

Topologically Sorted Skylines for Partially Ordered Domains

Dimitris Sacharidis ^{#1}, Stavros Papadopoulos ^{*2}, Dimitris Papadias ^{*3}

[#]National Technical University of Athens
Athens, Greece

¹dsachar@dbl-lab.ntua.gr

^{*}Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong

²stavros@cse.ust.hk

³dimitris@cse.ust.hk

Abstract—The vast majority of work on skyline queries considers totally ordered domains, whereas in many applications some attributes are partially ordered, as for instance, domains of set values, hierarchies, intervals and preferences. The only work addressing this issue has limited progressiveness and pruning ability, and it is only applicable to static skylines. This paper overcomes these problems with the following contributions. (i) We introduce a generic framework, termed TSS, for handling partially ordered domains using topological sorting. (ii) We propose a novel dominance check that eliminates false hits/misses, further enhancing progressiveness and pruning ability. (iii) We extend our methodology to dynamic skylines with respect to an input query. In this case, the dominance relationships change according to the query specification, and their computation is rather complex. We perform an extensive experimental evaluation demonstrating that TSS is up to 9 times and up to 2 orders of magnitude faster than existing methods in the static and the dynamic case, respectively.

I. INTRODUCTION

The skyline query returns all points not dominated by another point [1]. A point p_i is said to dominate p_j , if p_i is at least as good as p_j on all dimensions, and there is at least one dimension where p_i is preferred. Since tuples can be viewed as points in the multi-dimensional space defined by their attribute domains, we use the terms tuple and point interchangeably. Intuitively, the skyline query retrieves the best tuples, irrespective of how a user weighs each dimension. The distinctive property of the skyline is that it contains the top-1 result of any preference function that is monotone on each attribute. Conversely, for any skyline point there is at least one preference function for which this point is the top-1 result.

Assume, for instance, a flight reservation system. A user specifies origin, destination and travel dates, and the system returns all tickets that satisfy these requirements. Figure 1(a)

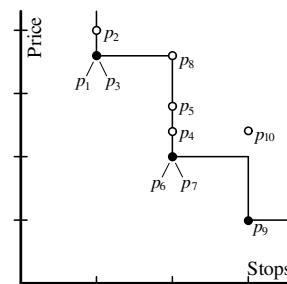
This work has been funded by the project PENED 2003. The project is cofinanced 75% of public expenditure through EC - European Social Fund, 25% of public expenditure through Ministry of Development - General Secretariat of Research and Technology and through private sector, under measure 8.3 of OPERATIONAL PROGRAMME “COMPETITIVENESS” in the 3rd Community Support Programme.

This work was supported from grant HKUST 6184/06 from Hong Kong RGC.

enumerates the result of such a query. The defining characteristics of a ticket are Price, number of Stops and the Airline company. For the moment, let us focus on the first two attributes, assuming that all airlines are equally desirable. A cheap ticket is preferred to a more costly one, and a direct flight to one with multiple stops. Figure 1(b) draws the tickets as points in the 2-dimensional space defined by Stops and Price. Clearly, the lower left corner, corresponding to a ticket with the minimum price and number of stops, is the most preferable. Note that p_8 (resp. p_4) is dominated by p_1 and p_3 (resp. p_6 and p_7), as it has the same price (resp. number of stops) but more stops (resp. is more expensive) than p_1 , p_3 (resp. p_6 , p_7). On the other hand, p_1 , p_3 , p_6 , p_7 and p_9 , drawn as filled circles, are not dominated by any other ticket and are thus in the skyline. Any ticket that lies to the right of the line connecting the skyline points in Figure 1(b) is dominated.

Ticket	Price	Stops	Airline
p_1	1,800	0	a
p_2	2,000	0	a
p_3	1,800	0	b
p_4	1,200	1	b
p_5	1,400	1	a
p_6	1,000	1	b
p_7	1,000	1	d
p_8	1,800	1	c
p_9	500	2	d
p_{10}	1,200	2	c

(a) Example data set



(b) Skyline tickets



Fig. 1. Flight reservation example

Consider now the case where a user has specific preferences among the airline companies. Assume for example, that s/he favors a over both b and c , and any company over d . However, s/he states no preference between b and c , implying that they are equally desirable. The lack of preference for some pairs of values indicates that the domain is *partially ordered* (PO). Compare this to a *totally ordered* (TO) domain; a cheaper price is always preferred over a more expensive one. In the sequel, for simplicity we assume that for two values x , y of

a (partially or totally ordered) domain, $x < y$ denotes that x is preferred over y .

A domain can be represented by a directed acyclic graph (DAG), called the *Hasse diagram*. Each node corresponds to a value; a directed edge from x to y exists if $x < y$ and there is no z such that $x < z < y$. Then, a value x is preferred to y if there is a (directed) path from x to y in the DAG. Returning to our example, the user preferences on the airline attribute induce a partial order represented by the DAG in the first row of Table I. According to this order, p_3 and p_7 are no longer in the skyline; for both tickets, there is an alternative which has the same price and number of stops but better airline. In addition, p_5 and p_{10} are no longer dominated and are in the skyline.

TABLE I
DIFFERENT PARTIAL ORDERS ON THE AIRLINE ATTRIBUTE

Partial order	Skyline tickets
	$p_1, p_5, p_6, p_9, p_{10}$
	$p_3, p_6, p_7, p_8, p_9, p_{10}$

The problem of computing the skyline in the presence of partially ordered domains was introduced in [2]. Briefly, the main idea is to map each PO domain into two TO domains, transform the entire data set to the new space, and apply a traditional skyline algorithm. The mapping, however, is incomplete in that it fails to preserve all preferences stated in the original domain. Under this restriction, checking for dominance in the transformed domain enforces a *stronger* notion of dominance in the original space, termed m -dominance. If p_i m -dominates p_j , it follows that p_i also dominates p_j according to the conventional definition. However, the converse does not hold. Subsequently, the result of any skyline algorithm in the transformed space is bound to contain *false hits*, i.e., points not m -dominated, but in fact dominated when all preferences are considered. To address this issue, cross-examination among skyline points is necessary. Depending on the number of preferences not captured, a large percentage of the pair-wise m -dominance checks can be inaccurate. The double-checking required reduces considerably the algorithm's *efficiency*. We refer to techniques that perform such dominance checks as *inexact*.

A desirable property of any skyline algorithm is *precedence*, which states that a point should be examined *after* all points that can potentially dominate it. A method that features *precedence* avoids unnecessary dominance checks by quickly identifying dominated, and more importantly, skyline points. To understand this, consider a point that has been checked for dominance against all previously seen points. Two outcomes are possible: the point is either dominated or not. In the second case, *precedence* ensures that it is a skyline point; on the

other hand, if *precedence* is not established, the point is only a candidate, and the algorithm should continue to check it for dominance against subsequent points. Unfortunately, the existing algorithms for visiting points in the transformed space cannot guarantee *precedence*.

The *progressiveness* of an algorithm, i.e., its ability to output skyline points before examining the entire data set, and its *efficiency*, in terms of dominance checks and page IOs, largely depend on *exactness* and *precedence*. In this work, we present a framework, termed Topologically-Sorted Skylines (TSS), that exhibits both properties and addresses all issues related to PO attributes. In short, TSS topologically sorts the DAG of a domain and extracts a spanning tree. Then, it associates with each PO value two pieces of information: (i) the ordinal number in a topological sort, and (ii) multiple intervals determined by the spanning tree structure. The former ensures *precedence*, while the latter guarantees the *exactness* of TSS. We apply our framework on the state-of-the-art index-based algorithm to obtain its PO counterpart, termed sTSS, and enhance its performance through various optimizations.

In many cases, the preferences among PO values are not uniquely defined, i.e., there is no objectively good partial order. Users can have different, often conflicting, preferences. To capture this scenarios we introduce the concept of *dynamic skylines* in PO domains. In the well studied case of only TO attributes, the dynamic skyline is defined with respect to a query point q . All dominance relationships are *implicitly* re-defined relative to q , so that p_i dominates p_j if p_i is not farther from q than p_j on all dimensions and is strictly closer on at least one. In the case of PO attributes, a dynamic skyline query *explicitly* redefines all dominance relationships, by specifying a partial order for each PO domain. Consider, for instance a dynamic skyline query over the airline attribute that defines the partial order indicated in the second row of Table I, according to which, the only preference is that of b over a . To compute the new skyline tickets shown in the table, the methodology in [2] needs to recompute the mapped coordinates for all points (a very expensive process) before running anew. The adaptation of the TSS framework to dynamic skyline queries, termed dTSS, avoids this pitfall, requiring only minimal pre-processing. In addition, dTSS allows for caching of past results to expedite the processing of future queries.

The remainder of this paper is organized as follows. Section II reviews related work on skyline query processing, focusing mostly on PO domains. Section III formally defines *exactness* and *precedence*, and introduces the TSS framework. Section IV proposes an efficient algorithm for computing skylines with PO domains. Section V discusses dynamic skyline queries and the re-use of previous results for improving performance. Section VI evaluates the benefits of our approach through extensive experiments, and Section VII concludes the paper.

II. RELATED WORK

Section II-A overviews methods for skylines and related queries in TO domains. Section II-B discusses the mapping

from PO to TO domains. Section II-C describes the only known methodology for skyline processing in PO domains.

A. Skyline Query Processing

Computing the skyline points, also known as finding the maxima in a set of vectors [3], has been thoroughly studied in the area of computational geometry where a large number of theoretical results exist. If the order within each dimension is treated as a preference, the skyline points correspond to the Pareto accumulation or composition of all preferences discussed in [4], [5]. The first work to address skyline computation in the context of databases is [1]. The authors devise an algorithm that iterates over all points using block nested loops (BNL), propose a B-tree based approach, and adapt the multidimensional divide and conquer algorithm [6] to external memory. An extension of the BNL algorithm, introduced in [7] and termed SFS, presorts the points according to a monotone preference function. SaLSa [8] and LESS [9] improve on SFS. Tan et al. [10] introduce two progressive techniques, Bitmap and Index, which output points guaranteed to belong in the skyline without having to scan the entire data set.

The NN algorithm [11] applies nearest neighbor search on data indexed by an R-tree, based on the observation that the NN point to the beginning of the axes is always part of the skyline. This point divides the dataset into overlapping partitions according to its coordinates. Then, NN repeats the search process iteratively in each resulting partition. Special care is taken to remove duplicates found in overlapping partitions. Branch and bound skyline (BBS) [12] avoids the pitfalls of the nearest neighbor approach. Starting from the root, BBS inserts R-tree entries, which correspond to minimum bounding boxes (MBB), into a heap in ascending order of their minimum distance to the beginning of the axes. The first point visited is the nearest neighbor and is inserted in the skyline list. When a subsequent entry is de-heaped, the lower-left corner of its MBB is checked for dominance against the skyline list. If it is dominated, the entire subtree can be pruned. Otherwise, its children are examined, and those not dominated are inserted into the heap. Execution terminates when the heap is depleted. BBS visits only the R-tree nodes that can potentially contain skyline points, and, hence, is IO optimal.

A large part of the recent literature focuses on skyline-related topics. The *skycube query* [13] returns the points not dominated in a specified subset of the dimensions. Li et al. [14] investigate *dominance relationships* from a data-warehouse perspective. The concept of *probabilistic skylines* [15] defines dominance for objects represented by multiple instances. Given a query point q , a *dynamic skyline* (for TO domains) reports the points that are not dominated in a new coordinate system centered at q [12]. The *reverse skyline query* [16] retrieves the points whose dynamic skyline includes an input point. Given a set of points Q , a *multi-source skyline query* [17], [18] reports the points not dominated with respect to Q . Wong et al. [19] identify the minimal set of preferences that cause the exclusion of a given point from the skyline.

B. Mapping PO to TO Domains

Determining preference in PO domains is analogous to the problem of detecting graph reachability. Several such techniques have been proposed in various application domains (for a survey see [20]). In the sequel we demonstrate a method adopted from [21], which creates a spanning tree on the DAG. Each node is associated with an interval $[minpost, post]$, where $post$ corresponds to the node's position in a postorder traversal, and $minpost$ to the minimum $post$, among its descendants. Consider the PO domain and the spanning tree shown in Figure 2(a). The spanning tree edges are shown with bold lines. The purpose of the small numbers on top of each node will become apparent later.

If the interval of a node is contained within that of another node, then the latter is preferred to the former, e.g., d is considered worse than a as $[3, 6] \subset [1, 9]$. If two intervals $([3, 3], [1, 1])$ are disjoint, the preference relationship between the corresponding nodes (f, h) cannot be determined based on the information preserved by the spanning tree (in Figure 2(a), f is better than h). Let I_1, I_2 be the integer domains associated with $minpost$ and $post$. Preference in these domains is defined by low values in I_1 and high values in I_2 . Figure 2(b) draws the intervals associated with each value on the $I_1 \times I_2$ domain and superimposes the spanning tree for reference.

C. Skylines in PO Domains

The only work with a scope similar to ours, which also serves as a baseline in our experimental evaluation is [2]. The authors use a method similar to that of Figure 2(a), to map all data points in a new space, where each PO value is substituted by the associated coordinates in the $I_1 \times I_2$ domain. Dominance in the new space corresponds to the notion of m -dominance in the original space. A point p_i m -dominates p_j if (i) p_i is at least as good in all totally ordered dimensions, (ii) its interval covers or coincides with p_j 's for all partially ordered dimensions, and (iii) there exists either (a) a totally ordered dimension in which p_i is strictly better, or (b) a partially ordered dimension for which p_j 's interval is contained in p_i 's. Since, not all preference relationships are captured by the interval representation, m -dominance is stronger than dominance. Consequently, the set of skyline points determined according to m -dominance is a superset of the actual skyline points, i.e., it may contain false hits.

Query processing is based on three variants of BBS, namely BBS+, SDC and SDC+, that operate on the transformed space and differ in false hit elimination. BBS+ checks every non m -dominated point for (actual) dominance against a list of intermediate skyline points found so far. BBS+ is not progressive because it cannot report any result until all points have been examined. SDC distinguishes between two disjoint subsets of points, called *strata*, based on the type of the DAG node representing their value in the PO domain. A node is *completely covered* if all edges of its incoming paths are included in the spanning tree; otherwise, it is *partially covered*. The definition can be extended to points in a straight-forward manner: a point is completely covered if its values

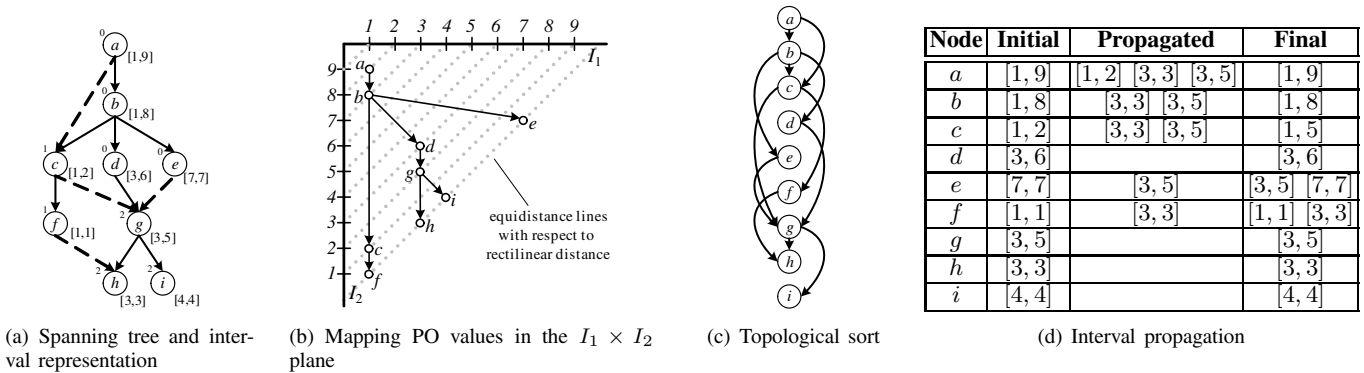


Fig. 2. A PO domain example

in all PO domains correspond to completely covered nodes. A useful property is that m -dominance among completely covered points corresponds to the actual dominance. During the execution of SDC, the stratum of a point is determined. If the point is not m -dominated and is completely covered, then it definitely belongs to the skyline and can be safely output, increasing progressiveness. Points in the other stratum cannot be reported before termination.

SDC+ further distinguishes partially covered points into multiple strata. The *uncovered level* of a node is the maximum number of non-tree edges in any of its incoming paths. All completely covered vertices have uncovered level of 0, whereas the partially covered vertices have a level strictly greater than 0. The small numbers on top of nodes in Figure 2(a) are their uncovered levels. For instance, the uncovered level of g is 2 because of the two non-tree edges in the path a, c, g . This definition can be extended to points, as before. The main intuition is that points from a particular uncovered level cannot be dominated by points from higher levels. SDC+ partitions the entire data into strata based on their uncovered level and stores them in a separate R-tree for each stratum. Then, strata are processed in sequence, sorted increasingly by their uncovered level. SDC+ maintains a global list of actual skyline points and a local list per stratum that may contain false hits. MBBs are checked for m -dominance against both lists. When a leaf entry p_i is de-heaped, it is checked for (actual) dominance against the local list. If p_i is not dominated, all points in the local list are cross-examined for dominance by p_i ; false hits are, thus, identified and removed. Point p_i is also checked for (actual) dominance against the global list. Once the heap is empty, the local skyline list contains actual skyline points, i.e., no false hits, and is appended in the global list.

III. TSS FRAMEWORK

In this section we present *Topologically-sorted Skylines* (TSS), a methodology that achieves efficiency and progressiveness in the presence of partially ordered domains. Since TSS does not restrict, or modify in any other way the dominance definition, it is not hindered by false hits and does not have to cross-examine skyline points. Furthermore, TSS can output skyline points immediately, long before the entire data set is

considered. In fact, TSS is progressive in the *strictest* sense. Our framework is also independent of the underlying skyline algorithm¹. Last, but not least, TSS is applicable to dynamic skyline queries for partially ordered domains.

Section III-A discusses some desirable properties for any skyline algorithm, and compares the existing methods with respect to these features, focusing mostly on SDC+. Section III-B describes the enforcement of these properties by TSS and provides the corresponding dominance definitions.

A. Motivation

Most external memory skyline algorithms, e.g., BNL, Index, SFS, LESS, SaLSa, BBS, follow a common scan-based paradigm [9]. Specifically, they maintain a candidate list of skyline points and check each point in turn for dominance against the candidates. Note that the candidate list contains the skyline points with respect only to the points seen so far. As such, a point in the candidate list is not an actual result and may be evicted if a subsequent point dominates it. The goodness of an algorithm is measured by (i) its *progressiveness*, i.e., its ability to output results before termination, and (ii) its *efficiency* in terms of the number of pair-wise dominance checks and IO accesses. Ideally, the algorithm should only examine the skyline points, immediately output a skyline point as soon as it is identified, and terminate upon discovering the last skyline point disregarding all the rest. The progressiveness and the efficiency depend on two crucial properties.

Property 1 (Precedence): A point should be examined before all points that it can potentially dominate.

Equivalently, a point should be examined *after all points that can potentially dominate it*. A straightforward consequence is that the first point considered is always a skyline point. Intuitively, *precedence* is desirable as it enables an algorithm to definitely determine if the point under examination is in the skyline. If no point in the candidate list dominates it and since not yet examined points cannot dominate it, it must belong in the skyline. The *precedence* property is explicitly exploited by SFS and its derivatives (LESS, SaLSa), which examine points ordered by a monotonic preference function. The Index and BBS algorithms also exhibit *precedence*, as they visit points ordered according to their minimum Chebyshev

¹To guarantee progressiveness, an appropriate method is needed.

and rectilinear, respectively, distance to the most preferable point.

On the other hand, none of the m -dominance methods satisfies *precedence*. Consider the PO domain and its interval representation in Figure 2(a). Assume that SDC+ uses the rectilinear distance to sort entries in the heap; similar results hold for any \mathcal{L}_p distance metric. The dashed lines in Figure 2(b) correspond to points that are within the same rectilinear distance of the most preferable point (top left corner). Observe that a point p_1 , which has PO value g , will be examined before another point p_2 , which has the same value with p_1 in the TO domains, but has PO value either e or c (both are preferred to g) in the PO. Clearly, this violates the *precedence* requirement.

Property 2 (Exactness): A dominance check should always identify dominance relationships, accurately.

This property implies that when examining a point against another, the algorithm should unambiguously ascertain whether it is dominated or not. All existing skyline algorithms for TO domains satisfy this property, as their dominance check accurately reflects the dominance relationship among points. For PO domains, however, the methodology of [2] applies a stronger, compared to the actual, dominance check. Depending on the density of the DAG, m -dominance can result in a large number of false hits, i.e., points incorrectly identified as not dominated. This necessitates a cross examination among all such points, as discussed in Section II.

We now revisit the two goodness measures and illustrate their relationship with the properties introduced. The progressiveness measure essentially summarizes the criteria for an online algorithm [22] as adapted to the skyline computation problem [11]. According to this notion, skyline points should be output (and never revoked) during the execution of the algorithm, before all points are considered. In this work, we impose harsher requirements on progressiveness. A skyline computation algorithm is *optimally progressive* if a point that belongs in the skyline is output immediately after it has been examined. It is easy to see that all progressive algorithms proposed in the literature, e.g., Index, SFS, LESS, SaLSa, BBS, are also optimally progressive. In comparison, BNL and BBS+ are neither progressive nor optimally progressive, as they output the entire skyline at once after all points have been examined. SDC and SDC+ output skyline points during the execution of the algorithm and are, thus, progressive. However, they are not optimally progressive: to output a skyline point they must first examine all points that belong in the same stratum. Since SDC+ organizes data in multiple strata, it can be considered more progressive than SDC.

An important remark is that any algorithm that features both the *precedence* and the *exactness* properties is guaranteed to exhibit optimal progressiveness. Indeed, when a point p is considered, the second property ensures that p is always accurately classified (as dominated or not dominated) with respect to any point considered so far. The first property ensures that all necessary dominance checks to accurately determine whether p is in the skyline have been performed.

Combining the two observations, we conclude that p can be safely output if not dominated, and hence the algorithm is optimally progressive.

We turn our attention to the efficiency of the skyline computation algorithms, as measured by the number of dominance checks and IO operations. According to the scan-based paradigm, each point will be checked against every point in the candidate list. The average number of checks performed per point is therefore determined by the size of the candidate list (if it does not fit in main memory extra IOs are required). It should be clear that to incur minimum checks and IOs the candidate list should only contain actual skyline points and no intermediate candidates. An algorithm that features both properties never maintains in the candidate list non-skyline points. Given that the second property is satisfied (i.e., all checks are accurate), the first property ensures that points enter the candidate list only if they belong in the skyline. To conclude, it is apparent that the existing techniques for computing skylines over PO domains are hindered by the lack of *precedence* and *exactness*. Next, we show that it is possible for an algorithm to respect both properties

B. TSS Design Issues

TSS constructs a totally ordered domain by mapping values from the partially ordered domain in a manner that preserves all preference relationships, direct (parent to child edges) and indirect (ascendant to descendant paths). In particular, to enforce total order we *artificially* insert additional preference relationships among incomparable values. Then, any monotonic preference function over the mapped domain is also monotonic in the original domain and *precedence* is ensured. The mapping from PO to TO domains is performed through topologically sorting the DAG nodes.

Figure 2(a) illustrates the DAG associated with a partially ordered domain of 9 values, labeled a through i . Figure 2(c), shows one admissible topological sort, including the edges of the DAG, of the nodes that corresponds to the total order $a < b < c < \dots < i$. It is straightforward to verify that every preference relationship in the DAG is preserved by the total order (all edges face down). TSS then maps a PO value into an integer TO domain, depending on its ordinal number in the topological sorting, i.e., 1 is assigned to a , 2 to b , and so forth. Let A_{PO} be the PO domain; we refer to the TO domain generated by topological sorting, as A_{TO} .

Unfortunately, the topological sort is inadequate for the second property, namely *exactness*. Consider, for example, two points p_1 and p_2 that are identical on all attributes except for the A_{PO} of Figure 2(a), where their values are c and d , respectively. The dominance check using their A_{TO} values, 3 and 4, respectively, suggests that p_1 dominates p_2 . However, these points are incomparable in A_{PO} . This occurs due to the artificial preferences inserted, which do not exist in the original domain.

To address this issue, TSS employs an efficient dominance check unrelated to the TO domain, which has neither false hits nor misses and, thus, allows for *exactness*. Similar to

the m -dominance notion, TSS’s check is based on the transformation of Section II-B. Unlike m -dominance, however, it guarantees accurate checks. Specifically, TSS extracts a spanning tree (containing all nodes, but not all edges) from the DAG of the PO domain. Next, it associates with each node the $[\text{minpost}, \text{post}]$ interval. Assume the spanning tree for the previous DAG as shown in Figure 2(a). The second column in Figure 2(d) illustrates the intervals associated to each node. As discussed in Section II-B, these intervals encode the preferences associated with the tree paths only.

To capture all preferences, TSS includes, in each node, information about non-tree edges by propagating intervals. For instance, to accurately reflect all preferences involving h , its interval $[3, 3]$ must be propagated along any incoming path to h that includes a non-tree edge. Therefore, $[3, 3]$ is copied to f and subsequently to c , b and a . The process is repeated for all non-tree edges and all propagated intervals are shown in the third column of Figure 2(d). Each node is, therefore, associated with multiple intervals. However, most of them can be merged with, or subsumed by others, decreasing the number of intervals required for capturing all dominance relationships. The fourth column of Figure 2(d) illustrates the final intervals after these operations. Next, we formally introduce the notion of preference among PO values and finally define TSS’s exact dominance check.

Definition 1 (t -preference): Given two values $x \neq y$ of a PO domain, x is t -preferred over y , iff for every interval i_y associated with y there exists an interval i_x , associated with x , such that either i_y is contained in i_x or i_x, i_y coincide. As an example, consider values f, h from the domain of Figure 2(a). The single interval $[3, 3]$ associated with h coincides with one of f ’s intervals; hence, f is t -preferred over h .

Definition 2 (t -dominance): A point p_i t -dominates p_j if (i) p_i is at least as good as p_j in all TO dimensions, (ii) p_j is not t -preferred over p_i in any PO dimension, and (iii) there exists either (a) a TO dimension in which p_i is strictly better, or (b) a PO dimension in which p_i is t -preferred over p_j .

Summarizing, since the interval encoding captures *all* dominance relationships in the DAG, TSS exhibits *exactness*. Furthermore, the topological sorting enforces *precedence*. TSS is a general framework and can be implemented with various skyline algorithms. Following [2], we next describe an implementation based on BBS, which permits a direct comparison with the previous work on PO skylines.

IV. STATIC SKYLINE IN PO DOMAINS

We first propose a concrete implementation of the TSS framework, called sTSS for static, i.e., conventional, skylines. Unlike the BBS-based methods of [2], sTSS achieves optimal progressiveness and efficiency due to *precedence* and *exactness*. Section IV-A presents the basic functionality of the algorithm and Section IV-B focuses on optimizations for expediting t -dominance checks.

A. Basic Algorithm

For ease of exposition, we describe sTSS through an example. We assume that there are only two attributes in the

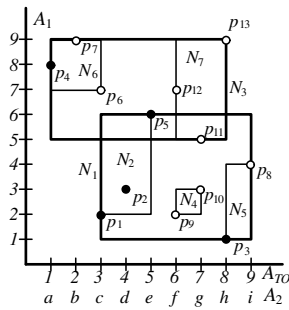
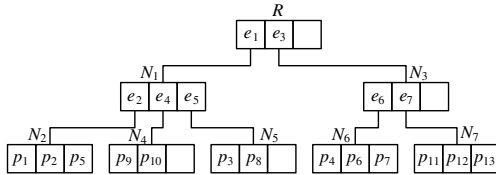
skyline criteria: A_1 , with a totally ordered domain, and A_2 , with the partially ordered domain of Figure 2(a). The extension to multiple PO domains is straightforward. sTSS constructs the DAG corresponding to A_2 (we slightly abuse notation by referring to the attribute and its domain using the same symbol) and obtains (i) a topological sort of its nodes, and (ii) a spanning tree and the resulting intervals associated with each node, as shown in Figure 2(d). In accordance to the TSS framework, A_2 is mapped to an integer domain, A_{TO} , by assigning to each value (DAG node) the ordinal number in the topological sort. Also, let I_1, I_2 be the domains of the minpost and post number, so that all intervals are expressed in $I_1 \times I_2$. As a result, any value in the original domain, A_2 , is associated with a single value in A_{TO} and with possibly multiple intervals (or 2D points) in $I_1 \times I_2$. Note that all constructed domains, A_{TO}, I_1, I_2 , have the same cardinality (9) with A_2 .

Consider the example data set in Figure 3(a). Each tuple corresponds to a 2-dimensional point in the $A_1 \times A_{TO}$ domain as shown in Figure 3(b). For better reference, we also include the A_2 values. Skyline points are denoted with filled circles. Figure 3(c) illustrates an R-tree on the transformed space, assuming a maximum node capacity of 3 entries. Each node N_i (except for the root R) corresponds to a minimum bounding box (MBB), drawn in Figure 3(b), and is associated with an entry e_i in its parent node. An entry in a leaf node corresponds to a data point. sTSS, similar to BBS, uses a heap to buffer R-tree entries and determine the nodes’ visiting order. The heap always contains at its head the entry with the minimum distance (mindist) to the most preferable point of the data set, which, in this case, is the lower-left corner of the transformed space. Throughout this example, we use the rectilinear distance (\mathcal{L}_1 norm) for calculating distances. In particular, assuming that all points have non-negative coordinates, the mindist of a point equals the sum of its coordinates, and the mindist of a node equals the mindist of the lower left corner of its MBB. Furthermore, the definition of t -dominance can be extended to cover nodes, i.e., a point p_i t -dominates node N_j if p_i t -dominates the lower left corner of N_j .

The sTSS algorithm proceeds in a manner similar to BBS. Table II demonstrates the contents of the heap and the skyline points found at each step of sTSS’s execution. Pruned entries are typed in bold. Initially, the heap is empty. Then, the root node R is expanded and e_1, e_3 , are inserted in the heap (step 1). The entry e_1 with the minimum mindist is de-heaped and its children e_2, e_4, e_5 are en-heaped (step 2). Next, e_2 is replaced by its children p_1, p_2, p_5 (step 3), among which point p_1 has the minimum mindist and is, thus, de-heaped and inserted in the skyline (step 4). The head now contains e_3 , which needs to be checked for t -dominance against p_1 . The MBB N_3 associated with this entry contains values that extend from a to h in the PO domain A_2 . In the interval domain, these values correspond to intervals which are subsumed by $[1, 8]$. Since $[1, 8]$ is not t -preferred over the interval $[1, 5]$ of p_1 , N_3 is not t -dominated by p_1 . Thus, N_3 is visited and its entries e_6, e_7 are en-heaped (step 5). Subsequently, sTSS has to examine

id	A_1	A_2	id	A_1	A_2
p_1	2	c	p_8	4	i
p_2	3	d	p_9	2	f
p_3	1	h	p_{10}	3	g
p_4	8	a	p_{11}	5	g
p_5	6	e	p_{12}	7	f
p_6	7	c	p_{13}	9	h
p_7	9	b			

(a) Example data set

(b) Points in $A_1 \times A_{TO}$ 

(c) R-tree structure

Fig. 3. Example data

TABLE II
HEAP AND SKYLINE CONTENTS

Step	Heap	Skyline
1	$\langle e_1, 4 \rangle \langle e_3, 6 \rangle$	\emptyset
2	$\langle e_2, 5 \rangle \langle e_3, 6 \rangle \langle e_4, 8 \rangle \langle e_5, 9 \rangle$	\emptyset
3	$\langle p_1, 5 \rangle \langle e_3, 6 \rangle \langle p_2, 7 \rangle \langle e_4, 8 \rangle \langle e_5, 9 \rangle \langle p_5, 11 \rangle$	\emptyset
4	$\langle e_3, 6 \rangle \langle p_2, 7 \rangle \langle e_4, 8 \rangle \langle e_5, 9 \rangle \langle p_5, 11 \rangle$	$\{p_1\}$
5	$\langle p_2, 7 \rangle \langle e_4, 8 \rangle \langle e_5, 9 \rangle \langle p_5, 11 \rangle \langle e_7, 11 \rangle$	$\{p_1\}$
6	$\langle e_4, 8 \rangle \langle e_6, 8 \rangle \langle e_5, 9 \rangle \langle p_5, 11 \rangle \langle e_7, 11 \rangle$	$\{p_1, p_2\}$
7	$\langle e_6, 8 \rangle \langle e_5, 9 \rangle \langle p_5, 11 \rangle \langle e_7, 11 \rangle$	$\{p_1, p_2\}$
8	$\langle e_5, 9 \rangle \langle p_4, 9 \rangle \langle p_6, 10 \rangle \langle p_5, 11 \rangle \langle e_7, 11 \rangle \langle p_7, 11 \rangle$	$\{p_1, p_2\}$
9	$\langle p_3, 9 \rangle \langle p_4, 9 \rangle \langle p_6, 10 \rangle \langle p_5, 11 \rangle \langle e_7, 11 \rangle \langle p_7, 11 \rangle \langle p_8, 13 \rangle$	$\{p_1, p_2\}$
10	$\langle p_4, 9 \rangle \langle p_6, 10 \rangle \langle p_5, 11 \rangle \langle e_7, 11 \rangle \langle p_7, 11 \rangle \langle p_8, 13 \rangle$	$\{p_1, p_2, p_3\}$
11	$\langle p_6, 10 \rangle \langle p_5, 11 \rangle \langle e_7, 11 \rangle \langle p_7, 11 \rangle \langle p_8, 13 \rangle$	$\{p_1, p_2, p_3, p_4\}$
12	$\langle p_5, 11 \rangle \langle e_7, 11 \rangle \langle p_7, 11 \rangle \langle p_8, 13 \rangle$	$\{p_1, p_2, p_3, p_4\}$
13	$\langle e_7, 11 \rangle \langle p_7, 11 \rangle \langle p_8, 13 \rangle$	$\{p_1, p_2, p_3, p_4, p_5\}$
14	$\langle p_7, 11 \rangle \langle p_8, 13 \rangle$	$\{p_1, p_2, p_3, p_4, p_5\}$
15	$\langle p_8, 13 \rangle$	$\{p_1, p_2, p_3, p_4, p_5\}$

the new head p_2 , which is incomparable to p_1 on A_2 because their intervals $[3, 6]$ and $[1, 5]$ are not related by t -dominance. Since *precedence* is guaranteed by topological sorting, p_2 is a skyline point (step 6). The head now contains e_4 . The MBB N_4 associated with this entry contains values that extend from f to g on A_2 . In the interval domain, these values correspond to $[1, 1]$, $[3, 3]$, $[3, 5]$, which are merged to $[1, 1]$, $[3, 5]$. Observe that both intervals are contained in p_1 's interval $[1, 5]$, p_1 's value on A_1 (2) is equal to N_4 's corresponding minimum value. Therefore, p_1 t -dominates any point in N_4 and the node is discarded (step 7). BSS* continues in a similar fashion until the heap is depleted; the final skyline points are p_1, p_2, p_3, p_4, p_5 .

B. Optimizations

Depending on the size of the skyline and the intervals associated with the skyline points and the MBBs, determining

t -dominance can be expensive. We describe two optimizations that together help dramatically reduce the cost. The first concerns the identification of all the intervals associated with the PO range of an MBB. Let r denote such a range extending across the topologically sorted PO domain A_{TO} . Since for each value in r there is at least one interval, the final set of intervals, computed after subsumption or merging, requires the retrieval and examination of at least $|r|$ intervals. For large MBBs, i.e., those high up in the R-tree, this computation overhead becomes more pronounced as almost $|A_{TO}|$ intervals must be considered. One solution for this problem is to pre-compute for each possible range $r \in A_{TO} \times A_{TO}$ the set of intervals associated with r (after subsumption/merging) and store them in a hash table. When examining an MBB for t -dominance, its range in the PO domain is used as the key to retrieve the intervals in constant time. This method's applicability is limited by the quadratic (in the PO domain size) space required to store all ranges.

We present a more attractive solution that balances the space and time trade-off. Specifically, we pre-compute and store the intervals associated with a small subset of all, i.e., $|A_{TO}|^2$, possible ranges. We keep the $|A_{TO}| - 1$ dyadic ranges (DR) of the domain. Assuming $|A_{TO}|$ is a power of 2, the DRs are $I_{l,k} = [k \frac{|A_{TO}|}{2^l}, (k+1) \frac{|A_{TO}|}{2^l})$ for $0 \leq l < \log |A_{TO}|$ and $0 \leq k < 2^l$, and correspond to the internal nodes of a binary tree built on top of A_{TO} . For example, $I_{0,0}$ corresponds to the root covering the entire domain, and its two children, $I_{1,0}$, $I_{1,1}$, correspond to the first and last half of A_{TO} , respectively. An important property of DRs is that any given range r can be decomposed into at most $\log |r|$ DRs. Hence, identifying the intervals associated with a range requires only logarithmic time at a linear storage overhead, with respect to $|A_{TO}|$.

The second and most important optimization concerns the t -dominance check itself. Returning to our running example, sTSS uses a 3-dimensional main memory R-tree, T_m , to store the skyline points and to efficiently perform the dominance check against points and MBBs. The R-tree's dimensions include the original A_1 and the two interval dimensions I_1, I_2 . Consequently, each skyline point is represented by possibly multiple virtual points in $A_1 \times I_1 \times I_2$, depending on the number of intervals associated with the point's PO value. For example, p_1 has a single virtual point, $(2, 1, 5)$ in the R-tree, whereas p_5 has two, $(6, 3, 5)$ and $(6, 7, 7)$, one for each interval associated with PO value e . Figure 4(a) shows the virtual points for the skyline of Figure 3(b). For ease of illustration, we draw their projection in the $I_1 \times I_2$ plane. The A_1 value of a virtual point is the single number above it. For example, p_1 's virtual point is at coordinates $(1, 5)$, while its A_1 value is 2. The most preferable point in this plane is the upper-left corner as it corresponds to $[0, 10]$ containing every possible interval in $I_1 \times I_2$. Virtual points are inserted in T_m according to the order that skyline points are discovered by sTSS; in our example, p_1, p_2, p_3, p_4, p_5 .

Next, we demonstrate the fast t -dominance check based on the main memory R-tree T_m . Consider step 7 in sTSS's

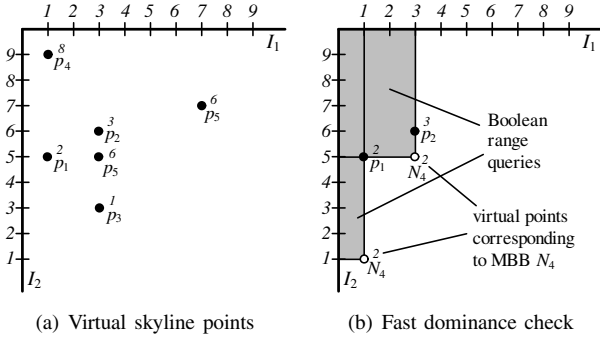


Fig. 4. Main memory R-tree

execution of Table II. In this step, the MBB of N_4 must be checked for t -dominance. Note that T_m contains only two virtual points corresponding to the current skyline p_1, p_2 . Using the first optimization, sTSS obtains the intervals $[1, 1]$, $[3, 5]$, associated with the A_2 range defined by N_4 's MBB. Transformed to the $A_1 \times I_1 \times I_2$ domain, this MBB corresponds to two virtual points, $(2, 1, 1)$ and $(2, 3, 5)$. Figure 4(b) draws with hollow circles their projections in the $I_1 \times I_2$ plane. sTSS needs to examine every virtual point for t -dominance. For each such point, it issues a range query on T_m . The range extends from the most preferable point in the $A_1 \times I_1 \times I_2$ domain (i.e., $(0, 0, 10)$) to the virtual point itself. Referring to N_4 's MBB, the two ranges are $[0 : 2] \times [0 : 1] \times [1 : 10]$ and $[0 : 2] \times [0 : 3] \times [5 : 10]$; their projection in the $I_1 \times I_2$ plane is shown shaded in Figure 4(b). If all range queries return at least one skyline point, then the MBB is t -dominated. In our case both queries return p_1 's virtual point; hence, sTSS safely prunes N_4 . Note that the range query can be efficiently processed as a Boolean query [16], where the answer is a single Boolean value that is false when the range is empty, and true otherwise.

V. DYNAMIC SKYLINES IN PO DOMAINS

In the context of PO attributes, a dynamic skyline explicitly specifies the dominance relationships. sTSS and the methods of [2] require rebuilding the index structures for each query. For sTSS this is because of changes in the topological sort, whereas in [2] this is due to the fact that the spanning tree (in fact, the entire DAG) is no longer valid.

Section V-A presents the dynamic counterpart of the TSS framework, termed dTSS, that avoids the pitfalls of existing static algorithms. Specifically, dTSS does not re-compute the coordinates, or re-build the index structure, and exhibits TSS properties. Section V-B proposes optimizations to enhance the performance of dTSS.

A. The dTSS Algorithm

Let A_1, A_2 be two TO attributes, and $A_3 = \{a, b, c\}$ be a PO attribute domain. Given the data set in Figure 5(a), dTSS partitions the points into disjoint groups, G_a, G_b, G_c based on their A_3 value. Upon receiving a dynamic skyline query defining a partial order on attribute A_3 , dTSS (i) topologically

sorts A_3 obtaining the integer domain A_{TO} , and (ii) extracts a spanning tree, and (iii) associates with each value a set of intervals in $I_1 \times I_2$. Assume that the query specifies the partial order on A_3 , topologically sorted as shown in Figure 5(a). In particular, the skyline query states that value b is better than c , and that no other preference exists.

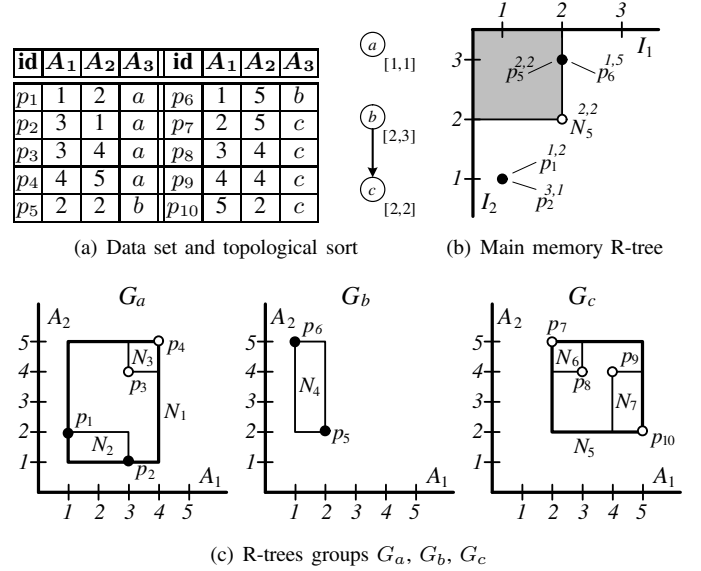


Fig. 5. Dynamic skyline query processing

The dTSS algorithm visits each group according to its topological order and processes all points within the group, before progressing to the next. To enforce *exactness*, a global main memory R-tree, T_m , in the $A_1 \times A_2 \times I_1 \times I_2$ domain is maintained. The t -dominance of a point is determined by performing range queries in this global tree, as described in Section IV. Within each group, the skyline is computed using any algorithm that exhibits *precedence*; here we use BBS. Observe that this is sufficient to ensure that *precedence* holds for all points. Therefore, any non-dominated point can be immediately output as it definitely belongs in the skyline. In addition, all virtual points corresponding to the intervals associated with its PO value are inserted in T_m .

The following discussion refers to the R-trees of Figure 5(c), which index points within each group G_a, G_b, G_c . Since all points in a group have the same A_3 value, and, thus, the same A_{TO} value, all R-trees are two-dimensional and are built in the $A_1 \times A_2$ domain. Consider the first group G_a . First, MBB N_1 is expanded to N_2 and N_3 , and then N_2 is expanded to p_1 and p_2 . The former is the entry with the minimum *mindist* and is a skyline point. Its A_3 value is associated with the interval $[1, 1]$ and leads to the insertion of a single virtual point with coordinates $(1, 2, 1, 1)$ in T_m . Figure 5(b) shows the projection of all virtual skyline points in the $I_1 \times I_2$ domain; the coordinates of the points in the A_1, A_2 dimensions are depicted on top of each point. Currently, the tree contains only p_1 's virtual point. The next entry examined is p_2 , which is not t -dominated by p_1 and, hence, is a skyline point and inserted

in the global R-tree. The entire N_3 MBB is t -dominated by both skyline points and is discarded, concluding the processing in the first group. dTSS continues to examine G_b . Since points in G_b cannot be t -dominated by those in G_a , p_5 and p_6 are in the skyline and their virtual points are inserted in the main memory R-tree. Finally, G_c is examined starting with its root entry N_5 . The lowest-left corner of N_5 has coordinates $(2, 2)$, and the interval associated with the c value is $[2, 2]$ (see Figure 5(a)). A Boolean range extending from $(2, 2, 2, 2)$ to the most preferable point, i.e., $(0, 0, 0, 4)$, is then issued on the global R-tree, returning p_5 's virtual point $(2, 2, 2, 3)$. Figure 5(b) depicts as shaded the projection of the range in the $I_1 \times I_2$ domain. The non-empty result implies that any point in G_c is dominated. Hence, the execution terminates without considering the group's R-tree entries at all. The final skyline and the virtual points are shown with filled circles in Figure 5(c) and Figure 5(b), respectively.

Suppose, now, that another dynamic query arrives, defining the new partial order illustrated in Figure 6(a). Values a and c are more desirable than b , but no preference exists between them. Observe that the groups are not affected and, hence, dTSS does not need to recalculate the coordinates of any point, or rebuild the R-trees associated with each group. It simply needs to topologically sort the new domain, extract a spanning tree and determine the intervals associated with each value. The result is illustrated in Figure 6(a). The global tree T_m from the previous run is, of course, discarded. Then, dTSS proceeds anew, visiting groups, applying BBS in each one and populating T_m . Points p_7, p_8, p_{10} , from the first group G_c , and p_1, p_2 from the second G_a , are inserted in the skyline (in that order). The projections of the contents of T_m in the $A_1 \times A_2$ and the $I_1 \times I_2$ planes are shown in Figure 6(b). Note that all points in G_b are dominated by p_1 because the virtual point $(1, 2, 2, 2)$ corresponding to the root MBB N_4 is dominated by p_1 's virtual point $(1, 2, 2, 3)$. Hence, the R-tree associated with this group is not examined.

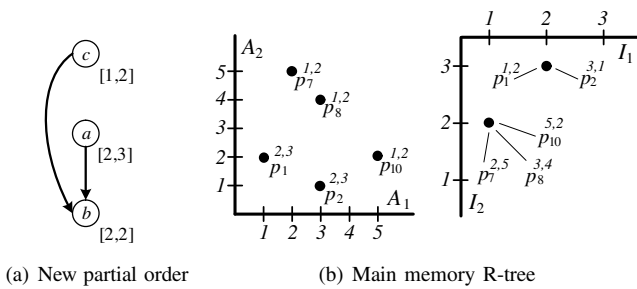


Fig. 6. New dynamic skyline query

B. Optimizations

We discuss two optimizations, pre-processing and caching of past dynamic queries, that improve the performance of dTSS. Initially, we assume that queries affect the dominance relationships only in the partially ordered attributes. Later, we consider *fully* dynamic skyline queries that may alter all relationships.

The dominance relationships among points of the same group are unaffected by changes in the partial order; only cross-group relationships may change. Therefore, completely ignoring the partial order, we pre-compute the *local* skyline points within each group and store them labeled with their group identifier. Then, for each new partial order that requires processing, we obtain a topological sort and a spanning tree of its DAG. We also identify the intervals associated with each node. Next, we consider groups in topological order, and examine only their local skyline points. For each such point, we perform a dominance check in the global R-tree. If the point is not t -dominated, it is a result. Upon examining the local skyline points of all groups, the R-tree will contain the skyline virtual points. We prove the algorithm's correctness by contradiction. Assume that there is a skyline point p_i not identified and let G_x be its group. Clearly, p_i must not belong in G_x 's local skyline, otherwise it would have been examined and correctly identified (due to the accuracy of the dominance check). Therefore, there is a skyline point p_j in G_x that dominates it locally. Since, p_i, p_j are in the same group, all their PO values are equal and p_j should dominate p_i globally as well; hence, a contradiction.

In the case of a fully dynamic skyline, which specifies a partial order on the PO attributes and the ideal values in the TO attributes, the pre-computed local skyline points are no longer valid. The skyline within each group must be recomputed. An interesting observation, however, is that *not all* preference relationships in TO domains are affected by the new query. Caching past results can help reduce the processing cost of dynamic queries, as shown in [23].

VI. EXPERIMENTAL EVALUATION

In this section, we evaluate TSS framework using synthetically generated data. We use the SDC+ algorithm as a benchmark to our methods since it outperforms BBS+ and SDC [2], the only other applicable methods for skyline queries with PO domains. All algorithms were implemented in C++, compiled with gcc and executed on a 2Ghz Intel Core 2 Duo CPU. Section VI-A presents the experimental setup and Sections VI-B, VI-C evaluate sTSS and dTSS on static and dynamic skylines, respectively.

A. Experimental Setup

We have modified a publicly available generator² to construct synthetic data sets using two types of distribution. In Independent, attribute values are drawn from a uniformly random distribution. In the Anti-correlated distribution, tuples with preferable values in one dimension are more likely to have bad values in the others, e.g., tickets with few stops are more expensive. The size of each TO domain is set to 10000. The cardinality of the data set N varies from 100K up to 10M points. The number of TO and PO attributes in a data set varies among 2, 3, 4 and 1, 2, respectively.

In the following we discuss the construction of the DAGs in PO domains. Each PO value corresponds to a DAG node

²<http://randdataset.projects.postgresql.org>

TABLE III
PARAMETERS AND VALUES

Parameter	Range
Data cardinality (N)	100K, 500K, 1M, 5M, 10M
Number of TO attributes ($ TO $)	2, 3, 4
Number of PO attributes ($ PO $)	1, 2
DAG height (h)	2, 4, 6, 8, 10
DAG density (d)	0.2, 0.4, 0.6, 0.8, 1

and the preference among two values is determined by the existence of a path between them. Initially, we obtain a DAG constructed by the containment partial order for sets. The maximum cardinality of the sets determines the number of nodes $|V|$ and the height (h) of the DAG, i.e., the maximum length of any path in the graph (diameter). For a domain of 8 distinct objects, the lattice corresponding to all possible subsets has $h = 8$ and contains $|V| = 2^8 = 256$ nodes. DAGs obtained in this manner are rather dense. To explicitly control the density d , defined as the ratio $d = \frac{|V|}{2^h}$, we retain lattice nodes (along with their incoming and outgoing edges) with a probability equal to d .

Table III enumerates all parameters involved in our experimental evaluation along with their examined values. In all settings, to segregate the effect of a specific parameter, we vary its value within a range, while we set the remaining ones to default values.

B. Static Skyline Queries

We compare the performance of sTSS (henceforth, denoted as TSS) and SDC+ in terms of total (IO and CPU) processing time required to construct the skyline in the presence of PO attributes. Note that for fairness we implement TSS without the main memory R-tree optimization for reducing CPU time described in Section IV-B. In all experiments of this section, the default values for the examined parameters are $N = 1M$, $|TO| = 2$, $|PO| = 2$, $h = 8$ and $d = 0.8$.

We first study the effect of the data set cardinality in the algorithms' performance. Figure 7(a) draws the total processing time for the Independent data set after charging 5 msec for each IO. On average, SDC+ requires roughly twice as much time to execute compared to TSS. In the best setting, $N = 500K$, TSS costs 2.19 times less (24.5 vs. 11.2 sec), whereas in the worst, $N = 10M$, 1.72 times less (96.4 vs. 56.2 sec). The percentage shown next to the markers corresponds to the ratio of CPU over total time. SDC+ is much more CPU intensive than TSS as it requires 3 times more processor cycles, on average. This is an important observation, because the IO cost, unlike the CPU cost, can be mitigated (to some extent) using buffers. As a result, if the cost of a single IO is reduced, TSS can potentially execute 3 times faster.

In Anti-correlated the processing times, shown in Figure 7(b), increase for both methods, as there are more skyline points to be found compared to Independent (8811 vs. 3167 when $N = 1M$). In the best setting, $N = 100K$, TSS is more than 3 times faster (31.3 vs. 10.2 sec), whereas in the worst, $N = 10M$, 2.33 times (1996 vs. 858 sec). Observe that

in Anti-correlated the skyline query is more CPU intensive, contributing roughly to 75% and 50% of the total time for SDC+ and TSS, respectively. However, TSS requires almost 5 times less processor cycles than SDC+ in the default setting ($N = 1M$). A general remark is that the performance benefit of TSS over SDC+ in terms of CPU time increases with the number of results. This is because as the skyline size increases, SDC+ needs to cross-check a larger number of points to eliminate false positives.

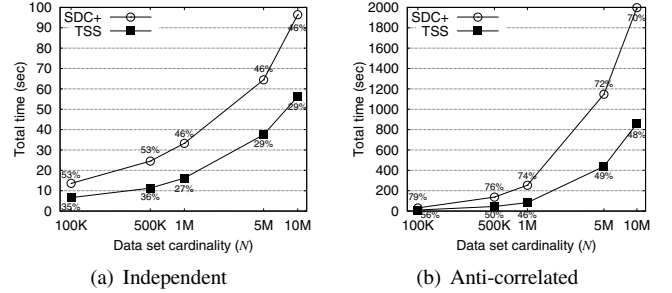


Fig. 7. Static: total time vs. data set cardinality

Figure 8 investigates the effect of the data set dimensionality. Each pair of bars indicates the total processing time and is labelled with two numbers in the x -axis identifying the setting: the first (second) corresponds to the number of TO (PO) attributes. The total time in both methods grows with the number of dimensions due to the increase in the skyline size. For fixed dimensionality the computation cost is larger when more PO attributes exist, e.g., as in the case of (3, 1) compared to (2, 2). This is because in PO domains there are fewer preference relationships among values and hence more non-dominated, i.e., skyline, points. In all settings TSS significantly outperforms SDC+. In particular, Figure 8(a) shows that in Independent the performance benefit of TSS ranges from 1.44 times in the (2, 1) setting up to 5.3 times in the (4, 2) setting. Similar findings hold for Anti-Correlated, where both methods exhibit longer running times. Figure 8(b) shows that TSS's gain is larger as it is 1.7 up to 5 times faster than SDC+. Note that in the last setting, (4, 2), SDC+ did not finish within a time frame of one hour.

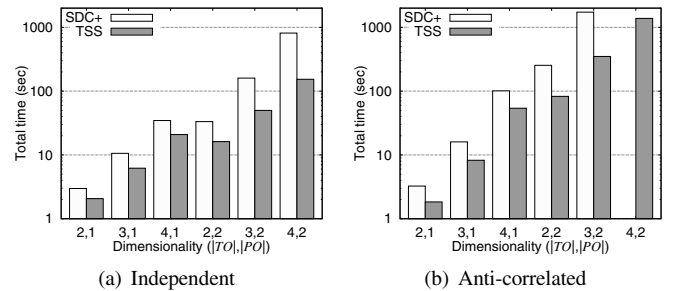


Fig. 8. Static: total time vs. dimensionality

In the next two sets of experiments, we study the effect of the DAG structure associated with the PO domains. First, in

Figure 9 we vary the DAG height h resulting in an exponential increase in the number of possible PO values. The relative benefit of TSS increases with the DAG height, as it executes 5 times faster in Independent and 9 times faster in Anti-correlated compared to SDC+ when $h = 10$.

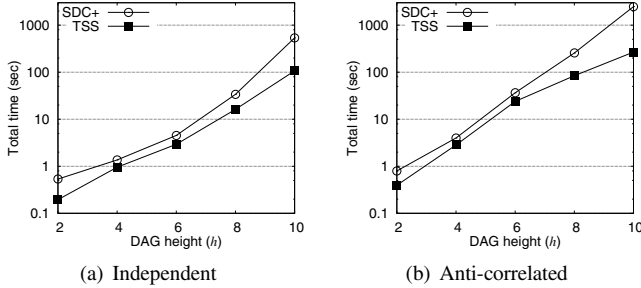


Fig. 9. Static: total time vs. DAG height

Next, in Figure 10 we vary the DAG density d . The skyline size increases with denser graphs; it ranges from 1332 to 3619, and from 4157 to 9781 points in Independent and Anti-correlated, respectively. Figure 10(a) illustrates that the gain of TSS in Independent increases with density reaching 2.4 times when $d = 1$. The benefit becomes more pronounced in the Anti-correlated data set where TSS is 4.5 times faster, as depicted in Figure 10(b). This demonstrates a serious drawback of the methodology in [2]. As d increases, a spanning tree contains fewer edges. Hence, SDC+ captures less preferences among PO values, increasing the number of inexact m -dominance checks.

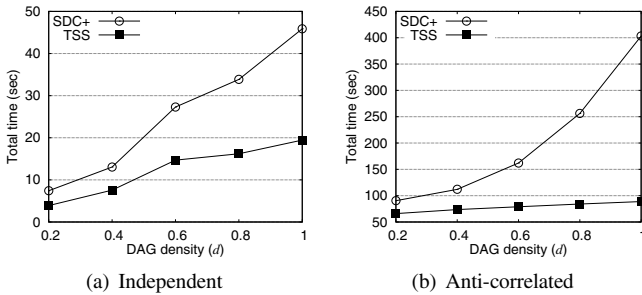


Fig. 10. Static: total time vs. DAG density

The final set of experiments investigates the progressiveness of TSS and SDC+ in the default setting. Figures 11(a) and 11(b) plot the time required to retrieve a subset of the skyline for Independent and Anti-correlated, respectively. Recall that SDC+ can only output results once all points inside a stratum have been examined. The 6 jumps in processing time shown in the SDC+ curves correspond to the computation within the data set's 6 strata. Clearly TSS is more progressive than SDC+ for both types of distribution. For example, in retrieving 50% of the results, TSS is 9 and 21 times faster than SDC+ in Independent and Anti-correlated, respectively.

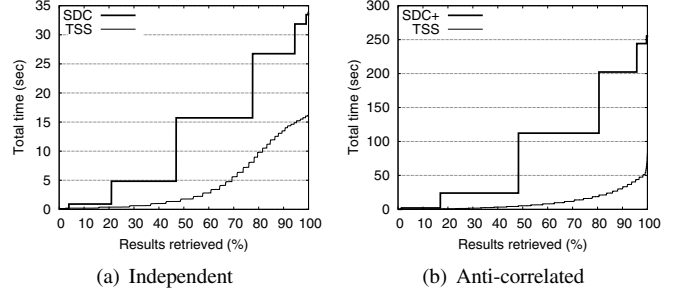


Fig. 11. Static: progressiveness

C. Dynamic Skyline Queries

In this section, we consider dynamic skyline queries that explicitly specify all preference relationships in PO domains and study the performance of the dTSS algorithm (henceforth denoted as TSS). To better gauge performance, we implement a simple adaptation of the SDC+ algorithm suitable for dynamic queries. Recall that SDC+ needs to extract a new spanning tree that complies with the input DAG. This has two serious implications. All node intervals need to be recalculated. Furthermore, the classification of tuples into strata is no longer valid. Therefore, SDC+ must build all index structures from scratch. To expedite the process we first perform an external sort to partition tuples according to strata. Subsequently, all R-tree indexes within a stratum are bulk loaded. The entire process imposes an IO overhead to the query processing as it requires at least two passes over the entire data set. It is important to note that this IO cost cannot be amortized across queries, e.g., using buffers, as suggested in the previous section. TSS, on the other hand, is subject to such optimizations. For fairness, no buffers, global main memory R-tree, pre-processing or caching mechanisms (Section V-B) are used in the implementation of TSS. The default values for all experiments presented in this section are $N = 1M$, $|TO| = 3$, $|PO| = 1$, $h = 6$ and $d = 0.8$.

Figure 12 examines scalability with respect to the data set cardinality N . In both types of distribution, TSS is around 7 times faster when $N = 100K$ and more than 100 times faster when $N = 10M$. In all settings, the IO cost dominates as it contributes to 90% of the total execution time of SDC+. For TSS the IO cost is 97% in Independent and ranges from 82% to 90% in Anti-correlated. The performance benefit of TSS can further increase by amortizing the cost of an IO operation.

In the next set of experiments we vary the dimensionality ($|TO|, |PO|$) of the data set. Compared to SDC+, Figure 13 shows that TSS is 2 orders of magnitude faster in the best case, (2, 1), and 2 times faster in the worst case, (4, 2). As the dimensionality increases, especially when $|PO| = 2$, the processing is dominated by CPU time since the skyline becomes larger. For example, in Anti-correlated when $|TO| = 4$, $|PO| = 2$, the CPU time is 75% and 89% of the total time for SDC+ and TSS, respectively. In such settings a global main memory R-tree would significantly improve the performance of TSS.

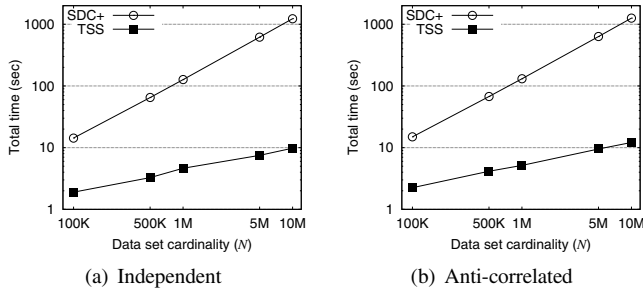


Fig. 12. Dynamic: total time vs. data set cardinality

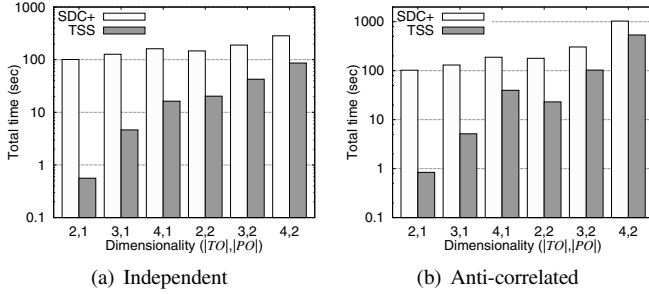


Fig. 13. Dynamic: total time vs. dimensionality

In Figure 14, we investigate the effect of the DAG structure associated with the PO domains in Anti-correlated; similar findings hold for Independent. For very small DAGs, $h = 2$, the performance benefit of TSS reaches 2 orders of magnitude, whereas in very large DAGs, $h = 10$ it amounts to 5 times. Interpolating from Figure 14(a), we estimate that the execution time of TSS would reach that of SDC+ in unrealistically large DAGs containing $2^{14} = 16384$ nodes. One factor, besides skyline size, that contributes to the rapid increase in the cost of TSS is the large number of R-trees present. Recall that in the dynamic case TSS maintains a tree for each group, i.e., for each PO value (Section V-A). During execution a large number of root nodes must be visited. To alleviate this IO cost we could store the roots in contiguous disk pages and retrieve multiple roots at a time.

Finally, Figure 14(b) shows that SDC+ and TSS are rather insensitive to the increase of DAG's density. TSS executes 38 times faster than SDC+ in sparse and 23 times faster in dense DAGs.

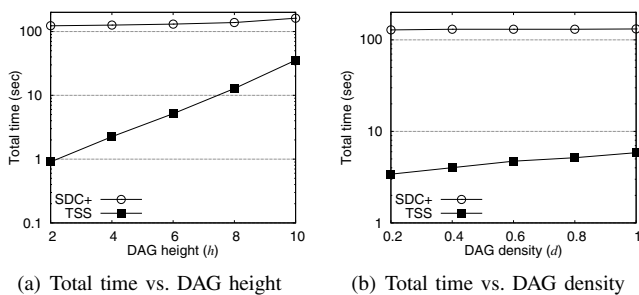


Fig. 14. Dynamic: DAG structure in Anti-correlated

VII. CONCLUSIONS

We have addressed the problem of finding the skyline points when the data include attributes with partially ordered domains. Unlike previous approaches, the proposed TSS framework exhibits two desirable properties, *precedence* and *exactness*, resulting in increased progressiveness and large pruning ability. Furthermore, TSS generalizes to the dynamic case, where the preference relationships among PO values are specified by the query. Our experimental evaluation has demonstrated that TSS outperforms the existing methods by up to 9 times and up to 2 orders of magnitude in the static and the dynamic case, respectively.

REFERENCES

- [1] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.
- [2] C. Y. Chan, P.-K. Eng, and K.-L. Tan, "Stratified computation of skylines with partially-ordered domains," in *SIGMOD*, 2005, pp. 203–214.
- [3] H. T. Kung, F. Luccio, and F. P. Preparata, "On finding the maxima of a set of vectors," *Journal of the ACM*, vol. 22, no. 4, pp. 469–476, 1975.
- [4] W. Kießling, "Foundations of preferences in database systems," in *VLDB*, 2002, pp. 311–322.
- [5] J. Chomicki, "Preference formulas in relational queries," *ACM Transactions on Database Systems*, vol. 28, no. 4, pp. 427–466, 2003.
- [6] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. Springer, 1985.
- [7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *ICDE*, 2003, pp. 717–816.
- [8] I. Bartolini, P. Ciaccia, and M. Patella, "Efficient sort-based skyline evaluation," *ACM Transactions on Database Systems*, vol. 33, no. 4, pp. 1–45, 2008.
- [9] P. Godfrey, R. Shipley, and J. Gryz, "Algorithms and analyses for maximal vector computation," *The International Journal on Very Large Data Bases*, vol. 16, no. 1, pp. 5–28, 2007.
- [10] K.-L. Tan, P.-K. Eng, and B. C. Ooi, "Efficient progressive skyline computation," in *VLDB*, 2001, pp. 301–310.
- [11] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: An online algorithm for skyline queries," in *VLDB*, 2002, pp. 275–286.
- [12] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 41–82, 2005.
- [13] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang, "Efficient computation of the skyline cube," in *VLDB*, 2005, pp. 241–252.
- [14] C. Li, B. C. Ooi, A. K. H. Tung, and S. Wang, "Dada: a data cube for dominant relationship analysis," in *SIGMOD*, 2006, pp. 659–670.
- [15] J. Pei, B. Jiang, X. Lin, and Y. Yuan, "Probabilistic skylines on uncertain data," in *VLDB*, 2007, pp. 15–26.
- [16] E. Dellis and B. Seeger, "Efficient computation of reverse skyline queries," in *VLDB*, 2007, pp. 291–302.
- [17] K. Deng, X. Zhou, and H. T. Shen, "Multi-source skyline query processing in road networks," in *ICDE*, 2007, pp. 796–805.
- [18] M. Sharifzadeh and C. Shahabi, "The spatial skyline queries," in *VLDB*, 2006, pp. 751–762.
- [19] R. C.-W. Wong, J. Pei, A. W.-C. Fu, and K. Wang, "Mining favorable facets," in *KDD*, 2007, pp. 804–813.
- [20] G. Gou and R. Chirkova, "Efficiently querying large xml data repositories: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 10, pp. 1381–1403, 2007.
- [21] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *SIGMOD*, 1989, pp. 253–262.
- [22] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas, "Interactive data analysis: The control project," *IEEE Computer*, vol. 32, no. 8, pp. 51–59, 1999.
- [23] D. Sacharidis, P. Boursos, and T. Sellis, "Caching dynamic skyline queries," in *SSDBM*, 2008.