LShape Partitioning: Parallel Skyline Query Processing using *MapReduce*

Heri Wijayanto, Wenlu Wang, Wei-Shinn Ku, and Arbee L.P. Chen*

Abstract—A skyline query searches the data points that are not dominated by others in the dataset. It is widely adopted for many applications which require multi-criteria decision making. However, skyline query processing is considerably time-consuming for a high-dimensional large scale dataset. Parallel computing techniques are therefore needed to address this challenge, among which *MapReduce* is one of the most popular frameworks to process big data. A great number of efficient *MapReduce* skyline algorithms have been proposed in the literature and most of their designs focus on partitioning and pruning the given dataset. However, there are still opportunities for further parallelism. In this study, we propose two parallel skyline processing algorithms using a novel *LShape* partitioning strategy and an effective *Propagation Filtering* method. These two algorithms are 2*Phase LShape* and 1*Phase LShape*, used for multiple reducers and single reducer, respectively. By extensive experiments, we verify that our algorithms outperformed the state-of-the-art approaches, especially for high-dimensional large scale datasets.

Index Terms—skyline query, parallel computing, partitioning strategy, MapReduce

1 Introduction

Given a large number of data points, a skyline query searches for the data points that are not dominated by others. A data point dominates another if it is as good or better in all attributes and better in at least one attribute [1]. The set of data points that are not dominated by other data points is called a skyline. Furthermore, the process to search the skyline is named skyline queries. For example, a customer wants to choose one hotel from many hotels that are available near a beach. Each hotel has two attributes: its distance to the beach, and its price. A hotel is dominant if no other hotel has a smaller distance to the beach nor a price equal to or smaller than its price. As shown in Fig. 1, the skyline data points are t_1, t_2, t_7, t_8, and t_{12}.

H. Wijayanto is with Department of Computer Science and Information Engineering, Asia University, No. 500, Lioufeng Rd., Wufeng, Taichung, Taiwan 41354, ROC, and Informatics Study Program, Engineering Faculty, Mataram University, No. 62, Majapahit Rd., Mataram, Indonesia 83115, EMAIL: heri@unram.ac.id.
W. Wang is with Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849, USA. EMAIL: wenluwang@auburn.edu.
W.S. Ku is with Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849, USA. EMAIL: weishinn@gmail.com.
A.L.P. Chen is with Department of Computer Science and Information Engineering, Asia University, No. 500, Lioufeng Rd., Wufeng, Taichung, Taiwan 41354, ROC. EMAIL: arbee@asia.edu.tw.
** corresponding author*
Manuscript received XXX XX, XXXX; revised XXX XX, XXXX.

The skyline query is a popular approach to derive valuable information in big data. Specifically, skyline queries provide an effective mechanism for multi-criteria decision making [1], wireless sensor networks [2], and product recommendation [3]–[5]. Moreover, it draws the attention of researchers to discover efficient algorithms since skyline queries are computationally intensive in large scale and high dimensional datasets.

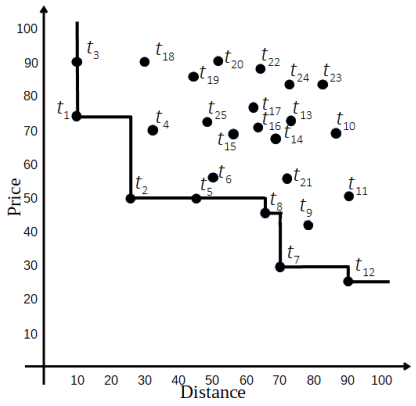


Fig. 1. Example Skyline

Since first introduced in 2001, skyline problems [1] have been extensively studied in the literature. Börzsonyi S. et al., [1] proposed *Block Nested Loops* (*BNL*) algorithm to determine the skyline data points in a large dataset that does not fit in the memory. This approach was designed for a single computer machine. It utilizes a *window* in the main memory and a temporary file in the disk to store incomparable data points. The *divide and conquer* algorithm of *BNL* approach was also proposed and has become the fundamental technique that is extended to the MapReduce approach today. Basically, it divides the data space into grids. Each grid computes skyline data points individually then all grids results are merged together to calculate the final result. Researchers also introduced varied algorithms of skyline queries processing such as *Bitmap* [6], *Sort Filter Skyline* (*SFS*) [7], and MBR-Oriented approach [8]. The skyline queries have several variants such as *dynamic skyline* [9], *reverse skyline* [10], *uncertain skyline* [11], *continuous skyline* [12], and *spatial skyline* [13].

When the number of skyline points is large, parallel skyline evaluation still has an important meaning for mak-

ing decisions such as in the market analysis based on the resulting skyline points. Based on our best knowledge, this problem was first introduced in 2007 by Lin, Yuan, Zhang, and Zhang, [14]. Methods were then proposed to select k of the skyline points as the most representative skyline points to make a decision. Following this work, we [3] proposed to select the most k-demanding products from all skyline products, which satisfy most customer preferences. In the most recent study, Zhou, Li, Yang, and Li [15] find the optimal combinations of skyline products for the marketing strategy.

For real-world skyline applications, it is highly possible that the skyline operator has to deal with a large-size input. The skyline operator is able to refine the data to be used in the subsequent analysis. Efficient algorithms and powerful computing machines play a significant role in today's big data era. Due to the increasing size and complexity of data, centralized approaches are no longer appropriate to compute skyline queries. As mentioned in [16], the centralized approach of skyline processing is not appropriate for a large amount of input data because the execution time increases significantly (up to quadratic) when the amount of input data increases. Our experiments also show this drawback of the centralized approaches. Therefore, researchers tend to develop parallel algorithms for skyline processing. Currently, *MapReduce* framework has become the de-facto standard for parallel processing [17]. Generally, the *MapReduce* framework has two phases: map phase and reduce phase. First, data are divided into partitions that are accessed and managed separately in mapper machines. The second phase is the reduce phase that merges the outputs of map phase to an integrated output.

A large number of *MapReduce* skyline algorithms have been published [18]–[26]. Calculating skyline queries in a parallel manner has several challenges: data partitioning, data pruning, data transferring overhead, and load balancing. Improper partitioning technique would inevitably cause ineffective pruning and unbalanced work load. Also, the data transfer among computer nodes becomes the major challenge since the communication cost is expensive in parallel processing. Therefore, it is worth the effort to develop an enhanced parallel algorithm to search skyline data points in a large dataset based on MapReduce framework.

Our contributions can be summarized as follows:

- We propose an *LShape* partitioning strategy for skyline query processing that is designed for *MapReduce* framework. In our merge phase, most of the local skyline points are identified as the global skyline points without extra computation, a sequential merge of $O(N^2)$ (N is the size of the union of the local skyline points) is avoided; instead, we only need a light merge of $O(n^2)$ ($n << N$, n is the number of data points in shared data cells).
- We devise a *Propagation Filtering*. Taking advantage of the properties of *LShape* partitions, as the filtering object we use skyline points that are found in the locations that prune most of the non-skyline points. Especially for high-dimensional large size input, our filtering strategy is able to avoid more expensive dominance tests, and is, thus, more effective in reducing computational cost.
- We evaluate the performance of the proposed approaches by extensive experiments. Our experiments show that the number of duplicating cells for multiple reducers of the grid partitioning approaches grows exponentially when the number of dimensions increases. Our two-phase *MapReduce* skyline algorithm named *2Phase LShape* also suffers from this problem. Therefore, we propose another one-phase *MapReduce* skyline algorithm named *1Phase LShape* to avoid this problem.

The rest of this paper is organized as follows. In Section 2 we discuss the related works that show the literature survey of the *MapReduce* skyline queries processing. We present the preliminaries and our proposed algorithms in Section 3. The results of experiments and analysis are presented in Section 4. Future works and conclusion of this study are provided in Section 5.

2 Related Work

The general approaches of implementing *MapReduce* to the skyline queries could be classified into two categories. The first category divides the data into partitions based on the grid that divides each dimension into several parts. The second approach is angle-based partitioning which divides the data space into several parts by using vectors which contain angle and magnitude. These vectors start from the origin and divide the data space based on the angle of the vectors.

The first implementation of *MapReduce* for skyline queries was published by B. Zhang et al., [18] in 2011. These algorithms are called by *MR-BNL*, *MR-SFS*, and *MR-Bitmap*. Those are the *MapReduce* implementations of the early centralized algorithms that are *BNL* [1], *SFS* [7], and *Bitmap* [6]. The first *MapReduce* grid-based partitioning was also presented in this work. It is a *MapReduce* implementation of *divide and conqueror* in [1]. However, this approach lacks strong pruning strategies that are mainly proposed in the next approaches. Besides that, the bottleneck problem on the reducing phase was identified in this study.

The next early *MapReduce* skyline algorithm was published by L. Chen et al., [19] in 2012. It introduced the angular partitioning approach. The first step is to translate the *Cartesian* coordinate into a hyperspherical space. Then, it separates the data space into sectors according to the angular coordinates. Next, each sector performs the *BNL* algorithm [1] in the mapper machine to find the local skyline data points. Finally, the reducer merges all skyline data points to determine the global skyline data points.

MapReduce Grid Partitioning based Multiple Reducers Skyline (*MR-GPMRS*) was proposed by K. Mullesgaard et al., [20] in 2014. This approach utilizes many reducer machines to avoid the bottleneck of the reducing phase. Actually, K. Mullesgaard et al., proposed two algorithms in [20]. Those are *MapReduce Grid Partitioning based on Single Reducers Skyline* (*MR-GPSRS*) and *MR-GPMRS* mentioned above. These two approaches consist of two *MapReduce* phases. In the first phase, *MapReduce* approach is used to construct *bitstring* where each bit "1" in the string represents a non-empty partition and bit "0" for empty partition respectively. The *bitstring* is constructed in column-major or row-major

order. It will be used in the second phase to prune dominated data points and calculate skyline data points. In the second phase, the *bitstring* is broadcasted to all mapper machines. Then, the mapper machines utilize the *bitstring* to determine whether a data point is pruned or kept for the local skyline computation. Based on our experiments in evaluating the number of partitions, this approach suffers from exploding the we called duplicated cells for high dimensional data.

In 2016, J. Zhang et al., [21] proposed a *MapReduce* algorithm that is called *Proportional Partition-Aware Filtering Partial-Presort Grid-Based Partition Skyline* (*PPF-PGPS*). It is a two-phase *MapReduce* algorithm for a large dataset that utilizes both Angle-Based partitioning and Grid-Based partitioning. First, they use Angle-Based partitioning to filter the dataset and split the dataset based on the angle of data points. In the second phase, each partition is processed in a mapper machine which performs Grid-Based partitioning to calculate the local skyline data points. Then, the outputs of mappers are combined in a reducer machine to compute the global skyline. In addition, it sorts the grids to find skyline data points in the second phase. However, according to our experiments, the filtering approach of the first phase is not effective for anti-correlated data.

Y. Park et al., [22] proposed an approach that is based on the *Quadtree* data structure in 2013 called *SKY-MR*. This approach has better load balancing because the number of data points in each partition is expected to be the same. The weakness of this this approach is that it needs sample data points to construct the *Quadtree* rapidly. The *Sky-MR+* algorithm is an improvement of *Sky-MR* by the same authors and was published in 2017 [23]. They improved the pruning technique by dominance power filtering and improved the load balancing. *Sky-MR+* has a different method to construct the *Quadtree*. It still needs samples to construct the *Quadtree*. The performance of these two algorithms depends on taking the sample data points to construct *Quadtree*.

J. L. Koh et al., introduced *MR-Sketch* algorithm [24] in 2017. Similar to *MR-GPMRS* [20], it is a multi-reducers approach to address the reducer bottleneck problem. It performs distributed dominance test to improve the performance of *MR-GPMRS*. However, this approach still suffers from communication overwhelm because mappers produce duplicated local skyline sets and send them to several reducers. In other words, the distributed dominance test or *DDT* replicates a partition to several reducer machines. It is not efficient because it overwhelms transmission media that are expensive in cluster computing. *MR-Sketch* also performs sampling of the data in the first phase. Consequently, if the random sampling does not take the proper samples then the performance of this algorithm will be decreased.

The recent algorithm of *MapReduce* skyline queries processing was published in 2018 by M. Tang et al., [25]. It utilizes *Z-Order* notation of data points. This approach resists data skew and data straggler. The data skew is a condition where the partition of the data is unbalanced. It becomes a problem in parallel processing because the execution time depends on the largest partition. The data straggler is a condition where the number of skyline data points in a partition is much larger than the others which need the longest execution time. The problem in this approach is the *Z-Order* that is only suitable for integer dataset. Besides that, this approach needs sampling data to construct partitions. It could fail if the sampling step does not take the proper sample. Similar to *Sky-MR*, and *Sky-MR+*, this algorithm performs the reservoir sampling technique that is the extension of random sampling.

Many efficient parallel skyline processing algorithms based on *MapReduce* framework have been proposed [18]–[26]. The main work of those approaches is in the partitioning the massive dataset. The pruning strategies are developed based on the partitioning method. In general, the partitioning strategies are based on the grid and angle. The grid-based approaches have the advantage that it is easy to avoid load balancing problems. On the other hand, angle-based approaches have better pruning power that can reduce more unnecessary data points. However, the most recent algorithms can be classified into two groups where the first is sampling based algorithms and the second is non-sampling algorithms. Sampling based algorithms utilize data samples to construct the partition to avoid load balancing problem. However, there is no guarantee that the sampling data represent the data distribution correctly. If the data distribution is not represented well by the sample data, then the performance of the algorithm is also less. The algorithms that perform this sampling step are [5], [22], [23], [25]. On the other hand, the algorithms that do not perform the sampling step are [20] and [21]. The proposed algorithm developed in this research is based on the non-sampling algorithms namely *MR-GPMRS* and *PPF-PGPS* algorithms.

3 METHODOLOGY

This section provides a detailed explanation of our proposed algorithm. The preliminary is presented first (Subsection 3.1) and followed by our definitions of *LShape* partitioning concepts (Subsection 3.2). Subsection 3.3 explains the proposed two phases of the *MapReduce* algorithm named *2Phase LShape* algorithm. Subsection 3.4 formalizes our algorithms developed in this study. The *1Phase LShape* algorithm is presented in Subsection 3.5.

3.1 Preliminaries

A data point t_i in d-dimensional space is denoted by $t_i = (t_{i,1}, \ldots, t_{i,j}, \ldots, t_{i,d})$ where $t_{i,j}$ is the value of the j^{th} dimension of t_i. We denote D as a set of data points and $|D|$ the number of data points in D.

Definition 1. (*Dominating*) A data point t_i dominates t_j if $\forall m,\ t_{i,m} \leq t_{j,m}$, and $\exists n,\ t_{i,n} < t_{j,n}$. It is denoted as $t_i \prec t_j$ and otherwise, $t_i \not\prec t_j$ which represents t_i does not dominate t_j.

Definition 2. (*Skyline*) A data point that is not dominated by any other data points in a dataset D is called a skyline data point; the set of all skyline data points in D is denoted as $Skyline = \{t_i | \forall j,\ j \neq i,\ t_i, t_j \in D,\ t_j \not\prec t_i\}$

TABLE 1
Notation Table

Symbols	Description
t	A data point
d	The dimension of t
D	A set of data points
dc	A data cell
DC	A set of data cells
$cmin$	The minimum coordinate of a data cell
$Skyline$	A set of skyline data points
$SkyCells$	A set of skyline data cells
$\prec$	Dominating
$\nprec$	not Dominating
$\prec\prec$	Strongly Dominating
$\nprec\nprec$	not Strongly Dominating
L	An *LShape* partition
LS	A set of *LShape* partitions
ppd	partition per dimension

3.2 Concepts of *LShape* Partitioning

In this subsection, we formally define our *LShape* partitioning concepts. For better clearly, we provide notation in Table 1.

A conventional grid partitioning strategy divides each dimension into a number of equal intervals. This number is called *partition per dimension* and denote as ppd. As a result, the whole search space is partitioned into ppd^d cells. A cell can be uniquely identified by its minimum coordinate denoted by $cmin$ – i.e., $cmin = (min[1], \ldots, min[j], \ldots, min[d])$ where $min[j]$ is the minimum value of j^{th} dimension in the corresponding cell, which is also denoted as $cmin_j$. A cell that contains data points is called **data cell**, denoted as dc, and uniquely identified by its minimum coordinate $dc.cmin$. The set of all data cells is denoted as DC. We assign each data cell an index i as dc_i for clear presentation.

Fig. 2 shows an example of data cells in $\mathbb{R}^2$ *Euclidean* space, which is partitioned by dividing each dimension into 5 equal intervals. Taking data point t_7(70,30) as an example, the corresponding data cell is identified by $dc.$(60,20).

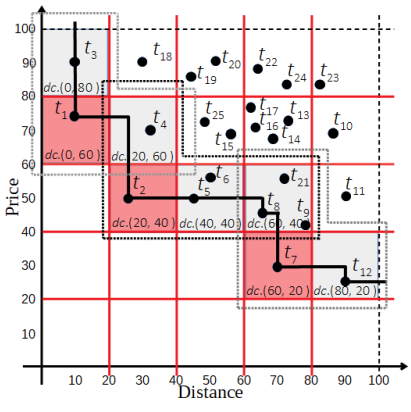


Fig. 2. *LShape* Partitioning Strategy

Concepts of Data Cell

Definition 3. (*Dominating of Data Cell*) Given two data cells dc_i and dc_j, dc_i dominates dc_j iff $\forall m, dc_i.cmin_m \leq dc_j.cmin_m$ and $\exists n, dc_i.cmin_n < dc_j.cmin_n$. It is denoted by $dc_i \prec dc_j$, and otherwise, $dc_i \nprec dc_j$ which represents dc_i does not dominate dc_j.

Definition 4. (*Strongly Dominating of Data Cell*) Given two data cells dc_i and dc_j, dc_i strongly dominates dc_j iff $\forall m, dc_i.cmin_m < dc_j.cmin_m$. It is denoted by $dc_i \prec\prec dc_j$. On the other hand, dc_j is strongly dominated by dc_i.

Definition 5. (*Skyline Data Cells*) A skyline data cell is a data cell that is not dominated by any other data cells. A set of skyline data cells is denoted by $SkyCells = \{dc_i \mid \forall j, j \neq i, dc_i \nprec dc_j\}$.

As shown in Fig. 2, $dc.$(60,20) dominates $dc.$(80,20). The $dc.$(0,20) strongly dominates $dc.$(20,40)). And, the $dc.$(0,60), $dc.$(20,40), and $dc.$(60,20) are the skyline data cells.

Concepts of *LShape* partitioning

With our proposed concept of *strongly dominating data cells*, we guarantee strongly dominated data cells do not contain any skyline data points, whose data points are dominated by data points in the skyline data cells. However, skyline data points may locate in the dominated data cells. Our main idea is to focus on **dominated but not strongly dominated** data cells. For example, in Fig. 2, $dc.(20, 40)$ strongly dominates $dc.(40, 60)$ and the points in $dc.(40, 60)$ (e.g., t_{15}, t_{25}) are pruned. By our definition, $dc.(40, 40)$ is dominated (not strongly dominated) by $dc.(40, 60)$, whose data points t_5, t_6 can not be pruned.

The formalization of an *LShape* partition is summarized as follows.

(i) We first identify skyline data cells. In Fig. 2, $SkyCells = \{dc.(0,60), dc.(20,40), dc.(60,20)\}$ (colored in red).

(ii) We define candidate data cells by the set of dominated but not strongly dominated data cells $DC_{candidate}$

$$DC_{candidate} = DC_{dominated} - DC_{Sdominated}$$
$$DC_{dominated} = \{dc_i \mid \exists dc_s \in SkyCells, dc_s \prec dc_i\}$$
$$DC_{Sdominated} = \{dc_i \mid \exists dc_s \in SkyCells, dc_s \prec\prec dc_i\}$$

In Fig. 2, $DC_{candidate} = \{dc.(0,80), dc.(20,60), dc.(40,40), dc.(60,40), dc.(80,20)\}$ (colored in gray). After pruning strongly dominated data cells, we have a union of $SkyCells$ and $DC_{candidate}$.

(iii) For each skyline data cell $dc_s \in SkyCells$, we collect data cells that belong to $DC_{candidate}$ and are dominated by dc_s as a set of candidate cells w.r.t. dc_s, denoted as $L^{set}(dc_s)$

$$L^{set}(dc_s) = \{dc_i \mid dc_i \in DC_{candidate}, dc_s \prec dc_i\}$$

(iv) We sort each $L^{set}(dc_s)$ based on *Z-order* to form a tuple $L = \langle dc_1, dc_2, \ldots, dc_l \rangle$ where dc_1 is dc_s because dc_s is the closest data cell to the origin. In general, these data cells in L resembles an L-like shape in a 2-dimensional space, and we call it an *LShape* partition.

An *LShape* partition L is identified by L's $dc_1.cmin$ denoted as $L.(dc_1.cmin)$ since dc_1 is the only skyline data cell in the partition.

In Fig. 2, there are three *LShape* partitions with each corner cell (skyline data cell) of an *LShape* partition colored by red. The first *LShape* partition is identified by L_1 =

$L_1.(0,60) = \{dc.(0,60), dc.(0,80), dc.(20,60)\}$. We denote the set of L partitions that cover the whole search space as LS.

A data cell may belong to more than one *LShape* partition. In Fig. 2, $dc.(20,60)$ is a member of $L_1.(0,60)$ and $L_2.(20,40)$. We propose an Intersection Data Cell concept as follows.

Definition 6. (*Intersection Data Cell*) A data cell dc that is a member of more than one L partition. $\exists\ m$ and n, $n \neq m$, $dc_i \in L_m$ and $dc_i \in L_n$.

3.3 The 2*Phase LShape* Proposed Algorithm

The first phase is the construction of *LShape* partitions and the second phase is the computation of skyline data points based on the *LShape* partitions. The first phase scheme is shown in Fig. 3 and the second phase in Fig. 4.

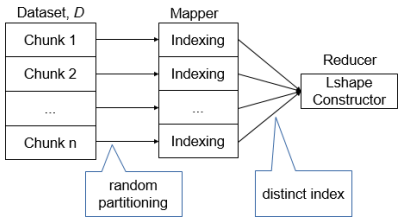


Fig. 3. The First Phase of 2*Phase LShape* Algorithm

In the first phase, the *LShape* partitions are constructed by distinct indices of every data point. Each data point is indexed by its corresponding data cell that is *dc.cmin*. It is a light *MapReduce* job because the data transmitted to a reducer are very small. For example, it is similar to the grid-based partitioning that divides each dimension into *ppd* equally sized intervals, and the number of the cells is ppd^d (d is the number of dimensions). However, not all the cells contain data points, and only data cells are processed in the reducer to construct a set of *LShape* partitions.

In detail, the first phase shown in Fig. 3 starts from dividing dataset D into n chunks randomly. Then, each chunk is processed in a mapper. A mapper determines each data point index. A data point index is a data cell identifier where it is located. A mapper only emits distinct data cell identities to a reducer. In the other words, only distinct cells that are contained data point/points (data cell) are processed in the reducer to construct a set of *LShape* partitions. Next, the reducer performs *LShapeSetConstruction* Algorithm that is presented in Algorithm 1 to construct a set of *LShape* partitions.

The scheme of the second phase is presented in Fig. 4. First, a set of *LShape* partitions produced in the first phase is broadcasted to all the mappers. Similar to the first phase, dataset D is randomly balanced partitioned into n chunks and each chunk is processed in a mapper. In this step, data points that are not located in any *LShape* partitions are filtered out. If a data point is a member of an *LShape* partition, it is entered to a data cell in one *LShape* partition or several *LShape* partitions for the intersection data cells. Next, the *Propagation Filtering* algorithm explained in Algorithm 4 is performed to find the skyline data points in a particular *LShape* partition. Each *LShape* partition produced in the mapper step is emitted to a corresponding

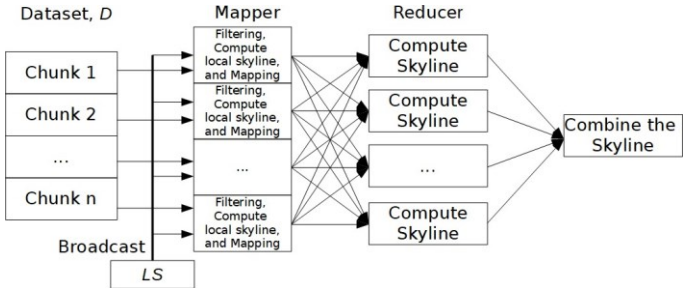


Fig. 4. The Second Phase of 2*Phase LShape* Algorithm

reducer to compute the skyline data points in entire dataset D. This reducer also performs *Propagation Filtering* in Algorithm 4. However, the skyline data points produced by a reducer still contain non-skyline data points that are located in the intersection data cells. Finally, the results of reducers are combined in a machine and non-skyline data points are removed by a light computation explained in Lemma 3. It is similar to the *MR-GPMRS* [20] that removes duplicated skyline data points in the final step. In the *LShape* partitioning strategy approach, skyline data points in an intersection data cell must be skyline for all the *LShape* partitions which contain this data cell. The technique is explained in detail in Subsection 3.4.

3.4 The Details of the Proposed Algorithms

3.4.1 Construction of a set of *LShape* Partitions

Construction of a set of *LShape* partitions is performed by the reducer at the first phase. It is not computationally intensive since the number of data cells is relatively small compared to the number of data points. Assuming each dimension is divided into the same number of partition *ppd* then the number of cells is ppd^d.

The reducer collects distinct data cells (without their data points) identified from the mappers. The output of this step is a set of *LShape* partitions *LS* that will be used in the next phase. According to Definition 5, the *SkyCells* is calculated based on the input data cells *DC* where each element of *SkyCells* becomes the first element and it is to be the identifier of each *LShape* partition. This step is shown in Algorithm 1 in line 1 of *LShapeSetConstruction* algorithm. The elements of an *LShape* partition are data cells that are dominated by the skyline data cell but not strongly dominated by the skyline data cell (as explained in Subsection 3.2). Therefore, all strongly dominated data cells by *SkyCells* in *DC* are removed in Line 2 of *LShapeSetConstruction* algorithm by running *removeStronglyDominatedCells* function. Next, the elements of each *LShape* partition L are determined in line 3 that calls *setLShapeMember* function. This function works as defined in Subsection 3.2 to find members of an *LShape* partition. A data cell that is not filtered in line 2 must be at least an element of an *LShape* partition. Furthermore, if it is an element of more than one *LShape* partition, then it is an *intersection data cell* according to Definition 6. Finally, members of each *LShape* partition L_j are sorted by *Z-Order* (line 4 to 6).

Algorithm 1 *LShapeSetConstruction* Algorithm

Input: DC
Output: LS
1: $SkyCells$ = *findSkylineCells*(DC)
2: DC = *removeStronglyDominatedCells*($SkyCells$, DC)
3: LS = *setLshapeMember*($SkyCells$, DC)
4: **for** each $L \in LS$ **do**
5: L = *zOrderSort*(L)
6: **end for**
7: **return** LS

3.4.2 LShape Partitioning Strategy

In the second phase, the set of *LShape* partitions LS is broadcasted to all the mappers. The input data D is divided into multiple balanced chunks D^l and each chunk is processed in a mapper. With the whole set of *LShape* partitions LS, a mapper filters out data points that are not a member of any data cell in LS. If a data point is an element of a data cell that belongs to an L ($L \in LS$), then this data point is added to the corresponding data cell. This step is performed in Lines 1 to 8 in *LshapeMapper* algorithm (Algorithm 2). Then, the mapper performs *Propagation Filtering* to search for the skyline data points in an L. It is operated for each L individually that is shown in Algorithm 2 Lines 9 to 11. Additionally, the *Propagation Filtering* is also performed in the reduce phase to determine the skyline data points in D. In this approach, the calculating of skyline data points can be performed independently for each *LShape* partition based on Lemma 2 and Lemma 3.

An important property of *Z-order* is presented in Lemma 1 according to [27].

Lemma 1. Given an *LShape* partition L and two data cells $dc_i \in L$, $dc_j \in L$, for any data point $t_i \in dc_i$, $t_j \in dc_j$, if dc_j is ordered after dc_i then t_j cannot dominate t_i.

Proof. The proof is by contradiction. We assume all the data cells in an *LShape* partition are sorted in Z-order based on the closeness of the data cell $cmin$ to the origin. Since dc_j is ordered after dc_i, there is at least a dimension k s.t. $dc_j.cmin_k > dc_i.cmin_k$. As a result, $\forall t_i, t_j$ that $t_i \in dc_i$, $t_j \in dc_j$, we have $t_{i,k} > t_{j,k}$. If $\exists t_i \in dc_i, t_j \in dc_j, t_j \prec t_i$ according to the definition of skyline dominance (Definition 1), $t_{i,k} \leq t_{j,k}$ which leads to a contradiction. Thus, this concludes the proof. □

Local skyline data points found in an *LShape* partition cannot be dominated by data points in other *LShape* partitions, except those found locally in the *intersection data cells*. It is presented in Lemma 2 as follows.

Lemma 2. (*Independence*) In an *LShape* partition L_i, $Skyline^{Li}$ is a set of skyline data points found in L_i. Assuming L_i contains a set of intersection data cells DC^{inter}, we have $DC^{inter} \subseteq L_i$ ($dc^{inter} \in DC^{inter}$). The skyline data points found in DC^{inter} are denoted as $Skyline^{inter}$. Therefore, $Skyline^l = Skyline^{Li} - Skyline^{inter}$, $t_n \in Skyline^l$, $t_n \in Skyline^{Lj}$, $L_j \neq L_i$, and $L_j \in LS$.

Proof. Based on Definition 5, the skyline data cell of L_i is dc_1^{Li} ($L_i = (dc_1^{Li}, \ldots, dc_n^{Li})$), and the skyline data points found in dc_1^{Li} are not dominated by any other data points in D. Based on Lemma 1, $dc_j^{Li} (2 \leq j \leq n)$ is possible to be dominated by other data cell dc_k^{Li} where $k < j$. If $dc_j^{Li} \notin DC^{inter}$ then for all $t_p \in dc_j^{Li}$ is only possible to be dominated by other data points $\in t_q$ dc_k^{Li} where $k \neq j$. Therefore, for any $t_p \in Skyline^l$, it cannot be dominated by other data points in D □.

Algorithm 2 *LShapeMapper* Algorithm

Input: LS, D' where $D' \subseteq D$.
Output: key value pairs (key, L); key is the id of L
LShape filtering
1: **for** each $t \in D'$ **do**
2: **for** each $L \in LS$ **do**
3: **if** t is located in L **then**
4: L = *add*(L,t); add t to a corresponding $dc \in L$
5: Break
6: **end if**
7: **end for**
8: **end for**
Propagation filtering
9: **for** each $L \in LS$ **do**
10: L = *propagationFiltering*(L)
11: **end for**
12: **return** key value pairs (key, L)

Skyline Data Points in an Intersection Data Cell

A local skyline data point in an intersection data cell is a global skyline data point if and only if it is a skyline data point for the union of *LShape* partitions which overlap with the intersection cell.

Lemma 3. For $dc^{inter} \in DC^{inter}$, a data point t_j is skyline data point in D if t_j is determined as skyline data point in all L_n where $dc^{inter} \in L_n$.

Proof. Based on the proof of Lemma 2 and that dc^{inter} is dominated by each skyline data cell dc_1^{Ln} for all L_n where $dc^{inter} \in L_n$, then, a data point $t_j \in dc^{inter}$ is probably dominated by data point t_k from any L_n where $dc^{inter} \in L_n$. Therefore, skyline data points in dc^{inter} have to be skyline data points in all L_n where $dc^{inter} \in L_n$ □.

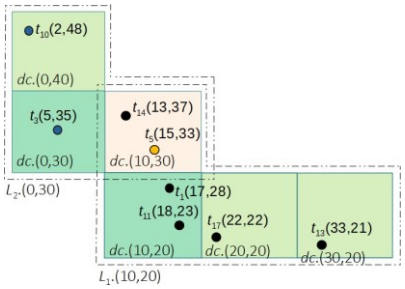


Fig. 5. Example Intersection Data Cell

Fig. 5 shows two *LShape* partitions in two dimensional space, L_1 and L_2. The $dc.(10, 30)$ is an *intersection data cell* because it exists in two *LShape* partitions. Based on Lemma 2, we can be determine the skyline data points in each *LShape* partition individually. The skyline data points that result in L_2 are t_{10}, t_3, and t_5. The skyline data points that result in L_1 are t_{14}, t_5, t_1, t_{11}, t_{17}, and t_{13}. When these

two *LShape* partitions are combined, additional selection is needed in the intersection data cell $dc.(10, 30)$. Skyline data points in this data cell must be skyline data points in both *LShape* partitions. In this case, since t_3 is available in both L_1 and L_2 then t_3 is the skyline data point and t_{14} is removed.

We formally provide a policy to merge partitions as follows:

(*Partitions Merge Policy*) We iterate each L_i in an *LS*. The first data cell in L_i is the skyline data cell and it is directly added to the final result without checking. We then iterate data cells from the second to the last. If the cell is not an intersection data cell, it is added to the final result. Otherwise, the skyline data points in this data cell must satisfy Lemma 3. After that, we add this intersection data cell to the final result and delete this intersection data cell in other *LShape* partitions.

Algorithm 3 *mergeLStoR* Algorithm

Input: *set of LShape partition LS, the number of reducers R*
Output: *set of LShape partition groups LG*

1: **for** each $L_i \in LS$ **do**
2: $G_i = \{L_i\}$
3: add G_i to LG
4: **end for**
5: **while** $|LG| > R$ **do**
6: G_f=find G with the smallest number of cells in LG
7: remove G_f from LG
8: G_s=find G with the smallest number of cells in LG
9: remove G_s from LG
10: $newG$=combine G_f and G_s
11: add $newG$ to LG
12: **end while**
13: **return** LG

To have the balanced load for reducers, Algorithm 3 merges a set of *LShape* partitions into R groups, which represents the number of reducers. Line 1 to 4 initializes a set LG of groups of the *LShape* partitions. Initially, each group contains one *LShape* partition. The process of merging the *LShape* partitions starts for finding the first and the second smallest groups G_f and G_s that is done on Line 6 and 8. Determining the smallest group is based on the total number of data cells contained in the *LShape* partitions in the group. Line 7 and 9 remove the selected groups from LG. Then, G_f and G_s are combined to $newG$ by merging all *LShape* partitions of the two groups. Because the *LShape* partitions are combined, there is no duplicated intersection cell in a group. The $newG$ is added to LG in Line 11. The process from Line 6 to 11 is repeated until the number of $|LG|$ is equal to or less than R. The detailed process of merging two or more *LShape* partitions follows the *Partition Merge Policy* that is explained in this Subsection.

3.4.3 Propagation Filtering

The outline of the *Propagation Filtering* is presented as follows:

- Starting from the skyline data cells, we traverse the rest of the data cells iteratively in *Z-Order*.
- When evaluating a data cell (non-skyline data cell), it is divided into two-levels checking: data cell and data point.
- In data cell level checking, we follow Lemma 4 that not all the data cells need to be compared.
- In data point level, we only keep data points that have a dimension smaller than all the skyline data points found in previous data cells.
- We find skyline data points in this data cell by *BNL* algorithm.

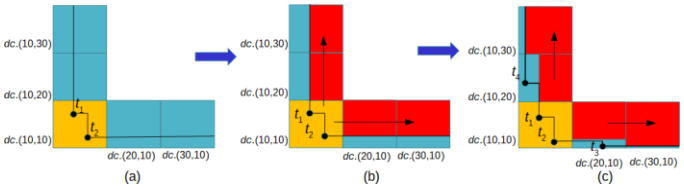


Fig. 6. Example of *Propagation Filtering* in 2-dimensional space

Similar to [21], our filtering technique runs on the *Z-order* sorted grids. This filtering is performed in an *LShape* partition to find skyline data points. An example of *Propagation Filtering* for 2-dimensional data is illustrated in Fig. 6. First, all data cells on an *LShape* partition are sorted based on *Z-order*. Therefore, the skyline data cell will be processed first, then followed by the data cells that are located nearest to the skyline data cell. In the example of Fig. 6, the order of data cells being processed is $\{dc.(10, 10), dc.(20, 10), dc.(10, 20), dc.(30, 10), dc.(10, 30)\}$; $dc.(10, 10)$ is the skyline data cell. First, the *Propagation Filtering* starts from the skyline data cell that is shown in Fig. 6(a). In this specific example, the result is two skyline data points from the skyline data cell $dc.(10, 10)$ that are t_1 and t_2. Then, based on the result from the skyline data cell, we evaluate the subsequent data cells that are closest to the skyline data cell, which are $dc.(20, 10)$ and $dc.(10, 20)$. However, based on Lemma 4, we do not need to evaluate all the data points in the subsequent data cells. In data cell $dc.(20, 10)$, we only need to consider data points whose y dimension is less than t_2's y dimension. Similarly, in data cell $dc.(10, 20)$, we only need to evaluate the data points whose x dimension is less than t_1's x dimension. Then, the result of this step is t_3 in data cell $dc.(20, 10)$, and t_4 in data cell $dc.(10, 20)$. These results are then propagated to data cell $dc.(10, 30)$ and $dc.(30, 10)$. Besides that, this approach does not need to compare the data points in data cell $dc.(10, 20)$ with $dc.(20, 10)$, data cell $dc.(10, 20)$ with $dc.(30, 10)$, data cell $dc.(30, 10)$ with $dc.(10, 20)$, or data cell $dc.(30, 10)$ with $dc.(10, 30)$ according to Lemma 4.

Lemma 4. Given data cell $dc_p.(x_1, \ldots, x_d)$ and data cell $dc_q.(y_1, \ldots, y_d)$, if $\exists i, j, i \neq j, x_i < y_i$ and $x_j > y_j$ then all data points in dc_p and dc_q are not comparable.

Proof. If data cell dc_p dominates data cell dc_q then it must satisfy condition $\forall m, dc_p.cmin_m \leq dc_q.cmin_m$, and $\exists k, dc_p.cmin_k < dc_q.cmin_k$. Therefore, $\forall k, dc_p.cmin_k > dc_q.cmin_k$ □.

Pseudo code of *Propagation Filtering* algorithm is presented in Algorithm 4, which is executed by mappers and reducers repeatedly for each L. The output is another

Algorithm 4 *PropagationFiltering* Algorithm

Input: L (L has already been sorted by *Z-order*)
Output: L' (It only contains skyline data points)
1: $L^l = \emptyset$
starting from the first data cell dc in L (skyline data cell)
2: **for** each $dc \in L$ **do**
3: **if** $L^l = \emptyset$ **then**
4: **for** each $dc' \in L'$ **do**
5: **if** *needToCompare*($dc.cmin$, $dc^l.cmin$) **then**
6: **if** $dc^l = \emptyset$ **then**
7: $indices$ = *getIndicesToCompare*($dc.cmin$, $dc^l.cmin$)
8: dc = *propagationFilteringItem*(dc^l,dc,$indices$)
9: **end if**
10: **end if**
11: **end for**
12: **end if**
13: dc = *skylineBNL*(dc)
14: L^l = *add*(L^l,dc)
15: **end for**
16: **return** L^l

LShape partition L^l that only contains skyline data points. In Line 1, the output of *LShape* partition is denoted as L^l, which is initialized as an empty *LShape* partition. The *LShape* partition L has been sorted by *Z-Order*, then it will examine each data cell in the *LShape* partition starting from the skyline data cell according to Lemma 2 and Lemma 1. When evaluating the skyline data cell, if the L^l is empty, then the data points in the skyline data cell are processed by *BNL* algorithm to locate skyline data points (line 13). The resultant skyline data points are stored in a new data cell dc and it is added to L^l. Then, the next data cell from the *LShape* partition is processed and compared with the data cells in L^l. We need to evaluate the necessity of comparison between the data cell dc from L and dc^l from L^l by *needToCompare* function, according to Lemma 4. If comparison is necessary, the attributes indices for comparison are determined by *getIndicesToCompare* function. Finally, to find skyline data points in dc, we execute *propagationFilteringItem* function. Then the skyline data points are added to L^l with the same data cell dc as the new data cell. It is repeated until all the data cells in an *LShape* partition are evaluated.

Algorithm 5 shows the algorithm to determine the necessity of comparison between two data cells. It works based on Lemma 4 that if a dimension x of a data cell $dc_A.cmin_x$ is less than $dc_B.cmin_x$, and another dimension y of data cell $dc_A.cmin_y$ is greater than $dc_B.cmin_y$, then dc_A and dc_B are not comparable.

We do not need to compare all the dimensions in the data point level since the *cmin*'s dimensions of data cells indicate necessity. Taking data cell $dc.(20, 20)$ and data cell $dc.(30, 20)$ as an example, we only need to compare the second dimension because the first dimension of data points in $dc.(30, 20)$ is larger than the first dimension of data points in $dc.(20, 20)$. Furthermore, those two data cells have already been sorted based on *Z-Order*, and the skyline data points in $dc.(20, 20)$ are calculated first before $dc.(30, 20)$. The implementation of this concept is explained in Fig. 6.

Algorithm 5 *needToCompare* Algorithm

Input: $dc_A.cmin$, $dc_B.cmin$
Output: *Boolean*
1: *greater* = false
2: *less* = false
d is the number of dimension
3: **for** i = 1 to d **do**
4: **if** $dc_A.cmin_i < dc_B.cmin_i$ **then**
5: *less* = true
6: **else if** $dc_A.cmin_i > dc_B.cmin_i$ **then**
7: *greater* = true
8: **end if**
9: **end for**
10: **return** $\neg$ (*greater* $\wedge$ *less*)

The algorithm to evaluate the necessity of comparison for dimensions between two data cells is presented in Algorithm 6; we record the dimensions where the values are equal between two data cells.

The data points level comparison of data points in two data cells is presented in Algorithm 7. We find a dimension of a data point in the current data cell (dc in Line 9 of Algorithm 4) that is less than a dimension of a data point in the other data cell. It previously has been evaluated (dc^l in Line 9 of Algorithm 4). This technique implements Lemma 1 that is the property of *Z-order*. For example, by the previous evaluation, the data point $t_{17}(22,22)$ in data cell $dc.(20, 20)$ is detected as a skyline data point. Then, when evaluating data cell $dc.(30, 20)$, we only need to find a data point where the second dimension is less than 22. This mechanism will reduce the computation cost for high dimensions of data because it does not need to compare all the dimensions and only finds one dimension that is smaller. It is shown in Line 7, 8, and 13 that if a dimension of a data point is less than a dimension of all the skyline data points in the previous data cell, then this data point is not comparable to all skyline data points found in the previous step. This step is done from the first data cell to the current data cell of the *LShape* partition. After this filtering is done, the final step in this algorithm is to find skyline data points on the current data cell by *BNL* algorithm. This is done in *PropagationFiltering* algorithm in line 14 in Algorithm 4.

Algorithm 6 *getIndicesToCompare* Algorithm

Input: $dc_A.cmin$, $dc_B.cmin$
Output: *indices*
1: $indices = \emptyset$
d is the number of dimensions
2: **for** i = 1 to d **do**
3: **if** $dc_A.cmin_i = dc_B.cmin_i$ **then**
4: $indices \cup$ i
5: **end if**
6: **end for**
7: **return** *indices*

3.5 The 1*Phase LShape* Algorithm

As explained in Subsection 3.2, the cells are constructed by dividing each dimension of the dataset by an integer number *ppd*. For a smaller number of *ppd*, we get a larger size

Algorithm 7 *propagationFilteringItem* Algorithm

Input: dc, dc^l, *indices*
where dc is current data cell and dc^l is previous data cell
Output: dc''

```
1: dc'' =∅
2: for each t ∈ dc do
3:    Boolean isSmaller = false
4:    for each t^l ∈ dc^l do
5:       for each index ∈ indices do
6:          if t_index < t^l_index then
7:             isSmaller = true
8:             Break
9:          end if
10:      end for
11:   end for
12:   if isSmaller then
13:      dc'' ∪ t
14:   end if
15: end for
16: return dc''
```

of cells. From our experiments shown in Fig. 13, we can see that the best execution time is achieved when ppd=2. This is because a smaller cell size produces a higher number of *LShape* partitions, which is then followed by an increased number of intersection data cells. Since the intersection data cells need to be duplicated to the corresponding *LShape* partitions, it slows down the execution time. The number of duplicated intersection data cells increases exponentially when the number of dimensions increases.

One effort to decrease the number of duplicated intersection cells is by merging *LShape* partitions into a number of groups, equal to the number of reducers. Algorithm 3 is designed to balance the processing time of the reducers. However, this effort does not change the result that the best execution time is achieved when the ppd=2. In this case, it only needs one reducer to find the global skyline points. We therefore propose the 1*Phase LShape* algorithm.

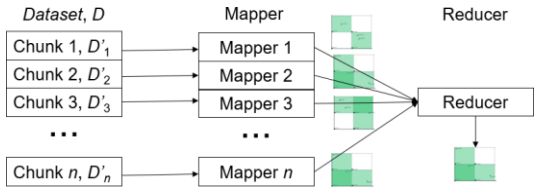


Fig. 7. 1*Phase LShape* Algorithm

The 1*Phase LShape* algorithm scheme is illustrated in Fig. 7. First, the dataset D is divided into n chunks and each chunk is processed into a mapper by Algorithm 8. It constructs a set of data cells DC by dividing data space into 2^d data cells (note that ppd=2) in Line 1. From Line 2 to 4, it assigns each data point to the corresponding its data cell. Then in Line 5 the *LShape* partitions are formed and strong dominated data cell pruned. Next in Line 6, the *Propagation Filtering* is performed to find the local skyline data points. Define the closest data cell to the origin as *origin cell*. If the *origin cell* is not empty then only one *LShape* partition is formed. However, if it is empty then the maximum number of the *LShape* partitions is equal to the number of the dimensions. Each mapper then emits the *LShape* partitions to the reducer.

In this scheme, only one reducer is used because the number of *LShape* partitions is small. Algorithm 9 shows the process in the reducer. The cells from each *LShape* partition are collected in Line 1 and a new *LShape* partition is constructed in Line 2. This step prunes the strong dominated data cell if it exists. Finally, *Propagation Filtering* is used to find the global skyline data points.

Algorithm 8 1*PhaseMapper* Algorithm

Input: *A chunk D' of Dataset D*
Output: *A set of LShape partition LS*

```
1: DC=construct data cells with 2 partition per dimension
2: for each t_i ∈ D^l do
3:    assign t_i to corresponding data cell C ∈ DC
4: end for
5: construct LS using Alg. 1
6: PropagationFiltering(LS) using Alg. 4
7: return LS
```

Algorithm 9 1*PhaseReducer* Algorithm

Input: *LShape partitions L_A and L_B*
Output: *set of LShape partitions LS*

```
1: DC=take all cells from L_A ∪ L_B
2: construct LS using Alg. 1
3: PropagationFiltering(LS) using Alg. 4
4: return LS
```

4 Experiments and Analysis

In this section, we present the performance of the *LShape* partitioning strategy algorithm that is compared to the state of the art approaches. We generate three types of dataset distributions that are generally used in the field of study. Those are anti-correlated dataset, independent dataset, and correlated dataset. However, the results of the correlated datasets are not presented because the results are similar to those of the independent datasets. We generate those three types of datasets for 10 million to 100 million data points and each in 2 dimensions to 10 dimensions. Besides that, we generate a real number of datasets in this experiment because the real world dataset could be a real number instead integer. A real dataset needs larger space in memory than integer dataset. The *ZDG+DM* utilizes *Z-Order* that is only suitable for integer data. Therefore, the *ZDG+DM* algorithm cannot be implemented to process real number datasets. However, the *LShape* partitioning strategy algorithm implements *Z-Order* in the data cell level that uses an integer to identify a data cell.

We implement our algorithms and state of the art approaches of this research, that are *MR-GPMRS* and *PPF-PGPS* by Java 8 and Apache Spark *MapReduce* framework version 2.2.0. Those algorithms are tested on a Hadoop cluster on Elastic *MapReduce* AWS cloud computing service [1] where the maximum number of nodes in a cluster is 16

1. https://aws.amazon.com/emr

machines. Each node has m4.large machines and each is 2.4 GHz Intel Xeon® E5-2676 v3, 2 cores, and 8 GB memory.

Based on our literature survey, we can classify algorithms of MapReduce Skyline into two: sampling based algorithms and non-sampling based algorithms. The *2Phase* and *1Phase LShape* algorithms, *PPF-PGPS*, and *MR-GPMRS* are classified into non-sampling algorithms. The *2Phase LShape*, *PPF-PGPS*, and *MR-GPMRS* are two phases parallel *MapReduce* skyline algorithms. Therefore, we compare the performance of *2Phase LShape* to the same class of skyline *MapReduce* algorithms: *PPF-PGPS* and *MR-GPMRS*. These comparations are presented in Subsection 4.1, 4.2, and 4.3. Next, we evaluate the performances of *1Phase* and *2Phase LShape* algorithms in Subsection 4.4, 4.5, and 4.6. We also present our experiments on the sampling effect in Subsection 4.7.

4.1 Varying Data Dimensions

The purpose of these experiments is to know the response time with increasing number of dimensions. Actually, 1M data can be processed in a single machine if the number of dimensions is low. However, if the number of dimensions increases, then the execution time would increase exponentially, which can not be processed in a single machine. The execution time not only depends on the data size but also depends on the data dimensions and the data distribution. Taking correlated data as an example, the execution time is much faster than anti-correlated data.

The comparisons of execution time of *2Phase LShape* compared to *PPF-PGPS* and *MR-GPMRS* in the varied dimensions are shown in Fig. 8. Fig. 8(a) shows that *2Phase LShape* partitioning strategy algorithm outperforms the *PPF-PGPS* and *MR-GPMRS* for anti-correlated dataset. In low dimensional data such as 2D and 3D, the differences of execution time are not as large as in high dimensional dataset. However, the *2Phase LShape* still runs faster than the two baseline algorithms shown in Fig. 9 when we present the comparisons in ranged size of dataset.

With increasing dimensions, the execution time of *PPF-PGPS* and *MR-GPMRS* increases exponentially for the three types of dataset shown in Fig. 8. However, *2Phase LShape* has a linear increase of execution time with an increasing number of dimensions. It shows that the *2Phase LShape* and *Propagation Filtering* are effective in high dimensional data.

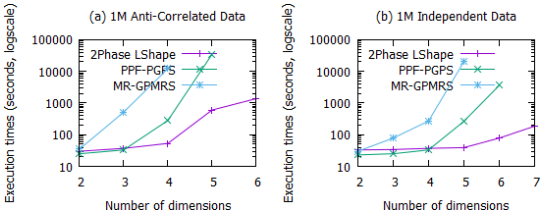


Fig. 8. Comparison by Number of Dimensions

The *PPF-PGPS* does not perform well compared to *2Phase LShape* because of the following reasons.: The *PPF-PGPS* implements *Progressive Filtering* to reduce the number of inputs in the first phase. This filtering is effective for correlated and independent dataset. However, the performance of the *Progressive Filtering* is poor for anti-correlated dataset [21]. Furthermore, the data are more widely spread in high dimensional data, which hurts the filtering performance in the *PPF-PGPS*. Actually, when the number of dimensions increases, we need to add more filtering objects. This challenge is addressed by *Propagation Filtering* in *2Phase LShape* algorithm where the user does not need to set the number of filtering objects. Our *Propagation Filtering* technique also selects the filtering object from the more effective locations in the search space (e.g., skyline data cells). In the *2Phase LShape* algorithm, the skyline data points that result from processing the skyline data cell can be treated as the filtering objects. Those filtering objects are used to filter the data points when processing the upcoming data cells, and the filtering objects are added automatically after processing the current data cell.

The data in the independent dataset are distributed evenly across the data space. The number of skyline data points in this data distribution is less than anti-correlated dataset because there are more data points located near the origin in the independent dataset than anti-correlated dataset. This condition makes *PPF-PGPS* filtering technique be able to filter more data points. It is shown in Fig. 8(b) that for 2D, 3D, and 4D data, the performance of *PPF-PGPS* is similar to *2Phase LShape*. However, when the dimension is 5 or more, the execution time of PPF-PGPS increases exponentially, whereas the *2Phase LShape* partitioning strategy performs in linear time response.

4.2 Varying Data Sizes

In this subsection, we evaluate the performance of *2Phase LShape* compared to the *MR-GPMRS* and *PPF-PGPS* in ranged data size. We use the fixed dimensions and range the data size from 10 million to 100 million. We stop the experiment if the difference of execution time is significant. Because the response time is much different between *MR-GPMRS* compared to *PPF-PGPS* and *2Phase LShape*, we separate it into two charts for independent and correlated dataset.

The execution time of *MR-GPMRS* and *PPF-PGPS* increases exponentially. Therefore, we need to choose a proper dimension to evaluate the execution time based on the varied data size, otherwise, the execution time between algorithms will be a very large difference. Moreover, the longest computation time is unacceptable (e.g., weeks) in real-life. For anti-correlated dataset, it is shown in Fig. 8.a that the performance of *2Phase LShape* and *PPF-PGPS* is similar in up to 3-dimensional anti-correlated dataset. Here, the *MR-GPMRS* has fast execution time. Therefore, we use 3-dimensional anti-correlated dataset for this experiment. For the independent dataset, based on Fig. 8.b, the execution time of *2Phase LShape*, *PPF-PGPS*, and *MR-GPMRS* is small in 4-dimensional dataset. Moreover, *PPF-PGPS* and *2Phase LShape* have similar execution time and then *PPF-PGPS* execution time increases significantly in 5-dimensional independent dataset compared to *2Phase LShape* performance. Therefore, in the independent dataset, we use a 4-dimensional dataset to evaluate the

MR-GPMRS, *PPF-PGPS*, and 2*Phase LShape* partitioning strategy. However, because the performances of 2*Phase LShape* and *PPF-PGPS* are similar, we use a 5-dimensional independent dataset to evaluate these two algorithms.

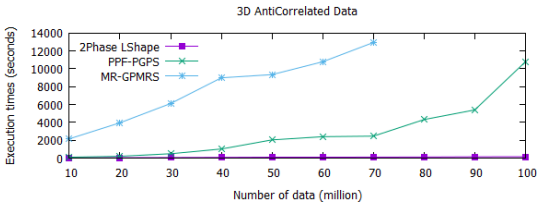


Fig. 9. Comparison by Data Sizes for Anti-Correlated Data

For the anti-correlated dataset, the performance of 2*Phase LShape*, *PPF-PGPS*, and *MR-GPMRS* is shown in Fig. 9. We choose 3-dimensional data for this evaluation since the performances of 2*Phase LShape* and *PPF-PGPS* are similar in 3-dimensional data as shown in Fig. 8.a. Therefore, we investigate more in this dimension from 10 million to 100 million data. The execution time of 2*Phase LShape* of the input dataset from 10 to 100 million increases slightly. However, the execution time of the other two algorithms increases dramatically. It shows that 2*Phase LShape* performs much better than the other two. For instance, with 70 million data, 2*Phase LShape* runs in 108 seconds (1.8 minutes), *PPF-PGPS* runs in 2460 seconds (41 minutes), and *MR-GPMRS* runs in 12960 seconds (3.6 hours).

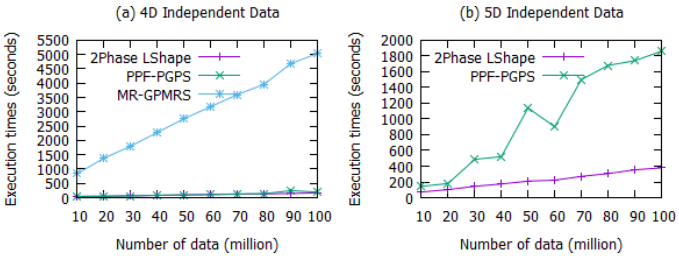


Fig. 10. Comparison by Data Sizes for Independent Data

The execution time of algorithms for independent dataset is shown in Fig. 10(a) and 10(b). As shown in Fig. 10(a), the performance of *MR-GPMRS* is worse than *PPF-PGPS* and 2*Phase LShape*. Fig. 10(b) shows the comparison of 2*Phase LShape* with *PPF-PGPS*. Based on these observations, the 2*Phase LShape* algorithm performs better in varied number of data for independent and correlated data. However, in the independent dataset, the execution time of *PPF-PGPS* drops significantly when the number of data increases such as in Fig. 10(b) on 60M data. In this case, data with such distributions benefit from the *PPF-PGPS* filtering process since a large number of data points are successfully removed during the mapper phase. As a result, the execution time of independent dataset does not increase smoothly when input data grows, showing that the performance of *PPF-PGPS* filtering depends on the distribution of input data [21]. Independent data has more probability to be close to the origin. Those data points have a large dominating region and are very sensitive to the filter process. If those data points are detected early in the progressive filtering on *PPF-PGPS*, then the overall execution will be fast. Besides, as mentioned in [21], *PPF-PGPS* performs heuristic to select the best filtering objects. It is commonly understood that heuristic algorithm could be trapped in the local optima. We found in our experiment that *PPF-PGPS* algorithm has fluctuated increasing execution time with the increasing number of data.

4.3 Varying Number of Computer Nodes

The next experiment is the evaluation of algorithms with varying number of computer nodes in a Hadoop cluster. We select a proper number of dimension and data size based on the experiment in Section 4.2 to anticipate a long execution time when the number of machines used is small. Next, we divide the comparison into two experiments and charts because the execution time of *MR-GPMRS* compared to 2*Phase LShape* and *PPF-PGPS* is much longer.

The execution time of *MR-GPMRS* has a logarithmic decreasing trend when the number of machines increases. It is shown in Fig. 11(a) and 12(a) that the execution time is much higher than that of *PPF-PGPS* and 2*Phase LShape*.

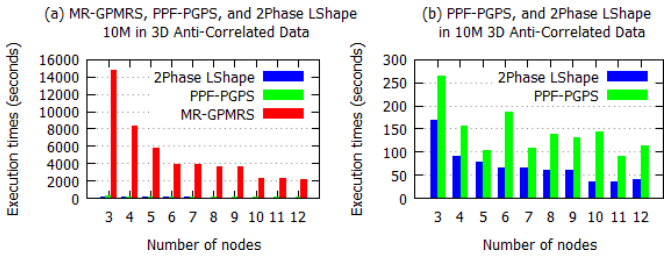


Fig. 11. Comparisons by the number of computer nodes for Anti-Correlated Data

In Fig. 11(b) and 12(b), 2*Phase LShape* also has a logarithmic decreasing with the increasing number of machines but it runs much faster than *MR-GPMRS*. *PPF-PGPS* has fluctuating execution time in these experiments although the trends are also in logarithmic decreasing. Similar to the analysis of the previous experiment, fluctuating execution time is caused by the data distribution and the choosing of filtering objects. The heuristic function may not select the best filtering object although the input data are the same, since in the first step of the *MapReduce* framework, it divides the data into several chunks according to the number of machines. Therefore, with different number of machines, the data chunks of the data are also different and so are the selected objects.

Based on the increasing number of executor machines, 2*Phase LShape* performs better than *PPF-PGPS* in the anticorrelated data as depicted in Fig. 11(b). As explained in Section 4.2, *PPF-PGPS* performs worse to filter out data in the *PPF* phase for the anti-correlated dataset. In the independent dataset dataset, *PPF-PGPS* performs better with small number of machines but when the number of machines grows, the execution time of 2*Phase LShape* becomes similar to *PPF-PGPS* as shown in Fig. 12(b).

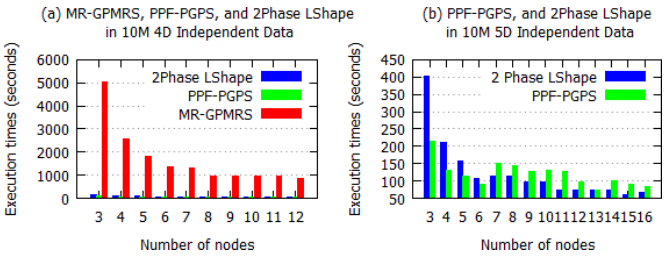


Fig. 12. Comparisons by the number of computer nodse for Independent Data

4.4 Varying Cell Sizes

The main purpose of these experiments is to know the effect of the number of partitions. Intuitively, a smaller size of cells produces a larger number of *LShape* partitions.

It has been explained in Subsection 3.5 to vary the cell sizes. Interestingly, our experiments show that the number of duplicated intersection cells increases exponentially when the number of dimensions increases, as shown in Fig. 13(a).

Because the partitioning strategy of our approach is extended from the *MR-GRPMRS* partitioning strategy, our experiments also show that it also suffers from the explosion of duplicated cells. However, the number of duplication cells in *MR-GPMRS* is a little bit higher than 2*Phase LShape*. It is because the 2*Phase LShape* produces a smaller number of partitions than *MR-GPMRS* as shown in Fig. 13(b). Furthermore, the *Propagation Filtering* proposed in this study performs well then the execution time of the 2*Phase LShape* runs is smaller than *MR-GPMRS* as shown in Fig8, 9 and 10.

Next, in Fig. 13(c) and 13(d), we evaluate the number of reducers used for the 2*Phase LShape* algorithm. These experiments show that one reducer is always better. When we use ppd = 1, a centralized *BNL* algorithm is used, and the performance is worse than that ppd = 2. In Fig. 13(d), the execution time of ppd = 3 increases significantly. By dividing each dimension by 3 or other odd numbers, there must be a cell located at the center of data space. Generally, data points are concentrated on the center of the data space for correlated and anti-correlated datasets. Therefore, for ppd = 3, the cell located at the center of the data space is the most populated cell which is not pruned and it is duplicated to the correlated *LShape* partitions. Consequently, the number of data processed by mappers and reducers is larger than the input dataset and the execution time increases significantly. This effect also happens when ppd = 5 although it is not as significant. However, when ppd is even or larger than 5, the most populated data cell is probably pruned or it is not duplicated.

4.5 2*Phase* and 1*Phase LShape* Algorithms

We compare the 2*Phase* and 1*Phase LShape* algorithms in this subsection. We merge the *LShape* partitions by Algorithm 3 into one group and use one reducer for the 2*Phase LShape* algorithm to make it comparable to the 1*Phase LShape* algorithm which uses one reducer. We use 10 million to 100 million 3-dimensional anticorrelated datasets and the result is shown in Fig. 14. Based on this

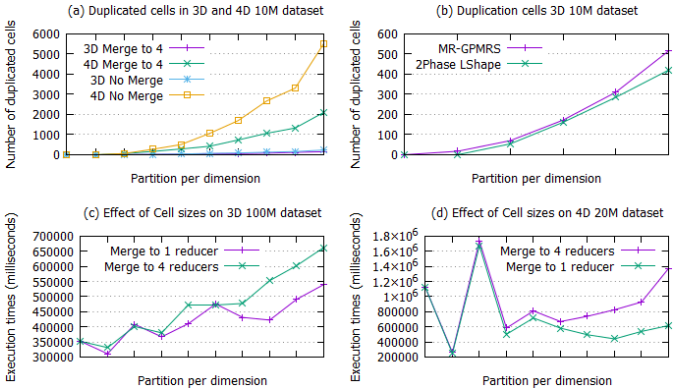


Fig. 13. Cell Sizes Experiments

experiment, the 1*Phase* algorithm has a better performance than the 2*Phase LShape* algorithm. It is because 1*Phase LShape* algorithm does not need to construct the *LShape* partitions in the first phase of the 2*Phase LShape* algorithm.

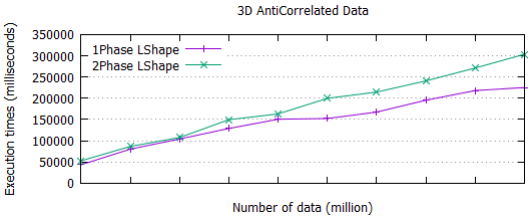


Fig. 14. 1*Phase* vs 2*Phase LShape* approach

4.6 Real-World Dataset

The experiments utilize the real-world dataset HEPMASS [2]. It is a dataset used in high-energy physics experiments for machine learning to find the signature of exotic particles. To find the exotic particles, Baldi, Cranmer, Faucett, Sadowski, and Whiteson [28] use a parameterized neural network to classify a new particle. In atomic reaction in quantum physics, when an elementary particle named quark collides a particle, it produces energy and new particles. Because the number of the new particles explodes in the atomic reaction, the dataset for this research is very large. It is interesting to find the dominating particles such that researchers can prune a large number of particles dominated by others.

This dataset includes 3 tables each containing 10.5 million records with 28 attributes. The amount of data is relatively small compared to our synthetically dataset, and we vary the number of attributes and the number of computer nodes used to find the performance of the 1*Phase LShape* algorithm. We perform two experiments in real-world datasets. The first evaluates the varying number of computer nodes, i.e. 3, 4, 8, 12, and 16 computer nodes. The second observes the varying number of dimensions of the dataset, i.e. the first 3, 4, ..., and 10 dimensions. The results are shown in Fig. 15(a) and 15(b). In Fig. 15(a), we

2. https://archive.ics.uci.edu/ml/datasets/HEPMASS

see that the best execution time is achieved when there are 8 computer nodes used. When there are more than 8 computer nodes used, the execution time increases slightly because it needs more data transfer among the nodes. Fig. 15(b) shows an exponential increase of the execution time for increasing number of dimensions of the HEPMASS dataset using 8 computer nodes.

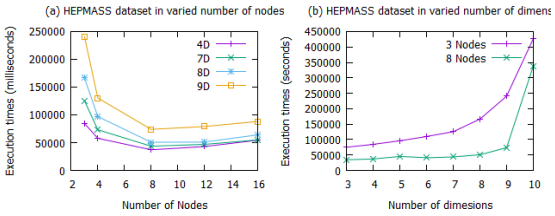


Fig. 15. Real-World Dataset

4.7 Effectiveness of Sampling

Our decision to develop a non-sampling parallel skyline algorithms is based on the experiments on the effectiveness of the sampling strategy. We implemented a random sampling method commonly used in the field named *Reservoir Sampling*. Our experiment to evaluate the effectiveness is as follows. First, we generate a fixed size of dataset D in three data distributions. Second, we take asamples S from 1% to 10% of the dataset D. Third, to measure the effectiveness of the sampling, we find skyline points Sky of the sample S and filter out the dataset D by Sky. The set of the remaining data points after the filtering is called F . Finally, we calculate the effectiveness ratio by $|(|D|-|F|)|/|D|$. The results are shown in Fig. 16. It can be seen that the sampling is only effective for higher dimensions for correlated dataset, and it is less effective for the independent dataset, and worst for the anticorrelated dataset. The results are shown in Fig. 16. As is shown in Fig. 16(c) the effectiveness of sampling is above 80% for the 1% sampling of a 10-dimensional correlated dataset. For the independent dataset, Fig. 16(b), the effectiveness drops to around 50% for the 10% sampling of 10-dimensional dataset. In Fig. 16(a), the effectiveness for the anticorrelated dataset drops to below 30% for all the experiments.

5 Conclusion and Future Research

In this study, we have presented our work on the parallel Skyline queries processing based on *MapReduce* framework. We introduced a new partitioning strategy that is called by *LShape* partitioning strategy, a variant of grid based partitioning. The *LShape* partitioning strategy has the advantage that we can utilize new filtering method that is called *Propagation Filtering*. The *LShape* partitioning strategy and *Propagation Filtering* work better than the state of the art of our approach, *MR-GPMRS* and *PPF-PGPS* which are the non-sampling algorithm approaches. It is shown in our intensive experiments for anti-correlated, independent, and correlated dataset that the performance becomes better in high dimension and high number of data.

Future research based on *MapReduce* skyline processing can be divided into categories as follows. The first is improving the existing algorithms to become more efficient with more balanced partitioning, increasing the pruning power, and reducing the communication overhead among the mappers and reducers. The second of the future research is the study of the *MapReduce* skyline queries in the variants of skyline queries such as dynamic skyline, reverse skyline, uncertain skyline, and continuous skyline. Each variant of skyline queries needs further technique to process the data because not all variants are composable. And the third is the utilization of the *MapReduce* skyline queries for multi-criteria decision making in real dataset to find valuable information in many fields such as health, transportation, economics, and others.

Acknowledgment

This research has been funded in part by the R.O.C. Ministry of Science and Technology grant 107-2221-E-468-013-, and the U.S. National Science Foundation grants IIS-1618669 (III) and ACI-1642133 (CICI).

References

[1] S. Borzsony, D. Kossmann, and K. Stocker, "The Skyline operator," *Proceedings 17th International Conference on Data Engineering*, pp. 421–430, 2001.

[2] G. Wang, J. Xin, L. Chen, and Y. Liu, "Energy-efficient reverse skyline query processing over wireless sensor networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 7, pp. 1259–1275, 2012.

[3] C. Y. Lin, J. L. Koh, and A. L. Chen, "Determining k-most demanding products with maximum expected number of total customers," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 8, pp. 1732–1747, 2013.

[4] G. Xiao, K. Li, and K. Li, "Reporting L Most Favorite Objects in Uncertain Databases with Probabilistic Reverse Top-k Queries," *Proceedings - 15th IEEE International Conference on Data Mining Workshop, ICDMW 2015*, pp. 1592–1599, 2016.

[5] J.-L. Koh, C.-Y. Lin, and A. L. P. Chen, "Finding k most favorite products based on reverse top-t queries," *The VLDB Journal*, vol. 23, no. 4, pp. 541–564, 2014.

[6] B. C. Tan, K.-L.; Eng, P.-K. Eng; Ooi, "Efficient progressive skyline computation," in *VLDB*, 2001.

[7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with Presorting," in *ICDE*, 2003.

[8] J. Zhang, W. Wang, X. Jiang, W.-S. Ku, and H. Lu, "An mbr-oriented approach for efficient skyline query processing," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 806–817, IEEE, 2019.

[9] D. Papadias, G. Fu, and B. Seeger, "An Optimal and Progresive Algorithm for Skyline Queries," in *ACM SIGMOID*, 2003.

[10] E. Dellis and B. Seeger, "Efficient Computation of Reverse Skyline Queries," *VLDB 07 Proceedings of the 33rd international conference on Very large data bases*, pp. 291–302, 2007.

[11] J. Pei, B. Jiang, X. Lin, and Y. Yuan, "Probabilistic Skylines on Uncertain Data," *33rd International Conference on Very Large Data Bases*, pp. 15–26, 2007.

[12] Y. Tao and D. Papadias, "Maintaining Sliding Window Skylines on Data Streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 3, pp. 377–391, 2006.

[13] M. Sharifzadeh and C. Shahabi, "The spatial skyline queries," in *Proceedings of the 32nd international conference on Very large data bases*, pp. 751–762, Citeseer, 2006.

[14] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang, "Selecting stars: The k most representative skyline operator," *Proceedings - International Conference on Data Engineering*, pp. 86–95, 2007.

[15] X. Zhou, K. Li, Z. Yang, and K. Li, "Finding optimal skyline product combinations under price promotion," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 1, pp. 138–151, 2019.

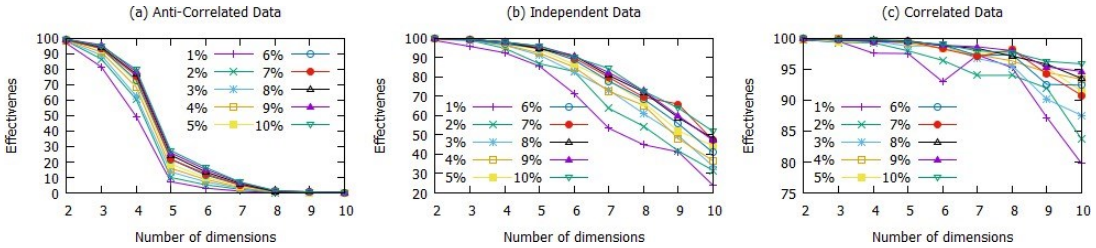


Fig. 16. Sampling effect of three data distributions

[16] A. Vlachou, C. Doulkeridis, and Y. Kotidis, "Angle-based space partitioning for efficient parallel skyline computation," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, (New York, NY, USA), p. 227–238, Association for Computing Machinery, 2008.

[17] M. Parsian, *Data Algorithms Recipes for Scaling up with Hadoop and Spark*. Boston; Farnhan; Sebastapol; Tokyo: O'Reilly, 2015.

[18] B. Zhang, S. Zhou, and J. Guan, "Adapting Skyline Computation to the MapReduce Framework: Algorithms and Experiments," *Lecture Notes in Computer Science*, vol. 6637, no. 60873040, pp. 403–414, 2011.

[19] L. Chen, K. Hwang, and J. Wu, "MapReduce skyline query processing with a new angular partitioning approach," *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012*, pp. 2262–2270, 2012.

[20] K. Mullesgaard, J. L. Pedersen, H. Lu, and Y. Zhou, "Efficient Skyline Computation in MapReduce," *Proceedings of the 17th International Conference on Extending Database Technology*, no. c, pp. 37–48, 2014.

[21] J. Zhang, X. Jiang, W. S. Ku, and X. Qin, "Efficient parallel skyline evaluation using MapReduce," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 1996–2009, 2016.

[22] Y. Park, J.-K. Min, and K. Shim, "Parallel computation of skyline and reverse skyline queries using mapreduce," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 2002–2013, 2013.

[23] Y. Park, J. K. Min, and K. Shim, "Efficient Processing of Skyline Queries Using MapReduce," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 5, pp. 1031–1044, 2017.

[24] J. L. Koh, C. C. Chen, C. Y. Chan, and A. L. Chen, "MapReduce skyline query processing with partitioning and distributed dominance tests," *Information Sciences*, vol. 375, pp. 114–137, 2017.

[25] M. Tang, Y. Yu, W. G. Aref, Q. M. Malluhi, and M. Ouzzani, "Efficient Parallel Skyline Query Processing for High-Dimensional Data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 10, pp. 1838–1851, 2018.

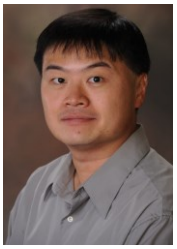
[26] W. Wang, J. Zhang, M.-T. Sun, and W.-S. Ku, "Efficient parallel spatial skyline evaluation using mapreduce," in *Proceedings of the 20th international conference on extending database technology*, 2017.

[27] K. C. K. Lee, B. Zheng, H. Li, and W.-C. Lee, "Approaching the skyline in z order," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pp. 279–290, VLDB Endowment, 2007.

[28] P. Baldi, K. Cranmer, T. Faucett, P. Sadowski, and D. Whiteson, "Parameterized neural networks for high-energy physics," *The European Physical Journal C*, vol. 76, 2016.

Heri Wijayanto received his BSc and MSc degree from Gadjah Mada University, Indonesia. Currently he is working toward his PhD degree in the department of Computer Science and Information Engineering at Asia University. He is a lecturer at Informatics Study Program, Engineering Faculty, Mataram University, Indonesia. His research interests include databases, software engineering, and information security.

Wenlu Wang received her BSc from Beihang University and MSc from New York University. Currently she is working toward her PhD in the department of Computer Science and Software Engineering at Auburn University. Her research interests include data science and data management.



Wei-Shinn Ku (S'02-M'07-SM'12) received his Ph.D. degree in computer science from the University of Southern California (USC) in 2007. He also obtained both the M.S. degree in computer science and the M.S. degree in electrical engineering from USC in 2003 and 2006, respectively. He is a program director at the National Science Foundation and a professor with the Department of Computer Science and Software Engineering at Auburn University. His research interests include databases, data science, mobile computing, and cybersecurity. He has published more than 100 research papers in refereed international journals and conference proceedings. He is a senior member of the IEEE and a member of the ACM SIGSPATIAL.

Arbee L.P. Chen received a Ph.D. degree in computer engineering from the University of Southern California, and is currently a Chair Professor of Computer Science at Asia University, and a professor of Department of Computer Science at National Tsing Hua University, by joint appointment. Dr. Chen was a Member of Technical Staff at Bell Communications Research, USA, and a Research Scientist at Unisys, USA. Dr. Chen organized the Eleventh IEEE Data Engineering Conference in Taiwan, and continuously serves in various capacities for international conferences and journals. He was invited to deliver a speech in the NSF-sponsored Inaugural International Symposium on Music Information Retrieval, USA, the IEEE Shannon Lecture Series, USA, and the Institute for Advanced Study of Hong Kong University of Science and Technology. Dr. Chen's current research interests include big data analytics, top-k queries, and multimedia information retrieval. He has published more than 250 papers in renowned international journals and conference proceedings, and was a visiting scholar at Kyoto University, King's College London, Stanford University, Boston University, Harvard University, and Hong Kong University of Science and Technology.