Stratified Computation of Skylines with Partially-Ordered Domains

Chee-Yong Chan Pin-Kwang Eng Kian-Lee Tan
Department of Computer Science, National University of Singapore
{chancy,engpk,tankl}@comp.nus.edu.sg

ABSTRACT

In this paper, we study the evaluation of skyline queries with partially-ordered attributes. Because such attributes lack a total ordering, traditional index-based evaluation algorithms (e.g., NN and BBS) that are designed for totally-ordered attributes can no longer prune the space as effectively. Our solution is to transform each partially-ordered attribute into a two-integer domain that allows us to exploit index-based algorithms to compute skyline queries on the transformed space. Based on this framework, we propose three novel algorithms: BBS$^+$ is a straightforward adaptation of BBS using the framework, and SDC (Stratification by Dominance Classification) and SDC$^+$ are optimized to handle false positives and support progressive evaluation. Both SDC and SDC$^+$ exploit a dominance relationship to organize the data into strata. While SDC generates its strata at runtime, SDC$^+$ partitions the data into strata offline. We also design two dominance classification strategies (**MinPC** and **MaxPC**) to further optimize the performance of SDC and SDC$^+$. We implemented the proposed schemes and evaluated their efficiency. Our results show that our proposed techniques outperform existing approaches by a wide margin, with SDC$^+$-MinPC giving the best performance in terms of both response time as well as progressiveness. To the best of our knowledge, this is the first paper to address the problem of skyline query evaluation involving partially-ordered attribute domains.

1. INTRODUCTION

Many decision support applications are characterized by several features: (1) the query is typically based on multiple criteria; (2) there is no single optimal answer (or answer set); (3) because of (2), users typically look for *satisfying* answers; and (4) for the same query, different users, dictated by their personal preferences, may find different answers meeting their needs. As such, it is important for the DBMS to present *all interesting* answers that may fulfill a user's need. In this paper, we focus on the set of interesting answers called the *skyline*. Given a set of points, the skyline comprises the points that are not *dominated* by other points [4]. A point dominates another point if it is *as good or better in all dimensions and better in at least one dimension*. As an example, a tourist looking for *budget* hotels that are *close* to the cities may issue the following SQL query [4]: `Select * From hotels Skyline of Price Min, Distance Min`, where `Min` indicates that the price and the distance should be minimized. Clearly, if hotel h1 dominates hotel h2 (i.e., h1 is cheaper and nearer to the city than hotel h2), then h2 can be pruned away. On the other hand, if h1 is cheaper but further away to the city than h2, then they are not comparable and both should be returned to the users (provided they are not dominated by other hotels).

While much work has been done to develop efficient schemes to evaluate skyline queries, these deal exclusively with totally-ordered attribute domains [4, 18, 11, 14]. Partially-ordered attribute domains which include interval data (e.g., temporal data), type/class hierarchies, and set-valued domains, have not been considered. In our hotel example, a hotel may store a set of interesting places within its vicinity, and our tourist may prefer a hotel that contains a superset of interesting places or amenities (e.g., gift shop, gymnasium, saloon, sauna, etc.).

As another example, categorical data involving roles are typically partially ordered, e.g., in an employee table, there is a hierarchy of reporting structure (project member reports to their project leader who in turn is accountable to the department head and so on) as well as incomparable roles (while the Heads of the manufacturing department and the the finance department report to the president of the organization, they do not dominate each other).

For totally-ordered attribute domains, index-based algorithms like NN algorithm [11] and BBS algorithm [14] have been shown to be superior over the nested-loop approach. However, because of the lack of a total ordering for partially-ordered attribute domains, it is unclear if index-based schemes can still maintain their competitiveness given that their effectiveness to prune the search space is reduced. To the best of our knowledge, this issue has not been investigated by any of the previous related work.

In this paper, we address the novel and important problem of evaluating skyline queries involving partially-ordered attribute domains. We propose a framework to compute such skyline queries. The basic idea is to (a) transform each partially-ordered attribute domain into two integer-domain attributes, (b) organize the transformed attributes in an existing indexing method, and compute the skyline answers via

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.

the index. We note that as the skyline computation is performed on the transformed space, false positives may arise and these have to be pruned away when answering skyline queries.

We also propose three algorithms based on the above framework. BBS$^+$ is a straightforward adaptation of BBS. Because of false positives, BBS$^+$ is no longer *progressive*, i.e., it needs to find all skyline points before answers can be returned. The second scheme, SDC (Stratification by Dominance Classification) exploits the properties of domain mappings to avoid unnecessary dominance checkings. In particular, it organizes the data into two *strata* at runtime - points that are definitely in the skyline (stratum 1) and those that may be false positives (stratum 2) . As such, it can return answers in the former category as soon as they are produced. In the third scheme, SDC$^+$, the data is partitioned into two or more strata offline so that points at stratum i cannot dominate points at stratum $i-1$. In this way, skyline points obtained from stratum $i-1$ can be returned before points in stratum i are examined. In addition, we also design two dominance classification strategies (**MinPC** and **MaxPC**) to further optimize the performance of SDC and SDC$^+$.

We have implemented the three proposed schemes, and evaluated their performance against two variants of the block nested-loop scheme: BNL that operates on the native domains and BNL$^+$ that operates on the transformed space. Our results show that our proposed techniques outperform the BNL variants by a wide margin (between a factor of 2 and 16), with SDC$^+$-MinPC offering the best performance both in terms of response time as well as progressiveness.

2. RELATED WORK

A large number of algorithms have been developed to compute skyline queries. These can be categorized into non-index-based (e.g., *block nested loop* [4], *divide and conquer* [4]), and index-based (e.g., *B-tree* [4], *bitmap* [18], *index* [18], *nearest neighbor* [11], *BBS* [14]). As expected, the non-index-based strategies are typically inferior to the index-based strategies. It also turns out the index-based schemes can progressively return answers without having to scan the entire data input. The nearest neighbour scheme, which applies the divide and conquer framework on datasets indexed by R-trees, was shown to be superior over earlier schemes in terms of overall performance [11]. BBS was also shown to be I/O optimal [14] and to outperform nearest neighbour scheme. It is thus considered currently the state-of-the-art skyline algorithm reported in the literature. An evaluation of these schemes can be found in [14]. More recently, algorithm to answer skyline queries over distributed sources has also been proposed [3]. Now, all these work on skyline query evaluation handle only totally-ordered attribute domains.

Another related direction is the work on preference query systems. A framework for *quantitative* preference queries that rank answers based on scoring functions has been proposed in [2], and performance issues have been addressed in work such as [8]. Both [7] and [9] have separately proposed frameworks for *qualitative* preference queries that deal with binary preference relations between tuples. Skyline queries can be seen as a special class of pareto preference queries [9, 10]. While there are several operators designed to evaluate preference queries (e.g., *winnow* operator [6], *Best Matches Only* [9] and the *Best* operator [19]), these schemes are designed for more general preference queries. Moreover, they require at least one scan through the dataset, making it unattractive for producing fast initial response time.

Algorithm BBS (T, S)
Input: T is an R-tree.
S is an intermediate set of skyline points.
Output: Set of skyline points.
1) Initialize heap H to be empty;
2) Insert all entries in the root node of T into heap H;
3) **while** (H is not empty) **do**
4) Remove top entry e from H;
5) **if** (e is an internal entry) **then**
6) **if** (e is not dominated by any entry in S) **then**
7) **for each** child entry e_i of e **do**
8) **if** (e_i is not dominated by any entry in S) **then**
9) Insert e_i into H;
10) **else**
11) S = `UpdateSkylines`(e, S);
12) **return** S;

Algorithm `UpdateSkylines` (e, S)
Input: e is a data point in some leaf node of an R-tree.
S is an intermediate set of skyline points.
Output: Return an updated set S.
1) **for each** $p \in S$ **do**
2) **if** (e is dominated by p) **then**
3) **return** S;
4) Insert e into S;
5) **return** S;

Figure 1: Algorithm BBS

Computing the skyline of a set of points is also known as the maximum vector problem [12]. Early works on solving the maximum vector problem typically assume that the points fit into the main memory. Algorithms devised include divide-and-conquer paradigm [12], parallel algorithms [17] and those that are specifically designed to target at 2 or very large number of dimensions [13]. Other related problems include top k [5], nearest neighbor search [16], convex hull [16], and multi-objective optimization [15]. These related problems and their relationship to skyline queries have been discussed in [4].

For the rest of this section, we shall review the BBS algorithm which our proposed algorithms are based upon. An overview of the BBS algorithm [14], is shown in Fig. 1. The set of skyline points is computed by invoking BBS$(R, \emptyset)$, where R is an R-tree index and S, which represents an intermediate set of computed skyline points, is initialized to the empty set[1].

The algorithm recursively traverses the R-tree, performs a nearest neighbour search to find regions/points that are not dominated by the current skyline points in S, and inserts these into a main-memory heap structure H. Because BBS visits entries in ascending order of their distances from the origin, each computed point is guaranteed to be a skyline point, and hence can be returned to the user immediately.

Note that our proposed algorithms (to be described in Section 4) are all based on the framework of BBS shown in Fig. 1 with modifications mainly to the `UpdateSkylines` function.

[1]We present a slightly more general form of the BBS algorithm (with an input parameter S) to facilitate the presentation of our proposed schemes in Section 4.

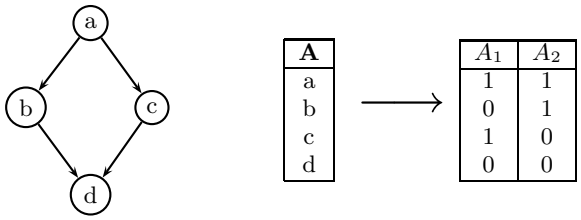


Figure 2: Example of domain transformation

3. MOTIVATION

In this section, we consider the possible evaluation strategies and motivate our proposed algorithms for processing skyline queries with partially-ordered attribute domains. For convenience, we refer to such queries as *partially-ordered skyline queries* (or POS-queries) in contrast to the *totally-ordered skyline queries* (or TOS-queries) that involve only totally-ordered attribute domains.

The most direct method to process POS-queries is to apply the well-known block nested loop approach (BNL) [4], which is the simplest and most versatile approach that works for all types of attribute domains. However, the performance of BNL has been shown to be inferior to that of index-based approaches (such as NN algorithm [11] and BBS algorithm [14]) due to the pruning effectiveness of index-based methods. Another limitation of BNL is that it is a "blocking" algorithm and lacks progressiveness (i.e., answers can only be returned after all skylines are computed).

Another strategy to evaluate POS-queries is to try to leverage the effectiveness of previous index-based approaches for TOS-queries by first transforming the partially-ordered attribute domains into totally-ordered domains such that the partial ordering of the original domains are "preserved" in the transformed domains. The most obvious transformation technique is to map each partially-ordered attribute domain into a set of boolean attribute domains, as illustrated by the following simple example.

Example 3.1 Consider the simple poset shown in Fig. 2 for an attribute A consisting of four domain values $\{a, b, c, d\}$ indicated by the nodes in the DAG representation shown. Attribute A can be transformed into a set of two boolean-valued attributes $\{A_1, A_2\}$ depicted by the mapping tables. Thus, given two records r and r', $r.A$ dominates $r'.A$ if r also dominates r' in the transformed domain (i.e., $r.A_1$ dominates $r'.A_1$ and $r.A_2$ dominates $r'.A_2$). □

By applying a suitable partial-to-total domain mapping to each partially-ordered attribute, the collection of transformed attributes is now amenable to be indexed using one of the efficient techniques proposed for TOS-queries (e.g., [11, 14]), which is particularly convenient for set-valued attribute domains. However, this boolean transformation suffers from the well-known "dimensionality curse" problem which will result in a large number of transformed boolean-valued attributes when the size of the partially-ordered attribute domain is large. Thus, the simple boolean mapping is not suitable for index-based methods.

4. AN INTERVAL-BASED APPROACH

To both enable the use of efficient index-based techniques (that are designed for totally-ordered attributes) as well as avoid the "dimensionality curse" problem with using simple domain transformations, the approach that we propose is a "middle-ground" solution that is based on using an approximate, space-efficient domain transformation. In a nutshell, our approach is based on using an approximate *interval* representation (in the form of a pair of integer attributes) for each partially-ordered attribute. This strategy, which increases the dimensionality by one for each partially-ordered attribute, provides a reasonable and practical approximate domain mapping that is amenable to efficient indexing.

In this section, we present three novel algorithms, namely, BBS^+, SDC, and SDC^+, that are all based on the interval-domain mapping idea to process POS-queries.

4.1 Basic Idea

For each partially-ordered attribute A_i with domain D_i, our approach constructs a one-to-one domain mapping f_i that transforms each value $v \in D_i$ into an interval $f_i(v) \in \mathcal{N} \times \mathcal{N}$, where $\mathcal{N}$ denotes the set of natural numbers. The domain mapping f_i is defined such that for any pair of distinct values $v, v' \in D_i$, if $f_i(v)$ contains $f_i(v')$, then v dominates v'.

Example 4.1 Consider again the poset for attribute A in Fig. 2. We can construct the following mapping f:

$$f(v) = \begin{cases} [0,3] & \text{if } v = a, \\ [0,2] & \text{if } v = b, \\ [1,3] & \text{if } v = c, \\ [1,2] & \text{if } v = d. \end{cases}$$

Here, f is an isomorphic mapping in the sense that v dominates v' iff $f(v)$ contains $f(v')$. □

In general, since the transformed values $f_i(v)$ is an approximate representation of the actual values v, it is possible for a pair of transformed attribute values to be incomparable (i.e., neither one of the transformed values contains the other) even though the original attribute values are actually comparable. This implies that when skyline points are computed using the transformed attribute values, it is possible to have *false positives* – points that are considered skylines in the transformed domains but are actually not skylines in the original domains. While appropriate domain mappings can be constructed for the special case of hierarchical partial orders (i.e., trees) to avoid false positives, false positives are generally inevitable for non-hierarchical partial orders. Therefore, skyline computation algorithms that are based on approximate domain representations need to take into account of false positives.

The idea of our proposed algorithms comprises two main steps:

(S1) For each partially-ordered attribute A_i, construct a domain mapping function f_i to transform its domain values v to $f_i(v)$. This effectively replaces each A_i attribute with two integer-domain attributes.

(S2) Organize the transformed data using an efficient indexing method, and use it to compute the skyline points taking into account of possible false positives.

We note that the above two steps are orthogonal, i.e., the design of the domain mapping function is independent of the choice of the index-based skyline computation algorithm. Moreover, any existing domain mapping functions

and index-based skyline computation schemes could be used in the two steps. As a first cut, we have adapted the encoding scheme of [1] in step (S1) and employ the BBS scheme [14], which has been shown to be very efficient for TOS-queries, in step (S2).

Step (S1) is discussed in Section 4.3; and the three algorithms that we propose for step (S2) are presented in Sections 4.4 to 4.6. Our first algorithm, BBS$^+$, which is the least progressive, is a simple extension of BBS that explicitly detects for false positives as the skyline points are computed. Both our second and third algorithms, SDC and SDC$^+$, exploit properties of domain mappings to avoid unnecessary dominance checkings. While SDC stratifies the data at runtime, SDC$^+$ creates the strata offline. SDC$^+$ is the most progressive, and processes the data in stages in such a way that there are no false positives in the intermediate results. Section 4.5.3 presents dominance classification schemes to further optimize the performance of SDC and SDC$^+$.

4.2 Definitions

We first introduce some notations and definitions.

Let $\mathcal{A} = \{A_1, A_2, \cdots A_n\}$ denote the set of attributes of interest, where $\mathcal{A} = \mathcal{A}_{total} \cup \mathcal{A}_{partial}$ with $\mathcal{A}_{total}$ and $\mathcal{A}_{partial}$ denote, respectively, the subset of totally- and partially-ordered attributes. For each attribute $A_i \in \mathcal{A}$, we use $(D_i, \preceq_i)$ to denote the partially order set (or poset) for its domain values D_i. Each $\preceq_i$ is a reflexive, antisymmetric, and transitive binary relation on D_i. We denote by $\prec_i$ the strict ordering associated with D_i; i.e., $y \prec_i x$ if $y \preceq_i x$ and $x \neq y$. Given $x, y \in D_i$, x and y are said to be *comparable* if either $y \prec_i x$ or $x \prec_i y$; otherwise, they are said to be *incomparable*. We say that x *dominates* y if $y \prec_i x$. A value $v \in D_i$ is a *maximal value* (resp. *minimal value*) if there is no value $v' \in D_i$ such that $v \prec_i v'$ (resp. $v' \prec_i v$).

Consider a finite set of data records R over the set of attributes in $\mathcal{A}$; i.e., $R \subseteq D_1 \times D_2 \times \cdots \times D_n$. Given two records $r_1, r_2 \in R$, we say that r_1 dominates r_2, denoted by $r_2 \prec r_1$, if (1) $r_2.A_i \preceq_i r_1.A_i$ for each attribute $A_i \in \mathcal{A}$, and (2) there exists some $A_j \in \mathcal{A}$ such that $r_2.A_j \prec_j r_1.A_j$.

Each partial order $(D_i, \preceq_i)$ can be represented by a DAG $G_i = (D_i, E_i)$, where $(v, w) \in E_i$ if $w \preceq_i v$ and there does not exist another value $x \in D_i$ such that $w \preceq_i x \preceq_i v$. For simplicity and without loss of generality, we assume that G_i is a single connected component.

For each partially-ordered attribute $A_i \in \mathcal{A}_{partial}$ with domain D_i, let $f_i : D_i \rightarrow \mathcal{N} \times \mathcal{N}$ denote the mapping function constructed for A_i that maps each value $v \in D_i$ into some interval of values (i.e., $f_i(v) = [v_1, v_2]$, $v_1, v_2 \in \mathcal{N}$) such that for any pair of distinct values $v, v' \in D_i$, if the interval $f_i(v)$ contains the interval $f_i(v')$, then v dominates v'.

Based on the transformed values for partially-ordered attributes, we can define a more restrictive form of dominance, called *m-dominance*, as follows. Given two records $r_1, r_2 \in R$, we say that r_1 *m-dominates* r_2, denoted by $r_2 \prec^m r_1$, if (1) $r_2.A_i \preceq_i r_1.A_i$ for each attribute $A_i \in \mathcal{A}_{total}$; (2) $f_i(r_2.A_i)$ is equal to or contained in $f_i(r_1.A_i)$ for each attribute $A_i \in \mathcal{A}_{partial}$; and (3) there exists (a) some $A_j \in \mathcal{A}_{total}$ such that $r_2.A_j \prec_j r_1.A_j$, or (b) some $A_j \in \mathcal{A}_{partial}$ such that $f_j(r_2.A_j)$ is contained in $f_j(r_1.A_j)$.

Observe that m-dominance is a stronger form of dominance in that if $r_2 \prec^m r_1$, then $r_2 \prec r_1$; but the converse does not necessarily hold.

The definition of dominance between records can be further generalized to between a record $r \in R$ and a subset of records $e \subseteq R$ as follows: we say that r dominates e (resp, r m-dominates e), denoted by $e \prec r$ (resp, $e \prec^m r$), if r dominates (resp, m-dominates) each record in e.

In both algorithm BBS as well as our proposed algorithms, we refer to the data points maintained in S as *intermediate skyline points*. In the case of BBS (which deals with only totally-ordered attributes), an intermediate skyline point is guaranteed to be a definite skyline point; thus, once a point is inserted into S, it can be output immediately. On the other hand, for two of our proposed algorithms (BBS$^+$ and SDC), the intermediate skyline points in S could be *false positives*.

4.3 Domain Mapping Function

For each partially-ordered attribute A_i, the domain mapping function f_i that we use to transform its domain D_i is adapted from the encoding scheme of [1] and works as follows: a spanning tree ST_i is first computed from the DAG G_i, and ST_i is then traversed in postorder with each node v being assigned a unique postorder number $post(v)$. Then, $f_i(v)$ is given by $[x, y]$, where $y = post(v)$ and x is the smallest postorder number assigned to a descendant of v. This mapping scheme satisfies the following *domain mapping property*:

If $(v, v') \in E_i$ is also an edge in the spanning tree ST_i, then $f_i(v)$ contains $f_i(v')$.

It follows that given any two nodes v and v' in G_i, $f_i(v)$ contains $f_i(v')$ iff there is a path from v to v' in the spanning tree ST_i.

Example 4.2 Refer once more to the poset for attribute A in Fig. 2. The domain mapping function f for A is constructed as follows. Let the spanning tree computed from the poset be equivalent to the DAG shown but without the edge (c, d). Then, the postorder numbers assigned to a, b, c, and d are respectively, 4, 2, 3, and 1; and their respective assigned interval values are $[1, 4]$, $[1, 2]$, $[3, 3]$, and $[1, 1]$. Observe that although c dominates d (w.r.t. the original domain), c does not m-dominate d (w.r.t. the transformed domain). □

Note that there are other alternative schemes that could be used for the domain mapping function (e.g., [20]). However, as our focus in this paper is mainly on skyline computation algorithms, we have selected a simple mapping function for our work here. It is important to point out that our optimized algorithms (to be presented in Sections 4.5 and 4.6) are orthogonal to the choice of the mapping function. Indeed, in Section 4.7, we present a new domain mapping scheme that is inspired by the properties of our optimizations to improve the efficiency and progressiveness of skyline computation.

4.4 Algorithm BBS$^+$

In this section, we present our first algorithm, called BBS$^+$, which represents the simplest extension of BBS [14] to process POS-queries.

Algorithm BBS$^+$ is similar to BBS (shown in Fig. 1) except for the following two changes shown in Fig. 3. First, since the R-tree index used in BBS$^+$ is based on transformed attribute domains for partially-ordered attributes,

Algorithm BBS$^+$ (T, S)
Same as Algorithm BBS in Fig. 1 except that each "dominated" comparison is replaced by a "m-dominated " comparison.

Algorithm `UpdateSkylines` (e, S)
Input: e is a data point in some leaf node of an R-tree.
S is an intermediate set of skyline points.
Output: Return an updated set S.
1) **for each** $p \in S$ **do**
2) **if** (e is dominated by p) **then**
3) **return** S;
4) **else if** (p is dominated by e) **then**
5) Delete p from S;
6) Insert p into S;
7) **return** S;

Figure 3: Algorithm BBS$^+$

the two dominance comparisons in BBS (steps 6 and 8) are replaced with m-dominance comparisons in BBS$^+$. Second, since there could be false positives in the set of intermediate skyline points maintained in S, the `UpdateSkylines` function in BBS$^+$ needs to detect and remove any false positives (steps 4 to 5) while comparing the new data point e against the intermediate skyline points in S.

4.5 Algorithm SDC

One major drawback of BBS$^+$ is that it is a non-progressive algorithm due to the possibility of false positives in the computed intermediate skyline points. Another limitation of BBS$^+$ is that it can incur many unnecessary comparisons for dominance; in the worst case, the `UpdateSkyline` function might need to compare the new data point e against every intermediate skyline point in S.

In this section, we present our second algorithm, called SDC (Stratification by Dominance Classification), which improves over BBS$^+$ in terms of both progressiveness (by separating the intermediate skyline points into definite skylines and potential false positives) as well as speed (by avoiding unnecessary checkings for dominance).

4.5.1 Dominance Classification

To overcome the limitations of BBS$^+$, Algorithm SDC exploits two simple characterizations of partially-ordered attribute values based on their domain mapping functions.

Recall that there is a spanning tree ST_i associated with each partially-ordered attribute A_i (induced by its domain mapping f_i) that is contructed from its partial order DAG $G_i = (D_i, E_i)$. We can classify each value in D_i based on its relationship with incoming and outcoming values (w.r.t. G_i and ST_i) in two ways as follows. A value $v \in D_i$ is said to be *completely covered* if every directed incoming path to v in G_i is also in ST_i; otherwise, v is said to be *partially covered*. A value $v \in D_i$ is said to be *completely covering* if every directed outgoing path from v in G_i is also in ST_i; otherwise, v is said to be *partially covering*.

Example 4.3 Consider the poset $(D, \preceq)$ with $D = \{a, b, \cdots, j\}$ in Fig. 4, where the edges included in (excluded from) its spanning tree are indicated by solid (dotted) arrows. The set of values $\{a, b, c, d, f, h\}$ are partially covering; and the set of values $\{f, g, h, i, j\}$ are partially covered. □

The above classifications of attribute values can be easily generalized to data points as follows. A data point $r \in R$ is said to be *completely covered* if the value of each of its partially-ordered attributes is completely covered; otherwise, r is said to be *partially covered*. Similarly, $r \in R$ is said to be *completely covering* if the value of each of its partially-ordered attributes is completely covering; otherwise, r is said to be *partially covering*.

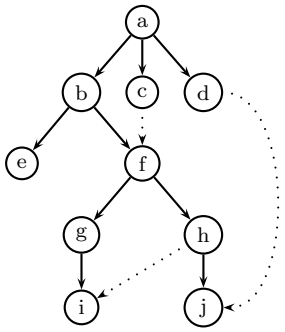


Figure 4: Example Poset $(D, \preceq)$

Based on these two orthogonal classifications, given a set of data points S, S can be partitioned into four disjoint subsets:

$$S = S_{c,c} \cup S_{c,p} \cup S_{p,c} \cup S_{p,p}$$

where each $S_{i,j}$ denote the subset of points in S that are (1) partially covered (resp. completely covered) if $i = p$ (resp. $i = c$), and (2) partially covering (resp. completely covering) if $j = p$ (resp. $j = c$).

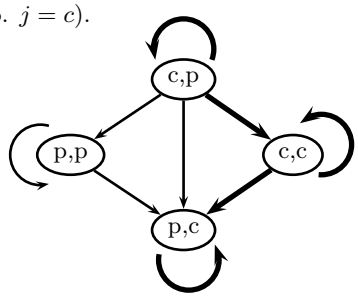


Figure 5: Dominance Graph DG

The dominance relationship among the four subsets of data points is depicted by the *dominance graph* shown in Fig. 5 and has the following property:

LEMMA 4.1. *A data point $p \in S_{i,j}$ dominates another data point $p' \in S_{i',j'}$ only if there is a directed edge (normal or bold) from node (i, j) to node (i', j') in the* dominance graph *DG shown in Fig. 5.*

Observe that the dominance relationship among the four subsets in Fig. 5 is reflexive, antisymmetric, and transitive. The significance of the bold edges will be explained in Section 4.5.3.

In the following subsections, we present three optimizations used in SDC that are based on the properties of the dominance graph.

4.5.2 Minimizing Dominance Comparisons

To avoid unnecessary dominance comparisons, SDC exploits Lemma 4.1 to organize the intermediate set of skyline points into four subsets. In contrast to BBS$^+$ which com-

Algorithm SDC (T, S)
Same as Algorithm BBS$^+$ in Fig. 3.

Algorithm `UpdateSkylines` (e, S)
Input: e is a data point in some leaf node of an R-tree.
S is an intermediate set of skyline points,
where $S = S_{c,c} \cup S_{c,p} \cup S_{p,c} \cup S_{p,p}$.
Output: Return an updated set S.
1) Let (i, j) be the category that e belongs, $i, j \in \{c, p\}$;
2) Let $C = \{(x, y) \mid$ edge from (x,y) to (i,j) in DG$\}$;
3) Let $C' = \{(p, y) \mid$ edge from (i,j) to (p,y) in DG$\}$;
4) **for each** $p \in S_{x,y}$, $(x, y) \in C \cup C'$ **do**
5) $ret =$ `CompareDominance` (e, p);
6) **if** $(ret == 1)$ **then**
7) **return** S;
8) **else if** $(ret == -1)$ **then**
9) Delete p from $S_{x,y}$;
10) Insert e into $S_{i,j}$;
11) **return** S;

Algorithm `CompareDominance` (x, y)
Input: x and y are two data points.
Output: Return -1 if x dominates y, or
1 if x is dominated by y, or
0 if neither x nor y dominates each other.
1) **if** (x is m-dominated by y) **then**
2) **return** 1;
3) **else if** (y is m-dominated by x) **then**
4) **return** -1;
5) **if** (x is partially covering) **and** (y is partially covered) **then**
6) **if** (x is dominated by y) **then**
7) **return** 1;
8) **else if** (y is dominated by x) **then**
9) **return** -1;
10) **return** 0;

Figure 6: Algorithm SDC

pares each new leaf entry e against all the intermediate skyline points in S, SDC only compares e against the necessary subsets of intermediate skyline points.

Referring to SDC's `UpdateSkylines` function in Fig. 6, step 1 first determines the category, denoted by $S_{i,j}$, of the input leaf entry e. Once this is known, step 2 then selects the categories of data points, denoted by C, that can possibly dominate e (based on Lemma 4.1), and step 3 selects the categories of data points, denoted by C', that e can possibly dominate (to be explained in Section 4.5.4). Steps 4 to 9 then compare e against the intermediate skyline points that belong to the selected categories by using an optimized function called `CompareDominance`. This function accepts two input data points x and y and returns -1 if x dominates y, 1 if y dominates x, and 0 otherwise; the details of `CompareDominance` are elaborated in Section 4.5.3.

4.5.3 Optimizing Dominance Comparisons

The second optimization in SDC aims to maximize the use of dominance comparisons that are based on the transformed domains (i.e., m-dominate comparisons) over dominance comparisons that are based on the original domains for partially-ordered attributes. This optimization is useful when dominance comparisons based on the original domains (e.g., set-valued domains) are more expensive to evaluate than dominance comparisons based on the transformed domains which involve two integer comparisons. Therefore, to improve performance for such cases, the more costly dominance comparisons involving the original domains for partially-ordered attributes should be used only as a last resort.

SDC exploits the following property to maximize m-dominate comparisons:

LEMMA 4.2. *If x is a completely covering point or y is a completely covered point, then x dominates y iff x m-dominates y.*

This lemma is depicted by the bold edges in Fig. 5: if $p \in S_{i,j}$, $p' \in S_{i',j'}$, and there is a bold edge from (i, j) to (i', j') in DG, then p dominates p' iff p m-dominates p'.

We briefly explain the correctness of the above lemma. Clearly, if x m-dominates y, then by the domain mapping property, x must necessarily dominate y. On the other hand, if x dominates y, then for each partially-ordered attribute $A_i \in \mathcal{A}_{partial}$, there is at least one directed path p from $x.A_i$ to $y.A_i$ in G_i. Since x is a completely covering point or y is a completely covered point, this implies that the path p must also be in ST_i which means that $f_i(x.A_i)$ contains $f_i(y.A_i)$; therefore, x m-dominates y.

Based on Lemma 4.2, SDC performs dominance comparisons in the `UpdateSkylines` function by using a new function called `CompareDominance` (shown in Fig. 6). `CompareDominance` first compares x and y using m-dominance, and only when the points are incomparable in terms of m-dominance but could be comparable in terms of dominance (by Lemma 4.2), `CompareDominance` then resorts to comparing them using their original domain values.

4.5.4 Enabling Progressive Computation

The third optimization in SDC aims to enable skyline points to be computed progressively.

SDC exploits the following property to identify definite skyline points from the intermediate skyline points.

LEMMA 4.3. *An intermediate skyline point that is completely covered is a definite skyline point.*

The correctness of the above lemma is based on the property of the BBS algorithm [14], and Lemmas 4.1 and 4.2.

Therefore, based on Lemma 4.3, the `UpdateSkylines` function in SDC is optimized by checking for false positives only from intermediate skyline points that are partially covered; this explains step 3 of SDC's `UpdateSkylines` which selects the categories of data points (denoted by C') that the input data point e could dominate. Thus, SDC is more efficient than BBS$^+$ which checks for false positives from all the intermediate skyline points in S.

More importantly, SDC enables the set of skyline points to be computed progressively: each newly determined intermediate skyline point e that is completely covered (i.e., $e \in S_c^c \cup S_p^c$) can be output immediately since it is a definite skyline point.

4.6 Algorithm SDC$^+$

In this section, we present our third algorithm, called SDC$^+$, which aims to further increase the progressiveness of SDC. Recall that SDC essentially organizes the intermediate skyline points into two strata at runtime - the completely covered intermediate skyline points (stratum 1) and the intermediate skyline points that are partially covered (stratum 2). While skyline points in stratum 1 can be progressively returned, those in stratum 2 could be false positives and

therefore need to be compared against all the intermediate skyline points to verify that they are indeed definite skyline points. This limitation restricts the progressiveness of SDC since the skyline points in stratum 1 are generally only a small percentage of the entire set of skyline points as indicated by our experimental results. To increase progressiveness, SDC$^+$ statically partitions the data into two or more strata.

4.6.1 Data Stratification

In SDC$^+$, the set of data points R is partitioned into a sequence of subsets called *strata* $< R_0, R_1, \cdots, R_k >$ for some value k, such that each $R_i \subseteq R$ and $\bigcup_{i=0}^{k} R_i = R$. By judiciously partitioning the data into separate strata, the skyline points can be computed one stratum at a time starting from R_0 to R_k such that each "local" skyline point in a stratum R_i can not be dominated by skyline points in the succeeding strata (i.e., R_j, $j > i$), which therefore guarantees that none of the computed skyline points from each stratum are false positives (as explained earlier). Thus, by computing skyline points from a sequence of smaller subsets instead of from a single large set, the skyline computation becomes more progressive.

An obvious strategy is to organize the data points based on the dominance graph into the following sequence of four strata: $< R_{c,p}, R_{c,c}, R_{p,p}, R_{p,c} >$. However, as the last two strata $R_{p,p}$ and $R_{p,c}$ are generally large which limits progressiveness, SDC further refines the last two strata based on the notion of *uncovered level* of attribute values and data points.

We define the *uncovered level* of an attribute value $v \in D_i$, denoted by $\mathcal{L}(v)$, as the maximum number of edges in a directed path to v that are in G_i but not in ST_i. The uncovered level of each value v can be computed recursively as follows:

$$\mathcal{L}(v) = \begin{cases} 0 & \text{if } v \text{ is a maximal value in } D_i, \\ \max\limits_{(w,v)\in E_i} \{\mathcal{L}(w) + c(w,v)\} & \text{otherwise.} \end{cases} \quad (1)$$

where $c(w,v) = 0$ if (w,v) is an edge in ST_i, and $c(w,v) = 1$ otherwise.

Example 4.4 Consider again the poset $(D, \preceq)$ in Fig. 4. We have $\mathcal{L}(v) = 0$ if $v \in \{a, b, c, d, e\}$, $\mathcal{L}(v) = 1$ if $v \in \{f, g, h, j\}$, and $\mathcal{L}(v) = 2$ if $v = i$. □

The *uncovered level of a data point* r, denoted by $\mathcal{L}(r)$, is defined as the maximum of the uncovered levels of its partially-ordered attribute values; i.e.,

$$\mathcal{L}(r) = \max_{A_i \in \mathcal{A}_{partial}} \{\mathcal{L}(r.A_i)\}.$$

The notion of uncovered level is useful for refining the dominance relationship among partially covered points as given by the following result.

LEMMA 4.4. *A partially covered data point p can not dominate another partially covered data point p' if $\mathcal{L}(p) > \mathcal{L}(p')$.*

The correctness of the above lemma follows from the fact that for any pair of attribute values $v, v' \in D_i$, v dominates v' iff there is a directed path from v to v' in G_i.

Lemma 4.4 provides a simple and effective way to further partition the data points in the strata $R_{p,p}$ and $R_{p,c}$ to increase progressiveness. $R_{p,p}$ is partitioned into $k = \max\limits_{r\in R_{p,p}} \{\mathcal{L}(r)\}$ strata, where each stratum $R^i_{p,p}$, $1 \le i \le k$, represents the subset of data points in $R_{p,p}$ with an uncovered level of i. It follows from Lemma 4.4 that intermediate skyline points from stratum $R^i_{p,p}$ will not be dominated by any data point in $R^j_{p,p}$, $j > i$. Similarly, $R_{p,c}$ is partitioned into $k' = \max\limits_{r\in R_{p,c}} \{\mathcal{L}(r)\}$ strata, where each stratum $R^i_{p,c}$, $1 \le i \le k'$, represents the subset of data points in $R_{p,c}$ with an uncovered level of i.

Thus, SDC$^+$ partitions the data points in R into $(k+k'+2)$ strata, where the data points in each stratum $R^i_{x,y}$ is (conceptually) indexed using a separate R-tree $T^i_{x,y}$. The strata is processed in the following sequence: $< R_{c,p}, R_{c,c}, R^1_{p,p}, R^1_{p,c}, R^2_{p,p}, R^2_{p,c}, \cdots >$ as shown in Fig. 7. Each stratum is processed by calling the function SDC$^+$`-sub` with two input parameters: the R-tree T for the stratum, and the intermediate set of skyline points S computed so far. The set of skyline points returned by SDC$^+$`-sub` for the input stratum can be output immediately as they are all definite skyline points. For the special cases of strata $R_{c,p}$ and $R_{c,c}$, the intermediate skyline points (which are completely covered) can be output even earlier: by Lemma 4.3, each intermediate skyline point e is a definite skyline point and can be output before it is inserted into L (step 12 in Fig. 7). Note that SDC$^+$`-sub` is similar to BBS$^+$ except for some minor changes.

SDC$^+$ progressively computes the skyline points in ascending order of their uncovered levels starting with the completely covered data points in $R_{c,p}$ and $R_{c,c}$ (which have uncovered level of 0) in steps 1 and 2. Steps 3 to 7 compute the partially covered skyline points. SDC$^+$ organizes the computed skyline points using two main sets: S stores the definite skyline points computed from the processed strata, while L stores the skyline points computed from the current stratum being processed (which could contain false positives). Thus, the `UpdateSkylines` function in SDC$^+$ processes an input data point e against L and S separately. First, e is compared against L (steps 1 to 6) to both check if e could be dominated by the points in L as well as check if there are any false positives in L that could be dominated by e. Next, e is compared against S (steps 7 to 11) to check if e could be dominated by the points in S. SDC$^+$ uses the same `CompareDominance` function as SDC.

Although each stratum is conceptually indexed independently by a separate R-tree, multiple consecutive strata can actually be indexed using a single R-tree by including an additional stratum number attribute for indexing to facilitate the conceptual approach of processing the sequence of strata.

4.7 Optimizing Dominance Classification

In this section, we present our final optimization technique, which is applicable to both SDC and SDC$^+$, that aims to reduce the number of dominance comparisons and maximize the use of m-dominance comparisons. The idea is to optimize the construction of the spanning tree for each partially-ordered attribute (Section 4.3) to maximize the occurence of certain dominance categories of attribute values over others. The following example illustrates the effect of the spanning tree structure on the dominance classification of the attribute values.

Example 4.5 Consider the two almost similar spanning trees ST and ST' that differ by only one edge for the same DAG in Figs. 8(a) and (b). The solid edges represent the

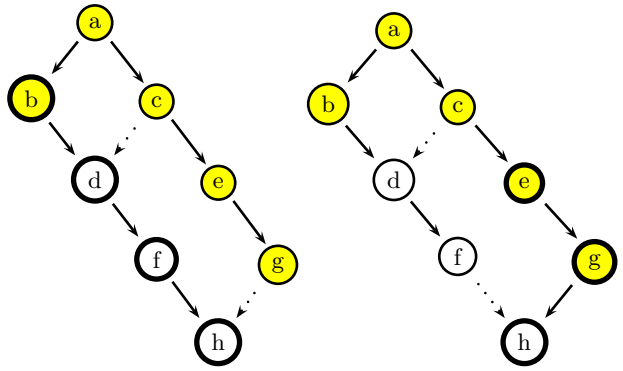
Algorithm SDC$^+$ ($\mathcal{T}$)
Input: $\mathcal{T} = \{T_{c,p}, T_{c,c}, T^1_{p,p} \cdots T^k_{p,p}, T^1_{p,c} \cdots T^{k'}_{p,c}\}$ is a set of $(k + k' + 2)$ R-trees.
Output: Set of skyline points.
1) S = SDC$^+$-sub($T_{c,p}$, $\emptyset$);
2) $S = S \cup$ SDC$^+$-sub($T_{c,c}$, S);
3) **for** $i = 1$ **to** $\max\{k, k'\}$ **do**
4) **if** $(i \leq k)$ **then**
5) $S = S \cup$ SDC$^+$-sub($T^i_{p,p}$, S);
6) **if** $(i \leq k')$ **then**
7) $S = S \cup$ SDC$^+$-sub($T^i_{p,c}$, S);
8) **return** S;

Algorithm SDC$^+$-sub (T, S)
Input: T is an R-tree.
S is an intermediate set of skyline points.
Output: Return an updated set L.
0) Initialize L to be empty;
1-10) Same as Algorithm BBS$^+$ in Fig. 3 except that in steps 6 and 8, S is replaced by $S \cup L$.
11) L = `UpdateSkylines`(e, S, L);
12) **return** L;

Algorithm `UpdateSkylines` (e, S, L)
Input: e is a data point in some leaf node of an R-tree.
S is an intermediate set of skyline points, where $S = S_{c,c} \cup S_{c,p} \cup S_{p,c} \cup S_{p,p}$.
L is the set of skyline points generated from the current stratum.
Output: Return an updated set L.
1) **for each** $p \in L$ **do**
2) ret = `CompareDominance` (e, p);
3) **if** $(ret == 1)$ **then**
4) **return** L;
5) **else if** $(ret == -1)$ **then**
6) Delete p from L;
7) Let (i, j) be the category that e belongs, $i, j \in \{c, p\}$;
8) Let $C = \{(x, y) \mid$ edge from (x,y) to (i,j) in DG, $(x, y) \neq (i, j)\}$;
9) **for each** $p \in S_{x,y}$, $(x, y) \in C$ **do**
10) **if** (`CompareDominance` $(e, p) == 1$) **then**
11) **return** L;
12) Insert e into L;
13) **return** L;

Algorithm `CompareDominance` (x, y)
Same as that in Algorithm SDC in Fig. 6.

Figure 7: Algorithm SDC$^+$



(a) ST **(b)** ST'

Figure 8: Optimizing Dominance Classification

edges that are in the spanning tree, and the dotted edges represent the edges that are in the DAG but excluded from the spanning tree. Completely and partially covered values are represented by shaded and unshaded nodes, respectively; and completely and partially covering values are represented by nodes with thick and thin lines, respectively. We observe the following differences: (1) b, d, and f are completely covering in ST but partially covering in ST'; and (2) e and g are partially covering in ST but completely covering in ST'. □

The above example shows that the dominance classification of the values for an attribute can be varied (to some extent) by changing the structure of the spanning tree constructed from the DAG representation of its poset.

More generally, the structure of the DAG determines whether the nodes are completely or partially covered, while the structure of the spanning tree determines whether the nodes are completely or partially covering. Specifically, a node v in a DAG G is a partially covered node in any spanning tree of G if v or an ancestor of v has multiple incoming edges in G; otherwise, v is a completely covered node in any spanning tree of G. The choice of the edges included in the spanning tree will determine whether the nodes are partially or completely covering. In particular, if an edge (v, w) is excluded from the spanning tree, then each ancestor node of v (including v itself) will be partially covering.

Thus, the spanning tree can affect the relative number of nodes between the categories (p, c) and (p, p), and between the categories (c, c) and (c, p). Comparing the two categories (p, p) and (p, c), having more data points in (p, p) relative to (p, c) can reduce the number of dominance comparisons since data points in (p, p) need not be compared against data points in (c, c) (refer to Fig. 5). On the other hand, having more data points in (p, c) can maximize the use of m-dominance comparisons, whereas comparisons involving data points in (p, p) must be performed in terms of the actual domain values. Thus, the two categories (p, p) and (p, c) have different tradeoffs. Comparing the categories (c, p) and (c, c), having more data points in (c, c) relative to (c, p) is better for performance because it not only reduces the number of dominance comparisons (since points in (c, c) need not be compared against points in (p, p)), but it also enables all the comparisons to be done using m-dominance. Thus, it is better to maximize the number of (c, c) nodes relative to the number of (p, c) nodes in the spanning tree.

Based on the above analysis, there are two main strategies, referred to as **MinPC** and **MaxPC**, for optimizing the spanning tree construction:

MinPC: Minimizes the number of (p, c) nodes relative to the number of (p, p) nodes.

MaxPC: Maximizes the number of (p, c) nodes relative to the number of (p, p) nodes.

For the above strategies, the primary optimization criterion is to minimize/maximize the number of (p, c) nodes (relative to the number of (p, p) nodes), and maximizing the number of (c, c) nodes (relative to the number of (c, p) nodes) is

Algorithm `OptimizeSpanningTree` (G)
Input: $G = (D, E)$ is the DAG representation of a poset for a partially-ordered attribute.
Output: A spanning tree ST.

1) Initialize ST to be G;
2) **for each** node v in ST in topological order **do**
3) **if** (v has more than one parent in ST) **or** (v's parent is classified as (p, c) in ST) **then**
4) Classify v as (p, c);
5) **else**
6) Classify v as (c, c);
7) Let $V = \{v \in D \mid |parent(v)| > 1\}$;
8) **while** (V is not empty) **do**
9) Choose $v \in V$, $w \in parent(v)$ such that $|PCSet_v(w)| \geq |PCSet_{v'}(w')|, \forall\, v' \in V, w' \in parent(v')$; Break ties by choosing $v \in V$, $w \in parent(v)$ such that $|CCSet_v(w)| \leq |CCSet_{v'}(w')|, \forall\, v' \in V, w' \in parent(v')$;
10) **for each** $u \in parent(v), u \neq w$ **do**
11) Delete (u, v) from ST;
12) Update u's classification from (x, y) to (x, p);
13) **for each** $u \in PCSet_v(w)$ **do**
14) Update u's classification to (p, p);
15) **for each** $u \in CCSet_v(w)$ **do**
16) Update u's classification to (c, p);
17) Delete v from V;
18) **return** ST;

Figure 9: Algorithm to optimize spanning tree

used as a secondary criterion (see Fig. 9). In Fig. 8, the spanning trees ST and ST' are created using the **MaxPC** and **MinPC** strategies, respectively.

Note that we have also experimented with two other variations of the above strategies, where the primary and secondary optimization criteria are swapped. Our experimental results indicate that these variations performed worse than their coumterpart strategies, showing that minimizing the number of (p, c) nodes is more important than minimizing the number of (c, c) nodes.

Algorithm `OptimizeSpanningTree` in Fig. 9 takes as input the poset of a partially-ordered attribute, $G = (D, E)$, and computes a spanning tree ST from G that is optimized based on the **MinPC** strategy[2]. Steps 2 to 6 of the algorithm first initializes the spanning tree ST to be the input DAG G and classifies the nodes into either completely or partially covered nodes, with a default completely covering classification. The classification is computed by a topological traversal of ST since the category of a node depends on the categories of its ancestor (but not descendant) nodes as explained earlier. Next, steps 7 to 17 then construct a spanning tree by using a greedy heuristic to delete edges to minimize the number of (p, c) nodes. Here, $parent(v)$ denotes the set of parent nodes of a node v in ST. $PCSet_v(w)$ denotes the set of nodes in category (p, c) that would become in category (p, p) when all the incoming edges to v, except for (w, v), are deleted from ST. In other words, $PCSet_v(w)$ is the set of nodes in category (p, c) such that each node is an ancestor of some node in $parent(v) - \{w\}$. $CCSet_v(w)$ is defined similarly for nodes in category (c, c) that would become nodes in category (c, p).

We use SDC-$MinPC$ and SDC-$MaxPC$, to denote SDC that is optimized using the **MinPC** and **MaxPC** strategies, respectively. SDC$^+$-$MinPC$ and SDC$^+$-$MaxPC$ are defined similarly.

[2]Changing the comparison operator in step 9 to $\leq$ would result in the **MaxPC** strategy.

Parameter	Values
$\lvert\mathcal{A}_{total}\rvert$, # of totally-ordered attributes	2, 1, 4
$\lvert\mathcal{A}_{partial}\rvert$, # of partially-ordered attributes	1, 2
Attribute correlation	independent, anti-correlated
Poset size (# nodes)	450, 1000
Poset height (# levels)	6, 13
Data size (# data points)	500K, 1000K

Table 1: Experimental parameters and values used

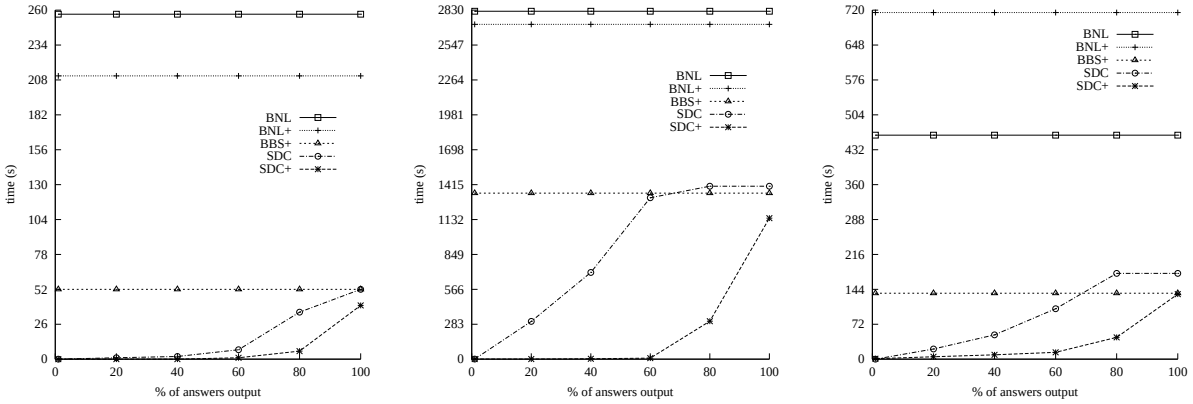
5. PERFORMANCE STUDY

To evaluate the effectiveness of our proposed algorithms, we conducted an extensive set of experiments to study their performance. Our results show that our proposed algorithms (BBS$^+$, SDC, and SDC$^+$) outperform existing techniques by a wide margin, with SDC$^+$-MinPC, which is SDC$^+$ using the **MinPC** strategy to optimize dominance classification, giving the best performance in terms of both response time as well as progressiveness.

Data Sets: We generated synthetic data sets by varying the number of attributes, the correlation among the attributes, the complexity of the posets for partially-ordered attributes, and the size of the data sets. The parameters and their values used are shown in Table 1, where the first value listed is the default value. In our experimental presentations, default parameter values are used unless stated otherwise.

For totally-ordered attributes, we used integer values from the domain $(0, 1000]$, where values are generated as described in [4] with possible correlation among different attributes. For partially-ordered attributes, we used set-valued attributes where dominance is based on set containment. The poset for each partially-ordered attribute is created by first generating a forest of trees, by varying the number of trees, their heights and branching factors. Next, the poset is then formed by randomly connecting nodes among the trees, such that two nodes can be linked only if their levels differ by one. The density of edges in the poset is controlled by the number of iterations of adding inter-tree edges and the probability of adding an edge for a node. The domain of the set-valued attribute values is then derived from the constructed poset. Each data point is generated by choosing a random attribute value from its domain; in particular, for a partially-ordered attribute, a value is selected by randomly choosing a node from its domain's poset.

Algorithms: We compared our proposed techniques (denoted by BBS$^+$, SDC, and SDC$^+$), and two variants of the block nested-loop algorithm, denoted by BNL and BNL$^+$. BNL is the basic algorithm proposed in [4], while BNL$^+$ is our optimized extension of BNL that works in a two-stage filter-and-postprocess manner as follows. First, BNL$^+$ executes the standard BNL algorithm (using the transformed attribute values) to quickly obtain a set of intermediate skyline points (possibly with false positives), which is then pipelined to a second BNL algorithm (using the actual attribute values) to eliminate any false positives. We also evaluated the effectiveness of our dominance classification optimization strategies (**MinPC** and **MaxPC**) on SDC and SDC$^+$. For our proposed algorithms, the transformed data values are indexed using R*-trees with page sizes of 4K bytes



(a) 2 numerical & 1 set attributes

(b) 1 numerical & 2 set attributes

(c) 4 numerical & 1 set attributes

Figure 10: Varying the number of numerical/set-valued attributes.

and node capacity of 50. Each R-tree index is constructed by first scanning the data points to extract the distinct domain values of each partially-ordered attribute. Their posets are then constructed, and each value in each poset is then mapped to an integer interval as explained in Section 4.3. Finally, the data points are then indexed with an R-tree on the set of totally-ordered attribute values and the transformed partially-ordered attribute values. Note that each entry in the index nodes has two additional bits indicating whether the entry is partially/completed covered/covering. The posets are therefore not needed once the index is built.

Our experiments were carried out on a Pentium 4 PC with a 2.4 GHz processor and 256 MB of main memory running the Linux operating system.

5.1 Response Time & Progressiveness

In this experiment, we examine the performance of the various algorithms in returning first answers as well as how fast *all* answers are returned progressively. For each algorithm, we recorded the time it took to output various percentages of the answers (20%, 40%, 60%, 80% and 100%), as well as the time it took to output the first answer (close to 0%). Although we also captured the time to output every 10 answers, up to 100, we found that the timings are generally the same as the time to output the first tuple and hence, the time to output the first answer is sufficient to illustrate the response time of the initial set of answers returned. Fig. 10(a) illustrates the performance when 2 numerical attributes and 1 set-valued attribute are used. In Figs. 10(b) and (c), we compare the algorithms' performance when the queries consist of more set-valued attributes and numerical attributes respectively.

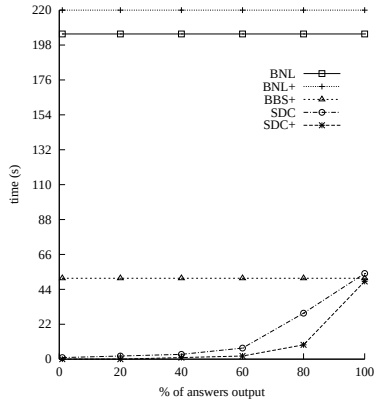
From Fig. 10(a), we can see that SDC and SDC$^+$ have fast initial response times and are progressive. Among them, SDC$^+$ has the best overall performance. There are 662 skyline points and 561 false positives in this experimental run. For BBS$^+$, it is not progressive because it cannot output any answers as they become available due to the possibility of false positives. Instead, each available answer has to be checked against the current skyline using actual set representation to ensure that it is not a false positive. Nonetheless, its performance is still better than the block nested loop algorithms.

For SDC, it is progressive as it can immediately output those intermediate skyline points that are completely covered. Furthermore, to find skyline points efficiently, intermediate skyline points are organized into subsets and m-dominate comparisons are used whenever possible. Comparing with BBS$^+$, this results in a 59% drop in actual set-valued comparisons. Consequently, the initial set of answers can be found very quickly and since most of them are definite skyline points, they can be output immediately. However, as processing continues, its progressiveness drops as remaining skyline points belong to $S_{p,p}$ and hence, cannot be output immediately. Moreover, since the various subsets are getting bigger as processing continues, this results in more comparisons and hence a poorer performance towards the end.

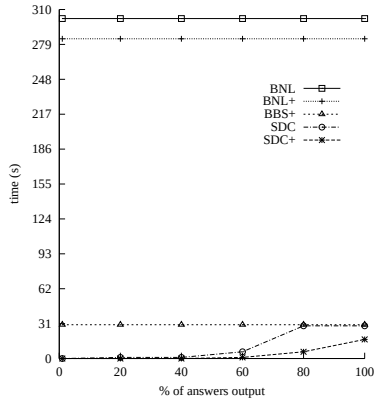
For SDC$^+$, it is clear that it is more progressive and produces answers faster than SDC. This is because the initial set of strata being processed consists of points belonging only to $S_{c,p}$ and $S_{c,c}$ and hence, any answer found can be output immediately. Since 80% of the skyline points belong to $S_{c,p}$, this explains why SDC$^+$ can output the first 80% of the answers significantly faster. Moreover, we found that SDC$^+$ incurs 30% fewer actual set-valued comparisons and 16% fewer m-dominate comparisons compared to SDC. This is because in SDC$^+$, data points belonging to subsets $S_{c,p}$ and $S_{c,c}$ (which have the highest potential of being in the skyline) are processed first while in SDC, the data points can belong to any subsets. Consequently, more comparisons which do not result in any meaningful outcome are incurred for SDC. Thus, SDC$^+$ has a better overall performance than SDC.

For BNL, its performance is relatively poor throughout as comparing using actual set representation is more expensive than comparing numerical values on the transformed data. This explains why BNL$^+$ can outperform BNL even though it requires a post-processing step.

Consider Figs. 10(b) and (c) where we increase the number of set-valued and numerical attributes respectively. It is a well known fact in the literature that the number of skyline points increases with increasing number of attributes.



(a) Increasing poset size



(b) Increasing poset height

Figure 11: Effect of poset structure.

For example, with 4 numerical attributes and 1 set-valued attribute, the number of skyline points is 8831 with 9990 false positives. Moreover, adding an additional set attribute increases the skyline points more rapidly than adding a numerical attribute. For example, in Fig. 10(b), an additional set-valued attribute alone increases the number of skyline points to 9203. As illustrated from these figures, the runtime of the algorithms increases with more numerical/set-valued attributes although their relative performance remain similar. Notice that SDC can be slower than BBS^+ after 60% of the answers are output. Furthermore, its progressiveness drops with increasing number of skyline points. This is because there are now more skyline points belonging to subsets $S_{p,p}$ and $S_{p,c}$ (they cannot be output immediately), resulting in more comparisons and thus, the progressiveness drops. Finally, notice in Fig. 10(c) that BNL^+'s performance is worse than BNL. Since each set-valued attribute is transformed to two numerical attributes, BNL^+ now need to find the skyline for a 6 numerical attributes dataset which by itself, is time-consuming. Coupled with a post-processing step using actual set representation, this results in a poorer performance compared to BNL.

Finally, by looking at the time 100% of the results are produced, we observe the algorithms' performance in terms of overall runtime. We can see that SDC^+ has the best overall runtime as it incurs fewer dominance comparisons compared to the rest.

5.2 Effect of poset structure

Size of poset. Fig. 11(a) shows the performance of the various algorithms when we increase the size of the poset from the default value of 450 nodes (in Fig. 10(a)) to 1000 nodes. We observe that the performance of our proposed algorithms remain relatively unchanged except for a slight increase in runtime for both SDC and SDC^+. Increasing the size of the poset has the effect of increasing the number of skyline points. For example, there are 1051 skyline points and 1881 false positives in this experiment. This, in turn, affects the runtime of the algorithms. We can see that BNL^+ is most significantly affected and performs worse than BNL.

Height of poset. Fig. 11(b) shows the performance of the various algorithms when we increase the height of the poset to 13 by generating a tall and relatively sparse poset to make the number of answers comparable to Fig. 10(a). This resulted in 25 strata for SDC^+. Again, the relative performance is unchanged compared to previous experiments. However, notice that both BNLand BNL^+ have a higher runtime. This is because a poset with more levels results in sets whose cardinalities are larger. Consequently, the set comparisons become more expensive and this has the largest impact on both BNLand BNL^+.

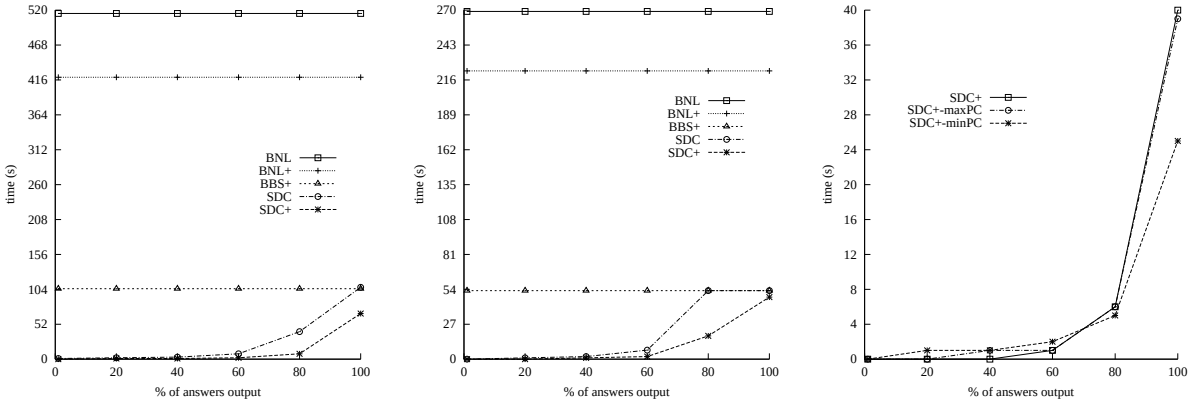
5.3 Other experiments

Effect of Large Dataset Fig. 12(a) shows the results when the size of the data set is increased to one million data points. We see that the overall runtime for all the algorithms increased significantly due to the need to process more data tuples. However, both SDC and SDC^+ still maintain an advantage over the rest by being able to produce nearly all the answers before the rest do so.

Effect of Anti-Correlated Attributes Fig. 12(b) shows the results when the totally-ordered attributes are anti-correlated. This means that if a numerical attribute of a data point has a low value for one attribute, it would have another attribute with high value and so on. From the figure, the relative performance of the various algorithms remain unchanged except for higher runtime. This is because anti-correlation increases the number of skyline points. For example, in this experiment, there are 898 answers compared to 662 when the attributes are independent. With more skyline points, the overall runtime of all algorithms are thus higher compared to the case when independent attributes are used.

Effect of Optimization Schemes Fig. 12(c) compares the effect of optimizing the dominance classification (discussed in Section 4.7) for SDC^+. From the figure, we can see that SDC^+-MaxPC has only slight improvement over SDC^+ while a more significant improvement is observed in SDC^+-MinPC. This significant improvement is due to the decrease in the number of dominance comparisons involving data points in the category (c, c). We also conducted experiments comparing SDC against SDC-MinPC and SDC-MaxPC, and our results (not shown) indicate that the impact of optimized dominance classification on SDC is not too significant.

We have also studied the effect of the various optimization techniques discussed in Sections 4.5.2-4.5.4 on SDC (re-



(a) Increasing size of data set

(b) Anti-correlated dataset

(c) Optimizing dominance classification

Figure 12: Results of other experiments.

sults not shown). Among the three optimizations, optimizing dominance comparisons has the most impact and could improve the performance by as much as a factor of 18. Minimizing dominance comparison turns out to have little impact with only marginal improvement. While the progressive optimization alone does not contribute to performance improvement, it is the key to progressiveness.

6. CONCLUSIONS

In this paper, we have addressed the novel problem of evaluating skyline queries with partially-ordered attributes. Our solution transforms each partially-ordered attribute into a two-integer domain that enables us to exploit index-based algorithms to compute skyline queries on the transformed space. Based on the framework, we have proposed three novel algorithms: BBS^+ is a straightforward extension of BBS, and SDC and SDC^+ are optimized versions that handle false positives and facilitates progressive evaluation. Our extensive performance study show that our proposed algorithms outperform existing techniques by a wide margin (between a factor of 2 and 16), with SDC^+-MinPC, which is SDC^+ using the **MinPC** strategy to optimize dominance classification, giving the best performance in terms of both response time as well as progressiveness.

For future work, we intend to explore the tradeoffs of using different domain mapping functions, and examine the evaluation of other skyline-related queries that involved partially-ordered domains. We are also exploring efficient methods to update the domain mappings and indexes when the data points are modified.

7. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD'89*.

[2] R. Agrawal and E. Wimmers. A framework for expressing and combining preferences. In *SIGMOD'00*.

[3] W. Balke, U. Güntzer, and X. Zheng. Efficient distributed skylining for web information systems. In *EDBT'04*.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE'01*.

[5] M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *SIGMOD'97*.

[6] J. Chomicki. Querying with intrinsic preferences. In *EDBT'02*.

[7] J. Chomicki. Preference formulas in relational queries. *ACM TODS*, 24(4), 2003.

[8] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: a system for the efficient execution of multi-parametric ranked queries. In *SIGMOD'01*.

[9] W. Kießling. Foundations of preferences in database systems. In *VLDB'02*.

[10] W. Kießling and G. Köstler. Preference SQL - design, implementation, experiences. In *VLDB'02*.

[11] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: an online algorithm for skyline queries. In *VLDB'02*.

[12] H. Kung, F. Luccio, and F. Preparata. On finding the maxima of a set of vectors. *JACM*, 22(4), 1975.

[13] J. Matousek. Computing dominances in E^n. *Information Processing Letters*, 38(5):277–278, 1991.

[14] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD'03*.

[15] C. H. Papadimitriou and M. Yannakakis. Multiobjective query optimization. In *PODS'01*.

[16] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[17] I. Stojmenovic and M. Miyakawa. An optimal parallel algorithm for solving the maximal elements problem in the plane. *Parallel Computing*, 7(2), June 1988.

[18] K. L. Tan, P. K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB'01*.

[19] R. Torlone and P. Ciaccia. Which are my preferred items? In *Workshop on Recommendation and Personalization in E-Commerce*, May 2002.

[20] Y. Zibin and J. Y. Gil. Efficient subtyping tests with PQ-encoding. In *OOPSLA'01*.