


MapReduce-based skyline query processing scheme using adaptive two-level grids

Hyeong-Cheol Ryu¹ · Sungwon Jung¹ 

Received: 25 May 2017 / Revised: 12 June 2017 / Accepted: 16 September 2017 / Published online: 26 September 2017
© Springer Science+Business Media, LLC 2017

Abstract Skyline queries are extensively used for solving many problems such as product recommendation, because skylines contain data to satisfy various user criteria. Currently, skyline queries for large databases are being investigated. In particular, the research using the existing index techniques to MapReduce for large databases in a parallel and distributed environment has been actively conducted. A characteristic of skyline queries is that the data closer to the origin dominate more data regions. In this paper, we propose a novel index technique using adaptive two-level grids, called *TLG*. It separates the data space into regions by considering the characteristic of the skyline queries. We also propose an efficient skyline query algorithm based on *TLG*. It computes the skylines for each data region for reducing the number of checking dominance relationship between data points in different regions.

Keywords Skyline · Location-based system · MapReduce · Multi dimensional databases

1 Introduction

Skyline queries [1] are beneficial in many diverse areas of applications including bio and medical areas [2,3], product recommendation, restaurant recommender system [4], review rate reflecting user feedback [5], and decision-making problem areas [1,6].

✉ Sungwon Jung
jungswng@sogang.ac.kr
Hyeong-Cheol Ryu
hchryu@gmail.com

¹ Department of Computer Science and Engineering,
Sogang University, Seoul, Korea

Investigations of skyline query processing on a single machine were performed [1,7–9] using centralized index structures such as *B⁺-tree* [10] and *R*-tree* [11]. However, these skyline query processing techniques are not suitable for large databases owing to their scalability and computation complexity problems. For instance, there are more than 1 million transactions per hour at Wal-Mart stores [12], and 1 billion photos are uploaded by Facebook users in a day [13]. The skyline queries are frequently used as primitive operators for quickly processing these large databases to provide pricing decisions and develop marketing strategies. A method to solve these problems is MapReduce [14], which has recently attracted the attention of researchers. A variety of fields of research [15,16] that needs for processing and analyzing large database has used MapReduce technique.

MR-BNL and *MR-SFS* [17], which extend the existing skyline query techniques to MapReduce, have been proposed. *MR-BNL* and *MR-SFS* algorithms partition dataset into sub-datasets and compute local skylines on each sub-dataset. Next, a single machine merges all local skylines and computes global skylines. If there are numerous local skylines, considerable running time will be required for processing them using a single machine. On the other hand, our proposed algorithm computes global skyline simultaneously on multiple machines.

A technique for computing skylines in *SpatialHadoop* has been proposed in [18]. To compute skylines, the technique first performs filtering. As the global index of *SpatialHadoop* has information about the region of a node, it can filter the regions that do not have skylines beforehand. However, the technique is only available in *SpatialHadoop*. On the other hand, our proposed algorithm can be used in any framework that supports MapReduce.

A novel MapReduce-based scheme *SKY-MR* has been proposed in [19]. *SKY-MR* uses a quad-tree, called *sky-quadtree*,

which is fabricated using randomly extracted data from the input dataset. The sky-quadtree serves as a basis of pruning non-skyline points on each map function and enhances the workload balance of available machines. *SKY-MR* also uses *virtual max point* and *sky-filter point* for reducing the amount of filter points required for independently computing global skylines on multiple machines. However, *SKY-MR* should search the *sky-quadtree* for every input data. Moreover, it has a disadvantage, that is, MapReduce job can be performed twice.

In this paper, we propose an effective one-phase MapReduce algorithm for computing skylines. First, we propose a novel index technique named *TLG*, which stands for two-level grids. In particular, our algorithm effectively prunes non-skyline points as it adaptively divides the data space. Next, we show the memory efficiency of *TLG*. Finally, we propose the effective skyline algorithm based on *TLG*. We showed experimentally that our proposed algorithm performs better than the existing algorithm in a low-dimensional data space. This paper performs the following contributions:

Novel pruning technique based on TLG We propose *TLG*-based pruning technique for decreasing the skyline computation overhead. Moreover, it does not require searching the tree and can be used in various data distributions.

Computing skylines in each data region independently To reduce the number of checking dominance relationship between all pairs of points, we compute partial skylines in each data region and merge them to obtain a complete skyline.

This paper is organized as follows. The proposed index *TLG* and techniques are presented in Sect. 2. Section 3 explains the proposed algorithm utilizing the *TLG* index. Section 4 reports the performance evaluation. Finally, the conclusions are provided in Sect. 5.

2 Adaptive two-level grids

This section describes the *TLG* index. First, we explain the grid block. Then, we extend the grid block to *TLG*.

2.1 Two-level grids index

Before explaining *TLG*, we define a grid block and describe the dominance power of the skyline queries. We then present the partitioning technique based on an arithmetic sequence and *TLG*. *TLG*, which considers the dominance power, can be utilized for datasets of several distributions. *TLG* can be applied to a multi-dimensional data space; however, we use a two-dimensional data space for ease of exposition.

Consider a d -dimensional data space consisting of n^d regions where each dimension of the data space is divided into n intervals. These n intervals are numbered 1 to n from

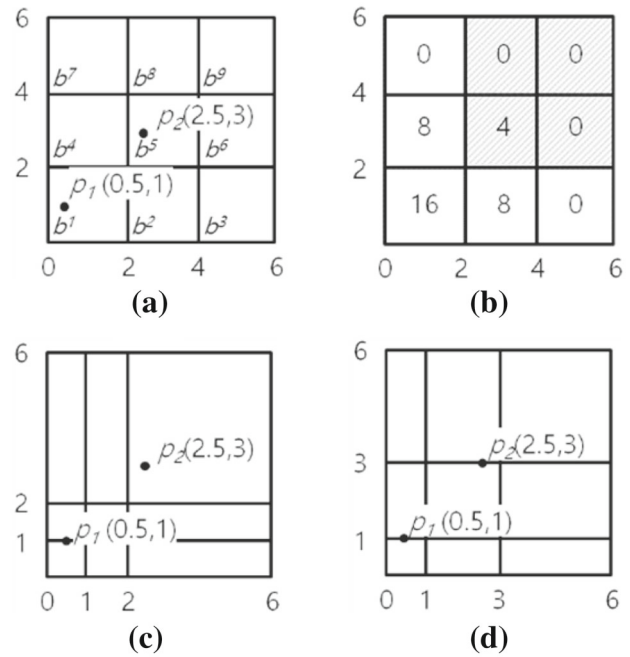


Fig. 1 Examples of grid blocks in 2D data space

left to right in each dimension. We name each region as a grid block. Moreover, we use the row-major order for numbering the grid blocks, as shown in Fig. 1a.

Definition 2.1 An i th grid block b^i is represented by $\langle b_1^i, b_2^i, \dots, b_d^i \rangle$ where b_k^i is the interval number of b^i corresponding to the k th dimension for all $k \in [1, d]$.

Without loss of generality, we assume that the minimum interval numbers are preferred for each dimension in this paper. We formally define a dominance relationship between grid blocks in Definition 2.2.

Definition 2.2 Given two grid blocks b^i and b^j , b^i dominates b^j , denoted as $b^i < b^j$, if and only if $\forall k \in [1, d], b_k^i < b_k^j$. If b^i contains a point, b^i is named a dominating grid block and b^j is named a dominated grid block.

A grid block b^1 , which has a point p_1 in Fig. 1a, dominates four grid blocks $b^5, b^6, b^8,$ and b^9 . By identifying the dominating grid blocks, we can remove all the points in the dominated grid blocks for skyline computation. Therefore, we divide the data space into grid blocks. There are many types of grid blocks including fixed-size grid blocks and variable-size grid blocks. Figure 1a, b shows 3^2 fixed-size grid blocks where the partition points of each dimension are 2 and 4. Figure 1c, d shows 3^2 variable-size grid blocks. We calculate the volume of grid block as follows:

Definition 2.3 Let $(L(b_k^i), U(b_k^i)]$ be the range of the k th dimension of a grid block b^i . We define the volume V of b^i as follows:

$$V(b^i) = \prod_{k=1}^d |U(b_k^i) - L(b_k^i)|$$

Considering the volumes of the grid blocks dominated by b^i , we define the dominance power of b^i as follows:

Definition 2.4 Given a grid block b^i , we denote $b^i.DGB$ as a set of grid blocks that are dominated by b^i . We define the dominance power DP of b^i as:

$$DP(b^i) = \sum_{b^j \in b^i.DGB} V(b^j)$$

For example, in Fig. 1b, grid blocks that are dominated of b^1 are filled with diagonal patterns. $DP(b^1)$ is $V(b^5) + V(b^6) + V(b^8) + V(b^9) = 16$. Figure 1c shows finely divided partial data space near the origin. In Fig. 1c, we can observe the domination of more grid blocks in a dataset that contains a point near the origin such as $p_1 = \langle 0.5, 1 \rangle$. However, if we use the datasets that only contain $p_2 = \langle 2.5, 3 \rangle$, the grid in Fig. 1c will not contain dominated grid blocks. In order to dominate many grid blocks even if we use a dataset that does not contain a point near the origin, we suggest a grid consisting of variable-size grid blocks based on an arithmetic sequence to apply to diverse datasets, as shown in Fig. 1d.

Definition 2.5 We divide the data space into variable-size grid blocks by using the partition points of each dimension, which increase the interval in the order of the arithmetic sequence.

Let n be the number of grid blocks per dimension, c be the common difference, and f be the first term of the arithmetic sequence. We use the same value of c and f to reduce the calculation cost. The equation of an arithmetic series $n(2f + (n - 1)c)/2$ is simplified to $nc(n + 1)/2$.

Let us now assume that the range of a dimension is $[0,30]$, as shown in Fig. 2. In this example, if we divide the range by five intervals, c becomes 2 owing to $5 \times c \times (5 + 1)/2 = 30$. Each partition point on the range is calculated by the equation of the arithmetic series. For instance, the first partition point is $1 \times 2 \times (1 + 1)/2 = 2$, and the second partition point is $2 \times 2 \times (2 + 1)/2 = 6$. Other partition points are also calculated in the same manner. If we divide the range by four intervals, c becomes 3 and the respective partition points are 3, 9, and 18 in the same manner. As the range of each dimension in the

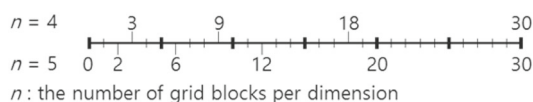


Fig. 2 Example of dividing dimension

d -dimensional data space is divided in the same manner, we can easily find the grid block to which a point belongs using the arithmetic series without an additional overhead such as searching the tree.

The reasons for using the arithmetic sequence are to reduce the calculation cost and avoid oversized grid blocks. If we adopt other methods such as a geometric sequence, we should use exponentiation on floating point to find a grid block to which a point belongs, thereby greatly increasing the calculation cost. For every input point, as we observe for the grid block to which each point belongs, the performance of our proposed algorithm will be considerably degraded if the calculation cost is high. In the case of oversized grid blocks, oversized grid blocks are hardly dominated by other grid blocks, as shown in Fig. 1c. Consequently, we use the arithmetic sequence for preventing these problems.

However, when we consider a dataset that does not contain a point near the origin, the number of dominated grid blocks in the dataset is still few. We propose adaptive Two-Level Grids named TLG . An example of TLG is shown in Fig. 3. Figure 3a shows TLG . Figure 3b, c shows TLG_1 and TLG_2 , which are sub-indices of TLG , respectively.

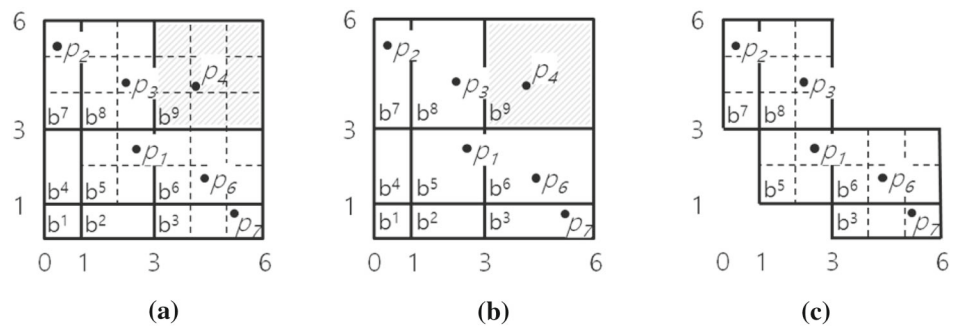
Definition 2.6 The adaptive two-level grids, denoted as TLG , consist of a grid block level grid TLG_1 and a sub-grid block level grid TLG_2 . The sub-grid block in TLG_2 is the same as the smallest grid block in TLG_1 . A grid block $b^i \in TLG_1$ is subdivided into sub-grid blocks if and only if it satisfies the following criteria: (1) b^i that belongs to a point (2) b^i is not dominated by other grid blocks. If b^i is subdivided, we name it as subdivided grid block.

2.2 Memory estimation

If we finely divide the data space into numerous grid blocks to dominate as many grid blocks as possible, we should check dominance relationship between numerous grid blocks and compute skylines among the numerous dominating grid blocks. Hence, the numerous grid blocks cause a higher computation cost. Moreover, they are entirely loaded into main memory to avoid disk I/O. As a result, the number of grid blocks is limited.

Before exhibiting the memory efficiency of TLG , we first describe the definitions and properties of TLG . Given a set of grid block $GBS \subseteq TLG_1$ and grid block $b^i \in GBS$, as a $V(b^i)$ can be divided by the volume of a sub-grid block, $V(GBS)$ can be divided by the volume of the sub-grid block. We represent $V(GBS)$ as the number of sub-grid blocks in GBS as follows:

Fig. 3 Examples of *TLG* **a** *TLG*, **b** grid block level, **c** sub-grid block level



Definition 2.7 We define the number N of sub-grid blocks in a set of grid blocks GBS as follows:

$$N(GBS) = \sum_{b^i \in GBS} \prod_{k=1}^d b_k^i$$

Example Given a set of grid blocks $GBS = \{b^8, b^9\}$ in Fig. 3a, $N(GBS)$ is $\prod_{k=1}^2 b_k^8 + \prod_{k=1}^2 b_k^9 = 2 \times 3 + 3 \times 3 = 15$.

The following two properties of *TLG* present the relationship between the grid blocks. If we can identify a grid block that contains a point is already dominated, then we can also identify that some of the other grid blocks are dominated or empty.

Property 2.1 Given a grid block $b^i \in TLG_1$ containing a point is not dominated, $b^j \in TLG_1$ that satisfies the following criteria are **empty** states:

$$\forall k \in [1, d], b_k^i > b_k^j.$$

Example In Fig. 3a, three grid blocks $b^1, b^5,$ and b^9 are provided. As b^5 containing a point p_1 is not dominated, b^1 is an **empty** state. Also b^9 is dominated by b^5 .

Property 2.2 Two grid blocks b^i and b^j are provided. If b^i dominates b^j , then $N(\{b^i\})$ is invariantly less than $N(\{b^j\})$.

Example As shown in Fig. 3a, two grid blocks b^5 and b^9 are provided. $N(\{b^5\}) = \langle 2, 2 \rangle$ is invariantly less than $N(\{b^9\}) = \langle 3, 3 \rangle$.

As shown in Fig. 3c, only some grid blocks can be subdivided. We show the property that describes the criteria to subdivide a grid block.

Property 2.3 Let $SGBS$ be a set of subdivided grid blocks and $DGBS$ be a set of dominated grid blocks. Provided every grid block $b^u \notin DGBS$ is included in $SGBS$, if and only if it satisfies the following criteria:

- (1) b^u contains a point.
- (2) $\exists k \in [1, d], b_k^u = n \vee (\forall l \in [1, d], b_l^u = b_l^d \vee b_l^u = b_l^d - 1)$ where $b^d \in DGBS$.

Example In Fig. 3a, as only b^9 is dominated by other grid blocks, $SGBS = \{b^3, b^5, b^6, b^7, b^8\}$.

Let $GBS \subseteq TLG_1$ be a set of grid blocks. Lemma 2.1 shows the maximum sum of the numbers of subdivided grid blocks. Using Definition 2.6 and Property 2.1–2.3, we prove Lemma 2.1 as follows.

Lemma 2.1 Let $SGBS_{max} = \{b^j \in TLG_1 | \exists k \in [1, d], b_k^j = n\}$. $N(SGBS)$ is less than or equal to $N(SGBS_{max})$.

Proof To only subdivide the grid blocks of $SGBS_{max}$, all points in a dataset are located into the grid blocks of $SGBS_{max}$ by Property 2.1. If a grid block $b^i \notin SGBS_{max}$ is subdivided, according to Definition 2.6, b^i contains a point. It indicates that b^i dominates at least one grid block $b^j \in SGBS_{max}$ by Property 2.3. $N(\{b^i\})$ is invariantly less than $N(\{b^j\})$ according to the Property 2.2. Thus, $N(SGBS_{max} \cup b^i \setminus b^j)$ is invariantly less than $N(SGBS_{max})$. \square

Example Assuming *TLG* as shown in Fig. 3a, $SGBS_{max} = \{b^3, b^6, b^7, b^8, b^9\}$. As b^5 contains a point p_1 , b^5 subdivides and dominates b^9 (we indicate this by using diagonal pattern). That is, 2×2 sub-grid blocks are generated by Definition 2.6, and 3×3 sub-grid blocks are eliminated, thereby reducing the sum of numbers of sub-grid blocks in the subdivided grid blocks.

Let G_1 be a grid consisting of fixed-size grid blocks. If G_1 consists of $((n + 1)/2)^d$ fixed-size grid blocks, the volumes of dominated grid blocks of G_1 and *TLG* are the same. We compare the memory requirements of two grids *TLG* and G_1 . According to Lemma 2.1, $N(SGBS_{max})$ is Equation 1. The number of grid blocks in TLG_1 is invariantly n^d , so that the total number of grid blocks and sub-grid blocks in *TLG* does not exceed the sum of Equation 1 and n^d . As a result, *TLG* contains less memory requirement than Equation 2. For example, we assume that n is 10 on a two-dimensional data space, and *TLG* requires a main memory for up to 1100 grid blocks, but G_1 requires main memory for 3025 grid blocks. The dataset in this case, however, are extremely skewed. Moreover, the actual amount of memory requirement may be considerably smaller.

$$\left(\frac{n(n+1)}{2}\right)^d - \left(\frac{(n-1)n}{2}\right)^d \tag{1}$$

$$\left(\frac{(n-1)n}{2}\right)^d - n^d \tag{2}$$

3 Algorithm

This section presents the *TLG*-based skyline query processing algorithm on MapReduce framework, called *TLGSL*. Given a point set \mathbb{P} , we compute the global skyline points of \mathbb{P} by running map, combiner, and reduce functions simultaneously. In *TGLSL*, we build *TLG* using the samples of \mathbb{P} . We broadcast *TLG* into all map functions. Next, we prune non-skyline point on each map functions and compute local skyline on each combiner functions independently. Finally, we merge local skylines and compute global skylines on the reduce function simultaneously. *TLGSL* is shown in Algorithm 1.

Before explaining *TGLSL*, we first define filter points. In order to reduce the number of checking dominance relationship between all pairs of points in \mathbb{P} , we locate \mathbb{P} into the grid blocks in *TLG*. Next, we compute the respective skyline for each grid block. Meanwhile, we perform a filtering process for each grid block. The filtering process eliminates the points in b^i by comparing with the points in other grid blocks. We name the points in b^i as candidate points and the points in other grid blocks as filter points. Without loss of generality, we assume that minimum coordinates are preferred for each dimension in this paper.

```

Algorithm 1 TLGSL algorithm
//INPUT:
// point dataset  $\mathbb{P}$ 
//OUTPUT:
// skyline
sample  $\leftarrow \emptyset$ ; TLG  $\leftarrow \emptyset$ ; skyline  $\leftarrow \emptyset$ ;
sample  $\leftarrow$  ReservoirSampling( $\mathbb{P}$ );
TLG  $\leftarrow$  calculateTLG(sample);
setGroupKey(TLG)
Broadcast TLG;
skyline  $\leftarrow$  RunMapReduce( $\mathbb{P}$ )
return skyline;
    
```

Definition 3.1 The filter points of b^i are the unions of points in other grid blocks that satisfy the following conditions:

- (1) $b^j \in \text{TLG}_1$ contains points and is not dominated by other grid blocks.
- (2) $i \neq j$
- (3) $\forall k \in [1, d], b_k^j \leq b_k^i$

A filter point fp^p is represented by $\langle fp_1^p, fp_2^p, \dots, fp_d^p \rangle$ where fp_k^p is the k th dimensional coordinate of fp^p .

For example, p_5 is the filter point of four grid blocks $\{b^7, b^8, b^{10}, b^{14}\}$, as shown in Fig. 4.

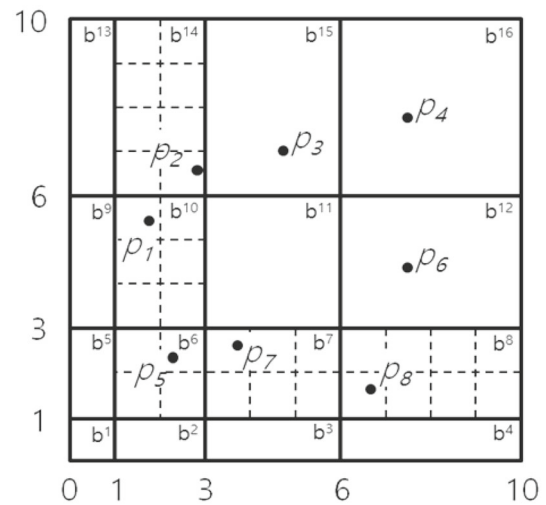


Fig. 4 Example of skyline computation except surface

3.1 TLG construction

In this section, we describe the construction of building *TLG*. We first obtain samples from \mathbb{P} using the Reservoir sampling technique [20]. Next, we divide data space into grid blocks of *TLG* and all samples are inserted into *TLG*. Meanwhile, we identify the dominating grid blocks for pruning all points in the dominated grid blocks. *TLG* based on samples will be sent to all map functions and will be used for pruning non-skyline points in \mathbb{P} on each map function. The setGroupKey function is described in Sect. 3.2.

```

Algorithm 2 The map function of TLGSL algorithm
//INPUT:
// point id  $i$ , a point  $p_i$ 
//OUTPUT:
// virtual id  $vid$ , a point  $p_i$ 
 $b^{gid} \leftarrow$  getGB( $p_i$ );
if ( $b^{gid}$  is unpruned) then
  prune grid blocks based on  $b^{gid}$ ;
  if ( $b^{gid}$  is subdivided) then
     $b^{gid}.sb^g \leftarrow$  getSubGB( $p_i$ );
    if ( $b^{gid}.sb^g$  is unpruned) then
      prune sub-grid blocks based on  $b^{gid}.sb^g$ ;
      Output  $\langle 0, p_i \rangle$ ;
    end if
  else
    Output  $\langle 0, p_i \rangle$ ;
  end if
end if
    
```

3.2 Local skyline computation

The map function of *TLG* is shown in Algorithm 2. The map function is called with point id i as key and a point p_i as value. Next, the map function calls *getGB* function to obtain grid block b^{gid} . The function *getGB* returns the grid block b^{gid} to which p_i belongs. If b^{gid} is not pruned, it can dominate the remaining unpruned grid blocks by b^{gid} . If b^{gid} is sub-

divided, it calls the *getSubGB* function to observe sub-grid block $b^{sid}..sb^s$ to which p_i belongs. If $b^{sid}..sb^s$ is not pruned, it can dominate the remaining unpruned sub-grid blocks by $b^{sid}..sb^s$. If b^{sid} or $b^{sid}..sb^s$ is pruned, the map function does not output anything. Otherwise, it outputs $\langle \text{key}, \text{value} \rangle$ with virtual id 0 as the key and point p_i as the value.

The combiner function runs on the same machine as the map function to decrease the network traffic between the map function and the reduce function. As the combiner function only contains the intermediate results of subset of \mathbb{P} divided by the splitter of MapReduce, it computes the local skyline points to eliminate the non-skyline points as much as possible in advance. To dwindle down the number of comparisons for checking dominance relationship between all pairs of points, we compute the skyline points for each grid block using the filtering process. We first introduce a filtering technique, named *computation except surface*, which uses a novel buffer named “surface”.

Algorithm 3 The combiner function of TLGSL algorithm
<pre> //INPUT: //list of <virtual id vid, a point list P_in> //OUTPUT: //group key gk, ((grid block id i, grid block's surface id surface), a point list P_out) for each (<vid, P_in> in list) do for each (p_p ∈ P_in) do put p_p into TLG; for each (b^i ∈ TLG_i) do // row-major order if (b^i belongs a point) then prune grid blocks based on b^i; tot_filter ← ∅; for each (sf from 1 to d) do filter ← b^i.surface[sf]; b^i ← getSkylineExceptSurface(b^i, filter, sf); tot_filter ← tot_filter ∪ filter; b^i.SL ← Skyline(b^i); Output <b^i.gk, ((i, -1), b^i.SL)>; local_filter ← tot_filter ∪ b^i.SL; for each (b^x ∈ TLG_i) do if (i ≠ x AND b^x is not dominated AND ∀ k ∈ [1, d], b_k^i ≤ b_k^x ≤ b_k^i + 1) then sf ← surface between b^i and b^x; mv_filter ← getFilterExceptSurface(local_filter, sf); put mv_filter into b^x.surface[sf]; for each (b^i ∈ TLG_i) do // row-major order if (b^i belongs a point) then for each (groupk_key from 1 to GK) do if (b^i.gk ≠ groupk_key) then Output <groupk_key, ((i, d + 1), b^i.SL)>; </pre>

Given the grid block b^i and its filter point fp^p , according to Definition 3.1, $\exists k \in [1, d]$, $fp_k^p < L(b_k^i)$. Therefore, it is not essential to compare all the candidate points in b^i with fp^p at the k th dimension. We generate $d+1$ surfaces of each grid block. The surfaces of the grid block contain the following property.

Property 3.1 *The surfaces of the grid blocks consist of a combination of 1 to $d-1$ dimensions. Let nsf be the number*

of dimensions that composes the surface. The nsf number of comparisons is omitted.

Property 3.1 shows that, as the number of dimension increases, the number of surfaces increases exponentially. Assuming a d -dimensional data space, the number of 1-dimensional surface of a grid block is $2 \times d$. As half of all surfaces are associated with its filter points, we use up to the d number of 1-dimensional surface. Therefore, we generate only d surfaces. The surfaces of a grid block are built by an array structure. We maintain $d+1$ surfaces of a grid block. From the 1st to the d th surface maintains filter points that skip comparison at that dimension. $(d+1)$ th surface is used for temporary storage. For example, fp^p where $\exists k \in [1, d]$, $fp_k^p < L(b_k^i)$ is stored into the k th surface of b^i .

As all grid blocks are visited by the row-major order from the 1st to the d th dimension, the grid block visited later is invariantly greater than the grid blocks visited at least one dimension previously. This indicates that the visited grid block does not affect the previously visited grid block. Therefore, we search the entire grid blocks in TLG in a row-major order. For instance, we visit in the order of b^6, b^7, b^8, b^{10} , and b^{14} in Fig. 4. Note that the empty grid blocks are not visited. $\{b^{11}, b^{12}, b^{15}, b^{16}\}$ are dominated by b^6 when we visited b^6 .

When we visit the grid block b^i , we only send the local skyline points of b^i to every non-visited grid block b^x that reaches b^i . If the grid block b^m , which does not reach b^i , is affected by the points in b^i , then b^m received the points of b^i from the grid blocks that are in between b^i and b^m .

For instance, in Fig. 4, as p_2 in b^{14} is dominated by both of p_5 in b^6 and p_1 in b^{10} , b^{14} requires p_1 or p_5 . When visiting b^6 , we compute the skyline points and send the skyline point p_5 to b^{10} . Next, when visiting b^{10} , we eliminate the candidate points by comparing with p_5 . We compute the skyline points and determine the filter points to send to b^{14} . Meanwhile, we use the *computation except surface* technique for calculating the filter points. As we do not consider the 2nd dimension, p_1 is the filter point. We send p_1 to b^{14} . Finally, when visiting b^{14} , p_2 is dominated by the received filter point p_1 .

The combiner function is shown in Algorithm 3. The combiner function is called with virtual id 0 as key and point list \mathbb{P}_{in} as value. The combiner function locates all the points in \mathbb{P}_{in} into TLG. The combiner function searches the entire grid blocks in a row-major order and finds the grid block b^i that holds the points. Meanwhile, some remaining grid blocks can be dominated by b^i . Next, for each surface, the combiner function obtains the filter points *filter* from a surface of b^i and calls *getSkylineExceptSurface* function for eliminating the points of b^i by comparing with *filter*. Note that the *getSkylineExceptSurface* functions use the *computation except surface* technique to skip comparison at the sf th dimension. The combiner function calls the *Skyline* function to obtain

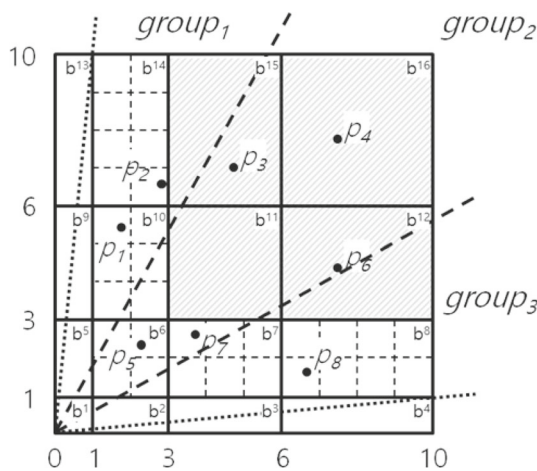


Fig. 5 Example of grouping grid blocks

the local skyline points. The local skyline points are used as the output of the combiner function with the surface id -1 . It will be used as the candidate points of reduce function.

To check dominance relationship between the points in different grid blocks, we send a novel filter point mv_filter of b^i to the surface of b^x , which is not dominated and reaches $b^i \cdot b^x$ has also not been visited yet. The mv_filter is obtained by using the $getSkylineExceptSurface$ function. This function takes the union of the local skyline points and all the filter points of b^i and returns the filter points that are calculated by skipping the comparison at the sf th dimension.

For every grid block b^i , if the reduce function is called with b^i and its candidate points, then every reduce function requires all the filter points of b^i by Definition 3.1. It indicates that the same filter points can be duplicated. For instance, in Fig. 5, assume p_1-p_8 are points and the grid blocks filled by diagonal pattern are pruned by samples. According to Definition 3.1, a point p_5 is sent to five grid blocks b^6, b^7, b^8, b^{10} , and b^{14} . That is, p_5 duplicates five times. As the number of filter points increases, the network cost increases, which is a prime factor in degrading the overall performance.

To decrease the number of duplication of the filter points, we separate the entire grid blocks into groups. After constructing TLG , we use the samples for pruning the grid blocks that do not contain skyline points. Next, we separate the unpruned grid blocks into arbitrary gk groups by using the angular partitioning approach [21]. Let the virtual center point $vcp = \left\langle \frac{U(b'_1)-L(b'_1)}{2}, \frac{U(b'_2)-L(b'_2)}{2}, \dots, \frac{U(b'_d)-L(b'_d)}{2} \right\rangle$. For every grid block b^i , if the vcp of b^i belongs to a group, then assign b^i to the group.

The reason to use samples for grouping is due to the workload balancing of the reduce function. As there is less chance to be many real points in the grid blocks that do not have samples, workload balancing in the reduce function can be enhanced by reducing the influence of these grid blocks. We

set the group key of the grid blocks on Algorithm 1 via the $setGroupKey$ function.

```

Algorithm 4 The reduce function of TLGSL algorithm
//INPUT:
// group key  $gk$ ,  $\langle$ grid block id  $gid$ , grid block's surface id  $surf$  $\rangle$ , a point
list  $\mathbb{P}_{in}$ -list
//OUTPUT:
// skyline
for each ( $p_p \in \mathbb{P}_{in}$ ) do
  if ( $surf = -1$ ) then
    put  $p_p$  into  $b^{gid}$ ;
  else
    put  $p_p$  into  $b^{gid}.surface[surf]$ ;
  end if
for each ( $b^i \in TLG_i$ ) do // row-major order
  if ( $b^i.surface[d+1]$  belongs a filter point) then
    prune grid blocks based on  $b^i$ ;
     $filter \leftarrow b^i.surface[d+1]$ ;
    for each ( $b^x \in TLG_i$ ) do
      if ( $i \neq x$  AND  $\forall k \in [1,d], b_k^i \leq b_k^x \leq b_k^i + 1$ ) then
         $sf \leftarrow$  surface between  $b^i$  and  $b^x$ ;
         $mv\_filter \leftarrow getFilterExceptSurface(filter, sf)$ ;
        put  $mv\_filter$  into  $b^x.surface[sf]$ ;
    if ( $b^i$  is not pruned by samples
    AND  $b^i$  has candidate or filter points) then
      prune grid blocks based on  $b^i$ ;
       $tot\_filter \leftarrow \emptyset$ ;
      for each ( $sf$  from 1 to  $d$ ) do
         $filter \leftarrow b^i.surface[sf]$ ;
         $b^i \leftarrow getSkylineExceptSurface(b^i, filter, sf)$ ;
         $tot\_filter \leftarrow tot\_filter \cup filter$ ;
       $SL \leftarrow Skyline(b^i)$ ;
      Output  $\langle 0, SL \rangle$ ;
       $local\_filter \leftarrow tot\_filter \cup SL$ ;
      for each ( $b^x \in TLG_i$ ) do
        if ( $i \neq x$  AND  $\forall k \in [1,d], b_k^i \leq b_k^x \leq b_k^i + 1$ ) then
           $sf \leftarrow$  surface between  $b^i$  and  $b^x$ ;
           $mv\_filter \leftarrow getFilterExceptSurface(local\_filter, sf)$ ;
          put  $mv\_filter$  into  $b^x.surface[sf]$ ;
  
```

For example, when the number of groups is 3 ($gk = 3$), as there are no points in $b^1, b^2, b^3, b^4, b^5, b^9$, and b^{13} in Fig. 5, the remaining grid blocks (indicated by short dotted lines) are divided by 3 (indicated by long dotted lines). Then, the grid blocks are allocated based only on the long dotted line. Therefore, $b^5, b^9, b^{10}, b^{13}, b^{14}$, and b^{15} correspond to $group_1$, b^1, b^6, b^{11} , and b^{16} correspond to $group_2$. Note that the grid blocks that do not contain samples are also divided.

We visit the entire grid blocks in a row-major order and find the grid block b^i containing points. When visiting b^i , all the local skyline points of b^i are sent to other groups as filter points. To indicate that they are a filter point, we set the surface id of the filter point to $d+1$.

3.3 Global skyline computation

The reduce function is shown in Algorithm 4. The reduce function is called with group key gk as key and a point list \mathbb{P}_{in} as value. \mathbb{P}_{in} consists of filter points and candidate points.

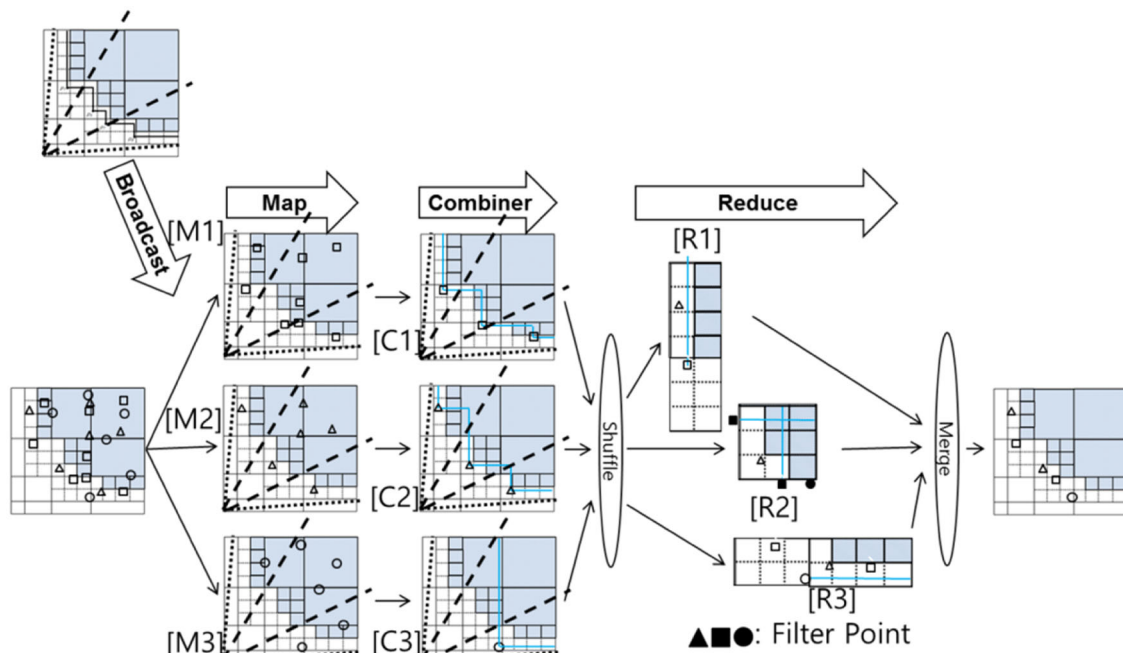


Fig. 6 Example of TLGSL algorithm

If surface id sf of each point $p_p \in \mathbb{P}_{in}$ is -1 , point is inserted into b^{sid} as candidate points. Otherwise, the point is inserted into b^{sid} . $surface[surf]$ as filter points. Next, we visit the entire grid blocks in the row-major order and find the grid block b^i containing filter points. The filter points are transmitted to the grid blocks that reach b^i . The rest of the lines are similar to the combiner function. Therefore, we only mention the difference between them. Sending filter points to the reduce function is excluded. We visit b^i containing filter points even if it does not contain candidate points.

Example Let M1, M2, and M3 be the map function, C1, C2, and C3 be the combiner function, and R1, R2, and R3 be the reduce function, as shown in Fig. 6.

First, MapReduce randomly selects the samples from \mathbb{P} using the reservoir sampling and builds TLG using the samples. The grid blocks filled with color are pruned by samples. Every grid blocks are divided by the angular partitioning approach. MapReduce broadcasts TLG into all the map functions.

M1, M2, and M3 accept points represented by square, triangle, and circle, respectively. M1 is independently called with each square point. M1 obtains the grid block b^i and $b^i.sb^s$ by calling $getGB$ and $getSubGB$ to which the square point belongs, and then checks whether b^i and $b^i.sb^s$ are pruned. If they are pruned, we omit the square point. Therefore, the points in the grid blocks and sub-grid blocks filled with color are eliminated. We perform this operation for all mappers in the same manner. Moreover, if grid blocks or sub-grid blocks is dominated by each input point, we dominate them.

If the point of each map function is not eliminated, it is sent to the combiner function. C1 computes the local skyline points for each grid block. At this point, we visit all the grid blocks in the row-major order and compute the local skyline points using the filtering process with the *computation except surface* technique. C1 sends the local skyline points to the grid block in the same group. The transmitted points become the candidate points in the reduce function.

The C1 computes the filter points among the local skyline points and send them to all the other groups. In Fig. 6, the filter points on the left and lower surfaces of R2 are filter points received from the combiner C1 and C3, respectively. Performing the same in the remaining combiner functions, the filter points and candidate points are sent to R1–R3 as input.

The filter points and candidate points are input to the reduce function. The filter points obtained by the surfaces of grid block are used for eliminating the non-skyline points. In Fig. 6, the lines starting at the filter points represent the filtering process. Finally, we compute the global skyline points among the remaining points. All reducer functions such as R1, R2, and R3 are independently called with group key and MapReduce merges the global skyline points of each reduce function and returns them.

4 Performance evaluation

In this section, we empirically evaluated the performance of the proposed algorithm using the parameters shown in

Table 1 Parameter

Parameter	Description	Range
s	The number of samples	400
D	The number of dimensions	2–5
$ \mathbb{P} $	The number of points	$2 \times 10^6 - 10^7$
n	The number of grid blocks per dimension	5–12
gk	The number of groups	3

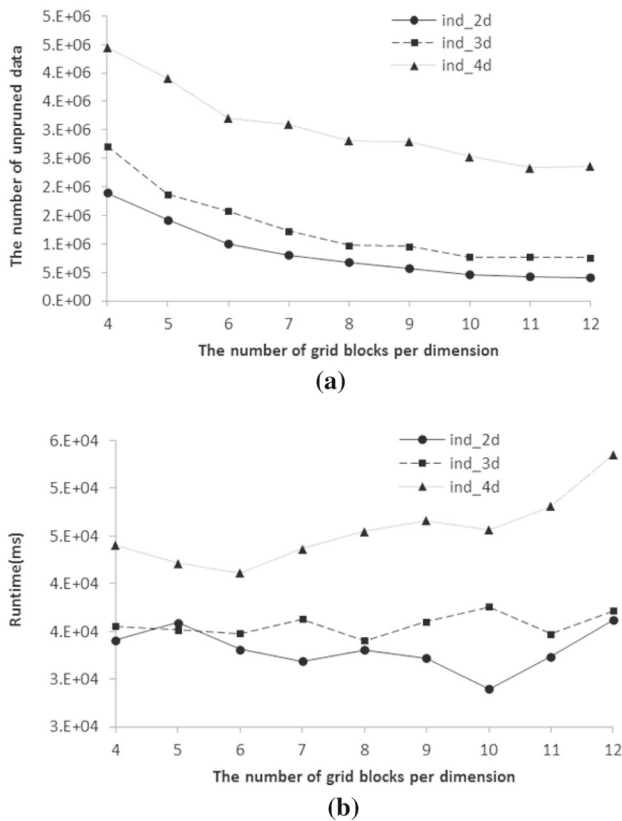


Fig. 7 Effect of the number of grid blocks per dimension for the datasets with independent data distribution **a** unpruned data, **b** runtime

Table 1. To perform all experiments, a total of nine Mac machines were used for configuring a cluster. The cluster consisted of one master and eight slaves nodes of Intel(R) Core(TM) i5-2500S CPU 2.70GHz processor with 16GB (Giga bytes) main memory. We compared our proposed algorithm *TLGSL* with SKYMR [19]. We used javac 1.8 for compiling all the implemented algorithms. The framework used was Hadoop 2.4.1 on Mac OS X Sierra 10.12.3. We generated two synthetic datasets that are commonly used for evaluating the performance of skyline algorithms [1]. The respective datasets with independent and anti-correlated data distributions were randomly generated.

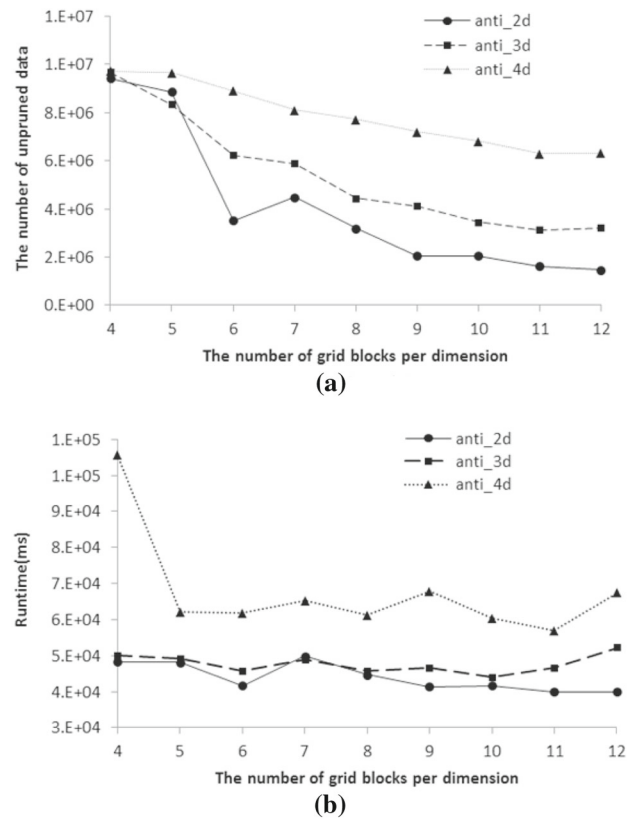


Fig. 8 Effect of the number of grid blocks per dimension for the dataset with the anti-correlated data distribution **a** unpruned data, **b** runtime

4.1 Effect of the number of grid blocks per dimension

We first evaluated the effect of *TLGSL* by varying n (i.e., the number of grid blocks per dimension). We used 400 samples. Let d be the number of dimensions. We ran our experiments on 2–4 dimensional datasets of size 10^7 by increasing n from 4 to 12.

Figure 7 shows the effects of n for 2–4 dimensional datasets *ind_2d*, *ind_3d*, and *ind_4d* that are the datasets with the independent data distribution. The number of unpruned data was decreasing with increasing n in Fig. 7a, because further dividing the data space increased the pruning power. However, the runtime did not always decrease, even if the pruning power increased. This is because the number of grid blocks increased with increasing n . For large number of grid blocks, the cost of checking dominance relationship between the grid blocks and the cost of traversing the unpruned grid blocks become considerably high. It adversely affected the overall performance. In Fig. 7b, the values of n with the lowest runtimes were 10 for 2D, 8 for 3D, and 6 for 4D. Since *TLG₁* requires n^d grid blocks, the number of grid blocks became larger with increasing dimensionality, which in turn increases computational costs.

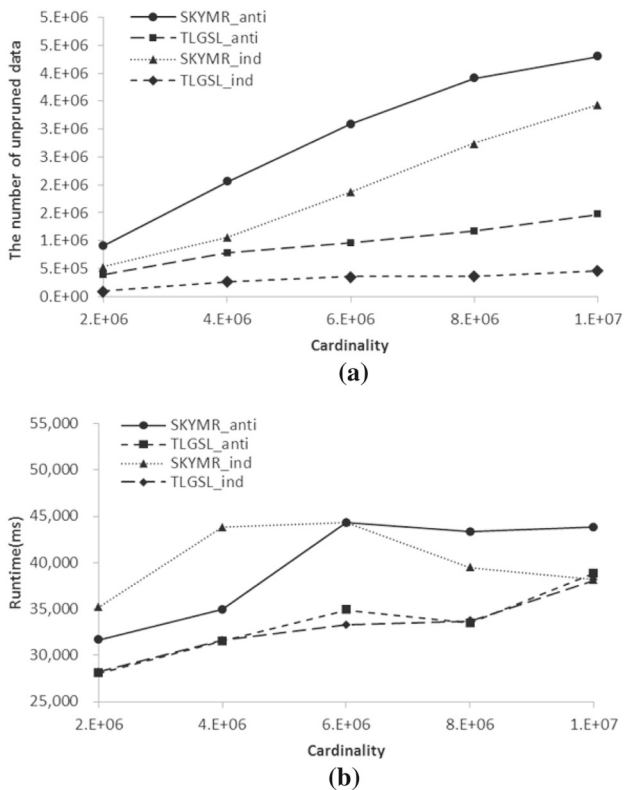


Fig. 9 Effect of cardinality for a two-dimensional dataset **a** unpruned data, **b** runtime

Figure 8 shows the experimental results for 2–4 dimensional datasets *anti_2d*, *anti_3d*, and *anti_4d*, which are the datasets with the anti-correlated data distribution. The results of Fig. 8 were similar to those of the experiment in Fig. 7.

4.2 Effect of data cardinality

In this section, we evaluated the effect of data cardinality on MapReduce skyline algorithms. 400 samples were extracted and tested for both algorithms. Other parameters of *SKYMR* used the values they used in the experiment [19]. That is, the split threshold was 20, the number of machines was 50, the buffer size was 500,000, and the maximum depth of the quad-tree was 50. Let n be the number of grid blocks per dimension. We set n as 10 for a two-dimensional dataset and 8 for a three-dimensional dataset. The respective *SKYMR_ind* and *TLGSL_ind* are experimental results of *SKYMR* and *TLGSL* for dataset with independent data distribution. Likewise, The Respective *SKYMR_anti* and *TLGSL_anti* are experimental results of *SKYMR* and *TLGSL* for dataset with anti-correlated data distribution.

Figure 9 shows the effect of data cardinality for the two-dimensional dataset. Figure 9a shows the number of unpruned data, when all the map functions are completed. Moreover, Fig. 9b shows the total runtime including sam-

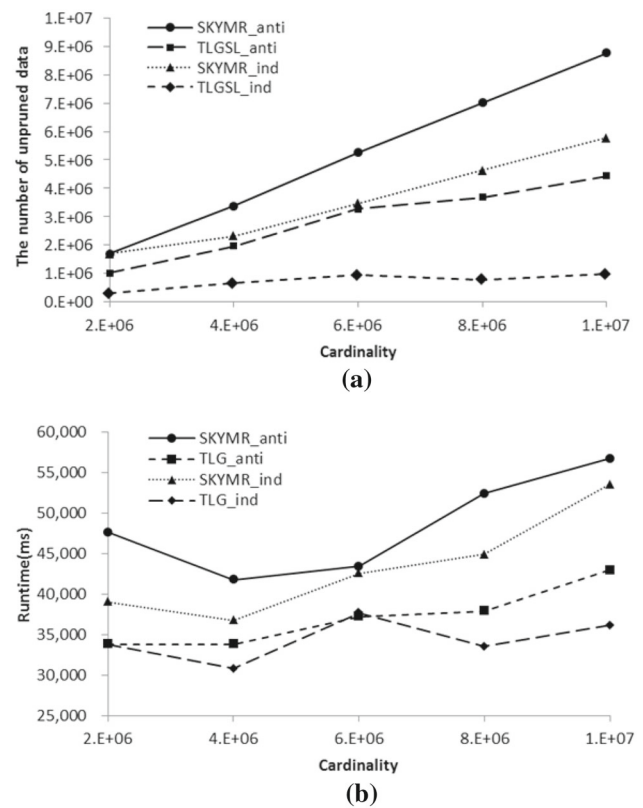


Fig. 10 Effect of cardinality for a three-dimensional dataset **a** unpruned data, **b** runtime

pling, building *TLG*, and broadcasting *TLG* in milliseconds. *TLGSL* effectively prunes non-skyline points. In MapReduce, the intermediate data as a result of the map functions are stored into a machine on which the map function runs and are transmitted to the reduce function through the network. Therefore, it is advantageous to eliminate as much data as possible on the map and combiner functions, so that the total execution time was shorter in Fig. 9b.

Figure 10 shows the same experiment performed for the three-dimensional dataset. Figure 10a, b shows results similar to the above experiment. However, as the number of dimension increases, the number of pruned data sharply decreases for the dataset with the anti-correlated data distribution. We have conducted similar experiments and will discuss an effect of dimensionality in Sect. 4.3.

4.3 Effect of dimensionality

Finally, we evaluated the effect of dimensionality d on MapReduce skyline algorithms. We ran experiments with d from 2 to 5. We set n as 10 for 2D, 8 for 3D, 6 for 4D, and 5 for 5D. We fixed the cardinality at 10^7 . The results are shown in Fig. 11.

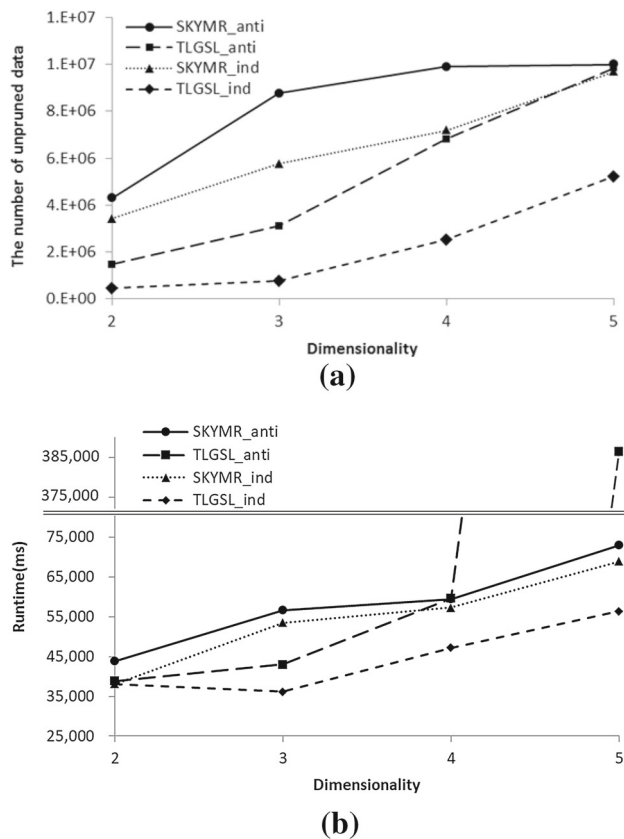


Fig. 11 Effect of dimensionality **a** unpruned data, **b** runtime

As *TLGSL* effectively pruned non-skyline points rather than *SKYMR* for the dataset with independent data distribution, it had a steady performance up to five-dimensions, as shown in Fig. 11b. On the other hand, we could see that the number of pruned points sharply decreases as the number of dimensions increases on the anti-correlated distribution, as shown in Fig. 11a. This is because the volume of the data space exponentially increases with increase in dimensionality. If we divide the data space by the number of grid blocks used in the lower dimensionality, not only the volume of each grid block increases considerably but also the number of pruned grid blocks can be reduced considerably.

On the other side, if we finely divide the entire data space by the volume we used on the lower dimensionality such as 2–4 dimensions, it requires a considerable number of grid blocks. However, it is not desirable to be divided into more than the certain number of grid blocks owing to the physical memory limit and other computation overhead, as discussed earlier. Finally, the data space should be divided into grid blocks, but the number of grid blocks should not exceed a certain number. Hence, the number of pruned grid blocks is reduced.

In this manner, as the anti-correlated distribution has no point close to the origin, the dominance relationship between

grid blocks did not occur most of the time. As a result, we can understand that our algorithms cannot effectively prune non-skyline points when the data space is not finely divided.

Above discussed problems lead to performance degradation for the dataset with the anti-correlated data distribution, as shown in Fig. 11b. Subsequently, we should compute skyline points among a large amount of unpruned points through searching the entire grid blocks in a row-major order. Therefore, the performance of *TLG* was deteriorated when the dimensionality was greater than 5 for the dataset with the anti-correlated data distribution.

On the other hand, the *SKYMR* algorithm partitions data space into balanced nodes that reflect the data distribution using quad-tree. As 400 samples are used, the data space is not divided into numerous nodes. Therefore, unlike *TLG*, the cost to check dominance relationship between nodes is low. Through the above experiment, although the proposed algorithm exhibited poor performance for the dataset of anti-correlated data distribution, it showed prominent performance for the dataset of independent data distribution. Our proposed algorithm also exhibited prominent performance at the low-dimensional dataset, for which it is not essential to divide the entire data space finely.

5 Conclusion

We proposed a *TLG-based* skyline query algorithm with MapReduce, called *TLGSL*. *TLG* is fabricated using variable-size grid blocks that increase its interval in the order of arithmetic sequence in each dimension and fixed-size grid blocks. The *TLG* achieved the same pruning power as a grid consisting of a fixed-size grid block while using less main memory, because *TLG* utilized the dominance power of the skyline. Moreover, we can rapidly find the grid block to which the points belong for pruning non-skyline points. The skyline points are expeditiously computed for each grid block using the filtering process with skyline *computation except surface* technique. In order to decrease network costs, we applied the existing angular partitioning approach-based grouping technique. Finally, we obtained the conclusion from the performance evaluation. Our proposed algorithm shows prominent performance on a low-dimensional data space, wherein it is not essential to divide the data space finely.

Acknowledgements This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2017-2017-0-01628) supervised by the IITP (Institute for Information & communications Technology Promotion) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2015R1D1A1A01058001).

References

- Borzsony, S., Kossmann, D., Stocker, K.: The skyline operator. In: 17th IEEE International Conference on Data Engineering, pp. 421–430 (2001)
- Böhm, C., Fiedler, F., Oswald, A., Plant, C., Wackersreuther, B.: Probabilistic skyline queries. In: 18th ACM Conference on Information and Knowledge Management, pp. 651–660 (2009)
- Lee, W., Eom, C.S.H., Jo, T.C.: Path skyline for moving objects. In: 14th Asia-Pacific Web Conference, pp. 610–617 (2012)
- Lee, J., Hwang, S.W., Nie, Z., Wen, J.R.: Navigation system for product search. In: 26th IEEE International Conference on Data Engineering, pp. 1113–1116 (2010)
- Lappas, T., Gunopulos, D.: Efficient confident search in large review corpora. In: Machine Learning and Knowledge Discovery in Databases, pp. 195–210 (2010)
- Alrifai, M., Skoutas, D., Risse, T.: Selecting skyline services for QoS-based web service composition. In: 19th International Conference on World Wide Web, pp. 11–20 (2010)
- Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: an online algorithm for skyline queries. In: 28th International Conference on Very Large Data Bases, pp. 275–286 (2002)
- Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: 2003 ACM SIGMOD International Conference on Management of data, pp. 467–478 (2003)
- Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient progressive skyline computation. In: 27th International Conference on Very Large Data Bases, pp. 301–310 (2001)
- Comer, D.: The ubiquitous b-tree. *ACM Comput. Surv.* **11**(2), 121–137 (1979)
- Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: 1990 ACM SIGMOD International Conference on Management of Data, pp. 322–331 (1990)
- Data, data everywhere. *The Economist*. <http://www.economist.com/node/15557443> (2010). Accessed 9 June 2017
- Beaver, D., Kumar, S., Li, H.C., Sobel, J., Vajgel, P.: Finding a needle in haystack: Facebook's photo storage. In: 9th USENIX Conference on Operating Systems Design and Implementation, vol. 10, pp. 1–8 (2010)
- Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. *Commun. ACM* **51**(1), 107–113 (2008)
- Lee, S., Kim, J., Moon, Y.S., Lee, W.: Efficient level-based top-down data cube computation using MapReduce. *Trans. Large-Scale Data Knowl. Centered Syst.* **XXI**, 1–19 (2015)
- Lim, Y., Choi, E.: Research on Map-Reduce mechanism for time series big data processing and analysis. *J. Inf. Technol. Archit.* **12**(1), 91–98 (2015)
- Zhang, B., Zhou, S., Guan, J.: Adapting skyline computation to the MapReduce framework: algorithms and experiments. In: 16th International Conference on Database Systems for Advanced Applications, pp. 403–414 (2011)
- Eldawy, A., Li, Y., Mokbel, M.F., Janardan, R.: CG_Hadoop: computational geometry in MapReduce. In: 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 294–303 (2013)
- Park, Y., Min, J.K., Shim, K.: Parallel computation of skyline and reverse skyline queries using mapreduce. *Proc. VLDB Endow.* **6**(14), 2002–2013 (2013)
- Vitter, J.S.: Random sampling with a reservoir. *ACM Trans. Math. Softw. (TOMS)* **11**(1), 37–57 (1985)
- Chen, L., Hwang, K., Wu, J.: MapReduce skyline query processing with a new angular partitioning approach. In: 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pp. 2262–2270 (2012)



Hyeong-Cheol Ryu received the B.S. degree in Computer Science from Hanshin University, Korea, in 2009 and the M.S. degree in Computer Science and Engineering from Sogang University, Korea, in 2014. He is currently a Ph.D. candidate in Computer Science and Engineering Department at Sogang University. His research interests include spatial databases and data mining.



Sungwon Jung received the B.S. degree in Computer Science from Sogang University, Seoul, Korea in 1988. He received the M.S. and Ph.D. degrees in Computer Science from Michigan State University, East Lansing, Michigan 1990 and 1995, respectively. He is currently a professor in the Computer Science and Engineering Department at Sogang University. His research interests include spatial and mobile databases, data mining, and blockchain technology.