HI-Sky: Hash Index-Based Skyline Query Processing

Jong-Hyeok Choi 1, Fei Hao 2,3 and Aziz Nasridinov 1,*

1 Department of Computer Science, Chungbuk National University, Cheongju 28644, Korea; leopard@chungbuk.ac.kr
2 School of Computer Science, Shaanxi Normal University, Xi’an 710119, China; fhao@snnu.edu.cn
3 Department of Computer Science, College of Engineering, Mathematics and Physical Sciences, University of Exeter, Exeter EX4 4PY, UK
* Correspondence: aziz@chungbuk.ac.kr

Received: 6 January 2020; Accepted: 27 February 2020; Published: 2 March 2020

Abstract: The skyline query has recently attracted a considerable amount of research interest in several fields. The query conducts computations using the domination test, where “domination” means that a data point does not have a worse value than others in any dimension, and has a better value in at least one dimension. Therefore, the skyline query can be used to construct efficient queries based on data from a variety of fields. However, when the number of dimensions or the amount of data increases, naïve skyline queries lead to a degradation in overall performance owing to the higher cost of comparisons among data. Several methods using index structures have been proposed to solve this problem but have not improved the performance of skyline queries because their indices are heavily influenced by the dimensionality and data amount. Therefore, in this study, we propose HI-Sky, a method that can perform quick skyline computations by using the hash index to overcome the above shortcomings. HI-Sky effectively manages data through the hash index and significantly improves performance by effectively eliminating unnecessary data comparisons when computing the skyline. We provide the theoretical background for HI-Sky and verify its improvement in skyline query performance through comparisons with prevalent methods.

Keywords: database; query processing; skyline query

1. Introduction

The skyline query [1] returns data points that are not dominated by other data points in a given database. We refer to such data points as skylines, and domination indicates that a data point is better than other data points in at least one dimension while being equal to or better than other data points in the remaining dimensions. Figure 1 demonstrates an example of a skyline query. Here, Figure 1a lists a given dataset, and Figure 1b identifies the skylines of the dataset (i.e., A, G, H).

Generally, the skyline query can be used in many real-world applications, such as retail [2], road networks [3], and Web services [4]. Among these, it is commonly known that retail applications deal with dynamic datasets. For example, recently, it was reported that Walmart has high-volume data (i.e., four petabyte) that contains purchase records, where the skyline is frequently used to obtain a set of non-dominant data [5]. Considering that in such retail applications, data points change frequently (i.e., the addition of new purchase records, deletion of the records, or updates on the field values), it is essential to have a skyline query technique that scales well in the presence of the intense amount of updates. In addition, when the dimensionality or quantity of the data points is increased, naïve skyline queries lead to degradation in overall performance owing to the higher cost of pairwise data comparisons [1]. This degradation is the result of a significantly increased frequency of the comparison operations needed for performing dominance tests. Thus, to efficiently compute the skylines among a
large amount of high-dimensional data, a more efficient method is required to improve performance relative to existing methods.

| Point | Dimension 1 | Dimension 2 |
|-------|-------------|-------------|
| A     | 0.3         | 0.3         |
| B     | 0.4         | 0.4         |
| C     | 0.7         | 0.35        |
| D     | 0.35        | 0.8         |
| E     | 0.6         | 0.6         |
| F     | 0.9         | 0.8         |
| G     | 0.6         | 0.1         |
| H     | 0.9         | 0.05        |
| I     | 0.55        | 0.9         |
| J     | 0.8         | 0.55        |

(a) example dataset

Figure 1. An example of skyline query.

Among recent proposals, research interest has tended toward index structure-based methods due to their efficiency in handling large amounts of data. These methods generate indices on data for efficient skyline computation when a query is requested. The branch-and-bound skyline (BBS) is a method proposed by Papadias et al. [6,7] that utilizes the features of the R-tree [8,9]. Z-SKY, proposed by Lee et al. [10,11], computes the skyline through a ZBtree by combining the B+-tree with a Z-order curve [12,13]. However, BBS and Z-SKY are not suitable for skyline computation when the data frequently changes, as these methods require a large number of resources to maintain the indices [14,15]. Moreover, these methods fail to overcome issues related to high-volume data and dimensionality, which greatly influence the performance of indices.

In this paper, we propose a hash index-based skyline (HI-Sky), which is a skyline query method based on the hash index. HI-Sky can efficiently create and maintain index structures for large and high-dimensional data, and can eliminate unnecessary pairwise data comparisons when computing the skylines. More precisely, we make the following contributions in this paper:

- We propose an index structure, which is a fusion of the hash index and grid partitioning (simply HashGrid). HashGrid manages data by grid location address (GLAD), which is a special form of the hash key. We assign GLAD to each grid-partitioned region and perform a region-wise data comparison. This enables us to prune the data space without access to the entire dataset.

- We propose a hash index-based skyline (HI-Sky), which is a skyline query method that utilizes the features of HashGrid. HI-Sky computes the skyline through HashGrid using data that only belong to regions not dominated by other GLADs, which is therefore predicted to contain the skyline. This feature allows HI-Sky to avoid accessing unnecessary data that cannot be a skyline and simultaneously performs the dominance test using GLAD to enable efficient computation of the skyline only when domination is possible.

- We demonstrate the superiority of HI-Sky through various experiments. Specifically, we compare the index creation time and skyline query processing time with state-of-the-art skyline query methods including sort-filter skyline (SFS) [16,17], BBS [6,7], and Z-SKY [10,11] in a single system environment. The results of our experiment demonstrate that HI-Sky can generate indices faster than other methods and perform faster skyline query processing at higher dimensions by effectively reducing the number of dominance tests.

The remainder of this paper is organized as follows: In Section 2, we discuss existing work related to the proposed method. Section 3 describes the theoretical background of HashGrid and
HI-Sky. Section 4 presents the results of a comparison with existing methods. Section 5 summarizes the conclusions of this study.

2. Related Study

In this section, we review related studies on the skyline query. Numerous methods have been proposed to process skyline queries efficiently, and we classify them into traditional skyline computation, parallel and distributed skyline computation, and index structure-based skyline computation.

2.1. Traditional Skyline Computation

The traditional skyline query methods compare all data to compute the skyline. One such technique is the block nested loop (BNL), a basic skyline query method proposed by Borzsonyi et al. [15]. The BNL conducts the dominance test according to the order in which data are stored on a disk. However, the BNL is inefficient due to frequent changes in skyline candidate data point selection. Chomicki et al. [16,17] proposed the sort filter skyline (SFS), which computes the skyline by relying on the fact that data with better entropy scores are more likely to become skylines. To utilize this feature, the SFS calculates the entropy score while sorting the data and then performs the dominance test similarly to the BNL. The entropy score provides a monotonic ordering characteristic to the skyline candidate, which enables the SFS to compute the skyline more effectively than the BNL. However, the requirement of additional computation for entropy score calculation and sorting for all data makes this method unsuitable for the fast computation of skylines from large amounts of data.

2.2. Index Based Skyline Computation

The main problem with skyline queries is the need to perform dominance tests for the entire set of data. Index structure-based methods have been proposed to solve this problem, such as branch-and-bound skyline (BBS) [6,7] and Z-SKY [10,11].

The BBS was proposed by Papadias et al. [6,7]. It generates an R-tree [8,9] index structure for the given data through preprocessing. The skyline query is then requested using the spatial features of the minimum bounding rectangle (MBR), which is an index component of the R-tree. The main feature of BBS is that it removes MBRs dominated by other MBRs or skyline candidates through dominance testing. Thus, the removal of MBRs can be used to prune large amounts of data using a single comparison. However, the R-tree also has the problem of overlapping being increasingly likely among the MBRs as dimensionality increases. Because this overlapping significantly reduces the probability of domination in the dominance test, it also reduces the efficiency of the skyline computation.

Z-SKY was proposed by Lee et al. [10,11]. It computes the skyline through a ZBtree by combining a B+-tree and a Z-order curve index structure [12,13]. The Z-order curve generates an address for the given multidimensional data called a Z-address, and the Z-address is used as a basis to map the data to a one-dimensional space. Z-addresses have the characteristic of monotonic ordering, which allows the location information of the given data to be extracted. This in turn allows Z-SKY to compute the skyline without access to the actual data stored on the disk. For more efficient skyline computations, Z-SKY also uses RZ-regions, which are clusters that employ the B+-tree and Z-order curve segments. An RZ-region is generated by searching the Z-order curve segments that can be included in all Z-addresses belonging to the corresponding node using the internal node of the B+-tree. Therefore, as the generated RZ-region shows the data space with various lattices, it enables easy data space pruning with a small number of comparisons. However, when data at fine intervals (i.e., real values) are used in Z-SKY, the Z-address becomes very long. Because of this, Z-SKY can only be used with a limited range of integer data or with real values normalized to a limited range of integers. Even if real values are normalized, the exact skyline cannot be computed by the Z-address alone, because the actual data may be different. Thus, Z-SKY is highly dependent on the data type. Moreover, if the RZ-region of the skyline does not dominate the given data, all the RZ-regions belonging to this region are stored in a queue. Consequently, when the threshold of the node is high, a large number of nodes is entered into
the queue, and a dominance test is performed on them that may decrease computation performance. Thus, Z-SKY cannot guarantee fast performance in all cases due to the potentially massive storage and memory space requirement of the Z-address.

2.3. Parallel and Distributed Skyline Computation

Parallel and distributed skyline query methods have been proposed to rapidly compute skylines in parallel and distributed environments. The critical aspect of performance for these skyline queries is the assignment of data distributed to each system or processor. Therefore, various data-partitioning techniques have been proposed. In particular, Vlachou et al. [18] proposed a skyline query method that has received significant attention and is based on angle-based space partitioning (ABSP). Grid partitioning, which is a traditional partitioning approach, can potentially create a partition that does not contain a skyline. By contrast, these unnecessary partitions are less likely to be created when using ABSP because it reflects the topological characteristics of the skyline. Because the local skyline generated in each partition using ABSP is likely to become the global skyline, which is the final skyline query result, ABSP can respond more quickly to the skyline query request because it operates using distributed systems. However, in ABSP, as the numbers of dimensions and partitions increase, the likelihood of these unnecessary local skylines occurring significantly increases, and though they do not become global skylines at the partition boundary, overall efficiency is significantly reduced as a consequence. Therefore, although this method can respond quickly when used in a distributed environment, it is unsuitable for high-dimensional data.

New skyline query methods based on MapReduce technology have recently been proposed and have shown excellent results in the field of big data processing. Early applications of MapReduce included traditional skyline query methods [19], and existing distributed environments were also converted into Hadoop environments [20]. However, these methods are still ineffective on big data because they require dominance tests on all data to compute the skyline. Therefore, recent work has yielded a new method of combining MapReduce with index structure-based skyline methods. These methods can obtain skylines from a large number of data in a single scan by using group-to-group comparison instead of data-to-data comparison [21–24]. Also, skyline queries using an environment like SpatialHadoop [25], an extended Hadoop that supports various indexes for spatial data, have been proposed [26]. However, because their performance is highly dependent on the index structure used, which attracts continuous attention in the field of skyline computation.

3. Hash Index-Based Skyline Query Processing

This section describes the theoretical background of HI-Sky, the proposed skyline query method to more effectively handle high-dimensional big data. Section 3.1 describes HashGrid in detail. Section 3.2 demonstrates how HI-Sky performs skyline computation using HashGrid.

3.1. Hash Index for Skyline

In the past, major index structures have used a tree structure to accomplish data computation with a minimal number of operations. Existing tree structure methods such as the R-tree and B+-tree store data in a cluster under a certain standard. This feature makes it possible to quickly remove large amounts of data without requiring a comparison. Therefore, though conventional methods have the disadvantage of requiring dominance testing on all data, this can be obviated through the index structure to enable the efficient processing of large volumes of data, such as big data.

However, tree structure-based approaches based on the R-tree or B+-tree methods, such as BBS and Z-SKY, increase data sparsity in the data space by increasing the absolute distance between data points as the number of dimensions increases. This significantly reduces the probability of data space pruning in the MBR or the RZ-region. Moreover, these methods are expensive to preprocess or traverse for skyline computation. It is also difficult to utilize these methods in environments where data insertion and deletion are frequent because it leads to expensive operations such as changes to the tree
structure [27–29]. In such cases, skyline computation requires a new index structure method capable of clustering the given data and maintaining it at low cost.

HashGrid is a fusion of the hash index [30,31] and grid partitioning [32,33], and allows for chaining. The most important feature of hash indices is that very little time is needed to find the bucket where data is stored, or will be stored. Chaining-based hash indices are also robust in terms of maintenance because they do not require changes to other buckets when storing or deleting data. Therefore, in generating and maintaining indices using large amounts of data, a hash index can be generated more quickly than a conventional tree-based method, and maintenance cost is thus reduced. Further, hash index clusters similar data points in the same hash bucket, similarly to tree structures. At the same time, it is common to use a formula that can distribute the data as uniformly as possible. Therefore, to effectively use the hash index in skyline computation, it is important to select a hash function that can cluster data based on the data space and sparsity. Moreover, the generated hash key must be able to prune the data space without needing to access the real data. To do this, the hash function used in HashGrid divides the given data at regular intervals in each dimension and combines them sequentially to generate a hash key. Then, the number of partitions (np) for dividing each dimension is input by the user. Alternatively, given the number of dimensions d, the minimum bucket capacity C, and the number of data N, the maximum value of np can be obtained from the following Equation (1).

$$\max(np) = \begin{cases} \left\lfloor \sqrt[2]{N/C} \right\rfloor, & \sqrt[2]{N/C} \geq 2 \\ 2, & \sqrt[2]{N/C} < 2 \end{cases}$$

Equation (1) computes the maximum value of np required to store the amount of data close to the minimum capacity into the partitioned space, assuming that a uniformly distributed dataset is given. However, it returns 2, the minimum value of np, when there is no value satisfying this criterion. Then, the final np is selected from the minimum value of np (np = 2) to the maximum value of np. Finally, if the data is correlated or anti-correlated, better performance can be obtained by using a value slightly larger than the maximum value of np obtained through Equation (1). The hash key generated using this np represents the position of the grid-partitioned region, which we call GLAD.

Given the data points belonging to 2-D data space [0, 1]² as shown in Figure 1a, and assuming the number of partitions (np) of each dimension is four, each dimension is divided into regular intervals and marked in an ordered manner starting from zero (e.g., 0, 1, 2, 3) as shown in Figure 2. Thus, given the data point $G = [0.6, 0.1]$, we obtain values of two for the horizontal axis and zero for the vertical axis. On the other hand, if the data point is on a partition boundary, the higher order is applied (e.g., 0.5 obtains two). The hash function computes the order along each axis for the given data, i.e., the number of dimensions, and generates GLAD based on this order. In this case, GLAD is obtained by converting the order of each dimension into bits and combining them in sequence. The number of digits representing all intervals in order needs to be the minimum number of bits that can represent them. The required number of minimum digits can be obtained by taking $\log_2(np)$ and performing a subsequent decimal point round-up. Therefore, if np is four, as shown in this example, two bits are required to indicate the marked intervals of each dimension. Because data point G is located in the third interval along the horizontal axis (bit value 10), and in the first interval along the vertical axis (bit value 00), the result of their combination in GLAD is eight (1000). Figure 2 shows the partition results for the data space used in this example as well as the GLAD assigned to each partition. The generated GLADs were not duplicated, and were instead obtained by combining the interval order corresponding to each partition in a bitwise sequence. Furthermore, the generated GLADs have the column major-order characteristic for the data space when the bit is changed to decimal, allowing GLADs to be more easily managed.

HashGrid uses the GLAD obtained to store the data in the appropriate hash index. Once the hashing and storing of all data is complete, only the hash index with the bucket is stored in the index storage space on the disk according to the order of GLAD, thereby minimizing the use of unnecessary
index storage space. Because of these features, HashGrid is very robust against frequent insertion and deletion, unlike index structures used in previous studies, and can also prune the data space because the clustering of the given data is based on the lattice region. Moreover, dividing the data into a uniformly sized lattice space minimizes the reduction in the efficiency of data space pruning resulting from partition overlap.

3.2. Hash Index-Based Skyline Query Processing

To increase the processing efficiency of skyline computation, we propose a HashGrid-based skyline query method called hash index-based skyline (HI-Sky). HashGrid is an index structure that manages the given data by dividing the data space into uniformly sized partitions. In this section, we discuss the basic background, data space pruning step, and skyline computation step of HI-Sky.

3.2.1. Background

Most operations in skyline computation perform dominance tests to determine the dominance among the data. As mentioned in Section 1, domination means that a data point is better than other data points in at least one dimension while being equal to or better than other data points in the remaining dimensions. Here, the criterion for determining a good value is “<” or “>,” and in this paper, smaller values are considered better for the consistency of contents. The definition is as follows:

**Definition 1.** Dominance [1011]. Given d-dimensional space $S = \{s_1, s_2, \ldots, s_d\}$ and data points $p$ and $p'$ in this space, $p$ is said to dominate $p'$ if $\forall s_d \in S, p[s_d] \leq p'[s_d] \land \exists s_k \in S, p[s_k] < p'[s_k]$, and is not said to do so otherwise.

HI-Sky performs most operations in the dominance test similarly to existing skyline methods. Therefore, for a quick computation of the skyline, it is necessary to remove unnecessary data with minimal comparison. GLAD as used by HI-Sky has the feature of column-major ordering along with the location information of the data space, which has the following characteristics in the skyline:

**Lemma 1.** Given d-dimensional space $S = \{s_1, s_2, \ldots, s_d\}$ and arbitrary GLADs $a$ and $b$ composed from a combination of the sequential order of each dimension $O_k$ ($k = 1 \ldots d$), if $a.O_k < b.O_k$ in all dimensions that constitute the GLADs, all data that belong to $a$ dominates all data belonging to $b$.

**Proof.** A GLAD is formed by a combination of the order of regions divided by the partition in each dimension. Therefore, when the comparison operation is performed on the GLADs, $O_k$ ($k = 1 \ldots d$) orders can be obtained in each dimension. In this case, if arbitrary GLADs $a$ and $b$ satisfy $a.O_k < b.O_k$ in all dimensions, the data belonging to GLAD $a$ have a better value in all dimensions than those belonging to GLAD $b$. Thus, Definition 1 is always satisfied. \(\square\)
Lemma 1 shows that HI-Sky can prune the data space using GLAD and can remove a large amount of data early, which significantly reduces the number of dominance tests. In this case, data space pruning is performed only if the corresponding GLAD has a smaller order in all dimensions. This is because, as shown in Figure 2, data point A may dominate data points such as C and D which have the same order in a certain dimension. However, data point B may not dominate other data points C and D. As these data do not satisfy the conditions of Lemma 1, i.e., neither point is in a partition with smaller corresponding dimensions than the other point given the order of the partition, data space pruning is not performed. Moreover, even if Lemma 1 is not satisfied, the characteristics of the column-major order in GLAD lead to the following characteristics of Lemma 2 and 3 that can be used to evaluate the need for dominance tests between partitions, and this results in the effective reduction of unnecessary dominance tests:

Lemma 2. Given arbitrary GLADs a and b, if a < b, a has a smaller value in at least one dimension in which its order is less than that of b.

Proof. GLAD is characterized by a column-major order. Therefore, when each of d-dimensions is divided into np partitions, the order \( O_k \) \( (k = 1 \ldots d) \) of each dimension satisfies \( O_k \in [0, np - 1] \), whereas the GLAD is obtained through the following formula:

\[
(...(((O_1 \times np) + O_2) \times np) + O_3) \times np \ldots) \times np + O_d = \sum_{k=1}^{d} np^{d-k} \times O_k
\]  

(2)

Therefore, if the condition \( a < b \) is satisfied, \( a \) has at least one dimension smaller than \( b \) from \( O_1 \) to a specific \( O_k \).

Due to the characteristics of Lemma 2, data in relatively smaller GLADs have at least one dimension where they have a better value than data with relatively larger GLADs, and are thus not dominated by larger GLADs. For example, in Figure 2, data point B belonging to GLAD 5 (0101) has a larger value in dimension 1 than data point D in GLAD 7 (0111), but it is guaranteed through the GLAD that it has a smaller value in dimension 2. Likewise, data point B has a larger value in dimension 2 than C in GLAD 9 (1001), but a smaller value in dimension 1 is guaranteed through the GLAD. Therefore, it is never the case that data belonging to a relatively large GLAD, such as C and D, dominate B belonging to a relatively small GLAD. The proof of this is in Lemma 3.

Lemma 3. Given arbitrary GLADs a and b, if a < b, data belonging to a are not dominated by those belonging to b.

Proof. As evidenced in the Lemma 2, if \( a < b \) is satisfied, \( a \) has a smaller value than \( b \) in at least one dimension. Therefore, data belonging to \( b \) cannot dominate data belonging to \( a \) according to Definition 1. □

3.2.2. Data Space Pruning Step

HI-Sky conducts a two-step skyline computation to maximize the features of Lemmas 1–3. In the first step, the data space is pruned using the features of Lemma 1 to remove the partitioned region where the skyline cannot occur. To this end, we create a window in main memory to store the GLAD (simply GLAD_window) that is not dominated, and prune the data space through a comparison between the GLAD stored in GLAD_window and the next GLAD in sequence. As the GLAD is stored in an ordered manner at the final HashGrid, the characteristic of monotonic ordering is obtained when the GLAD is used sequentially. Therefore, it is unnecessary to check whether the given GLAD...
dominates GLADs stored in GLAD_window, thereby enabling a more efficient computation. The proof of this is in Lemma 4.

**Lemma 4.** When the data space is pruned using ordered GLADs, the order of the GLADs is characterized by monotonic ordering.

**Proof.** When ordered GLADs are used in sequence, GLADs stored in GLAD_window are always smaller than the next GLAD in the sequence to be compared. As demonstrated in Lemmas 2 and 3, large GLADs cannot dominate small GLADs. The number of GLADs stored in memory always increases monotonically. Therefore, GLADs are characterized by monotonic ordering. □

Once the first step completes data space pruning for GLADs that cannot contain the skyline, GLADs stored in GLAD_window are used to start the second step of computation for the actual skyline. For an effective skyline computation, a comparison operation is needed between the previously stored GLAD and the given GLAD because a dominance test can only proceed between data if domination can occur. This comparison operation is performed via a general dominance test using Definition 1 when the given GLAD is dominated by one previously stored in GLAD_window. This means that even though the order of some dimensions is the same, data belonging to the preceding GLAD may have a smaller value than those belonging to the given GLAD. However, as demonstrated by Lemma 3, the data in the next GLAD in sequence cannot dominate those in the preceding GLAD; thus, a dominance test is not necessary. Further, these comparisons can be processed more efficiently as a result of Lemma 5.

**Lemma 5.** When GLADs are ordered and GLAD a and b are given, if \( a < b \) and if \( a.O_k > b.O_k \) is satisfied in the \( k \)-th dimension (\( k = 1 \ldots d \)), then \( b \) is not dominated by GLAD \( a \), and \( b \) cannot be dominated by any subsequent GLAD with identical dimensional order until reaching a GLAD with its last order in the \( k \)-th dimension.

**Proof.** When ordered GLADs are used and current GLAD \( b \) is not dominated by GLAD \( a \) due to the order of the \( k \)-th dimension obtained by decomposing GLAD, then there is no need to perform a comparison until reaching a specific GLAD which has the same order in all dimensions except \( k \)-th dimension and has the last order in the \( k \)-th dimension, compared to \( a \). Because GLAD \( b \) has a smaller order than the others in the \( k \)-th dimension, Lemma 1 is not satisfied. Therefore, there is no need to perform a comparison because dominance cannot occur, and we can skip the comparison from the current GLAD to the specified GLAD using Lemma 5. □

### 3.2.3. Skyline Computation Step

After the data space pruning is completed, the skyline computation step is performed to compute the actual skyline using the GLADs in GLAD_window. At that stage, because skyline candidates also need to be managed according to their GLADs, a window for each GLAD (simply Sky_window) is created in main memory, and the selected data are stored in the Sky_window of each GLAD as a skyline candidate. In this process, we can perform the skyline computation step more effectively through the dominance test using GLADs. If the current GLAD is not dominated by a GLAD that was previously stored a skyline candidate in Sky_window, Lemma 5 allows us to skip unnecessary dominance testing between the data in each GLAD. Data that belong to the same GLAD stored in the Sky_window then need to be compared with one another. However, data belonging to the same GLAD are not monotonically ordered, and thus the efficiency of the computation can be drastically reduced if the data are compared in a naïve manner. To solve this problem, HI-Sky calculates the entropy score of the data to check for dominance, minimizing the efficiency reduction.
**Definition 2.** Entropy score [16,17]. Given d-dimensional space \( S = \{s_1, s_2, \ldots, s_d\} \) and a data point \( p \) belonging to that space, the entropy score of \( p \), \( E(p) \), is calculated using the actual values of each dimension of \( p \) as follows:

\[
E(p) = \sum_{k=1}^{d} \ln(p[s_k] + 1)
\]  

(3)

Such an entropy score enables monotonic ordering of the given data. Thus, to dominate specific data points, the given data point must have an entropy score smaller than a specific data point. On the contrary, when the entropy score of the given data point is greater than that of the specific data point, the latter is likely to dominate the former. The proof of this is as follows:

**Lemma 6.** Given d-dimensional space \( S = \{s_1, s_2, \ldots, s_d\} \) and data points \( p \) and \( p' \) belonging to that space, if the calculated entropy score of both data points satisfies the \( p \) dominates \( p' \) relationship, this condition is sufficient to show that \( E(p) < E(p') \).

**Proof.** For \( p \) to dominate \( p' \) by Definition 1, it must satisfy \( \forall s_d \in S, p[s_d] \leq p'[s_d] \land \exists s_x \in S, p[s_x] < p'[s_x] \). Therefore, to dominate a specific data point, the data point must have values smaller than or equal to that specific data point in all dimensions, with a smaller value in at least one dimension. When this condition is extended to Definition 2, \( E(p) < E(p') \) is always satisfied when \( p \) dominates \( p' \). \( \Box \)

Using these characteristics, HI-Sky confirms by use of entropy score whether the data point belonging to the same GLAD is dominated by the given data point, or whether the given data point is itself dominated. To perform these operations more efficiently, data that are not dominated are sorted by entropy score and stored in a Sky_window.

However, the features of Lemma 6 apply equally to dominance tests of all data and data points from the same GLAD. In particular, even if a comparison is made with a skyline candidate belonging to a relatively small GLAD, the given data can be dominated only if Lemma 6 is satisfied. If this is not the case, according to Lemma 3, the dominance test for the relevant GLAD can be completed early because the given data do not dominate the remaining skyline candidates. Therefore, HI-Sky prunes the data space early where the skyline cannot occur through Lemma 1, and compares all skyline candidates that can dominate the given data or can be dominated by them. Thus, HI-Sky computes for the exact number of skyline points.

In this way, HI-Sky performs fast skyline computation by data space pruning in data space pruning step, through various features of HashGrid, GLAD, and entropy score, and by the removal of most dominance tests in skyline computation step when absolute dominance cannot occur.

### 3.2.4. Skyline Computation using HI-Sky

In this section, we provide a step-by-step explanation of how HI-Sky handles skyline queries. HI-Sky involves a first step of data space pruning using HashGrid and the GLAD stored therein, as described in the previous section. The second step of computing the skyline using data involves the non-pruned GLAD.

For HI-Sky, when two-dimensional (2-D) data are given as shown in Figure 1a, they are represented in a 2-D space as shown in Figure 3a. For effective skyline query processing, HI-Sky generates a HashGrid through a hash function that allows grid partitioning using the given data before a skyline query is requested. In this case, if the np used by the hash function is 4, 16 GLADs can be stored in HashGrid in total, from zero (0000) to 15 (1111), as in Figure 3b. They have the property of column-major order due to the combination of bits. Once the HashGrid is completed, only the GLAD that has data as in Figure 3a is stored as the final HashGrid like Figure 3b; if the skyline query is requested, the stored HashGrid is used for it.
Sky_window and repeats the process. However, if there is a match, using the characteristics of Lemma 5, the skyline candidate is determined as either dominating or dominated, and the dominated data are removed. If data are not dominated according to the dominance test, they are stored as a skyline candidate in a Sky_window with the same GLAD. The storing is carried out in an aligned manner by entropy score for efficiency of comparison. Therefore, in the second step, data points A and B stored in GLAD 5 (0101) are used first, and A is determined as the skyline of 5 (0101), as shown in Figure 3e. When B is used first instead of A and is stored in Sky_window as a skyline candidate, we can see through Lemma 6 that A is likely to dominate B, and we can thus remove B and add A to the list of skyline candidates through comparison. In the subsequent case of GLAD 7 (0111), a comparison with the skyline candidates of GLAD 5 (0101) should be performed because 7 (0111) was dominated by 5 (0101). In this process, data point D is removed from the comparison because it is dominated by the skyline candidate A of 5 (0101) as shown in Figure 3f, and there are no more data points in 7 (0111). Thus, it does not have a skyline candidate.

Figure 3. An example of skyline query processing based on HI-Sky. (a) 2-D data space; (b) result of HashGrid; (c) data space pruning; (d) data space pruning (cont.); (e) skyline computation; (f) skyline computation (cont.); (g) skyline computation (cont.); (h) skyline computation (cont.); (i) result of skyline query

When a skyline query is requested, the first step of HI-Sky compares the GLAD stored in the HashGrid for data space pruning. At this time, to store the non-pruned data space, a GLAD_window is created and the data spaces shown in dotted lines in Figure 3c are pruned. More specifically, as the order of comparison is in accordance with that of GLAD values, 5 (0101), which is the smallest GLAD of HashGrid, is used first for comparison. This compares the GLAD stored in the GLAD_window with the given GLAD read from the HashGrid based on Lemma 1; if Lemma 1 is not satisfied, it is repeatedly compared with subsequent GLADs stored in the GLAD_window. Initially, 5 (0101) is stored in GLAD_window because GLAD_window is empty, and no regions can be pruned using 5 (0101) through Lemma 3. Then, 7 (0111), 8 (1000), and 9 (1001) are compared in order, and are all stored in the
GLAD_window because no GLAD that can prune them is present in GLAD_window. However, in case of 10 (1010), 11 (1011), 14 (1110), and 15 (1111) (excluding 12 (1100) with data point \( H \)), they are not stored in GLAD_window because we can know that all data in these GLADs are dominated by all data in 5 (0101) according to Lemma 1. Therefore, GLAD_window stores only 5 (0101), 7 (0111), 8 (1000), 9 (1001), and 12 (1100), where the skyline is expected to occur as shown in Figure 3d after the completion of Step 1.

Following data space pruning for all GLADs stored in HashGrid, HI-Sky begins the second step, where it computes for the skyline using data from non-pruned GLADs. In this step, the GLAD stored in the GLAD_window is sequentially visited, and the skyline is computed using the data stored in each GLAD. The entropy score minimizes unnecessary comparisons as demonstrated in Lemma 6, and it also monotonically orders the data to enable an efficient skyline computation. Thus, for each piece of data, the entropy score is immediately calculated once the reading is complete. Following entropy score calculation, HI-Sky checks the given GLAD and the GLADs already stored in the Sky_window to determine whether the data belonging to the given GLAD can be dominated by GLADs in the Sky_window. This check is confirmed according to Definition 1 rather than according to Lemma 1 used in data space pruning. If the given GLAD is dominated by the GLAD in Sky_window, there is a possibility of it being dominated by the skyline candidates belonging to the GLAD held in Sky_window, and the skyline candidates stored in the corresponding GLAD and those resulting from the Lemma 6-based comparison are used to confirm dominance. Conversely, if the GLAD of the Sky_window is not a comparison object, HI-Sky checks whether the given GLAD is identical to the GLAD of the Sky_window, and if it is not, HI-Sky moves to the next GLAD in the Sky_window and repeats the process. However, if there is a match, using the characteristics of Lemma 5, the skyline candidate is determined as either dominating or dominated, and the dominated data are removed. If data are not dominated according to the dominance test, they are stored as a skyline candidate in a Sky_window with the same GLAD. The storing is carried out in an aligned manner by entropy score for efficiency of comparison. Therefore, in the second step, data points \( A \) and \( B \) stored in GLAD 5 (0101) are used first, and \( A \) is determined as the skyline of 5 (0101), as shown in Figure 3e. When \( B \) is used first instead of \( A \) and is stored in Sky_window as a skyline candidate, we can see through Lemma 6 that \( A \) is likely to dominate \( B \), and we can thus remove \( B \) and add \( A \) to the list of skyline candidates through comparison. In the subsequent case of GLAD 7 (0111), a comparison with the skyline candidates of GLAD 5 (0101) should be performed because 7 (0111) was dominated by 5 (0101). In this process, data point \( D \) is removed from the comparison because it is dominated by the skyline candidate \( A \) of 5 (0101) as shown in Figure 3f, and there are no more data points in 7 (0111). Thus, it does not have a skyline candidate.

In case of GLAD 8 (1000), as it is not dominated by 5 (0101), it is not necessary to compare it with its skyline candidate. Because dominance has not occurred between GLAD 5 (0101) and GLAD 8 (1000) due to dimension 2, we can skip the dominance test up to GLAD 7 (0111) using Lemma 5. Therefore, data point \( G \), the only data point in GLAD 8 (1000), is defined as the skyline candidate of 8 (1000) as shown in Figure 3g. The data point \( C \) of GLAD 9 (1001) is then dominated and removed by skyline candidate \( A \) belonging to GLAD 5 (0101). The last GLAD of the GLAD_window, 12 (1100), is not dominated by 5 (0101), but is dominated by 8 (1000) through the dominance test. Therefore, it is necessary to perform a comparison with the skyline candidate \( G \) of GLAD 8 (1000), and as data point \( H \) is not dominated by \( G \), it is confirmed as the skyline candidate of GLAD 12 (1100) as shown in Figure 3h. As the GLAD and data stored in GLAD_window no longer exist, Step 2 ends, and \( A \) of GLAD 5 (0101), \( G \) of 8 (1000), and \( H \) of 12 (1100) are determined as skylines and are returned as the results of the skyline query as shown in Figure 3i.

The processing of HI-Sky can be represented by the HI-Sky algorithm shown in Figure 4.
In case of GLAD 8 (1000), as it is not dominated by 5 (0101), it is not necessary to compare it with its skyline candidate. Because dominance has not occurred between GLAD 5 (0101) and GLAD 8 (1000) due to dimension 2, we can skip the dominance test up to GLAD 7 (0111) using Lemma 5. Therefore, data point G, the only data point in GLAD 8 (1000), is defined as the skyline candidate of 8 (1000) as shown in Figure 3g. The data point C of GLAD 9 (1001) is then dominated and removed by skyline candidate A belonging to GLAD 5 (0101). The last GLAD of the GLAD_window, 12 (1100), is not dominated by 5 (0101), but is dominated by 8 (1000) through the dominance test. Therefore, it is necessary to perform a comparison with the skyline candidate G of GLAD 8 (1000), and as data point H is not dominated by G, it is confirmed as the skyline candidate of GLAD 12 (1100) as shown in Figure 3h. As the GLAD and data stored in GLAD_window no longer exist, Step 2 ends, and A of GLAD 5 (0101), G of 8 (1000), and H of 12 (1100) are determined as skylines and are returned as the results of the skyline query as shown in Figure 3i.

The processing of HI-Sky can be represented by the HI-Sky algorithm shown in Figure 4.

**Algorithm HI-Sky**

**Input** $H$: A HashGrid with sorted GLAD.

**Output** $Skyline$: A set of skyline candidates

**Process**

1: /* Step 1: Data space pruning */
2: GLAD_window initialization;
3: FOR EACH (GLAD $g$ in $H$)
4: IF (GLAD_window is empty)
5: GLAD_window.insert($g$);
6: ELSE
7: FOR EACH (GLAD $c$ in GLAD_window)
8: IF ($g$ pruned by $c$)
9: BREAK;
10: IF ($g$ has not been pruned)
11: GLAD_window.insert($g$);
12: /* Step 2: Skyline Computation */
13: Sky_window[GLAD_window.size()] initialization;
14: FOR EACH (GLAD $c$ in GLAD_window)
15: FOR EACH (DATA $p$ in $c$)
16: $p$.entropyScoreCalc();
17: FOR EACH (GLAD $s$ in Sky_window)
18: IF ($s$ dominate $c$)
19: FOR EACH (DATA $sc$ in $s$)
20: IF ($p$ dominated by $sc$)
21: BREAK;
22: IF ($p$.entropyScore < $sc$.entropyScore)
23: BREAK;
24: ELSE IF ($s$ equal $c$)
25: FOR EACH (DATA $sc$ in $s$)
26: IF ($sc$.entropyScore < $p$.entropyScore)
27: IF ($p$ dominated by $sc$)
28: BREAK;
29: ELSE
30: IF ($sc$ dominated by $p$)
31: Sky_window[] remove(sc);
32: ELSE
33: $s$ = skippedGLAD($s$, $c$)
34: IF ($p$ is not dominated)
35: Sky_window[] insert($p$);
36: $Skyline = Sky_window$;
37: RETURN $Skyline$;

Figure 4. The pseudocode of the HI-Sky algorithm.

In line 2, the GLAD_window is initialized, and in lines 3–11, HI-Sky performs data space pruning through a comparison between GLADs. If the GLAD_window is empty, the smallest GLAD will be saved immediately without comparison, as in lines 4 and 5. Otherwise, GLAD_window stores only the undominated GLAD, as shown in line 6–11, by comparing the GLAD read from HashGrid with the GLAD stored in GLAD_window. After finishing the data space pruning, HI-Sky performs the skyline computation on lines 12–35. In line 13, a Sky_window is created for each of the GLADs stored in the GLAD_window. In lines 18–23, a comparison is made between the data belonging to different GLADs. If there is no dominant relation to the GLAD according to line 18, the comparison is not performed, and if Lemma 6 is not satisfied in the comparison process, the comparison with the current Sky_window is ended as shown in lines 22 and 23. In lines 24–31, a comparison is made between data belonging to the same GLAD. At this time, because the data inside the GLAD is not ordered, the comparison is also performed when the input data dominates the skyline candidate according to lines 29–31. However, if dominance does not occur, and it is also not the same GLAD, the dominance test can be skipped according to Lemma 5 as shown in lines 32 and 33. Finally, the undominated data is stored in the Sky_window as in lines 34 and 35. Once all data belonging to the GLAD in GLAD_window has been compared, the entire dataset stored in the Sky_window is returned to the user as a skyline query result as in lines 36 and 37.
4. Performance Evaluation

The results of the performance evaluation of HI-Sky are detailed in this section. The experimental environment is described first, and is followed by experimental results along with a description of the state-of-the-art methods used for comparison against our method.

4.1. Experimental Environment

Datasets are important for the evaluation of skyline query methods. Therefore, to determine a fair performance, datasets were generated using the generator proposed by Borzsonyi et al. [15]. To simulate various scenarios, we generated various distributions, such as anti-correlated (correlation: −0.5), independent, and correlated (correlation: 0.5) datasets. Each dataset consists of four, eight, twelve, and sixteen dimensions. All datasets had 10 K (ten thousand), 100 K (one hundred thousand), 1 M (one million), and 10 M (ten million) data points to demonstrate the scalability of the proposed method. For experiments, we separated datasets into two types, one of which contains integer values whereas the other contains real values. Integer datasets used a range of [0, 1023] whereas real value datasets used a range of [0, 1] with up to 10 decimal places. Lastly, we also performed this experiment with a real dataset called the Gas Sensors for home activity monitoring [34]. This dataset has a size of 919 k and contains ten dimensions of real values. For real dataset experiments, we normalized the dataset to real values in the range of [0, 1].

To evaluate our proposed method, we first experimented with the performance change of our method according to the np, and then we compared it with the state-of-the-art skyline query methods SFS [16,17], BBS [6,7], and Z-SKY [10,11]. Traditionally, BBS is implemented using R-tree that constructs the overlapping structure. As the proposed method is a non-overlapping structure, we additionally performed experiments on quadtree-based BBS (BBS-Q). These experiments consisted of three factors of assessment as follows: indexing time, skyline computation time, and the number of dominance test calls. The indexing time refers to the total time taken to generate an index when data are entered sequentially. Skyline computation time shows the total time taken to request and complete the skyline query. The number of dominance test calls shows the dominance tests performed during the skyline query. HI-Sky, BBS-R, BBS-Q, and Z-SKY compute the skyline through indexing, whereas SFS does not use indexing. Therefore, indexing time was used only for HI-Sky, BBS-R, BBS-Q, and Z-SKY, whereas skyline computation time was measured by including preprocessing time. In other words, the SFS performs preprocessing to calculate entropy scores and sort the data according to the entropy score, and the HI-Sky performs preprocessing for data space pruning.

All methods were implemented using VC++ 12.0, and the experiments were carried out on an Intel Core i7-6700 3.4 GHz processor with 64-bit Windows 10 and 16 GB of main memory. The original data and the index structure used in the experiments were all placed in the main memory to mitigate fluctuations in performance due to the influence of disk access speed. Finally, the maximum capacity of nodes for BBS-R, BBS-Q, Z-SKY, and HI-Sky was fixed at 64, and the np of HI-Sky was used as the maximum value obtained from Equation (1) when half of the maximum capacity was the minimum capacity.

4.2. Result of Experiments

In this subsection, we will show the result of experimental evaluation on varying aspects such as performance changes of HI-Sky by the np, indexing time, and skyline computation time of each method.

4.2.1. Comparison of Changes in np

We first measured performance changes of HI-Sky such as indexing time, skyline computation time, and the number of dominance test calls according to the np. This experiment was conducted using a varying np based on the eight-dimensional independent data on 1 M (one million) and 10 M (ten million).
Figure 5 shows the results of performance changes of HI-Sky according to varying np. In this case, the maximum number of np that can be obtained from Equation (1) is 4 for the 1 M dataset and 5 for the 10 M dataset. The experimental result shows that the maximum number of np does not guarantee optimal performance, but it can work properly as a criterion to prevent performance degradation because the small np reduces the benefit of the data space pruning step as it creates a smaller number of GLADs and stores more data in each GLAD. On the other hand, a large np reduces the benefit of the skyline computation step because it raises the number of GLADs created to be closer to the number of data and performs more operations to determine whether a comparison between GLADs is required. However, the maximum np can be used to store the number of data close to the minimum capacity, thus reducing the number of comparisons between GLADs while properly pruning the data space.

Figure 5. The performance changes according to varying np.

4.2.2. Comparison of Indexing Time

We compared the indexing time of HI-Sky, BBS-R, BBS-Q, and Z-SKY. The experiments were conducted by varying the data type, dimensionality, and data distribution for various scenarios. We also show that the proposed method (i.e., HashGrid, the index structure of HI-Sky) can work well when data points are frequently inserted or deleted from the datasets.

Figure 6 shows the results in terms of indexing time for each data type and distribution. The experimental results show that HashGrid was faster than the R-tree of BBS-R or the ZBtree of Z-SKY because both needed to repeatedly change the structures of their trees to maintain optimal conditions for computing. In particular, the optimal RZ-region of ZBtree was computed using a split process to maintain the optimum RZ-region. However, the cost of computing the optimal RZ-region required more computation than a simple partitioning of the R-tree or quadtree. Thus, the indexing time was longer overall. By contrast, the hash bucket search of HashGrid and storage in the hash bucket were reduced to a time complexity of O(1), which enabled fast data storage. Further, as the hash table did not structurally change due to the data input, there was no additional cost to maintain the optimal structure. However, HashGrid was slower than the quadtree of BBS-Q above twelve dimensions. This is because the asymmetric tree structure of the quadtree can be relatively inexpensive to maintain. However, HashGrid increases the number of buckets generated due to the high-dimensionality, which increases the cost of managing them.
reduced to a time complexity of $O(1)$, which enabled fast data storage. Further, as the hash table did not structurally change due to the data input, there was no additional cost to maintain the optimal structure. However, HashGrid was slower than the quadtree of BBS above twelve dimensions. This is because the asymmetric tree structure of the quadtree can be relatively inexpensive to maintain. However, HashGrid increases the number of buckets generated due to the high-dimensionality, which increases the cost of managing them.

(a) correlated (integer)  
(b) correlated (real value)  
(c) independent (integer)  
(d) independent (real value)  
(e) anti-correlated (integer)  
(f) anti-correlated (real value)

Figure 6. Indexing time (1 Million).

Figure 7 shows the processing time when data is repeatedly inserted or deleted into the HashGrid. The experiments were conducted using varying dimensions and data distribution. In the case of deletion, Figure 7a,c,e show the consumed time when 10 K data are deleted 20 times consecutively from HashGrid composed of 1 M dataset. In the case of insertion, Figure 7b,d,f show the consumed time when inserting 10 K data 20 times consecutively into HashGrid composed of 800 K dataset. The result
of experiments demonstrates that for skewed data (i.e., correlated or anti-correlated), the time taken for the deletion process was relatively increased due to a large number of data stored in specific GLADs in a higher dimension. However, we did not observe a significant change in the overall performance of HashGrid caused by the volume of insertion and deletion operations. This is because HashGrid is a hash index-based approach that is not a volume-sensitive, unlike tree structure-based approaches [35].

![Graphs showing processing times for deletion and insertion](image.png)

Figure 7. Results of HashGrid maintenance performance on dynamic dataset.

4.2.3. Comparison of Skyline Computation Time

The third experiment measured the time required to complete the skyline query to determine the performance of skyline computation. In this case, the SFS required preprocessing to calculate entropy score and perform sorting, and HI-Sky used preprocessing to perform data space pruning.
Therefore, because the SFS and HI-Sky performed preprocessing, skyline computation time included preprocessing time, whereas because BBS-Q, BBS-R, and Z-SKY did not require preprocessing, only the time required for skyline computation was included. We experimented with dimensionality and cardinality using varying data distributions.

Figure 8 shows the results of skyline computation times for each dimension and distribution on the log scale. The experimental results show that the BBS-Q and BBS-R yielded the best performance in the case of independent and correlated data with four dimensions. Because the MBR of the BBS-R and the bucket of the BBS-Q have a hierarchical inclusion relationship, it was possible to remove large amounts of data even with a small number of comparisons. However, in the case of BBS-R, as the number of dimensions increased, the problem of overlapping among the MBRs increased significantly and, owing to a lack of monotonic ordering, the efficiency of the dominance test was low. In the case of BBS-Q, 2^d buckets are created for each split, and the creation of child buckets exceeding than the capacity of the bucket results in empty buckets, which reduces the efficiency of the dominance test. Therefore, as the number of dimensions increased, skyline computation in the BBS-R and BBS-Q were drastically affected. However, in the case of the SFS and HI-Sky, as the dominance test was performed effectively by utilizing the characteristics of monotonic ordering, comparatively small changes in performance occurred even when the number of dimensions increased. Meanwhile, in the case of HI-Sky, the dominance test was not performed when dominance did not occur because GLAD was used, which rendered HI-Sky more effective than the SFS.

Lastly, Z-SKY is demonstrated only using integers because it depends heavily on the data type and data range due to the Z-Address. Z-SKY demonstrated poor performance in most cases as a result of three identifiable problems. First, the ZBtree has poor performance because of its repeated dominance tests from root to leaf and its performance sensitivity to node capacity. Z-SKY uses the ZBtree to store the skyline in a similar manner to indexing, but this requires costly tree construction and maintenance, as mentioned in the second experiment. Second, Z-SKY performs a dominance test on the RZ-regions of each node. This test is performed every time new data is entered, starting from the root, until it is dominated or selected as skyline. In this process, child nodes of the RZ-region that need to be compared are repeatedly stored in queue and sequential comparison with them is performed in turn. The data selected as the skyline need to be compared with all RZ-regions or candidate skylines in the queue, which often requires more comparisons than performing a comparison using actual data with monotonic ordering. Third, depending on the node’s capacity, the height of the ZBtree and the maximum number of child nodes stored in the queue changes, which has a significant effect on overall performance. Therefore, choosing an incorrect capacity can significantly affect the overall performance, and it is difficult to select the optimum capacity in advance.

Figure 9 shows the results of skyline computation times for each cardinality and distribution on the log scale using an eight-dimensional dataset. For HI-Sky, to check only the effect due to cardinality, np is set to four, which is that of the default dataset (8 D, 1 M). The experimental results show that the SFS yielded the best performance in the case of independent and correlated data with a 10 K dataset. This is because a small number of data has a small number of skylines, where the sequential comparison is more efficient than using other structures. On the contrary, in the case of an anti-correlated dataset where more skylines occur, HI-Sky is faster than SFS. Additionally, indexing methods based on tree structures have a greater increase in computation time with increased data. This is because the number of nodes increases with the increase of data, and the number of nodes to be stored in the heap/queue increases for the skyline computation. The number of data stored in the GLAD also increases as the data increases when using HI-Sky, but because the comparison is performed only when dominance can occur using the GLAD, the increase of the skyline computation time is more directly correlated with the increase in the number of data.
4.2.4. Comparison of a Number Dominance Test

The fourth experiment measured the number of dominance tests in the skyline query. Using these experiments, we compared the performance of the methods through common dominance test calls that have a direct impact on performance for various skyline query methods and therefore have an absolute criterion for query processing time. Finally, we experimented by varying the dimensionality and data distributions to compare various scenarios.
Figure 9 shows the results of the number of dominance test calls for each dimensionality and data distribution on the log scale. The experimental results show that SFS and HI-Sky featured fewer dominance test calls than the other methods. These results show that dominance test calls in the MBR, bucket, and RZ-region were not few and, as the number of dimensions increased, the sparsity of the data increased to such an extent that they were not easily dominated. Therefore, an increase in the number of dimensions led to more dominance test calls. Conversely, in the case of SFS, a dominance test between the skyline and actual data was conducted immediately, and thus there was no need for unnecessary comparisons as in the case of dominance tests between MBRs. This was also true for
HI-Sky. However, as HI-Sky removed data that could not be skylines through data space pruning, it operated with fewer data and made fewer calls than the SFS. Moreover, when data were incomparable due to the use of GLAD, dominance test was not performed, which drastically reduced the number of calls. In particular, the efficiency of incomparability is prominently shown in Figure 10e,f, which show the experimental results using anti-correlated data. Anti-correlated data are characterized by a higher number of skylines than other types of data, and also have smaller data spaces or number of actual data points that each skyline can dominate. Therefore, if anti-correlated data are used, the SFS, BBS-R, and BBS-Q make more dominance calls. Meanwhile, Z-SKY can calculate dominance by using the dominance relation of the RZ-region, whereas HI-Sky can determine dominance by using GLAD. Therefore, they conducted fewer dominance tests only when the given data could be dominated, so fewer calls were made than in other methods.

Figure 10. The number of dominance tests according to dimensionality (1 Million).
Figure 11 shows the results of the number of dominance test calls for each cardinality and distribution on the log scale using an eight-dimensional dataset. The experimental results show that the number of dominance test calls increases as the cardinality increases. However, BBS-R and BBS-Q showed a higher rate of increase than other methods. This is because the number of nodes increases with the increase of data, and it implies more comparisons for the skyline computation. In case of Z-SKY, unlike the previous method, there is relatively little increase because it is a structure that does not cause node overlap and unnecessary node creation as in R-tree and quadtree. On the other hand, SFS or HI-Sky responds less sensitively to changes in cardinality because the structure, such as the height of the tree and the number of nodes, does not change even if the data increases.

Figure 11. The number of dominance tests according to cardinality.
Finally, as described above, because HI-Sky eliminated unnecessary dominance tests by data space pruning and by Lemma 5, it performs skyline computation with the smallest number of dominance tests in most scenarios.

4.2.5. Comparisons Using Real-World Dataset

The last experiment demonstrated the real world application of HI-Sky using a real world dataset called Gas Sensors for home activity monitoring [34]. The dataset is freely available at UCI Machine Learning Repository. In addition, Z-SKY was excluded from the experiment because the data is real valued data, and normalization into an integer type causes a large number of skylines to disappear.

Figure 12 shows the results of performance for each method. In this case, the real world dataset has correlations between specific dimensions and has clustered data distribution. Therefore, BBS-R and BBS-Q show better performance in skyline computation time than SFS in this experiment. However, in case of the dominance test calls, the comparison between the nodes occur more than the SFS as in the previous experiments. However, in all cases, HI-Sky performed better than other methods because the distribution of the real world dataset greatly influenced by the pruning of data space and the elimination of unnecessary dominance test between the GLADs.

![Figure 12. The experimental results using a real dataset.](image)

5. Conclusions

In this paper, we proposed a hash index structure and a special hash key for a skyline query method called HI-Sky, which reduces unnecessary dominance tests through a hash index and grid partitioning-based approach. For this purpose, we showed the theoretical background of HashGrid, a new hash index, and GLAD, a hash key used in the corresponding index structure. The theoretical background of HI-Sky, which efficiently computes the skyline using HashGrid, was presented. It was also tested through experiments against state-of-the-art methods. The results showed that HI-Sky can generate indices faster than other methods and can perform faster skyline query processing at higher dimensions by effectively reducing the number of dominance tests. This is because HI-Sky has an indexing method called HashGrid and a hash key called GLAD that enable data space pruning where a skyline cannot occur, eliminating unnecessary dominance tests when dominance cannot occur. Therefore, HI-Sky can process skyline queries more efficiently than other methods. To solve the data skewness problem, in the future work, we are planning to apply a method like box-counting [36]. The box-counting determines the data skewness based on number of grids that are penetrated by data points. This feature can be utilized in HI-Sky to make it choose the best np by itself. In addition, we are planning to utilize a system like SpatialHadoop [25,26] to extend HI-Sky into a distributed environment.
Author Contributions: J.-H.C. designed the algorithm. J.-H.C. also performed the bibliographic review and writing of the draft and developed the proposed algorithm. A.N. shared his expertise with regard to the overall review of this paper. A.N. and F.H. supervised the entire process. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2017R1D1A3B03035729). This work was also supported by the Industrial Strategic Technology Development Program (No.20003991, Development of Korean Wave Convergence Service for AI-based Motion Evaluation and Learning Technology) funded by the Ministry of Trade, Industry & Energy (MOTIE, Korea).

Conflicts of Interest: The authors declare no conflict of interest.

References
1. Chomicki, J.; Ciaccia, P.; Meneghetti, N. Skyline queries, front and back. ACM SIGMOD Rec. 2013, 42, 6–18. [CrossRef]
2. Cui, B.; Lu, H.; Xu, Q.; Chen, L.; Dai, Y.; Zhou, Y. Parallel distributed processing of constrained skyline queries by filtering. In Proceedings of the IEEE 24th International Conference on Data Engineering, Cancun, Mexico, 7–12 April 2008; pp. 546–555.
3. Nasridinov, A.; Ihm, S.-Y.; Park, Y.-H. Skyline-based aggregator node selection in wireless sensor networks. Int. J. Distrib. Sens. Netw. 2013, 9, 356194. [CrossRef]
4. Skoutas, D.; Sacharidis, D.; Simitsis, A.; Sellis, T. Ranking and clustering web services using multicriteria dominance relationships. IEEE Trans. Serv. Comput. 2010, 3, 163–177. [CrossRef]
5. Park, Y.; Min, J.-K.; Shim, K. Parallel computation of skyline and reverse skyline queries using mapreduce. Proc. VLDB Endow. 2013, 6, 2002–2013. [CrossRef]
6. Papadias, D.; Tao, Y.; Fu, G.; Seeger, B. An optimal and progressive algorithm for skyline queries. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA, 9–12 June 2003; pp. 467–478.
7. Lin, X.; Xu, J.; Hu, H.; Lee, W.-C. Authenticating location-based skyline queries in arbitrary subspaces. IEEE Trans. Knowl. Data Eng. 2014, 26, 1479–1493. [CrossRef]
8. Guttman, A. R-trees: A dynamic index structure for spatial searching. SIGMOD Rec. 1984, 14, 47–57. [CrossRef]
9. Berchtold, S.; Keim, D.; Kriegel, H. An index structure for high-dimensional data. In Readings in Multimedia Computing and Networking; Morgan Kaufmann: Burlington, MA, USA, 2001; pp. 451–462.
10. Lee, K.C.; Lee, W.-C.; Zheng, B.; Li, H.; Tian, Y. Z-sky: An efficient skyline query processing framework based on z-order. Int. J. Very Large Data Bases 2010, 19, 333–362. [CrossRef]
11. Lee, K.; Zheng, B.; Li, H.; Lee, W.-C. Approaching the skyline in Z order. In Proceedings of the 33rd International Conference on Very large data Bases, Vienna, Austria, 23–27 September 2007; pp. 279–290.
12. Gaede, V.; Günther, O. Multidimensional access methods. ACM Comput. Surv. 1998, 30, 170–231. [CrossRef]
13. Lawder, J.K.; King, P.J.H. Using Space-Filling Curves for Multi-dimensional Indexing. In Proceedings of the 17th British National Conference on Databases, Exeter, UK, 3–5 July 2000; pp. 20–35.
14. Zhang, S.; Mamoulis, N.; Cheung, D.W. Scalable skyline computation using object-based space partitioning. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 29 June–2 July 2009; pp. 483–494.
15. Borzsonyi, S.; Kossmann, D.; Stocker, K. The Skyline operator. In Proceedings of the 17th International Conference on Data Engineering, Heidelberg, Germany, 2–6 April 2001; pp. 421–430.
16. Chomicki, J.; Godfrey, P.; Gryz, J.; Liang, D. Skyline with presorting. In Proceedings of the 19th International Conference on Data Engineering, Bangalore, India, 5–8 March 2003; pp. 717–719.
17. Chomicki, J.; Godfrey, P.; Gryz, J.; Liang, D. Skyline with presorting: Theory and optimizations. Intell. Inf. Process. Web Min. 2005, 31, 595–604.
18. Vlachou, A.; Doukeridis, C.; Kotidis, Y. Angle-based space partitioning for efficient parallel skyline computation. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, 9–12 June 2008; pp. 227–239.
19. Zhang, B.; Zhou, S.; Guan, J. Adapting Skyline Computation to the MapReduce Framework: Algorithms and Experiments. In Proceedings of the 16th International Conference on Database Systems for Advanced Applications, Hong Kong, China, 22–25 April 2011; pp. 403–414.

20. Chen, L.; Hwang, K.; Wu, J. MapReduce Skyline Query Processing with a New Angular Partitioning Approach. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Shanghai, China, 21–25 May 2012; pp. 2262–2270.

21. Han, X.; Li, J.; Yang, D.; Wang, J. Efficient Skyline Computation on Big Data. IEEE Trans. Knowl. Data Eng. 2013, 25, 2521–2535. [CrossRef]

22. Park, Y.; Min, J.-K.; Shim, K. Efficient Processing of Skyline Queries Using MapReduce. IEEE Trans. Knowl. Data Eng. 2017, 29, 1031–1044. [CrossRef]

23. Islam, M.S.; Liu, C.; Rahayu, W.; Anwar, T. Q+Tree: An Efficient Quad Tree based Data Indexing for Parallelizing Dynamic and Reverse Skylines. In Proceedings of the 25th ACM International Conference on Information and Knowledge Management, Indianapolis, IN, USA, 24–28 October 2016; pp. 1291–1300.

24. Tang, M.; Yu, Y.; Aref, W.G.; Malluhi, Q.M.; Ouzzani, M. Efficient Parallel Skyline Query Processing for High-Dimensional Data. IEEE Trans. Knowl. Data Eng. 2018, 30, 1838–1851. [CrossRef]

25. Eldawy, A.; Mokbel, M.F. SpatialHadoop: A mapreduce framework for spatial data. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, Seoul, Korea, 13–17 April 2015; pp. 1352–1363.

26. Pertesis, D.; Doulkeridis, C. Efficient skyline query processing in spatialhadoop. Inf. Syst. 2015, 54, 325–335. [CrossRef]

27. Bayer, R.; McCreight, E. Organization and maintenance of large ordered indexes. In Software Pioneers; Springer: Berlin/Heidelberg, Germany, 2002; pp. 245–262.

28. Jensen, C.; Lin, D.; Ooi, B. Query and update efficient B+-tree based indexing of moving objects. In Proceedings of the 30th International Conference on Very Large Data Bases, Toronto, Canada, 31 August–3 September 2004; pp. 768–779.

29. Lee, M.L.; Hsu, W.; Jensen, C.S.; Cui, B.; Teo, K.L. Supporting frequent updates in R-trees: A bottom-up approach. In Proceedings of the 29th International Conference on Very Large Data Bases, Berlin, Germany, 9–12 September 2003; pp. 608–619.

30. Lehman, T.J.; Carey, M.J. A Study of Index Structures for Main Memory Database Management Systems. In Proceedings of the 12th International Conference on Very Large Data Bases, Kyoto, Japan, 25–28 August 1986; pp. 294–303.

31. Song, Z.; Roussopoulos, N. Hashing moving objects. In Proceedings of the International Conference on Mobile Data Management, Hong Kong, China, 8–10 January 2001; pp. 161–172.

32. Ihm, S.-Y.; Nasridinov, A.; Park, Y.-H. Grid-PPPS: A Skyline Method for Effectively Handling Top-k Queries in Internet of Things. J. Appl. Math. 2014, 2014, 1–10. [CrossRef]

33. Rocha-Junior, J.B.; Vlachou, A.; Doulkeridis, C.; Nørvåg, K. AGiDS: A grid-based strategy for distributed skyline query processing. In Proceedings of the International Conference on Data Management in Grid and P2P Systems, Linz, Austria, 1–2 September 2009; pp. 12–23.

34. Huerta, R.; Mosqueiro, T.; Fonollosa, J.; Rulkov, N.F.; Rodriguez-Lujan, I. Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring. Chemom. Intell. Lab. Syst. 2016, 157, 169–176. [CrossRef]

35. Gani, A.; Siddiqa, A.; Shamshirband, S.; Hanum, F. A survey on indexing techniques for big data: Taxonomy and performance evaluation. Knowl. Inf. Syst. 2015, 46, 241–284. [CrossRef]

36. Belussi, A.; Migliorini, S.; Eldawy, A. Detecting skewness of big spatial data in SpatialHadoop. In Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, New York, NY, USA, 6–9 November 2018; pp. 432–435.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).