A fast and progressive algorithm for skyline queries with totally- and partially-ordered domains

Hyungsoo Jung\(^a\), Hyuck Han\(^a\), Heon Y. Yeom\(^a\), Sooyong Kang\(^b,\ast\)

\(^a\) School of Computer Science and Engineering, Seoul National University, Seoul 151-742, Republic of Korea
\(^b\) Division of Computer Science and Engineering, Hanyang University, Seoul 133-791, Republic of Korea

**ABSTRACT**

We devise a skyline algorithm that can efficiently mitigate the enormous overhead of processing millions of tuples on totally- and partially-ordered domains (henceforth, TODs and PODs). With massive datasets, existing techniques spend a significant amount of time on a dominance comparison because of both a large number of skyline points and the unprogressive method of skyline computing with PODs. (If data has high dimensionality, the situation is undoubtedly aggravated.) The progressiveness property turns out to be the key feature for solving all remaining problems. This article presents a FAST-SKY algorithm that deals successfully with these two obstacles and improves skyline query processing time strikingly, even with high-dimensional data. Progressive skyline evaluation with PODs is guaranteed by new index structures and topological sorting order. A stratification technique is adopted to index data on PODs, and we propose two new index structures: stratified R-trees (SR-trees) for low-dimensional data and stratified MinMax treaps (SM-treaps) for high-dimensional data. A fast dominance comparison is achieved by using a reporting query instead of a dominance query, and a dimensionality reduction technique. Experimental results suggest that in general cases (anti-correlated and uniform distributions) FAST-SKY is orders of magnitude faster than existing algorithms.

© 2009 Elsevier Inc. All rights reserved.

**1. Introduction**

The database research community has recently given considerable attention to skyline computation, particularly for progressive algorithms that can immediately return intermediate skyline results. Using the common definition in the literature, given a set of objects \(p_1, p_2, \ldots, p_n\), the skyline operator returns all objects \(p_i\) such that \(p_i\) is not dominated by another object \(p_j\). The underlying query type in skyline computation is known to be a dominance query, which requires a series of pairwise comparisons.

Since the skyline operator was formally introduced in Börzsönyi et al. (2001), many researchers have devised novel algorithms for skyline queries in various environments (see Balke et al., 2004; Papadias et al., 2005; Chan et al., 2005; Sacharidis et al., 2009; Kossmann et al., 2002; Pei et al., 2006; Wang et al., 2007; Lee et al., 2007; Lian and Chen, 2008; Vlachou et al., 2008). Most skyline evaluation methods developed recently are aimed at efficient computation with TODs, and we have seen remarkable progress in that area. On the other hand, skyline computation with TODs and PODs has remained in a premature state since it was first tackled by Chan et al. (2005) until recent work by Sacharidis et al. (2009) advances the performance of skyline processing with PODs using topological sort.

The performance of skyline computation is mainly affected by two activities. The first one, which has already been researched extensively, is the IO overhead of accessing data. Many researchers think that reducing the number of IO accesses to a disk would have a great benefit, and lots of excellent algorithms showed substantial improvements. However, the IO overhead becomes less meaningful as data increases to millions of records. Rather, the CPU cost of carrying out a series of dominance checks with existing skyline points becomes the major factor in performance overhead, which is the second activity. Existing skyline techniques do not fully take the second type of overhead into consideration, and as a result, leave much room for further enhancement. In addition, if data has high dimensionality, the entire overhead grows nontrivially; in particular, skyline processing with high-dimensional data on PODs has not yet been researched thoroughly.

The main focus of this paper is on developing a fast and progressive algorithm for skyline queries with TODs and PODs. We propose the FAST-SKY algorithm that has the progressiveness property with TODs and PODs, and as a result, FAST-SKY can use a reporting query in dominance comparisons to tremendous effect.
In order to achieve IO-optimality\(^1\) and progressiveness (Tan et al., 2001) in skyline computation with PODs, we adopt a stratification technique and design two new index structures, each of which has merits and demerits. The stratification that is applied to PODs through domain classification, which partitions data into disjoint sets according to a value from POD, enables the FAST-SKY algorithm to perform IO-optimal and progressive skyline evaluation.

The new index structures are stratified R-trees (SR-trees) and stratified MinMax treaps (SM-treaps). Due to the stratification, both index structures are not concerned with attributes in PODs, only attributes in TODs. An SR-tree considers full d-dimensionality of data, while an SM-treap sees only two dimensions, \(\text{min}\) and \(\text{max}\) coordinates, when indexing data. An SR-tree has an advantage in that it supports all types of progressive evaluations such as arbitrary dimensionality. The drawback, however, is the severe performance degradation of the R-tree that indexes high-dimensional data (\(d > 10\)). To overcome the performance deterioration, an SM-treap enforces dimensionality reduction on multidimensional data in such a way that data is represented by two coordinates. It then inserts data into a treap with \(\min\) (key) and \(\max\) (priority) values. The merits of an SM-treap are twofold. An SM-treap can index high-dimensional data and support progressive computation, and unlike the R-tree-based algorithm (i.e., BBS (Papadias et al., 2005)), it does not require a heap space to buffer skyline candidates because the preorder traversal of an SM-treap guarantees progressive processing. Eliminating the heap overhead by guaranteeing progressive processing, which was discussed in ZBtree (Lee et al., 2007) using data on an integer domain (TODs only), contributes to a substantial performance gain in our problem domain as well (real TODs and PODs). To the best of our knowledge, this is the first work on space efficiency and the progressiveness property of an SM-treap for high-dimensional data in a real domain.

For an efficient dominance comparison, we use the well-known property in orthogonal range searching that if an intermediate skyline set is an insert-only set, then a dominance check can be realized by using a reporting query, instead of a dominance query. In other words, if skyline computation is progressive, then a dominance check can be optimized directly by the above property. Because of the progressiveness property, FAST-SKY can also exploit a reporting query. To deal with high-dimensional skyline points as well, we reduce all intermediate skyline points in 2-dimensional space by using a dimensionality reduction technique; we adopt and modify the iMinMax technique (Yu et al., 2004). Performing a dominance check using a reporting query on the reduced dimensions improves the entire query processing time significantly.

The main contributions of this paper follow:

- We address the problem of skyline computation with TODs and PODs even in high-dimensional space. We also note that the CPU cost of a dominance test becomes the major factor in performance overhead as datasets grow over millions of records.
- For progressive skyline evaluation with PODs, we design two new index structures: SR-trees and SM-treaps, each of which has strengths and weaknesses when used for different purposes. In particular, SM-treaps resolve progressive skyline processing with PODs in high dimensional space efficiently.
- We achieve a fast dominance test by using a reporting query on reduced dimensions. An IMax based B+tree with novel pruning techniques decreases the cost of dominance tests.

The rest of the paper is organized as follows: Section 2 reviews the related work in further detail and discusses its advantages and limitations. Section 3 introduces fundamental definitions. Section 4 gives an in-depth explanation of the FAST-SKY algorithm, and Section 5 experimentally evaluates FAST-SKY and compares it to SDC+ in a variety of settings. Finally, Section 6 concludes the article and describes directions for future work.

2. Related work

Previous work on algorithms for computing skyline queries can be grouped into two categories, namely non-index-based (e.g., block nested loop (Börzsönyi et al., 2001) and divide & conquer (Börzsönyi et al., 2001)) and index-based (e.g., B-tree-based scheme (Börzsönyi et al., 2001), R-tree-based schemes (Börzsönyi et al., 2001; Papadias et al., 2005), bitmap (Tan et al., 2001), and index (Tan et al., 2001)). Typically, the index-based approaches outperform the non-index-based approaches. They can also be guaranteed to return answers progressively without scanning whole datasets. This property is called “progressiveness” (Tan et al., 2001). Lin et al., 2005 studied efficient skyline computation over a data stream from the viewpoint of a reporting query. Under stream insert/delete conditions, they exploit the append-only (Babcock et al., 2002) feature of a data stream to compute Skylines using a reporting query.

---

1 An algorithm performs a single access to those nodes that may contain skyline points (Papadias et al., 2005).
2 It is also called an append-only property in Babcock et al. (2002).
Much research work focuses on the preference query processing, which is related to PODs in our work. In Agrawal and Wimmers (2000), a framework that processes quantitative preference queries was proposed. Cohen (1997) proposed a two-stage approach in which one first learns a preference function by conventional means. In Kießling (2002), a framework that processes qualitative preference queries was proposed. Skyline queries with PODs (Chan et al., 2005; Sacharidis et al., 2009) are similar to Pareto preference queries (Kießling, 2002; Kießling and Köstler, 2002). LatticeSky (Morse et al., 2007) transforms multiple low-cardinality domains into two integer-domain (interval) and (ii) maintain the transformed attributes using an existing indexing scheme (R-tree or its family) and compute the skyline extending BBS. To transform a POD, a directed-acyclic graph (DAG), which represents original ordering information, is reduced to a spanning tree as shown in Fig. 1a and b. The reduced spanning tree is lossy since multiple edges (much ordering information) might be removed from the original DAG during the construction of the spanning tree, and such loss leads to false positives in the current set of skyline points.

BBS+ is a direct branch of BBS. However, since there may be false positives in the current set of skyline points, the UpdateSkyline function in BBS+ should be able to detect and remove them. In other words, if a query point dominates some points in the current skylines, the dominated points are removed immediately, and the query point is inserted into the intermediate skyline set.

To avoid unnecessary dominancechecking, SDC partitions the data into two strata by introducing the dominance graph. Points that could immediately be placed in the skyline are grouped into stratum 1, and points that might be false positives are grouped into stratum 2. By doing so, SDC outperforms BBS+ in terms of progressiveness and speed in most cases.

SDC+ increases the progressiveness further than SDC by defining and using the uncovered level of the attribute value. It organizes data into two or more strata exploiting the dominance graph and the uncover-level. In SDC+, one R-tree is built per stratum. The skyline is now computed a single stratum at a time from $R_0$ to $R_n$, and the sequence of $R_i$ systematically guarantees that each “local” skyline point in a stratum $R_i$ cannot dominate any skyline point in previous strata. In other words, skyline points obtained from the stratum $R_{i-1}$ can be returned immediately before the computing skyline in the stratum $R_i$. However, SDC+ does not have complete progressiveness since there might be false positives in the “local” skyline points.

Recently, Sacharidis et al. (2009) proposed a different approach, called the TSS algorithm, for skylines with PODs. The fundamental motivation of the TSS algorithm is to guarantee complete progressiveness with which BBS+, SDC and SDC+ do not provide in processing data on PODs. Like BBS+, SDC and SDC+, TSS is based on the BBS algorithm. The difference is that when TSS builds the R-tree-based index structure, it stores the topological sorting value of each node in a DAG, instead of the interval value used in BBS+, SDC and SDC+. For the dominance-check operation, the $t$-dominance-check technique is devised. The $t$-dominance-check uses propagated intervals as well as scanning tree-based intervals to capture all preferences. Propagated intervals are used to make up for lossy ordering information arose from BBS+, SDC and SDC+ algorithms. For example, the propagated interval of the node $d$ in Fig. 1c is [4,4] that correctly captures the lossy ordering information between nodes $d$ and $j$ in Fig. 1a. Both topological sort and the $t$-dominance-check guarantee IO-optimality and complete progressiveness. However, TSS requires more indexing time for computing propagated intervals of every leaf and internal node of an R-tree, and propagated intervals incur more IO overhead. Moreover, TSS has a less pruning effect of branch-and-bound style algorithms because internal nodes in an R-tree might have many intervals.

In summary, SDC+ is more progressive than SDC, while BBS+ is not progressive at all. SDC+ shows the best performance by minimizing unnecessary dominance checking. However, these algorithms are still not optimal in terms of the number of I/O operations and dominance checks. They are also vulnerable to a change in the standard unit in TODs or an order-relation in PODs, as well as to the addition of a new value in PODs. A change in order-relation and the addition of a new value involve a change in the interval-based transformation, and even a small alteration in a transformation may lead to re-indexing a significant portion of the data points that have already been indexed. A change in a standard unit may also modify an existing R-tree structure significantly. TSS is completely progressive and can cope with dynamic changes of PODs without rebuilding an R-tree-based index structure.

3. Preliminaries

This section introduces some notations and definitions we use in the rest of the paper. All data points are defined on the set of
attributes \( \mathcal{A} = \{ A_1, A_2, \ldots, A_n \} \), where \( \mathcal{A} = \mathcal{A}_{\text{total}} \cup \mathcal{A}_{\text{partial}} \) and where \( \mathcal{A}_{\text{total}} \) and \( \mathcal{A}_{\text{partial}} \) denote, respectively, the subset of totally-ordered and partially-ordered attributes.

**Definition 1** (Partially-ordered set = poset). For each attribute \( A_i \in \mathcal{A}, \langle D_i, \preceq \rangle \) denotes the poset for its domain values \( D_i \). Each \( \preceq \) is a reflexive,3 antisymmetric, and transitive binary relation on \( D_i \).

**Definition 2** (Dominance). Given \( x, y \in D_i, x \) and \( y \) are said to be comparable if either \( y \preceq x \) or \( x \preceq y \); otherwise, they are said to be incomparable. We say that \( x \) dominates \( y \) if \( y \preceq x \).

**Definition 3** (m-Dominance (Chan et al., 2005)). Given \( x, y \in D_i, x \) is said to \( m \)-dominate \( y \) if \( y \cdot A_i \preceq x \cdot A_i \) for each \( A_i \in \mathcal{A}_{\text{total-partial}} \).

**Definition 4** (A range reporting problem (Chazelle, 1990a)). Given a collection of \( n \) points in \( d \)-dimensional space and a box \([a_1, b_1] \times \cdots \times [a_d, b_d]\), report every point whose \( i \)-th coordinate lies in \([a_i, b_i]\), for each \( i = 1, \ldots, d \).

**Definition 5** (A dominance search problem (Chazelle, 1990b)). Given a collection of \( n \) weighted points in \( d \)-dimensional space and a query point \( q \), compute the cumulative weight of the points dominated (in all coordinates) by \( q \).

### 4. Fast and progressive skyline algorithm

Before we delve into the details, we would like to state that the skyline considered in this paper is the \( \min \) operator for all attributes. In this section, key motivations are discussed in Section 4.1. We then describe the detailed algorithmic techniques of the FAST-SKY algorithm in the remaining sections.

#### 4.1. Motivation

**4.1.1. Progressive skyline computation with TODs and PODs**

Unlike totally-ordered attributes, in which we can build an index directly, partially-ordered attributes have intrinsic awkwardness when we apply direct indexing on such values. The complex ordering structure underlying partially-ordered attributes is the main cause of the difficulty. One may ease this problem by transforming the values on PODs to TODs, which can be done by increasing the dimensionality of a data point. The transformed values now enable us to reduce any of the efficient algorithms in the literature. This, however, can incur another notorious problem: “the curse of dimensionality.” To avoid this obstacle, one may try a lossy but quite efficient transformation method, as adopted in Chan et al. (2005).

While a lossy transformation scheme does not lead to the dimensionality curse, as the name implies, it does have a drawback: it may create false positive mappings for some values whose ordering structures are intricately interrelated with other values. The false positive mappings are definite blockages to achieving IO-optimality and progressiveness, because transformed values may suppress some of their original ordering relations. This leads to redundant IO accesses.

We approach the problem of indexing data points (with partially-ordered attributes) from a different standpoint. To mitigate the dimensionality curse and avoid false positive mappings, we do not use the values of partially-ordered attributes when indexing a data point; instead, data points are partitioned according to the PODs values. We then index data in each partition, viewing TODs only. Having data points indexed only with totally-ordered attributes while ignoring partially-ordered attributes has a great benefit: we can build multiple index structures for all data points on the reduced dimensions (TODs) without false positive mappings.

#### 4.1.2. Fast dominance comparison

In skyline computation, as a data set grows in size, a dominance check becomes one of the main sources of performance overhead. Existing skyline algorithms for data on PODs have difficulty coping with the burden because they have to adopt a dominance query when performing dominance checks due to the lack of the progressiveness property. If any algorithm is progressive, we can improve the performance enormously by adopting a well-known practice: the use of a reporting query instead of a dominance query. The efficacy of this practice was recently reported in Lin et al. (2005). When processing data on PODs, we devised an efficient dominance check routine by exploiting the iMinMax indexing technique (Yu et al., 2004).

According to Definitions 4 and 5, dominance and reporting queries seek different types of points. We observe an interesting property: if skyline computation is progressive, then an intermediate skyline set has an insert-only property, and in consequence, computing a dominance relationship in skyline computation can be reduced to the range reporting problem, which only requires us to find any intermediate skyline point inside a query rectangle. If we find any point, the query point should not be a skyline point. Otherwise, we now accept a new skyline point. Fig. 2a shows a dominance check in the context of a dominance search problem. A query \( q \) should be compared with all intermediate skyline points, and this takes linear time. In contrast, Fig. 2b depicts the same situation with a reporting query. In this case, a reporting query can be more efficient for dominance checks. This intuitive property enables us to reduce the comparison time in skyline computation substantially.

#### 4.2. Indexing strategy for data with PODs

In this section, we discuss the indexing strategy for data on PODs for IO-optimality and progressiveness. We then propose two index structures: SR-trees and SM-treaps.

**4.2.1. Preprocessing for progressiveness**

Before indexing data, we first have to preprocess each partially-ordered attribute \( A_{\text{partial}} \). While we do not use \( A_{\text{partial}} \) in indexing, we need to assign a specific value to each value \( v \) in \( A_{\text{partial}} \). This value has a pivotal role in making our algorithm progressive. In general, given a partially-ordered attribute \( A_{\text{partial}} \) on a domain \( D_{\text{partial}} \), each value \( v \in D_{\text{partial}} \) is assumed to be unique. We then assign a scheduling number \( tso_v \) to each \( v \). As stated above, we never use the \( tso_v \) when indexing data; instead, we use it in skyline computation. To get \( tso_v \) values, we apply topological sorting (Cormen et al., 2001). Given the fact that a partial ordering in \( D_{\text{partial}} \) is represented by a directed-acyclic graph (DAG), topological sorting on a DAG determines a linear ordering of its nodes that is compatible with the partial order \( R \) that is induced on the nodes where \( x \) comes before \( y \) (\( x \prec R y \)); this assumes that there is a directed path from \( x \) to \( y \) in the DAG.

Usually, a topological sort is very useful in solving job scheduling problems. In job scheduling problems, the scheduling order works as a group ID, such that jobs having the same group ID can be scheduled in the same period, because there is no dependency among these jobs. This property is remarkably useful when we are deciding on a scheduling order that can guarantee progressiveness of skyline computation.
Among many variants, we use the zero in-degree sorting algorithm. If a vertex in a DAG has in-degree zero, then it can be next in the topological order. We remove this vertex and look for an other vertex of in-degree zero in the resulting DAG. We repeat until all vertices have been added to the topological order. At each stage, this way, we assign a tso_v to each v \in A_{parset} of A_{rpart}. The resulting group partitioned by tso has many advantages: (i) the disjoint partition of data points through stratification causes each SR-tree to have small tree depth, which contributes to fast extraction of data from SRTv, and (ii) skyline computation with each SR-tree is definitely progressive.

4.2.2. Multiple partially-ordered attributes

When data has multiple partially-ordered attributes, as shown in Fig. 4a, we simply transform the two domains into a unified one. The unified domain, shown in Fig. 4b, is the genuinely Cartesian-product space. We then apply topological sorting on that space and create the resulting tso table that is shown in Fig. 4c.

4.2.3. Stratified R-trees (SR-trees)

In this section, we present stratified R-trees (SR-trees), a new secondary memory structure for indexing data with PODs. The basic structure of each SR-tree is based on an R-tree, and SR-trees consist of stratified R-trees, i.e., \( FR = \{ SRT_1, SRT_2, \ldots, SRT_n \} \), such that each SR-tree \( SRT_v \) is built on data points whose partially-ordered attribute value is \( v \). In other words, for all data points in \( SRT_v \), the partially-ordered attribute value \( (v) \) is the same. This stratification, which is based on \( v \) of \( A_{parset} \), has many advantages: (i) the disjoint partition of data points through stratification causes each SR-tree to have small tree depth, which contributes to fast extraction of data from \( SRT_v \), and (ii) skyline computation with each SR-tree is definitely progressive.

4.2.4. Stratified MinMax treaps (SM-treaps)

It is well-known that using an R-tree for indexing multidimensional \((d > 10)\) data is nearly as costly as linear scanning. The recent work on the 2Btree index (Lee et al., 2007), despite its excellent features, can be applied to data in the integer-domain only. To guarantee progressive computation and handle multidimensional data in the real domain efficiently, we apply the Min-Max indexing technique to high dimensional data. Our solution is

---

**Table 1:**

<table>
<thead>
<tr>
<th>tso value</th>
<th>POD value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b, c, d</td>
</tr>
<tr>
<td>3</td>
<td>e, f</td>
</tr>
<tr>
<td>4</td>
<td>g, h</td>
</tr>
<tr>
<td>5</td>
<td>i, j</td>
</tr>
</tbody>
</table>

---

**Fig. 2.** (a) A dominance query and (b) a reporting query to check the dominance relationship between a query and intermediate skyline points are shown.

---

**Fig. 3.** The scheduling order for the example poset in Fig. 1a.

---

**Fig. 4.** Example of multi-attributes.
stratified MinMax treaps (SM-treaps), which guarantee progressive skyline evaluation and index high-dimensional data in the real domain. SM-treaps use the same stratification strategy as SR-trees do.

The basic structure of an SM-treap is based on a treap, and SM-treaps consist of stratified MinMax treaps, i.e., $$\mathcal{F} = \{\text{SMT}_1, \text{SMT}_2, \ldots, \text{SMT}_n\}$$. A treap has been developed as a priority search

![Diagram](image)
Lemma 1. Given two data points $p_1$ and $p_2$ ($p_1 \neq p_2$) accessed in turn (i.e., $p_2$ after $p_1$), $p_2$ never dominates $p_1$, but $p_2$ can be incomparable to $p_1$ (i.e., $p_1 \not\preceq p_2$ or $p_2 \not\preceq p_1$).

Proof. We prove the lemma by contradiction. Assume that $p_2$ dominates $p_1$. This assumption implies $K_{p_2} \leq K_{p_1} \land \text{Pr}_{p_2} \leq \text{Pr}_{p_1} \land (K_{p_2} < K_{p_1} \lor \text{Pr}_{p_2} < \text{Pr}_{p_1})$. Then, by the preorder traversal rule of an SM-treap, it is impossible to read $p_1$ before $p_2$. This contradicts the assumption and completes the proof.

Lemma 1 indicates that an SM-treap with a preorder access method guarantees progressive processing and does not incur the heap overhead of maintaining skyline candidates, which was inevitable in the BBS algorithm. The benefit of removing heap overhead is discernable in skyline processing of high-dimensional data.

4.2.5. Progressive evaluation

Given a group of stratified index structures ($\text{SIS}$), we need to determine a sort of scheduling order for each $v \in \text{D}_{\text{partial}}$ of $\text{A}_{\text{partial}}$ when extracting data points from an index in order to guarantee progressive computation. Intuitively, we read data points whose values $v$ have the smallest $\text{tso}_v$ and process them using the FASTSKY algorithm that will be explained in detail.

To determine the scheduling order, we use $\text{tso}_v$ for each value $v \in \text{D}_{\text{partial}}$ of $\text{A}_{\text{partial}}$. We first partition $\mathcal{S}$ into a set of index groups, i.e., $\Gamma = \{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k\}$, by using $\text{tso}_v$ for each $v$. For example, an SIS, should be included in $\mathcal{G}_i$ if $\text{tso}_v$ is $j$, i.e.,

\[ i \] is a matter of choice between a group of SR-trees ($\mathcal{S}$) and a group of SM-treaps ($\mathcal{S}$). A stratified index structure SIS can be an SRT or an SMT.
If two index structures $SIS_x$ and $SIS_y$, where $x \in A_{parial}$ are included in the same index group $\mathcal{H}$, i.e., $SIS_x, SIS_y \in \mathcal{H}$, then a data point $p_i \in SIS_x$ is incomparable to a data point $p_j \in SIS_y$.

**Proof.** The proof idea is quite straightforward. As we see in Fig. 6, each $\mathcal{H}$ groups SMTs whose topological sort value is $i$. All SMTs included in the same index group have no dominance relationship with each other. For example, $\mathcal{H}^2$ has two SMTs, SMT1 and SMT2, which are not comparable to each other. This is the intuition we use to prove Lemma 2. We now prove the lemma formally by contradiction. Assume that a data point $p_i \in SIS_x$ dominates a data point $p_j \in SIS_y$, where $SIS_x, SIS_y \in \mathcal{H}$. If $p_i \preceq p_j$, this implies that $x \preceq y$ on a POD; henceforth, both SISs and SISs will never belong to the same index group $\mathcal{H}$, because topological sorting does not assign the same TSO to both values when they have dependencies. This contradicts the assumption and completes the proof. □

After classifying $\mathcal{H}/ \mathcal{F}$ into $\mathcal{I}$, we compute a skyline set using appropriate algorithms. In each stage, we select an index group $\mathcal{H}$ that is in increasing order, and perform the skyline computation for each SIS $\in \mathcal{H}$. By Lemma 2, the selection order for each SIS $\in \mathcal{H}$ does not affect skyline computation; more importantly all intermediate skyline points can be returned to a user immediately since two data points from two different runs of algorithms are incomparable.

**Lemma 3.** An intermediate skyline point $s_i$ that is found in the $\mathcal{H}$ cannot be dominated by any data point in $\mathcal{H}$, where $i < j$.

**Proof.** The proof idea of Lemma 3 is also simple. As we see in Figs. 6 and 7, data points from lower level index groups (i.e., $\mathcal{H}$ or $\mathcal{H}^2$) never dominate data points from a higher level index group (i.e., $\mathcal{H}^3$) because of the dominance relationship of the given poset. Using this proof idea, we also prove this lemma formally by contradiction. Assume that there exists a point $p_i \in \mathcal{H}$ that dominates an intermediate skyline point $p_j \in \mathcal{H}$, where $i < j$. With the assumption of $p_i \preceq p_j$, this indicates $ts_{ij} < ts_{ij}$, which means $ts_{ij}$ should be included in the other index group $\mathcal{H}$, where $j < k$. This is demonstrably impossible as per the construction rule of $\mathcal{I}$; therefore, it contradicts the assumption and completes the proof. □

Lemma 3 ensures that (i) we guarantee progressive skyline computation across index groups, and (ii) all skylines that are computed from preceding index groups can be returned to a user immediately. By Lemmas 2 and 3, we satisfy the progressiveness property during the entire skyline computation, even with PODs.

### 4.3. Efficient dominance comparison

Once we guarantee the progressive skyline processing with PODs via the aforementioned techniques, we can present a method of fast dominance comparison. We adopt dimensionality reduction for efficient comparisons. Yu et al. (2004) proposed the iMinMax(0) method to process high dimensional queries. iMinMax(0) maps high dimensional points onto a single dimensional space. We denote $x_{min}$ and $x_{max}$ as the smallest and largest values among all coordinate values, and $d_{min}$ and $d_{max}$ as the corresponding dimensions for $x_{min}$ and $x_{max}$. The data point is mapped to $y$ over a single dimensional space as follows:

$$y = \begin{cases} d_{min} \times c + x_{min} & \text{if } x_{min} + 0 < 1 - x_{max} \\ d_{max} \times c + x_{max} & \text{otherwise} \end{cases}$$

The authors use the transformation to partition the data space into different partitions based on which dimension has the largest or smallest value. Because the transformation provides an ordering (Yu et al., 2004), $B+$-trees can be exploited for efficient search and practical deployment. Thus, iMinMax(0) can be an effective technique for the range reporting problem. Since values in databases may not be transformed into real numbers between 0 and 1, we use real values instead of transformed values, use multiple $B+$-trees instead of a single $B+$-tree, and split $B+$-trees into iMinMax-based or iMax-based $B+$-trees. iMinMax-based $B+$-trees maintain data points that are indexed based on their minimum/maximum values, and each $B+$-tree is assigned to each coordinate.

Fig. 8 gives an approximation of our iMax-based dominance check algorithm. Let $S_1$ and $S_2$ denote intermediate skyline points...
with their maximum coordinate values indexed over B+-trees. The query point $q_1$ can be inserted into the intermediate skyline set after it is confirmed that neither $s_1$ nor $s_2$ dominates $q_1$. In contrast, neither $q_2$ nor $q_3$ can be inserted into the skyline set since the minimum coordinate value of the query point is greater than the maximum coordinate value of any skyline points (e.g., $s_1$ or $s_2$). We describe the dominance check algorithm using $iMinMax(0)$ in skyline computation.

**Algorithm 2. Algorithm DominanceCheck_Over_iMinMax**

**Data Structures**
- $\mathcal{F}_{iMinMax}/\mathcal{F}_{iMax}$: a group of iMinMax/iMax-based B+-trees built from an intermediate set of skylines
- $\mathcal{F}_v \in \mathcal{F}_{iMinMax}/\mathcal{F}_{iMax}$: iMinMax/iMax-based B+-trees for a partially-ordered attribute value $v$ in $A_{partial}$
- $T_v \in \mathcal{F}_v$: a B+-tree for each coordinate $i \in \{1, 2, \ldots, d\}$
- $q$: a query point
- $v_q$: the partially-ordered attribute value of the query $q$
- $s$: an intermediate skyline point

**Output**
- true if $q \preceq s$ in $\mathcal{F}_{iMinMax}$
- false otherwise

```plaintext
1: for each $T_v \in \mathcal{F}_{iMinMax}$ do
2: if $v_q \preceq v$ then /* relevant iMinMax-based B+-trees */
3: for each $T_v \in \mathcal{F}_v$ do /* for each coordinate */
4: while ($s = \text{ScanLeafNodes}(T_v)) \neq \text{null}$ do
5: if $KV_v > Min_q$ then break; /* pruning by $KV_v$ */
6: endif
7: if $s$ dominates $q$ then return true;
8: endif
9: endwhile
10: endfor
11: endif
12: endfor
13: return false;
```

**Algorithm DominanceCheck_Over_iMax**

**Output**
- true if $q \preceq s$ in $\mathcal{F}_{iMax}$
- false otherwise

```plaintext
1: for each $T_v \in \mathcal{F}_{iMax}$ do
2: if $v_q \preceq v$ then /* relevant iMax-based B+-trees */
3: for each $T_v \in \mathcal{F}_v$ do /* for each coordinate */
4: while ($s = \text{ScanLeafNodes}(T_v)) \neq \text{null}$ do
5: if $KV_v < Min_q$ then return true; /* early termination */
6: endif
7: if $KV_v > q[i]$ then break; /* pruning by $KV_v$ */
8: endif
9: if $s$ dominates $q$ then return true;
10: endif
11: endwhile
12: endfor
13: endfor
14: return false;
```

In order to efficiently manage the data on PODs, we now address an optimization technique in advance. We know that, given a value $v$ in a domain $D_{partial}$ of a partially-ordered attribute $A_{partial}$, there may exist many values that have no dependency with $v$, and that maintaining a single iMax-based B+-tree for all intermediate skylines may cause unnecessary dominance checks and increase computation cost. We optimize this in such a way that we create a single B+-tree $T_v$ for each value $v$ of $A_{partial}$ and maintain a set of B+-trees $\mathcal{F} = \{T_{v_1}, T_{v_2}, \ldots, T_{v_k}\}$. Thus, intermediate skyline points are stratified according to a value $v$ and are inserted into the appropriate B+-tree $T_v \in \mathcal{F}$. If we process data only with TDOs, then there exists a single B+-tree.

As per Algorithm 2, given a set of iMinMax(0)-based B+-trees $\mathcal{F}$ and a query point $q$ having $v_q \in D_{partial}$ of $A_{partial}$, our algorithm first extracts a relevant set of B+-trees $\mathcal{F}_{rel}$, and it examines all trees in $\mathcal{F}_{rel}$ beginning from the nearest ancestor trees. When it traverses each B+-tree in ascending order (from left to right node), we use simple yet powerful pruning techniques: (i) if the key value $KV_v > Min_q$ (the minimum coordinate value of the query point $q$) in iMinMax-based B+-trees, then the algorithm prunes the remaining nodes and visits the next tree. (ii) if $KV_v > q[i]$ (the $i$th coordinate value of the query point) in iMax-based B+-trees, then the algorithm prunes the remaining nodes and visits the next tree. (iii) if any $KV_v < Min_q$ in iMax-based B+-trees, $q$ is dominated.

**Performance analysis:** Here we discuss the performance of the iMinMax-based dominance check algorithm in terms of the worst case processing time. The worst case in the dominance-check procedure is that a query point $q$ is not dominated by any current skyline points, so it is inserted into the intermediate skyline set. Let $m$, $N_{total}$, and $N_{rel}$ be the number of skyline points, the number of all B+-trees, and the number of all relevant B+-trees (size of $\mathcal{F}_{rel}$), respectively. If we assume that skyline points are distributed equally over all B+-trees, a single B+-tree maintains $\frac{m}{N_{total}}$ skyline points. The three pruning techniques allow our algorithms to avoid evaluating all skyline points. When traversing an iMinMax-based B+-tree, the DominanceCheck_Over_iMinMax algorithm examines only $\frac{m}{N_{rel}} \times \frac{Min_q}{Max_q}$ points where $Min_q$ and $Max_q$ are the minimum coordinate value of the query point and the maximum key value of the tree. In the case of iMax-based B+-trees, Min$\_q$ is replaced by Max$\_q$, which is the maximum coordinate value of the query point. Thus, in the worst case, the processing time is $O\left(\frac{N_{total} \times m}{N_{rel}} \times \frac{Min_q}{Max_q}\right)$. Our technique runs faster by a factor of $O\left(\frac{N_{total} \times m}{N_{rel}} \times \frac{Min_q}{Max_q}\right)$ than the naive dominance-check procedure (linear scan, $O(m)$).

In iMinMax-based B+-trees, the distribution of key values is not uniform because each key value is the smallest value among all dimensions of a skyline point (smaller $KV_{max}$). This increases $\frac{Min_q}{Max_q}$ and the worst case processing time of the dominance-check operation. In fact, in many cases, $Min_q$ is greater than or equal to $KV_{max}$, and the fraction $\frac{Min_q}{Max_q}$ becomes 1. However, in iMax-based B+-trees, the key distribution is uniform. Thus, $\frac{Min_q}{Max_q}$ is smaller than 1, and iMax-based B+-trees show better performance than iMinMax-based B+-trees.

### 4.3.1. FAST-SKY algorithm

This section describes the complete version of the FAST-SKY algorithm. If an $SIS$ is an $SRT$, we use the SR-BBS algorithm, which is an $SRT$-tree-based $branch-and-bound$ skyline algorithm. Otherwise, we perform a dominance test directly for data points.
read from an SMT because an SMT does not incur heap overhead.

Algorithm 3. Algorithm FAST-SKY

Data Structures
\( S \subset \mathbb{R}^d \): a set of stratified index structures
\( \mathcal{F} \): an intermediate set of skyline points
\( \mathcal{F}^* \): a set of iMax-based B+-trees
\( \Gamma \): a set of disjoint groups of index structures
\( \text{SRT}_v \): an SR-tree for \( v \in \mathcal{A}_{\text{partial}} \)
\( \text{SMT}_v \): an SM-treap for \( v \in \mathcal{A}_{\text{partial}} \)

Algorithm FAST-SKY \((S, \mathcal{F}^*)\)
1: \( \Gamma = \text{Classifier}(S, \mathcal{F}) \); 
2: for \( i = 1 \) to \( k \) do
3: for each \( \text{SIS}_p \in \mathcal{F} \) do
4: if \( \text{SIS}_p = \text{SRT}_p \) then
5: \( \mathcal{F} = \text{SR-BBS}(\text{SRT}_p, \mathcal{F}, \mathcal{F}^*) \);
6: else 
7: \( \mathcal{F} = \text{SM-Skyline}(\text{SMT}_p, \mathcal{F}, \mathcal{F}^*) \);
8: endfor
9: endfor
10: return \( \mathcal{F} \)

Algorithm Classifier \((S, \mathcal{F}^*)\)
1: for each \( p \in \mathcal{A}_{\text{partial}} \) do 
2: \( s = \text{tso}_p \);
3: \( \mathcal{F}^* = (\text{SIS}_p \in \mathcal{F} \setminus \mathcal{F}^* \mid \text{tso}_p = s) \);
4: \( \Gamma = \Gamma \cup \mathcal{F}^* \);
5: endfor
6: return \( \Gamma \)

Algorithm SR-BBS(SRT\(_v\), \( \mathcal{F}, \mathcal{F}^* \))
SR-BBS is the same as the BBS algorithm in Algorithm 1, except that (i) each “dominated” comparison is replaced by a \text{DominanceCheck_Over_iMaxMin} routine, and (ii) \text{UpdateSkylines} is replaced by the new \text{UpdateSkylines} below.

Algorithm SM-Skyline(SMT\(_p\), \( \mathcal{F}, \mathcal{F}^* \))
max\(_{\text{min}}\) = min\(_{\text{max}}\) Pr\(_t\)
1: while \( (p = \text{Preorder}(\text{SMT}_p)) \neq \text{null} \) /* preorder traversal */ 
2: do
3: if max\(_{\text{min}}\) < \( K_p \) then
4: prune the right sub-treap of \( p \); /* pruning by max\(_{\text{min}}\) */
5: else
6: \text{UpdateSkylines}(p, \mathcal{F}, \mathcal{F}^*)
7: endwhile

Algorithm UpdateSkylines \((p, \mathcal{F}, \mathcal{F}^*)\)
\( v_p \) is the partially-ordered attribute value of a point \( p \)
1: if \( \text{DominanceCheck_Over_iMaxMin}(\mathcal{F}, p) = \text{true} \) then
2: return \( \mathcal{F}^* \);
3: Insert \( p \) into \( \mathcal{F}^* \);
4: Insert \( p \) into \( T_v \) in \( \mathcal{F}^* \), where \( v = v_p \); /* update \( \mathcal{F}^* \) */
5: return \( \mathcal{F}^* \);

As per Algorithm 3, the \( j \)th run of SR-BBS with \( \text{SRT}_i \in \mathcal{F}^* \) traverses the SR-tree \( \text{SRT}_i \), then searches for the nearest internal entry or data point(s) that are not dominated by any points of the current skyline. If such points are found, it inserts them into the Min-Heap \( H \). If the top entry point of the Min-Heap is not dominated by any other points of the current skyline set, it is inserted into the set and the set is updated. Both SR-BBS and SM-Skyline use Dominance-Check_Over_iMaxMin of Algorithm 2 in order to check whether an internal entry or a data point is dominated by other points of the current skyline set, and this minimizes the comparison time.

In SM-Skyline, we apply a simple pruning technique (lines 3–4). If \( K_p \geq \max_{\text{min}} \), then we do not have to read data points in the right sub-treap of a point \( p \) in an SM-treap because it is obvious that all data points in the right sub-treap have a larger \( K(\text{min}) \) value than \( \max_{\text{min}} \).

Theorem 1. The algorithm FAST-SKY achieves IO-optimality and progressiveness in skyline computation with TIDs and PIDs.

Proof. By Lemmas 1–3, there is not an intermediate skyline point that can be removed from an intermediate skyline set (progressiveness). By the IO-optimality of SR-BBS\(^5\) and the preorder traversal\(^6\) of SM-Skyline, (1) FAST-SKY reads only the nodes that may contain skyline points from the disk and (2) it does not read the same node twice from the disk (IO-optimality). \( \square \)

5. Performance study

In this section, we give performance results detailing (i) the efficiency and progressiveness of FAST-SKY and (ii) the applicability of the fast dominance check.

5.1. Experimental environments

The experiments were performed on a workstation with two dual-core Opteron 265 processors and 8 GB of main memory, which was running the SUSE Linux operating system. In TODs, values whose domain is \((0.0,1000.0]\) were generated by our synthetic data generator; in particular, the technique described in Bőrzsönyi et al. (2001) is used for the correlation between TIDs. In PIDs, the technique in Chan et al. (2005) is used to generate data points. All algorithms used were based on the Java version of the spatial index library (Hadjieleftheriou, 2003). We compared our algorithms to SDC\(^+\) and TSS. We denote SR-tree-based and SM-treap-based FAST-SKY as “FAST-SKY(SR-trees)” and “FAST-SKY(SM-treaps).”

We conducted the experiments by varying the number of data points, the number of attributes, the correlations between attributes, and the structure of a poset. Table 1 shows the parameters and values used in the experiments. Generally, the poset size increases as the number of partially-ordered attributes increases. Thus, we change the poset size instead of the number of partially-ordered attributes in order to investigate the effect of various numbers of partially-ordered attributes.

Because the core routine of SR-BBS is the same as BBS that has proven to be IO-optimal (Papadias et al., 2003) and the IO-optimality of SR-BBS can be derived directly from that of BBS.

Traversing itself means one time visit to a node.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td># of totally-ordered attributes</td>
<td>2–18</td>
</tr>
<tr>
<td># of partially-ordered attributes</td>
<td>1–2</td>
</tr>
<tr>
<td>Page size</td>
<td>4 KB</td>
</tr>
<tr>
<td># of tuples</td>
<td>1M–8M</td>
</tr>
<tr>
<td>( N_W ), poset size</td>
<td>511, 1023</td>
</tr>
<tr>
<td>Attribute correlation</td>
<td>Uniform, anti-correlated</td>
</tr>
<tr>
<td>Disk page size and node size of R-trees</td>
<td>4 KB</td>
</tr>
</tbody>
</table>
Unfortunately, SDC+ cannot adopt our fast dominance-check procedure because the stratification strategy of SDC+ does not guarantee the insert-only property. We therefore implement the dominance check routine based on the compareDominance algorithm in Chan et al. (2005).

### 5.2. IO efficiency

To assess the effect of our stratification technique, we modified the FAST-SKY(SR-trees) algorithm slightly. The basic idea of the implementation is (i) to index all data points using SR-trees, (ii) to iterate the BBS algorithm according to iso, and (iii) to replace the dominance check of Algorithm 1 (lines 6 and 8 of BBS, and line 2 of UpdateSkylines) with an m-dominance check (naive dominance check). That is, the detailed algorithm is the same as the algorithms given in Section 4, except that the “DominanceCheck” is invoked. We have designated this version “PRE-FAST-SKY.” Without loss of generality, we compare PRE-FAST-SKY to SDC+ and TSS since both SDC+ and TSS use R-trees while PRE-FAST-SKY adopts Min-Heap.

We evaluate the performance of PRE-FAST-SKY, measuring running time by varying the number of data points under four conditions. Then we break the measured time into IO and computation time. From Figs. 9 and 10, we can see that PRE-FAST-SKY outperforms SDC+ and TSS in terms of IO performance. Owing to the absence of lossy transformation and the stratification technique, PRE-FAST-SKY does not visit unnecessary internal nodes, and this leads to superior IO performance of PRE-FAST-SKY. This validates the IO superiority of our indexing scheme.

Figures also show that PRE-FAST-SKY computes all skyline points faster than SDC+ does. For easy analysis, we divide computation overhead into two main components: Min-Heap operations and m-dominance check operations. A node in the Min-Heap implies that the parent node of the node is read from the disk in PRE-FAST-SKY. A node that exists in the Min-Heap requires two m-dominance check operations; that is, one check operation is invoked before the node enters the Min-Heap, and another is invoked after the node leaves the Min-Heap. Thus, PRE-FAST-SKY can finish skyline computation earlier than SDC+, since it traverses the minimum number of internal nodes. This, in turn, leads to a small number of Min-Heap and m-dominance check operations, as shown in Fig. 11.

TSS outperforms PRE-FAST-SKY in terms of computation time because the t-dominance-check operation of TSS is much faster than the naive dominance-check operation of PRE-FAST-SKY. Although TSS has a larger number of Min-Heap operations than SDC+ as shown in Fig. 11, TSS always computes all skyline points faster than SDC+ due to the t-dominance-check operation. However, TSS shows the worst IO performance since the internal nodes of an R-tree in TSS might have many propagated intervals, and this leads to a less pruning effect of branch-and-bound style algorithms than SDC+ and PRE-FAST-SKY.

#### 5.2.1. Discussion

In principle, the three algorithms (i.e., SDC+, TSS and PRE-FAST-SKY) achieve the IO-optimality with different indexing strategies. What we learned from these experiments is that achieving IO-optimality itself does not always ensure better IO performance, rather the strategic differences make a noticeable distinction among the three algorithms. Progressiveness itself also does not contribute a lot to performance enhancement, but it gives us the vital opportunity to leverage the reporting query in dominance comparisons. The benefit of exploiting the reporting query is the main theme of the remaining part of performance study.

Some might doubt the scalability of the SR-trees-based approach; SR-trees appear to consume huge amounts of space. Our stratification technique does not lead to redundant tuples across trees because of the disjointedness property of the stratification. Our approach does not store values of partially-ordered attributes since each SR-tree is built for each value of partially-ordered attributes. The total space consumed by SR-trees, therefore, is less than that of R-trees in SDC+ and TSS.\(^7\) and Table 2 validates this.\(^8\)

Another issue is that it may be hard to process other queries (e.g., range query or nearest-neighbor query) over SR-trees. Processing such queries in the interval-based approach is much more difficult than it is in the SR-trees-based approach, since processing such queries over interval domains is very complex. In contrast, the SR-trees-based approach just increases the search space to process such queries.

#### 5.3. Computation efficiency

In this experiment, we evaluated the performance of FAST-SKY, measuring how fast all answers were returned when varying the number of data points. Dominance-check operations of FAST-SKY are based on the iMax-based technique since it shows better performance than the iMin-based approach. We will draw a comparison between iMin- and iMax-based approaches in the next section.

Each experiment consists of a series of two consecutive trials. In the first trial, the state of the buffer cache is cold, while in the second, the state is warm (since we have a workstation with fluent memory for all experiments, all data used in the former was in

---

\(^7\) An R-tree of TSS includes propagated intervals as well as initial intervals.

\(^8\) Total space of SM-treaps is less than that of SR-trees since the space complexity of SM-treaps is \(\Theta(n)\).
Today’s computing trend of using large-memory distributed systems, which render in-memory database processing very feasible, emphasizes the importance of the computation efficiency of FAST-SKY. We, therefore, measured the computation time and recorded the time it took to output a portion of the answers.

From Figs. 12 and 13, we can see that the SDC+ algorithm shows worst performance and FAST-SKY shows best performance under all conditions. Figs. 12a and 13a show running time, and results denoted as “Warm” indicate that our algorithm is very useful in computing skyline points with high-end servers that can do in-memory processing. Table 3 shows the number of skyline in both experiments. From the table, larger number of data points generates more skyline points and this leads to more computation time.

Figs. 12b and 13b show that FAST-SKY computes skylines much faster than SDC+ and TSS. It is noted that results in Fig. 12b are plotted on a logarithmic scale except those of FAST-SKY (SR-trees) and FAST-SKY (SM-treaps) which are plotted on a linear scale. Since SDC+ is based on the BBS algorithm, a dominance-check operation requires comparisons to all points of the current skyline set in the worst case (linear search). Such dominance-check operations cause...
SDC+ to incur tremendous computation overhead. The TSS algorithm uses the $t$-dominance-check operation that is an R-tree-based fast dominance check. In contrast, in FAST-SKY, a new data point can be inserted into a skyline set if there is no data point that precedes the new point. This enables our algorithm to prune unnecessary comparisons to existing skyline points and leads to optimized computation. Moreover, the fact that the dominance check is only invoked over relevant iMax-based B+-trees contributes greatly to computation efficiency as well. FAST-SKY(SR-trees) computes skylines 33 times (5–15 times) and 50–85 times (8–23 times) faster than PRE-FAST-SKY (TSS) in uniform and anti-correlated distributions, respectively. These observations definitively confirm that a fast dominance check, rather than optimal disk IO, is the key factor in improving performance. The result shows that our dominance check algorithm provides the most efficient computation as the number of skyline points increases.

FAST-SKY(SM-treaps) shows 1.2–2 times better performance than FAST-SKY(SR-trees) in uniform distribution, since FAST-SKY(SM-treaps) takes advantage of the dimensionality reduction and preorder traversal in 4-dimensional data. In contrast, in an anti-correlated distribution, FAST-SKY(SM-treaps) shows worse computation time than FAST-SKY(SM-trees) since the data dimensionality is only 2. The result indicates that skyline computation with multidimensional data (if $d > 2$) can be optimized via four key techniques: a fine-grained stratification, a fast dominance-check procedure, the dominance check over only relevant sets, and min–max indexing.

Figs. 12c and 13c show how fast each algorithm returns all answers progressively. In all experiments, as the size of intermediate skylines grows, the time to perform dominance checks also increases (the monotonicity of a skyline set). In SDC+ and TSS, the time slope rises more steeply than that of FAST-SKY(SR-trees) because SDC+ has $O(N)$ time overhead in dominance checks and TSS has IO overhead that arises from a less pruning effect. In PRE-FAST-SKY, although it outperforms SDC+ due to the IO-optimality of SR-trees, its time slope behaves similarly ($O(N)$) as intermediate skyline increases in both distributions. This is because computation time does not decrease as much as IO time. In contrast, FAST-SKY shows the most efficient behavior on account of the optimized dominance check and the IO-optimality. By this observation, we can validate the efficiency of FAST-SKY in progressive skyline computation.

5.3.1. Summary

The IO-optimality contributes to the reduction of the number of IO operations and $\text{Min-Heap}$ insert operations, each of which demands two dominance checks. Nonetheless, the IO-optimality does not reduce these operations as much as we expect. However, the new dominance check, which uses relevant iMax-based B+-trees, decreases computation time greatly. Based on this fact, we can safely confirm that FAST-SKY(SR-trees) is much better than the current state-of-the-art algorithm in terms of IO-optimality and efficient computation. Moreover, the dimensionality reduction mitigates pain from overlapping regions and the preorder traversal of FAST-SKY(SM-treaps) removes the $\text{Min-Heap}$ overhead. These make FAST-SKY(SM-treaps) compute skylines faster for multidimensional data than FAST-SKY(SR-trees).

5.4. Other experiments

In this section, we only report on the computation time of FAST-SKY, because efficient computation (dominance check) becomes the key factor in efficient skyline computation as data size increases. In particular, computing skylines with a complex poset structure (or by increasing the number of totally-ordered attributes) underlines the importance of computation time.

5.4.1. iMinMax vs. R-tree

Fig. 14 shows the computation time of FAST-SKY(SR-trees) with three different data structures -- an iMax-based B+-tree, an iMinMax-based B+-tree, and an R-tree -- for a reporting query, and we apply the optimization technique that was mentioned in Section 4.3 to all data structures. In Fig. 14a, which has 4-dimensional data, an iMax and an iMin run much faster than an R-tree. The superiority of an iMax and an iMin is the same in Fig. 14b. In both figures, an iMax outperforms an iMin since iMin-based B+-trees have less pruning effect than iMax-based B+-trees as explained in Section 4.3. These results validate the excellence of an iMax and indicate that the pruning technique adopted in $\text{DominanceCheck}$ works efficiently in higher dimensions as well. In all remaining experiments, only the iMax-based technique for FAST-SKY is used.

5.4.2. Effects of complex poset structure

Fig. 15 shows the elapsed time of FAST-SKY when the size and the height of the poset increase. The increase of the poset height leads to more skyline points as shown in Table 4. In case of SDC+, this causes more strata in SDC+ (Chan et al., 2005). More skyline points mean more invocations of $t$-dominance and $m$-dominance-check operations in TSS and SDC+. However, our optimized techniques (iMax-based dominance check and dominance check over only relevant B+-trees) enable FAST-SKY to finish the skyline computation faster than TSS and SDC+.

<table>
<thead>
<tr>
<th>Table 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>The size of skyline (A: case of Fig. 12, B: case of Fig. 13).</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

Fig. 13. Results (anti-correlated, two numerical and one set attributes, poset size = 511).
From the figures, we can see a bigger difference in computation time between SDC+ and FAST-SKY in uniform distribution (70–100 times with SR-trees and 130–250 times with SM-treaps) than that in Fig. 12, and a bigger difference in anti-correlated distribution (100–200 times with SR-trees and 120–160 times with SM-treaps) than in Fig. 13. In cases of TSS, we can see a similar gap of computation time between TSS and FAST-SKY in uniform and anti-correlated distribution. As explained in the previous section, FAST-SKY(SM-treaps) shows worse performance in many cases of Fig. 15b than FAST-SKY(SR-trees) since FAST-SKY(SM-treaps) does not have the effect of dimensionality reduction in two dimensional data points.

5.4.3. Effects of increasing totally-ordered domains

Fig. 16 and Table 5 show the results when the number of totally-ordered domains increases. A large number of totally-ordered domains increases the size of a skyline set, due to the high probability that a data point will become a skyline point. In both uniform and anti-correlated distributions, the difference in computation time between SDC+ and FAST-SKY(SR-trees) gets larger (120–230 times in uniform distribution and 170–260 times in anti-correlated distribution).
distributions) than that in the results of the previous section. FAST-SKY(SM-treaps) shows the best performance (160–370 times faster than SDC+ in uniform distribution and 210–370 times faster than SDC+ in anti-correlated distribution) since FAST-SKY(SM-treaps) exploits the effect of dimensionality reduction in both distributions. The difference in computation time between TSS and FAST-SKY(SR-trees) is 5 times in uniform distribution and 10–90 times in anti-correlated distribution. In the case of the anti-correlated distribution, the size of skyline increases much more (20 times) than it does in the case of two totally-ordered and one partially-ordered domain, and this leads to much longer computation in SDC+ and TSS.

5.4.4. Effects of increasing partially-ordered domains

Fig. 17 and Table 6 show the results when the number of partially-ordered domains increases. An increase in the number of partially-ordered domains necessitates increasing the cardinality of strata that we have to consider when building $\text{SIS}$. Multiple PODs cause two obvious changes. First, the size of skyline increases because of the high probability that a data point will become a skyline point, and this increases computation time. Second, a single SR-tree (or SR-treap) have less data points owing to the expansion of the cardinality of strata. As we discussed in the previous section, the total space consumed by SIS even with multiple PODs is less than that of R-trees in SDC+ and TSS, and the management overhead is negligible as well.

In both uniform and anti-correlated distributions, differences in computation time between SDC+ and FAST-SKY(SR-trees) are 30–500 times in uniform distribution and 45–600 times in anti-correlated distributions, and gaps of computation time between TSS and FAST-SKY(SR-trees) are 3–35 times in uniform distribution and 10–85 times in anti-correlated distributions. In Fig. 17a and b, we do not have a sharp difference between FAST-SKY(SR-trees) and FAST-SKY(SR-treaps) because the large cardinality of the product of two PODs makes each SR-tree (or SR-treap) have a very small number of data points. With a small SR-treap, the advantage of using an SR-treap is not prominent. When we increase the data points up to millions of records, we can see the performance gap between two stratified index structures, as shown in Fig. 17c and d.

Overall, Figs. 15–17 and Tables 4–6 demonstrate that when the number of skyline points increases, FAST-SKY is a scalable and efficient solution.

5.4.5. Effects of dimensionality reduction based indexing

Fig. 18 shows the results of FAST-SKY(SR-trees) and FAST-SKY(SM-treaps) when the number of TODs increases. Due to the excessive computation overhead of SDC+ in high dimensional space, similar to what we saw in previous results, results of SDC+ are omitted. Each percentile value of both plots means a skyline portion of whole data points. When the data dimensionality increases, the skyline portion increases as has already been discussed in Lee et al. (2007). FAST-SKY(SM-treaps) adopts the min–max

<table>
<thead>
<tr>
<th>Table 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>The size of skyline (A: case of Fig. 17a, B: case of Fig. 17b, C: case of Fig. 17c, D: case of Fig. 17d).</td>
</tr>
<tr>
<td>100K</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>1M</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
</tbody>
</table>

Fig. 17. Effects of increasing partially-ordered domains (computation time, poset size = 511).
indexing technique and the preorder traversal of treaps. The dimensionality reduction mitigates the overlapping region of search space and the preorder traversal eliminates the Min-Heap overhead. From Fig. 18a and b, we can see that FAST-SKY(SM-treaps) with multidimensional data points shows better performance (about 2 times) than FAST-SKY(SR-trees). In Fig. 18c and d, FAST-SKY(SR-tree) works the same way as the BBS algorithm with the iMax-based dominance check since poset size is one. FAST-SKY(SM-treap) outperforms (about 2 times) FAST-SKY(SR-tree) in both distributions. FAST-SKY(SR-tree) shows better performance (about four times and 2 times in uniform and anti-correlated distributions) compared with BBS. From the observation, we can safely conclude that our iMax-based pruning technique can be applied to an existing progressive algorithm and make the algorithm work better.

5.4.6. Effects of changing poset structure

We discuss the adaptability of FAST-SKY. Because the SDC+ algorithm cannot adapt to the change of the poset structure without rebuilding whole trees, we only explain how FAST-SKY can handle the situation. Two indexing strategies proposed in this paper index data points on a group of stratified R-trees (or MinMax Treaps), which means that for each POD value in a single POD or each product value in multiple PODs we build one index structure. Because we build each index for each POD value, our indexing strategy does not need extra time to build an index in case of the change of the poset. The only thing we need to do is recalculating the scheduling order according to the newly changed poset structure, without touching existing index trees (or treaps). In consequence, the overhead to adapt our index structures is negligible. This is our advantage compared to existing algorithms. The TSS algorithm allegedly dealt with the change of poset with minimal preprocessing, which is slight complicated processing than our methodology.

6. Conclusion

In this article, we address the problem of skyline evaluation with TODs and PODs even with high-dimensional data. We also note that most database algorithms for skyline computation are based on a dominance query, which incurs severe computation overhead as data size increases to millions of records. We design two new index structures for indexing data on TODs and PODs via a stratification technique, and they are very efficient in progressive skyline processing. By utilizing the progressiveness property, we can reduce the dominance check overhead. We use a reporting query, in addition to the iMax-based B+-tree, to enhance a dominance test. Based on our extensive performance investigation, the FAST-SKY algorithm shows that (i) it outperforms both SDC+ and TSS under any circumstance by a wide margin, and (ii) all novel pruning techniques work efficiently in processing a skyline query with PODs even in high dimensional space. For future work, we will further study the possibility of inventing an optimal algorithm for a reporting query in higher dimensions, because the optimal solution has a significant impact on many database problems that involve managing a huge amount of data.

Acknowledgement

This study was supported by the Seoul R&amp;BD Program (10561), Seoul, Korea.

References


Chazelle, B., 1990b. Lower bounds for orthogonal range searching: II. The arithmetic model. JACM 37 (3).


Hyungsoo Jung received the BS degree in mechanical engineering from Korea University, Seoul, Korea, in 2002; and the MS and the PhD degrees in computer science from Seoul National University, Seoul, Korea in 2004 and 2009, respectively. He is currently a post-doctoral research associate at Seoul National University, Seoul, Korea. His research interests are in the areas of distributed systems, database systems, and transaction processing.

Hyuck Han received his BS and MS degrees in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 2003 and 2006, respectively. Currently, he is a PhD candidate at Seoul National University. His research interests are distributed computing systems, parallel computing, and database systems.

Heon Y. Yeom is a Professor with the Department of Computer Science and Engineering, Seoul National Univ. He received his BS degree in computer science from Seoul National Univ. in 1984 and received the MS and PhD degree in computer science from Texas A&M Univ. in 1986 and 1992, respectively. From 1992 to 1993, he was with Samsung Data Systems as a Research Scientist. He joined the Department of Computer Science, Seoul National University in 1993, where he currently teaches and researches on distributed systems, multimedia systems and transaction processing, etc.

Sooyong Kang received his BS degree in mathematics and the MS and PhD degrees in Computer Science, from Seoul National University, Seoul, Korea, in 1996, 1998, and 2002, respectively. He was then a Postdoctoral researcher in the School of Computer Science and Engineering, SNU. He is now with the Division of Computer Science and Engineering, Hanyang University, Seoul. His research interests include Operating System, Multimedia System, Storage System, Flash Memories and Next Generation Nonvolatile Memories.