L(n))$ of every node n in *rsky-quadtrees*, we can obtain the reverse skyline.

To eliminate non-reverse skyline points in each node n, we need the local dynamic skyline midpoints $DSL(q, L_m(n))$ of every other node. For example, consider the points p_i, p_j, p_k $\in D_o$. If $mid(p_k, q) \prec_q mid(p_j, q)$ and $mid(p_j, q) \prec_q p_i$, we have $mid(p_k, q) \prec_q p_i$ and $p_i \notin RSL(q, D_o)$ by Lemma 5.3. Thus, if $mid(p_k, q) \prec_q mid(p_j, q)$, we only need $mid(p_k, q)$ to check whether p_i is a reverse skyline point or not.

The local dynamic skyline midpoints themselves are not sufficient, however, to eliminate all non-reverse skyline points from the strong reverse skylines. For instance, consider the point p_i in Figure 9. Although p_i is a strong reverse skyline point, $p_i \notin RSL(q, D)$ because $m_j (= mid(p_j, q)) \prec_q p_i$ holds. Since $m_i (= mid(p_i, q)) \prec_q m_j$ in $node(00)$, m_j is not a local dynamic skyline midpoint. Thus, if we only use the local dynamic skyline midpoints blindly, we cannot eliminate p_i correctly. However, since m_i is a local dynamic skyline midpoint, we can annotate m_i with a special symbol, representing that m_j dynamically dominates p_i, in order to utilize m_i to prune p_i. We call such annotated midpoints the *verification midpoints* as defined below:

Definition 5.7.: *Given a query point q and the set of midpoints $L_m(n)$ of an unpruned leaf node n in an* rsky-quadtree*, consider a point p such that $mid(p, q) \in DSL(q, L_m(n))$. The midpoint $mid(p, q)$ is a* verification midpoint *if there exists $m_j \in L_m(n)$ such that $mid(p, q) \prec_q m_j \prec_q p$.*

Lemma 5.8: *Given a query point q and an rsky-quadtree r of an orthant o, $p \in D_o$ is a reverse skyline point if and only if (1) p is in $SRSL(q, L(n))$ of an unpruned leaf node n in r,(2) $mid(p, q)$ is not a verification midpoint, and (3) for every unpruned leaf node n' in r, there does not exist $m (\neq mid(p, q)) \in DSL(q, L_m(n'))$ such that $m \prec_q p$.*

Proof: ($\Rightarrow$:) We prove the contrapositive: when one of the three conditions is not satisfied, $p \notin RSL(q, D)$.

When the condition (1) is not satisfied, by Definition 5.6, there is $m(\neq mid(p, q)) \in L_m(n)$ or $p_i(\neq p) \in L_p(n)$ s.t. $m \prec_q p$ or $mid(p_i, q) \prec_q p$. If the condition (2) is not satisfied, there is a midpoint m s.t. $mid(p, q) \prec_q m \prec_q p$ by Definition 5.7. When the condition (3) is not satisfied, there is $m(\neq mid(p, q)) \in DSL(q, L_m)$ s.t. $m \prec_q p$. Thus, whenever one of the three conditions is not satisfied, there exists a midpoint $m \neq mid(p, q)$ s.t $m \prec_q p$ and $p \notin RSL(q, D_o)$ according to Lemma 5.3. Therefore, $p \in RSL(q, D_o)$ implies that all three conditions are satisfied.

($\Leftarrow$:) For the purpose of contradiction, suppose $p \notin RSL(q, D_o)$. Based on Lemma 5.3, there exists a midpoint $m \neq mid(p, q)$ s.t. $m \prec_q p$ for a point p contained in the orthant o. Without loss of generality, assume that m is in an unpruned leaf node. (Otherwise, let m be m_u where m_u $(\neq mid(p, q))$ is a midpoint which is in an unpruned leaf node and dynamically dominates m. The midpoint m_u always exists by the properties of the *rsky-quadtrees* and we have $m_u \prec_q p$).

When p and m are located in the same node n of r, $p \notin SRSL(q, L(n))$ since $m \prec_q p$. It contradicts the condition (1) resulting that p and m should be located in different nodes of r. In the unpruned leaf node n_m containing m, if $m \in DSL(q, L_m(n_m))$, it contradicts the condition (3). Therefore, there exists another midpoint $m' \in DSL(q, L_m(n_m))$ s.t. $m' \prec_q m$. If $m' \neq mid(p, q)$, it also contradicts the condition (3) since $m' \prec_q m \prec_q p$. This implies that $m' = mid(p, q)$. Since m and $mid(p, q)$ are located in the same unpruned leaf node and $mid(p, q) \prec_q m \prec_q p$, $mid(p, q)$ is a verification midpoint by Definition 5.7. It contradicts the condition (2). Therefore, if all three conditions hold, p is a reverse skyline point.█

```
Function L-RSKY-MR.map(key, p)
key: null, p: a point
begin
1.  rsky-qtrees = LoadTrees(), q = LoadQuery();
2.  O(p) = FindOrthants(p, q);
3.  for each o ∈ O(p) do
4.      n_p = GetNode(p, rsky-qtrees[o]);
5.      if n_p.pruned == false then emit((o,n_p),("P",p));
6.      n_m = GetNode(mid(p, q), rsky-qtrees[o]);
7.      if n_m.pruned == false then emit((o,n_m),("M",mid(p,q)));
end
Function L-RSKY-MR.reduce(key, L)
key: (orthant id o, a node id n), L: a list of points and midpoints
begin
1.  q = LoadQuery();
2.  SRSL = StrongReverseSkyline(q,L);
3.  output(key, SRSL);
4.  L_m = {m ∈ L|m has symbol "M"};
5.  DSL = DynamicSkyline(q,L_m);
6.  for each midpoint m in DSL do
7.      if IsVerificationMidpoint(m, L_m) then output (key, ("V",m));
8.      else output (key, ("M",m));
9.  append(RSKY-FILTER, FilterPoint(DSL));
10. append(RVIRTUAL, VirtualMax(SRSL));
end
```

Figure 10: The L-RSKY-MR

We next define the *reverse virtual max point* of each leaf node of an *rsky-quadtree* and provide the property of the reverse virtual max points.

Definition 5.9.: *The reverse virtual max point of each leaf node n of an rsky-quadtree, denoted by rvp_n, is defined as the point whose k-th dimensional value $rvp_n(k)$ is $max_{p_i \in SRSL(q,L(n))} |p_i(k) - q(k)|$ for every k=1,2,$\cdots$,d.*

Proposition 5.10: *If a midpoint m does not dynamically dominate the reverse virtual max point of a leaf node n in an rsky-quadtree, m does not dynamically dominate every strong reverse skyline point in $region(n)$.*

We omit the proof of Proposition 5.10 because it is similar to that of Proposition 4.4 in Section 4.2.

5.3 L-RSKY-MR: The Local Reverse Skyline Computation Algorithm

Based on Lemmas 5.1, 5.3 and 5.8, the procedure *L-RSKY-MR* in Figure 10 computes the strong reverse skyline and local dynamic skyline midpoints in every unpruned leaf node of all *rsky-quadtrees*.

Each map function is called with a point p in D. To check whether a point p is a reverse skyline point or not, we examine only the points in each orthant containing p by Lemma 5.1. Thus, in the map function called with p, we examine each orthant o containing p independently. Note that if a point $p \in D_o$ is in the pruned leaf node of the

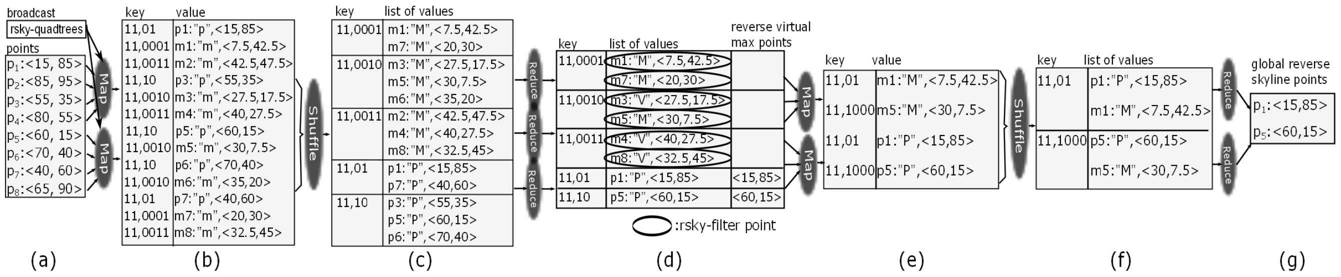

broadcast
rsky-quadtrees
points
p1:<15, 85>
p2:<85, 95>
p3:<55, 35>
p4:<80, 55>
p5:<60, 15>
p6:<70, 40>
p7:<40, 60>
p8:<65, 90>
Map
Map
key
value
11,01 p1:"p",<15,85>
11,0001 m1:"m",<7.5,42.5>
11,0011 m2:"m",<42.5,47.5>
11,10 p3:"p",<55,35>
11,0010 m3:"m",<27.5,17.5>
11,0011 m4:"m",<40,27.5>
11,10 p5:"p",<60,15>
11,0010 m5:"m",<30,7.5>
11,10 p6:"p",<70,40>
11,0010 m6:"m",<35,20>
11,01 p7:"p",<40,60>
11,0001 m7:"m",<20,30>
11,0011 m8:"m",<32.5,45>
Shuffle
key
list of values
11,0001 m1:"M",<7.5,42.5> m7:"M",<20,30>
11,0010 m3:"M",<27.5,17.5> m5:"M",<30,7.5> m6:"M",<35,20>
11,0011 m2:"M",<42.5,47.5> m4:"M",<40,27.5> m8:"M",<32.5,45>
11,01 p1:"P",<15,85> p7:"P",<40,60>
11,10 p3:"P",<55,35> p5:"P",<60,15> p6:"P",<70,40>
Reduce
Reduce
Reduce
key
list of values
reverse virtual max points
11,0001 m1:"M",<7.5,42.5> m7:"M",<20,30>
11,0010 m3:"V",<27.5,17.5> m5:"M",<30,7.5>
11,0011 m4:"V",<40,27.5> m8:"V",<32.5,45>
11,01 p1:"P",<15,85> <15,85>
11,10 p5:"P",<60,15> <60,15>
:rsky-filter point
Map
Map
key
value
11,01 m1:"M",<7.5,42.5>
11,1000 m5:"M",<30,7.5>
11,01 p1:"P",<15,85>
11,1000 p5:"P",<60,15>
Shuffle
key
list of values
11,01 p1:"P",<15,85> m1:"M",<7.5,42.5>
11,1000 p5:"P",<60,15> m5:"M",<30,7.5>
Reduce
Reduce
global reverse skyline points
p1:<15,85>
p5:<60,15>
(a)
(b)
(c)
(d)
(e)
(f)
(g)

Figure 11: The local and global reverse skyline phases of RSKY-MR

rsky-quadtree, p is not a global reverse skyline point due to Lemma 5.3 since there exists a midpoint of another point in D_o which dynamically dominates p wrt q. For each orthant o, we perform the following two steps: (1) We check whether p belongs to an unpruned leaf node of o's *rsky-quadtree*. If it does, we emit $\langle(o, n_p)$,("P", $p)\rangle$ where "P" represents that p is a point (in lines 2-5 of *L-RSKY-MR.map*). (2) If $mid(p, q)$ belongs to an unpruned leaf node n_m, we output $\langle(o,\ n_m)$, ("M", $mid(p, q))\rangle$ where "M" denotes that m is a midpoint (in lines 6-7).

After the shuffling phase groups the output of the map functions according to each distinct unpruned leaf node, a reduce function is called with each distinct group. For each distinct group (o, n), the list $L(n)$, which is $L_p(n) \cup L_m(n)$ as defined in Definition 5.6, is generated.

Consider the reduce function called with a distinct group (o, n) and the input value list $L(n)$. The reduce function computes the strong reverse skyline (i.e., $SRSL(q, L(n))$) and the local dynamic skyline of midpoints (i.e., $DLS(q,$ $L_m(n))$) according to Lemma 5.8 (in lines 1-5 of *L-RSKY-MR.reduce*). For every strong reverse skyline point, the reduce function outputs the key-value pair $\langle(o, n)$, ("P", $p)\rangle$ (in line 3). In addition, the reduce function emits $\langle(o, n)$, ("V", $v)\rangle$ for every verification midpoint v in $DSL(q,\ L_m(n))$ defined in Definition 5.7 (in lines 6-7). For every midpoint m in $DSL(q,\ L_m(n))$ which is not a verification midpoint, the reduce function outputs $\langle(o, n)$, ("M", $m)\rangle$(in line 8).

Similar to *L-SKY-MR*, for each dimension, the reduce function chooses a single midpoint, called an *rsky-filter midpoint*, in $DSL(q, L_m(n))$ which has the minimum value of the dimension. The reduce function also computes the reverse virtual max point of the leaf node n. Finally, the reduce function outputs the rsky-filter midpoints and the reverse virtual max point to the files called RSKY-FILTER and RVIRTUAL in HDFS respectively (in lines 9-10).

EXAMPLE 5.11: *Consider the rsky-quadtree in Example 5.5. In the local reverse skyline phase, a map function is invoked with each point $p \in D$. For instance, the map function with p_5 outputs $\langle(11, 10)$, ("P", $p_5)\rangle$ because p_5 belongs to the unpruned leaf node, node(10), in the orthant with id=11. In addition, since mid(p_5, q) belongs to an unpruned leaf node, node(0010), in the same orthant, the map function also outputs $\langle$(11, 0010), ("M", mid(p_5, q))$\rangle$. The key-value pairs output by all map functions are shown in Figure 11(b). The output of the shuffling phase is presented in Figure 11(c).*

For each distinct key (o, n), a reduce function is called with the list of points and midpoints in region(n). For instance, the reduce function invoked with the key (11,10) receives $\{p_3, p_5, p_6\}$ as input value list and outputs $\langle$(11,10),(p_5, "P")$\rangle$ since p_5 is a strong reverse skyline point. The reduce function next calculates the verification midpoints, reverse virtual max point and rsky-filter points. In node(0011), m_4 is annotated with "V" since m_2 is in region(n) and m_2 dominates p_4. In node(0011), m_4 and m_8 are selected as the rsky-filter midpoints since $m_8(1) = 32.5$ and $m_4(2) = 27.5$ are the minimum value on the first and second dimensions respectively. In Figure 11(d), we have shown the output emitted by all reduce functions where the rsky-filter midpoints are circled. ∎

```
Function G-RSKY-MR.map($key$, $p$)
$key$: (an orthant id $o$, a node id $n$), $p$: (a point or a midpoint $p$, $mark$)
begin
1.  $q$ = LoadQuery();
2.  if IsPoint($p$) then
3.     if DynamicDominatedByFilterPoints($p$,$q$,$o$) then return;
4.     emit($key$, ("P", $p$));
5.  if IsMidpoint($p$) then
6.     $rsky\text{-}quadtrees$ = LoadTrees()
7.     $nodes$ = LoadNonEmptyNodes(o);
8.     for each node id $n_i$ in $nodes$
9.         if IsNeeded($n$, $n_i$)then
10.            output(($o$,$n_i$), (mark, $p$));
end
```

Figure 12: The G-RSKY-MR.map

5.4 G-RSKY-MR: The Global Reverse Skyline Computation Algorithm

The parallel algorithm *G-RSKY-MR* finds the global reverse skyline points independently in each non-empty unpruned leaf node by Propositions 4.2, 5.10 and Lemma 5.8. We omit the details of the serial algorithm *G-RSKY* due to space limitations.

For every strong reverse skyline point p, we check whether (1) p is not dynamically dominated by a local dynamic skyline midpoint m (i.e., $m \not\prec_q p$) and (2) p's midpoint is not one of the verification midpoints. If both conditions are satisfied, p is a global reverse skyline point due to Lemma 5.8. To check the condition (1), we examine whether $m \prec_q p$ for every midpoint m contained in all unpruned leaf nodes n_i. However, we do not need to check whether $m \prec_q p$ if there is k such that $sub(id(n_i), k) > sub(id(n_j), k)$ where n_j is the node containing p and $sub(id(n), k)$ is the k-th substring of n's id defined in Section 4.1. The reason is that we have $m \not\prec_q p$ for every point p in n_j according to Proposition 4.2. In addition, if $m \not\prec_q rvp_{n_j}$ (i.e., the reverse virtual max point of n_j), since $m \not\prec_q p$ for every strong reverse skyline point p belonging to n_j by Proposition 5.10, we do not need to check $m \prec_q p$ either.

The pseudocode of *G-RSKY-MR* is presented in Figure 12. Each map function is invoked with a strong reverse skyline point (i.e., annotated with "P") or a local dynamic skyline midpoint (i.e., annotated with "M" or "V") which were generated at the previous phase. Consider a map function called with a strong reverse skyline point p in an unpruned leaf node n in an orthant o. Note that p is not a global reverse skyline point if p is dominated by another point's

Parameter	Range	Default
No. of samples (s)	$100 \sim 8,000$	400 (for skyline) 1,000(for reverse skyline)
Split threshold (ρ)	$10 \sim 60$	20 (for skyline) 40 (for reverse skyline)
No. of points (n)	$10^7 \sim 4 \times 10^9$	10^8
No. of dimensions (d)	$2 \sim 10$	6
No. of machines (t)	$5 \sim 20$	10

Table 1: Parameters

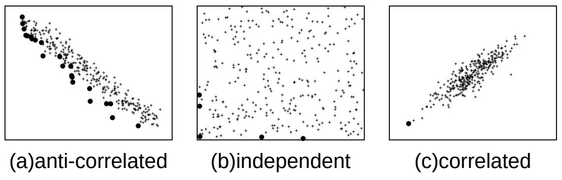


Figure 13: Examples of data sets

midpoint by Lemma 5.3. Thus, the map function emits $\langle(o, n)$, ("P",p)$\rangle$ if p is not dynamically dominated by rsky-filter midpoints (in lines 1-4 of *G-RSKY-MR.map*).

For the map function called with a local dynamic skyline midpoint m contained in an unpruned leaf node n_m in an orthant o, the map function should find out all unpruned leaf nodes n requiring m to check whether n's strong reverse skyline points are the global reverse skyline points. If $n_m \not\prec n$ or $m \not\prec_q rvp_n$, n does not require m to check n's strong reverse skyline points by Propositions 4.2 and 5.10. For each unpruned leaf node n which requires m, the map function outputs $\langle(o, n)$, ("V",m)$\rangle$ if m is a verification midpoint. Otherwise, it outputs $\langle(o, n)$, ("M",m)$\rangle$ (in lines 5-10).

The key-value pairs emitted by map functions are grouped according to each distinct unpruned leaf node in the shuffling phase and a reduce function is called with each distinct group. Each reduce function checks whether the strong reverse skyline points in a node are the global reverse skyline points based on Lemma 5.8. If a strong reverse skyline point p is dynamically dominated by the midpoints coming from the other nodes or p's midpoint is annotated with "V", p cannot be a reverse skyline point. Finally, the reduce function outputs the global skyline points.

EXAMPLE 5.12.: *Assume a map function is called with each point in the output of the local reverse skyline phase in Example 5.11. Since m_1 in node(0001) dynamically dominates the reverse virtual max point of node(01), the map function with m_1 emits $\langle(11, \mathbf{01})$, ("M", m_1)$\rangle$. However, since m_3 does not dynamically dominate the reverse virtual max point of node(1000), we do not emit $\langle(11, \mathbf{1000})$, ("V", m_3)$\rangle$. Figures 11(e) and 11(f) show the output of all map functions and the result of the shuffling phase respectively.*

For every unpruned leaf node, a reduce function is called to see whether each strong reverse skyline point is actually a global reverse skyline point. For example, the input value list of the reduce function with the key (11, 01) is $\{p_1, m_1\}$. Since m_1 is not a verification midpoint, p_1 is a global reverse skyline point. After every reduce function is finished, $\{p_1, p_5\}$ becoms the reverse skyline as in Figure 11(g). ▮

6. EXPERIMENTS

We empirically evaluated the performance of our proposed algorithms using the parameters as summarized in Table 1. All experiments on MapReduce were performed on the cluster of 20 nodes of Intel(R) Core(TM) i3 CPU 3.3GHz machines with 4GB of main memory running Linux. The implementations of all algorithms were complied by Javac 1.6. We used Hadoop 1.0.3 for MapReduce [2]. The execution times in the graphs shown in this section are plotted in log scale. We ran all algorithms five times and measured the average execution times. We do not plot the execution times of some algorithms when they did not finish within 8 hours.

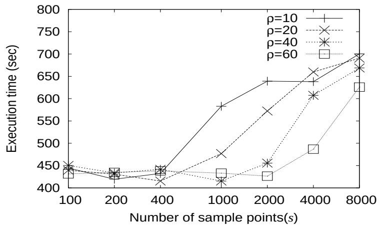


Figure 14: *SKY-MR* with varying s and ρ

Algorithm	Description
SKY-MR-S/M	*SKY-MR-S* utilizes the serial algorithm *G-SKY*. *SKY-MR-M* utilizes *G-SKY-MR*.
SKY-MR	*SKY-MR* adaptively selects *G-SKY-MR* or *G-SKY* wrt the number of local skyline points. If it is less than 7×10^5, *G-SKY* is selected.
MR-BNL	The state-of-the-art using MapReduce in [30].
PPPS-MR	The MapReduce implementation of PPPS in [14]. We set the sample size (s) to 1,000 which shows the best performance.
GRID-MR-1/2	The MapReduce implementations of the 1-step and 2-step algorithms in [1].
SKY-SC	The serial implementation of *SKY-MR*.
BBS	The state-of-the-art for a single core in [22].
SKY-MC	The implementation of *SKY-MR* for multi-cores.
PPPS	The state-of-the-art for multi-cores in [14].
SKY-MP	The implementation of *SKY-MR* using MPI.
GRID-1/2	The implementations of the 1-step and 2-step algorithms using MPI in [1].

Table 2: Implemented skyline algorithms

Data sets: We built three synthetic data sets which were randomly generated by *correlated*, *independent* and *anti-correlated* distributions. The three types of data sets are typically used to evaluate the performance of skyline algorithms [4]. Figure 13 shows the examples of such data sets where skyline points are represented by small bold circles. The sizes of resulting synthetic data sets are varied from 392MB to 153GB depending on the number of points (n) as well as the number of dimensions (d).

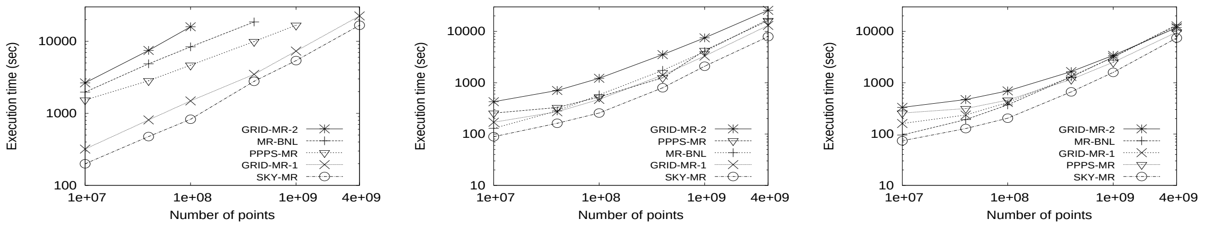
Implemented algorithms: The MapReduce algorithms implemented for skyline and reverse skyline are presented in Table 2 and Table 3 respectively. Furthermore, we also implemented the variants of *SKY-MR* for other environments such as using a single-core machine, multi-core machines and message passing interface (MPI) library [16] to see the effectiveness of our proposed algorithms compared to the existing algorithms[1, 14, 22] in such environments.

6.1 Performance Results for Skylines

Default values of s and ρ: To find the proper values of s and ρ, we ran *SKY-MR* with varying s from 100 to 8,000 and ρ from 10 to 60. The average execution times of *SKY-MR* for all data sets are shown in Figure 14. Since the best performance of *SKY-MR* is obtained with $s = 400$ and

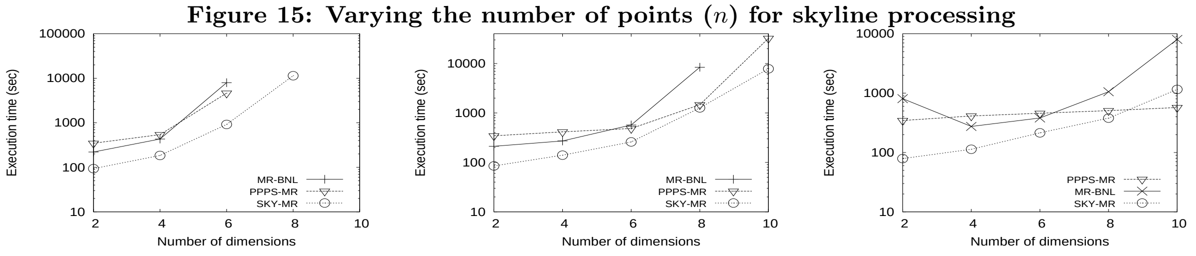
Algorithm	Description
RSKY-MR-S/M	*RSKY-MR-S* utilizes *G-RSKY*. *RSKY-MR-M* utilizes *G-RSKY-MR*.
RSKY-MR	*RSKY-MR* adaptively selects *G-RSKY-MR* or *G-RSKY* wrt the number of strong reverse skyline points. If it is less than 10^4, *G-RSKY* is selected.
BR-RSKY-MR	Our brute-force algorithm without using *rsky-quadtrees* in Section 5

Table 3: Implemented reverse skyline algorithms



(a) Anti-correlated (b) Independent (c) Correlated

Figure 15: Varying the number of points (n) for skyline processing



(a) Anti-correlated (b) Independent (c) Correlated

Figure 16: Varying the number of dimensions (d) for skyline processing

$\rho = 20$, we set $s = 400$ and $\rho = 20$ as the default values. When the sample size s decreases, since the samples do not reflect the data distribution precisely, the number of pruned points decreases and *SKY-MR* becomes inefficient. In *SKY-MR*, virtual max points, sky-filter points and local skyline points of an unpruned node are sent to the other unpruned leaf nodes. Thus, as the sample size s increases, the number of unpruned leaf nodes of a *sky-quadtree* which receives such points from other unpruned nodes increases and *SKY-MR* becomes inefficient due to high network costs. Decreasing ρ has also a similar effect of increasing the sample size s.

Varying n: We varied n from 10^7 to 4×10^9 and plot the running times of the algorithms in Figure 15. *SKY-MR* is always better than *SKY-MR-S/M* since it switches to *SKY-MR-S* or *SKY-MR-M* adaptively based on the number of local skyline points. Thus, we do not report the performance of *SKY-MR-S/M* in the rest of the paper.

Since the number of skyline points of the anti-correlated data sets is generally larger than those of the independent data sets and the correlated data sets, the algorithms with the anti-correlated data sets take generally more execution time than those of the other data sets.

GRID-MR-2 is always the worst performer due to the high cost of computing the relaxed skyline grids from t^d grids (e.g., when $t = 10$ and $d = 6$, we have $t^d = 10^6$ number of grids). *MR-BNL* performs better than *GRID-MR-2*, but it is still slower than *SKY-MR* because *MR-BNL* calculates the global skyline in a single machine only. *GRID-MR-1* performs poorly because it broadcasts all points p in each relaxed skyline grid to every other grid containing points which may be dominated by p. Since *SKY-MR* filters out non-skyline points effectively using the *sky-quadtree*, it shows the best performance.

Varying d: With varying d from 2 to 10, we plot the execution times of the algorithms except *GRID-MR-1/2* in Figure 16 because they show similar patterns with varying n.

The execution times of all algorithms increase gradually with increasing d since checking the dominance relationship between two points becomes more expensive with large values of d. Furthermore, when $d = 2$, *MR-BNL* becomes slow because *MR-BNL* utilizes only 4 ($= 2^d$) machines out of 10 machines. For the independent and anti-correlated data sets, *PPPS-MR* becomes slow since the last two partitions are merged in a single machine. However, *PPPS-MR* becomes fast for the correlated data sets, since there are a small number of local skyline points and merging them can be done quickly. The graphs confirm that *SKY-MR* is generally the best performer.

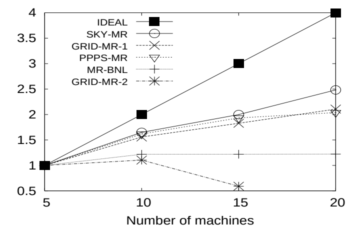


Figure 17: Relative speed with varying the number of machines (t) for skyline processing

SKY-MR	F and V	F	V	NONE
Correlated	218	229	234	242
Independent	260	281	283	283
Anti-correlated	840	1146	952	1207

Table 4: Effects of the virtual max points (V) and the sky-filter points (F) (sec)

Varying t: We show the relative speed of the tested algorithms averaged over all data sets in Figure 17. That is, for each algorithm, we plot its running time with 5 machines divided by its running time with t machines. For example, if the running times of *SKY-MR* with 5 and 20 machines are T_5 and T_{20} respectively, we plot the ratio T_5/T_{20} for t=20. In an ideal case, if the number of machines increases by 4 times from 5 to 20, the speed will be 4 times faster. We also plot the ideal speedup curve in the graphs of Figure 17. For the relative speed, our proposed algorithm *SKY-MR* shows the best scalability since *SKY-MR* effectively prunes data by partitioning with the *sky-quadtrees* and utilizes the virtual max points and sky-filter points to reduce the unnecessary comparisons.

The effects of the virtual max and sky-filter points: To evaluate the performance improvements by utilizing the

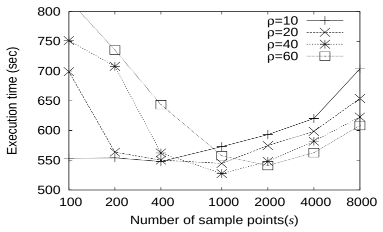


Figure 18: *RSKY-MR* with varying s and ρ

virtual max points and sky-filter points, we report the execution times of *SKY-MR* using virtual max point and the sky-filter points (F and V), *SKY-MR* using only the sky-filter points (F), *SKY-MR* using only the virtual max points(V), and *SKY-MR* without both points (NONE) in Table 4. For the correlated data set, the performance of *SKY-MR* is improved mainly by the sky-filter points, since most of the local skyline points are removed by sky-filter points in the third phase. For the anti-correlated data set, since the virtual max points reduce unnecessary comparisons between the local skyline points by Proposition 4.4, the running time decreases. Since *SKY-MR* utilizes both sky-filter points and virtual max points, it runs faster than the other algorithms.

6.2 Performance Results for Reverse Skylines

We next present the experimental results of the reverse skyline algorithms with randomly generated query points.

Default values of s and ρ: To choose the proper values of s and ρ, we varied s from 100 to $8,000$ and ρ from 10 to 60. Figure 18 presents the average execution time over all data sets. We utilize $s = 1,000$ and $\rho = 40$ as the default values since *RSKY-MR* shows the best performance with those values. Note that small and large values of s make *RSKY-MR* inefficient, as we mentioned in Section 6.1.

Varying n: We varied n from 10^7 to 4×10^9 and plot the execution times in Figure 19. Similar to the skyline experimental results with varying n, the performance of every algorithm on the anti-correlated data set is worse than that of itself on the other data sets. When there is a skew in data such that a lot of points belong to an orthant, *BR-RSKY-MR* shows the worst performance since *BR-RSKY-MR* computes the reverse skyline in each orthant independently.

RSKY-MR-M shows better performance than *RSKY-MR-S* due to its parallelization of the third phase when the number of strong reverse skyline points is large. As we expected, the performance of *RSKY-MR-S* is better than that of *RSKY-MR-M* only for small correlated data sets. Since *RSKY-MR* selects *RSKY-MR-M* or *RSKY-MR-S* adaptively depending on the number of strong reverse skyline points, *RSKY-MR* always shows the best performance.

Varying t and d: As expected, *RSKY-MR* shows the best scalability with varying t and d. The graphs with varying t show almost the same trends with those for skyline processing with varying t in Figure 17. Furthermore, the graphs varying d have almost the same trends with those with varying n in Figure 19. Thus, we do not provide the graphs for the experiments with varying t and d.

6.3 Performance Results in Other Environments

We finally present the experimental results by comparing the performance of our ported algorithms to other environments with the existing state-of-the-art algorithms in such

Distr.	Alg. \ n	10^5	10^6	10^7	10^8	10^9
Anti.	*SKY-SC*	1.5	91.3	949.8	8906.8	-
	BBS	39.9	603.2	16334.5	-	-
Ind.	*SKY-SC*	0.6	1.6	10.1	70.4	-
	BBS	0.3	1.9	12.6	2275.8	-
Cor.	*SKY-SC*	0.2	0.5	3.3	41.5	-
	BBS	0.1	0.2	1.3	11.6	-

Table 5: Varying n on a single core machine (sec)

Distr.	Alg. \ n	10^7	4×10^7	10^8	4×10^8	10^9
Anti.	*SKY-MC*	116.8	380.2	877.4	-	-
	PPPS	2201.5	10887	21736	-	-
Ind.	*SKY-MC*	11.3	37.7	81.4	290.8	666.0
	PPPS	14.9	66.5	193.1	1171.3	-
Cor.	*SKY-MC*	5.1	17.8	44.5	170.3	410.8
	PPPS	4.9	20.5	50.2	216.3	512.2

Table 6: Varying n on a multi-core machine (sec)

environments. We did experiments with varying n and d, but reported only the experimental results with varying n.

Single core machine: We compared our serial algorithm *SKY-SC* to the state-of-the-art serial algorithm *BBS* [22] which utilizes an R*-tree on a single core machine. We report the average execution times with varying n from 10^5 to 10^9 in Table 5. We do not include the construction time of R*-trees for *BBS*, but we include the construction time of sky-quadtrees for *SKY-SC* in Table 5. Whenever any algorithm did not finish due to lack of memory, we do not show the running time in Table 5.

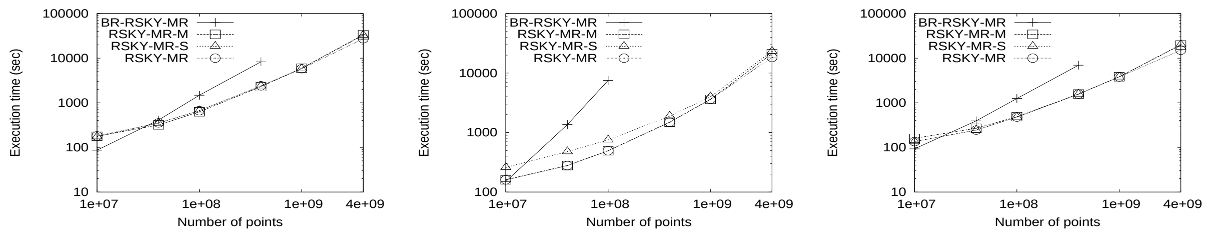
BBS finds skyline points progressively in increasing order of their distances to the origin. When the number of skyline points is small (i.e., correlated data), most of minimum bounding rectangles (MBRs) of R*-trees are pruned by the skyline points found at the beginning of *BBS* and thus *BBS* shows slightly better performance than *SKY-SC*. However, when the number of skyline points is large (i.e., independent or anti-correlated data), many MBRs are not pruned by the skyline points. Since *SKY-SC* filters out non-skyline points effectively using the *sky-quadtree* as well as virtual max points and sky-filter points, when the number of skyline points becomes large, *SKY-SC* performs much better than *BBS*.

Multi-core machine: We evaluated our *SKY-MC* and *PPPS* [14] devised for multi-core machines. Experiments were performed on a 32-core machine of Intel(R) Xeon(TM) E7 CPU 2.67GHz with 128GB of main memory running Linux. We show the average execution times with varying n from 10^7 to 10^9 in Table 6. Whenever any algorithm did not finish due to lack of memory, we do not show the running time in the table. As shown in Table 6, *SKY-MC* is much better than *PPPS* for all cases even if our work is originally developed for MapReduce. The reason is that *SKY-MC* filters out non-skyline points effectively using the *sky-quadtree* as well as virtual max points and sky-filter points.

MPI: We compared our *SKY-MP* with *GRID1* and *GRID2*

Distr.	Alg. \ n	10^7	4×10^7	10^8	4×10^8	10^9
Anti.	*SKY-MP*	190	665	1478	4513	9509
	GRID1	226	699	1586	4651	9698
	GRID2	5134	20955	-	-	-
Ind.	*SKY-MP*	12	53	183	669	1583
	GRID1	20	69	142	785	1801
	GRID2	641	860	1139	2300	4250
Cor.	*SKY-MP*	4.9	16	51	365	669
	GRID1	5.4	28	66	442	1126
	GRID2	237	447	642	1250	2322

Table 7: Varying n on MPI (sec)



(a) Anti-correlated (b) Independent (c) Correlated

Figure 19: Varying the number of points (n) for reverse skyline processing

proposed in [1]. We used MPICH2 [16] for the implementations of MP-model and GMP-model. We report the average execution times with varying n in Table 7. Whenever any algorithm did not finish within 8 hours, we do not show the running time in the table. Similar to the experiments with multi-core machines, our *SKY-MP* performs better than the others due to effective pruning.

7. CONCLUSION

We study the problems of skyline and reverse skyline computations using MapReduce. We introduce the parallel algorithms, *SKY-MR* and *RSKY-MR*, which compute the skyline and reverse skyline respectively. To filter out non-skyline points and non-reverse skyline points in advance, we propose new histograms, called the *sky-quadtree* and *rsky-quadtree*. Both *SKY-MR* and *RSKY-MR* partition the data based on the region split by the *sky-quadtree* and *rsky-quadtree* respectively and compute the candidate (reverse) skyline points independently for each partition. Our experiments confirm the effectiveness and scalability of our algorithms.

Acknowledgment

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (2009-0078828). This research was also supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning (2012M3C4A7033342) as well as Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (2012R1A1B3003060).

8. REFERENCES

[1] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman. Parallel skyline queries. In *ICDT*, pages 274–284, 2012.
[2] Apache. Apache hadoop. http://hadoop.apache.org, 2010.
[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.
[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *IEEE ICDE*, pages 421–430, 2001.
[5] R. E. Bryant. Data intensive scalable computing. *Carnegie Mellon University. Retrieved August*, 10:2009, 2008.
[6] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. On high dimensional skylines. In *EDBT*, 2006.
[7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *IEEE ICDE*, pages 717–719, 2003.
[8] D. Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.
[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD Conference*, 2010.
[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communication of the ACM*, 51(1):107–113, 2008.
[11] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, pages 291–302, 2007.
[12] R. Finkel and J. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1), 1974.
[13] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in manets. In *ICDE*, page 66, 2006.
[14] H. Köhler, J. Yang, and X. Zhou. Efficient parallel skyline processing using hyperplane projections. In *SIGMOD Conference*, pages 85–96, 2011.
[15] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.
[16] A. N. Laboratory. Mpich2. http://www.mpich.org/.
[17] T. Lappas and D. Gunopulos. Efficient confident search in large review corpora. In *ECML/PKDD (2)*, 2010.
[18] J. Lee, S. won Hwang, Z. Nie, and J.-R. Wen. Navigation system for product search. In *ICDE*, 2010.
[19] J. J. Levandoski, M. F. Mokbel, and M. E. Khalefa. Preference query evaluation over expensive attributes. In *CIKM*, 2010.
[20] X. Lian and L. Chen. Reverse skyline search in uncertain databases. *ACM Trans. Database Syst.*, 35(1), 2010.
[21] X. Lin, Y. Zhang, W. Zhang, and M. A. Cheema. Stochastic skyline operator. In *ICDE*, pages 721–732, 2011.
[22] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.
[23] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *VLDB*, pages 15–26, 2007.
[24] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *VLDB*, pages 751–762, 2006.
[25] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.
[26] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *IEEE Trans. Knowl. Data Eng.*, 18(2):377–391, 2006.
[27] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.
[28] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis. Skypeer: Efficient subspace skyline computation over distributed data. In *ICDE*, 2007.
[29] G. Wang, J. Xin, L. Chen, and Y. Liu. Energy-efficient reverse skyline query processing over wireless sensor networks. *IEEE Trans. Knowl. Data Eng.*, 24(7), 2012.
[30] B. Zhang, S. Zhou, and J. Guan. Adapting skyline computation to the mapreduce framework: Algorithms and experiments. In *DASFAA*, pages 403–414, 2011.
[31] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu. Probabilistic skyline operator over sliding windows. In *ICDE*, pages 1060–1071, 2009.
[32] L. Zhu, Y. Tao, and S. Zhou. Distributed skyline retrieval with low bandwidth consumption. *IEEE Trans. Knowl. Data Eng.*, 21(3):384–400, 2009.
[33] L. Zou, L. Chen, M. T. Özsu, and D. Zhao. Dynamic skyline queries in large graphs. In *DASFAA*, 2010.