Caching Support for Skyline Query Processing with Partially Ordered Domains

Yu-Ling Hsueh and Tristan Hascoet

Abstract—Existing methods have addressed the issue of handling each individual skyline query performed on data sets with partially ordered domains. However, it is still very challenging to process such queries for on-line applications with low response time. In this paper, we introduce a cache-based framework, called CSS, for further reducing the query processing time to support high-responsive applications. Skyline queries that were previously processed with user preferences similar to those of the current query contribute useful candidate result points. Hence, the answered queries are cached with both their results and user preferences such that the query processor can rapidly retrieve the result for a new query only from the result sets of selected queries with compatible user preferences. We introduce a similarity measure that establishes the level of similarity between the user preferences of a new query and a cached query; hence the system can start with the most similar candidates. Furthermore, if a new query is only partially answerable from the cache, then the query processor utilizes the partial result sets and performs less expensive constraint skyline queries guided by violated preferences. Furthermore, we introduce two access methods for cached queries indexed by their user preferences to only access a set of relevant cached queries for similarity measures. Extensive experiments are presented to demonstrate the performance and utility of our novel approach.

Index Terms—Query processing, multi-dimensional databases, indexing methods, data management

1 Introduction

The need for effective and real-time GIS applications for analyzing massive high-dimensional data collections to dynamically and judiciously make critical decisions under complex situations exists in a wide variety of disciplines. The skyline query computation, which has been studied extensively in the context of spatiotemporal databases, is a core technique employed to assist in such multi-criteria applications. Skyline queries have been defined as retrieving a set of points that are not dominated by any other points in multi-dimensional space. An object p dominates p', if p has more favorable values than p' in at least one dimension and does not have less favorable values than p' in other dimensions. In many applications, some data dimensions (for example, in the form of hierarchies, intervals, and preferences) are *partially ordered* (PO). In Fig. 1a, the domains d_1 and d_2 are totally ordered (TO), whereas domain d_3, with the options (or values) of $\{A, B, C, D, E\}$ is partially ordered. For the totally ordered domains, we assume that a smaller domain value is preferable. In each partially ordered domain, every user (i.e., query) can declare a *user preference* that describes the preference order among the options.

Fig. 1b shows user preferences g_1 and g_2 for two corresponding queries q_1 and q_2. Such a hierarchical order could, for example, represent the preferences of a frequent traveller with regard to flying with different airlines. We use a *directed acyclic graph* (DAG) to represent a user preference, and each node in the graph denotes a value in a partially ordered domain [2], [16], [24]. For example, g_1 represents that the user prefers airline A to C and D, and prefers C to B. The user preferences are normally obtained along with a user's request for a skyline query. The results of the skyline queries differ for different user preferences. Based on g_1, point p_5 dominates p_8, because the dimensional data in both the totally ordered and the partially ordered domains of p_8 are worse than those of p_5 (recall that query q_1 prefers A to C). Note that p_6 cannot dominate p_9 (although all of the TO attributes of p_6 are better than the TO attributes of p_9) because the preference has only an equivalence preference between B and D (i.e., B and D are sibling nodes).

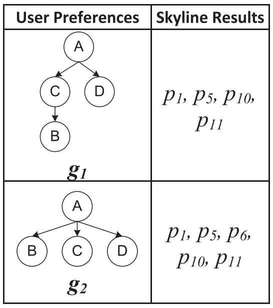
The traditional methods for executing queries over totally ordered domains cannot efficiently handle data sets with partially ordered domains. Related solutions [2], [16] convert each value of a partially ordered domain into integer intervals that enable traditional index-based skyline algorithms (e.g., the branch-and-bound skyline) to handle such queries. The topologically sorted skyline (TSS) [16] method further enhances the pruning ability and progressiveness of this idea by applying topological sorts on the user preferences. Skyline query computations with partially ordered domains are computationally complex in higher dimensions. The cost of the query evaluation process increases as either the number of options for a partially ordered domain or the number of partially ordered domains increases. Therefore, existing systems are often unable to provide up-to-date query results with a fast response time. To address this challenge, we propose a cache-based framework called Caching Support for Skyline Computations (CSS), which supports any existing algorithm for skyline queries with partially ordered domains. The main contribution of CSS is that it caches previous queries

- Y.-L. Hsueh is with the Department of Computer Science & Information Engineering, National Chung Cheng University, 168, University Rd., Min-Hsiung, Chia-Yi, Taiwan. E-mail: hsueh@cs.ccu.edu.tw.
- T. Hascoet is with the Telecommunications Department of the National Institute for Applied Sciences of Lyon, Villeurbanne, France. E-mail: tristan.hascoet@insa-lyon.fr.

Manuscript received 22 July 2013; revised 22 Dec. 2013; accepted 10 Feb. 2014. Date of publication 27 Feb. 2014; date of current version 26 Sept. 2014. Recommended for acceptance by X. Lin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2014.2309125

id	TO Domain		PO Domain
	d_1	d_2	d_3
p_1	1.8	0.3	A
p_2	2.0	0.3	A
p_3	1.8	0.3	B
p_4	1.2	1.0	B
p_5	1.4	1.0	A
p_6	1.0	1.0	B
p_7	2.0	1.0	D
p_8	1.8	1.2	C
p_9	1.5	1.2	D
p_{10}	0.8	1.0	C
p_{11}	2.0	1.4	E
p_{12}	3.0	1.8	E

(a) Sample data set.



(b) DAGs and query results.

Fig. 1. A partially ordered skyline query example.

with both their results and user preferences such that the query processor can rapidly retrieve the result for a new query from the result sets of the selected cached queries with *compatible* user preferences. One of the innovations of this approach lies in our proposed similarity function, which measures the degree of closeness between two user preferences. To facilitate our CSS framework, we test two access methods (i.e., index structures) to access only a relatively small set of cached queries that are similar to a new query before the more time-consuming similarity evaluation for the query selection operation is performed. The filtering process of the access methods results in a small and relevant cached set that is directly accessed by the query processor to retrieve the skyline points for a new query; therefore, the response time of the skyline computation can be greatly reduced. Furthermore, in light of the limited cache space, we adopt a query maintenance scheme that retains only the most popular user preferences and reduces the number of false hits.

2 Related Work

Many secondary storage based algorithms for computing skylines have been proposed. Borzsonyi et al. [1] introduced the non-progressive block-nested-loop (BNL) and divide-and-conquer (D&C) algorithms. The BNL approach recursively compares each data point p with the current set of candidate skyline points, which might be dominated at a later step. BNL does not require data indexing or sorting; however, its performance is influenced by the main memory size. The D&C technique divides the data set into several partitions and computes the partial skyline points for each partition. By merging the partial skyline points, the final skyline can be obtained. Both of these algorithms can incur many iterations and are inadequate for on-line processing. Tan et al. [18] presented two progressive skyline processing algorithms: the Bitmap approach and the index method. The Bitmap approach encodes dimensional values of data points into bit strings to speed up the dominance comparisons. The index method classifies a set of d-dimensional points into d lists, which are sorted in increasing order of minimum coordinate values. The index synchronously scans the lists from the first entry to the last. With pruning strategies, the search space can be reduced. The nearest neighbor (NN) method [6] indexes the data set with an R-tree and utilizes a nearest neighbor search to find the skyline results. This approach repeats the query-and-divide procedure and inserts new partitions that are not dominated by any skyline point into a to-do list. The algorithm terminates when the to-do list is empty. A special method is applied to remove duplicates retrieved from overlapping partitions. The branch-and-bound skyline (BBS) algorithm [14], [15] traverses an R-tree to find a set of skyline points. BBS recursively performs a nearest neighbor search to compute intermediate/leaf nodes that are not dominated by the currently discovered skyline points. Because BBS traverses R-tree nodes based on their *mindist* from the origin, each retrieved point is guaranteed to be a skyline point and can be immediately returned to the user.

Additionally, many recent techniques aim to support continuous skyline computations for moving objects and data streams. Lin et al. [9] utilized n-of-N skyline queries with the most recent n of N elements to support on-line computation against sliding windows over a rapid data stream. Morse et al. [11] illustrated a scalable LookOut algorithm for efficiently updating a continuous time-interval skyline. Sharifzadeh and Shahabi [17] introduced the concept of spatial skyline queries (SSQ). Given a set of data points P and a set of query points Q, SSQ retrieves from P a set of points that are not dominated by any other point in P, while considering their derived spatial attributes with respect to the query points in Q. For dynamic query points, a strategy of processing continuous skyline queries has been presented with a kinetic-based data structure [4]. A suite of novel skyline algorithms based on a Z-order curve [3] has also been proposed [7]. Among the solutions, ZUpdate facilitates incremental skyline result maintenance by utilizing the properties of a Z-order curve. Other related techniques can be found in the literature [5], [10], [12], [13], [19], [20], [23]. However, all of the aforementioned studies differ from the main goal of this research, which is to support the efficient evaluation of skyline queries with partially ordered domains.

The methods proposed in [2], [16], [22], [24] are the most relevant to our work. Chan et al. [2] presented three algorithms for evaluating skyline queries with partially ordered attributes. Their solution transforms each partially ordered attribute into a two-integer domain value, which allows the users to utilize index-based algorithms to compute skyline queries in the transformed space. However, all of the techniques proposed in [2] have limited progressiveness and pruning abilities. In real applications, dynamic preferences for categorical attributes are more common than a fixed ordering for skyline query evaluation. One straightforward solution is to enumerate all of the possible preferences and to materialize all of the results of the preferences; however, the costs of a full materialization are usually prohibitive. Therefore, Wong et al. [22] proposed a semi-materialization method named the IPO-tree search, which stores only partial useful results. With these partial results, the result of each possible preference can be efficiently returned. However, IPO-Tree only considers very simple totally-order-like user preferences. Sacharidis et al. designed a topological sort-based mechanism called topologically-sorted skylines [16], which involves a novel dominance check function to eliminate false hits and misses. In addition, TSS can handle

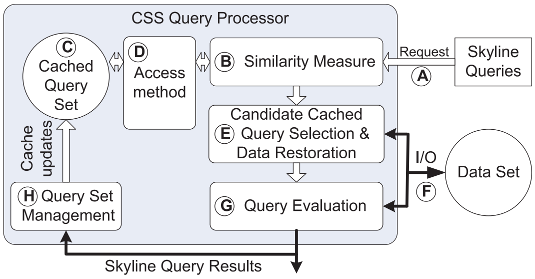


Fig. 2. The CSS system framework.

dynamic skyline queries. Zhang et al. [24] extended the lattice theorem and an off-the-shelf skyline algorithm and then designed a mechanism that employs an appropriate mapping of a partial order to a total order. Nevertheless, none of the aforementioned methods provides a framework which considers the utilization of previously cached query results to further improve the query evaluation performance.

3 System Overview

We consider a partially ordered domain in which a user q (note that we use the terms *user* and *query* interchangeably in this study) declares specific preferences for some data dimensions. Skyline query results vary for different user preferences (as illustrated in the examples of Fig. 1), and the computation is very costly. Our conjecture is that queries that were previously processed with user preferences similar to those of the current query could contribute useful candidate result points. We first introduce some terms to formally describe the problem. A user preference (denoted by $g = (V, E)$) can be represented by a directed acyclic graph, which is a set V of options and a set E of edges. The node set V includes a unique artificial root node r that has no predecessor together with the actual entry node(s) as the successor(s). A *primitive* relation in g from node v_i to node v_j is denoted by $v_i \rightarrow v_j$, and the edge $e \in E$ (with a solid arrow) is directly connected from v_i to v_j. A *transitive* relation is denoted by $v_i \dashrightarrow v_j$, where the edge (with a dashed arrow) does not exist in g. Hence, there is at least one additional node between nodes v_i and v_j. When a user has an *equivalent* preference for v_i and v_j, such a relation is denoted by $v_i \leftrightarrow v_j$, which indicates that the user does not prefer one over the other. To enable a quantitative comparison between two DAGs, we define a numeric similarity measure as an aggregate contribution of the preference relations by comparing pairs of nodes from both DAGs. We adopt an adjacency list to represent a DAG g and then compute g's *transitive closure* $g^+ = (V, E^+)$, which is composed of all of the primitive and transitive relations in g, such that for all v_i and v_j in V, there exists a non-null path (either $v_i \rightarrow v_j$ or $v_i \dashrightarrow v_j$) in E^+.

Fig. 2 shows the overall CSS framework. The query processor initially computes the skyline query results for a query request (A) and caches the result set with the associated preferences (C). Subsequently, when a new query request q enters the system, Task (B) performs a similarity evaluation by computing the similarity scores for q and for each cached query in a selected set accessed through an access method (D). Upon its completion, Task (B) forwards a sorted list of candidate queries to Task (E), which in turn selects a set of candidates from the list. If the new query q is not completely answerable from the cache, the data restoration component accesses the data set (F) to perform less expensive constraint queries to restore all of the possible missing answer points. Finally, in Task (G), given a list of candidate queries and the restored missing answer points, if any, the query processor evaluates the results based on the preferences of the new query to refine the final answer. Because cache space is limited, Task (H) purges the cache by preserving the most popular preferences; i.e., by eliminating queries with the least recently used preferences from the cache.

Before we describe each task in detail, we explain the main concept behind our work. Let $\mathbb{V}_k = \{\overline{V}_k, V_k\}$ be a node set of all of the possible node values allowed in the system for the PO_k dimension, where $\overline{V_k}$ is the *unspecified* node set and V_k is the *specified* node set indicated in g_k. An unspecified node is a node that is not specified by a user so that the node does not appear in a DAG, while a specified node is explicitly specified by a user. For example, the PO domain in Fig. 1a contains $\{A, B, C, D, E\}$ options. However, the user who specifies g_1 does not indicate his/her preference for E. We can say that $\overline{V}_k = \{E\}$ and $V_k = \{A, B, C, D\}$ are an unspecified node set and a specified node set, respectively. We define a data tuple set $T(P, \overline{V})$, where each tuple in P contains at least one unspecified preference node in one of the PO dimensions. In the $T(P, \overline{V})$ set, a data tuple p_i dominates p_j, if and only if all of the TO attributes of p_i is better than those of p_j and the PO attributes of p_i are equal to the values of p_j in their corresponding PO domains. For example, $T(P, \overline{V}) = \{p_{11}, p_{12}\}$ and p_{11} dominates p_{12} because p_{11} has better values than p_{12} in all TO domains and the value (i.e., E) of both tuples in the PO domain are equal. However, p_{10} can not dominate p_{11} because E is unspecified such that the processor has no dominance rule to follow. The following two equations describe the key concept in this paper:

$$P = T(P, \overline{V}) \cup T(P, V) \tag{1}$$

$$P' = T(P, \overline{V}) \cup T(D, V), \tag{2}$$

where P is the entire data set, P' is a subset of P, and D is a candidate result set obtained from results of the selected cached queries. Traditionally, the existing systems access the entire data set (P in Equation (1)) to retrieve the skyline query results. However, CSS handles a small data set $P' \subseteq P$, which contains the data tuples in D retrieved from the results of the cached queries (Equation (2)). In this paper, the unspecified preferences are not considered here because the computations involved are selection operations, which are relatively simple. We focus on retrieving $T(D, V)$ for our query processor. Therefore, CSS efficiently retrieves skyline points with partially ordered domains by reducing the search space from P to P'. Subsequently, because P' is considered, the number of calculations of the similarity scores is reduced as well. The details of each task in Fig. 2 are described in the following sections. Table 1 summarizes the symbols and functions used throughout the following sections.

TABLE 1
Symbols and Functions for the CSS Approach

Symbol	Description
q	A query with a user preference set for each PO domain.
P	The entire data set in the system.
D	The candidate result set obtained from the results of the cached queries.
$g = (V, E)$	A user preference DAG comprising a set V of the specified options with a set E of edges between nodes for one partially ordered dimension.
G	A cached preference set $G = \{g_1, ..., g_m\}$ maintained by the system for one partially ordered dimension.
$\hat{G}$	A cached preference set sorted by similarity scores in descending order for one partially ordered dimension.
$A \rightarrow B$	A primitive preference in which node A is directly connected to B.
$A \dashrightarrow B$	A transitive preference in which node A is indirectly connected to B.
$A \leftrightarrow B$	A user who has an equivalent preference for A and B.
$p \vdash p'$	p dominates p'.
$S(g, g')$	A similarity function that returns a number to measure the similarity between g and g'.
$T(S, N)$	A filtering function that selects from data set S and returns data tuples with one corresponding PO attribute equal to any node value in N.
d and n	Number of TO and PO domains, respectively.

4 Caching Skylines for Efficient Skyline Computations

4.1 User Preference Similarity Measures

To enable a quantitative comparison, we define a similarity function that returns an aggregate contribution of all preference pairs between two compared DAGs. The similarity function $S(g, g')$ measures the correlation between g (associated with a new query q) and g' (associated with a cached query) and returns a similarity score. $S(g, G)$ returns the sorted query list $\hat{G}$ in descending order of the similarity scores for g with respect to all DAGs in G. In the best case, the system finds one *perfectly similar preference* DAG g^*, which is the top one DAG in $\hat{G}$. In that scenario, the corresponding query associated with g^* contains the minimum-complete skyline results for q. Additionally, a perfectly similar preference DAG must satisfy the following two properties.

Property 1 (Violation-Free). *A perfectly similar preference DAG g^* has no* violations *that contradict the preferences in g.*

A violation exists when two preferences conflict. We give a more formal definition below. Consider the examples in Fig. 3, where g is the preference DAG of a new query and g_1 is the DAG of a cached query. In DAG g, $g.A \rightarrow g.B$, while $g_1.A$ and $g_1.B$ hold an equivalent preference. Therefore, there is no preference violation (i.e., only *matches*), and more importantly, we can observe that the structure of a perfectly similar preference DAG does not have to be identical to g. A violation refers to a preference conflict that occurs when the comparison of two corresponding relation pairs contradict each other. The formal definition of a violation and a match can be stated as follows:

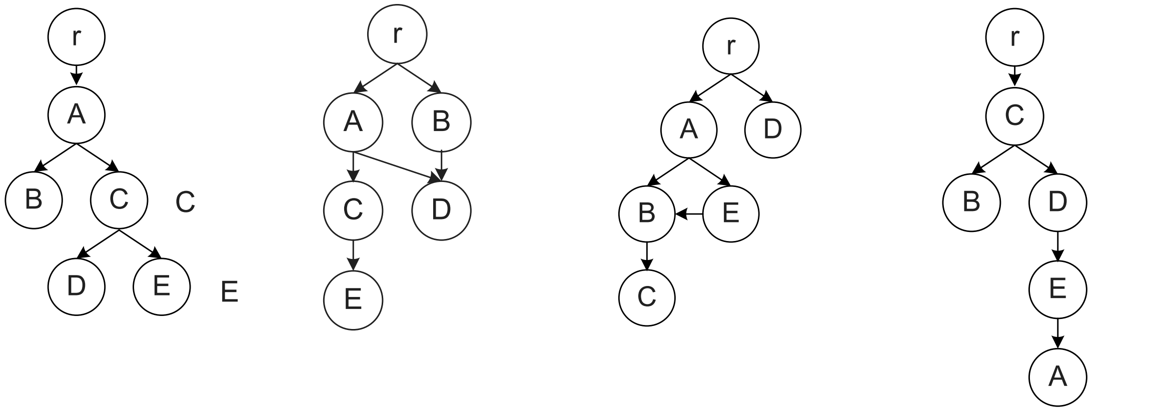
Definition 1 (Violation). *Given g and g', a pair of preference nodes are in violation of each other, if either $g.v_i \rightarrow g.v_j$ or $g.v_i \dashrightarrow g.v_j$ holds and at the same time either $g'.v_j \rightarrow g'.v_i$ or $g'.v_j \dashrightarrow g'.v_i$ exists. Additionally, if $g.v_i \leftrightarrow g.v_j$ is true, then a violation occurs when $g'.v_i \not\leftrightarrow g'.v_j$ holds.*

Definition 2 (Match). *Given g and g', if $g'.v_i \rightarrow g'.v_j$, $g'.v_i \dashrightarrow g'.v_j$ or $g'.v_i \leftrightarrow g'.v_j$, a match occurs with either $g.v_i \rightarrow g.v_j$ or $g.v_i \dashrightarrow g.v_j$. Additionally, if $g.v_i \leftrightarrow g.v_j$ is true, then a match occurs when $g'.v_i \leftrightarrow g'.v_j$ holds.*

We can restate Definition 1 and say that a violation condition does not hold if a match (Definition 2) is true. Because $g'.v_j \rightarrow g'.v_i$ or $g'.v_j \dashrightarrow g'.v_i$ contradicts $g.v_i \rightarrow g.v_j$ or $g.v_i \dashrightarrow g.v_j$, there is a violation. We interchangeably use the term *compatible* to describe two matched relations. A violation and a match are illustrated in the examples (a) and (b) in Fig. 3. The relation of $g_1.B \rightarrow g_1.D$ violates $g.B \leftrightarrow g.D$ so that the answer set of the query using g_1 may eliminate the data tuples which may be included in the answer set of the query using g due to the relation of $g_1.B \rightarrow g_1.D$. On the contrary, $g_1.C \leftrightarrow g_1.D$ matches $g.C \rightarrow g.D$, because the relation of $g_1.C \leftrightarrow g_1.D$ preserves the tuples which are necessary for the answer set of the query using g; hence, the system needs to further refine the answer set of the query using g_1 to retrieve the final answer set.

Property 2 (Inclusion). *The query result of a perfectly similar preference DAG g^* with respect to g is a minimum-complete result set of the query q using g among all cached queries G.*

Let R^* be the query result of q^* using g^*. Because a perfectly similar preference is free of violations, R^* must be an inclusive super set of q using g. For example, in Fig. 1b, assume that g_1 and g_2 are new and cached



(a) g (a new query) (b) g_1, $S(g, g_1) = -0.18$ (c) g_2, $S(g, g_2) = -2.27$ (d) g_3, $S(g, g_3) = -4.36$

Fig. 3. Example DAGs.

queries, respectively. The skyline result of the query q_2 using g_2 is a super result set of the query q_1 using g_1, because we can retrieve a complete result set for q_1 from the result set of q_2. We previously described the definition of a violation, and we now provide a proof for its properties. Given g and g', a violation occurs when a preference with more constraints is used in g' compared to the corresponding preference used in g. The query results of q' using g' could exclude some data points that would likely be result points of q using g. Therefore, R^* must contain a complete answer set for the corresponding query q because g^* is free of violations. Furthermore, the preferences of g and its perfectly similar preference DAG g^* must possess the closest degree of similarity, such that R^* does not contain many irrelevant results that would affect the performance of the query evaluation. A perfectly similar preference DAG g^* must exhibit identical preferences. However, in reality, such an optimal perfectly similar preference DAG rarely occurs, and hence, the system must choose an existing available DAG with a high degree of similarity to be as close as possible to an optimal selection. To enable quantitative measures for the degree of similarity, we define a similarity function in the following section.

4.2 Similarity Functions

For similarity comparisons, two states (*match* and *violation*) are expressed by $Q_{g,g'}(v_i, v_j)$, which compares two corresponding pairs of relations between v_i and v_j from both g and g', where g is the user preference of a new query q, and g' is the user preference of a cached query q'. Given g and g' (transitive closure forms), the function $Q_{g,g'}(v_i, v_j)$ returns a real number, which is used to aggregate the matching contributions, each of which is either a match or a violation, and is computed as shown in Equation (3).

$$S(g,g') = \frac{\sum\limits_{\forall v_i \in V', v_j \in V, v_i \neq v_j} Q_{g,g'}(v_i, v_j)}{|E^+_g|} \tag{3}$$

$$Q_{g,g'}(v_i, v_j) = \begin{cases} 1 & \text{(i) a match, or} \\ -|E^+_g| & \text{(ii) a violation.} \end{cases}$$

Here, the range of $S(g,g')$ is $-|E^+_{g'}| \leq S(g,g') \leq 1$. $|E^+_g|$ and $|E^+_{g'}|$ denotes the total number of edges of the transitive closure g and g', respectively. For all valid relations (v_i,v_j) in g', v_i is an intermediate node in g', and v_j is a specified node in g. $Q_{g,g'}(v_i, v_j)$ returns 1 for a match (case (i)). A violation incurs $-|E^+_g|$ as a penalty (case (ii)), which reduces the accumulation. The maximum similarity score is 1 for the case that there are matches for all of the comparisons (i.e., g and g' are identical). Likewise, the minimum similarity score is $-|E^+_{g'}|$ if there are violations for all of the comparisons. Therefore, a similarity score $S(g,g')$ ranges from $-|E^+_{g'}|$ up to 1. For example, in Fig. 3, $Q_{g,g_2}(A,C)$ returns 1 as a match even though $g_2.A \dashrightarrow g_2.C$, while $g.A \rightarrow g.C$. For a violation in case (ii), we deduct a maximum similarity score of $-|E^+_g|$ to cause the current summation to become negative. Consequently, user preferences that are free of violations are always ranked higher than user preferences with violations. For example, $g_1.B \rightarrow g_1.D$ violates $g.B \leftrightarrow g.D$. Therefore, the similarity score is negative, if at least one violation occurs. Because there are nine matches (five relations from r to A, B, C, D, E plus $A \rightarrow C$, $A \rightarrow D$, $A \rightarrow E$, $C \rightarrow E$) and one violation, the similarity score equals -0.18 (i.e., (9-11)/ 11). The overall similarity measure algorithm is outlined in Algorithm 1.

Algorithm 1 $S(g, g')$

```
let $V$ and $V'$ be the node sets of $g$ (used by a new query) and $g'$ (used
   by a cached query), respectively.
scores = $\phi$;
for (every $v_i \in V'$) do
   for (every node $v_j \in V, v_i \neq v_j$) do
      if ($g'.hasEdge(v_i, v_j)$) then /* A valid relation in $g'$ */
         if ($g.hasEdge(v_i, v_j)$) then /* A match */
            scores += 1;
         else then /* A violation */
            scores += $-|E^+_g|$;
   end for
end for
return scores; /* return final scores */
```

4.3 Cached Query Selection

Perfectly similar preferences are rarely found among cached queries, particularly when the maximum number of options allowed per user preference increases and because the users are more likely to specify very different preferences. For example, if the query processor accesses the top query from the cached queries sorted by the similarity scores in descending order, and it has a negative similarity score (i.e., indicating the existence of preference violations), this situation would imply that the system cannot retrieve a complete result set for the new query from the existing cached queries (because all of the queries have negative scores). To address this challenge, we introduce a novel approach in this study to select a minimum set of cached queries Q' from the sorted cached queries. The query processor can find a complete set of skyline results by combining the results of each query in Q', which is referred to as a *complementary query list*, and the definition is shown as follows.

Definition 3 (Complementary Query List). *Given a new query q, a complementary query list consists of a minimum set of top j queries $\{q_1, \ldots, q_j\}$, selected from the cache sorted by similarity scores in descending order such that every preference relation in g_k of q in the PO domain k, $\forall k = 1 \ldots n$, is matched with the relation of at least one query in the complementary list in the corresponding PO dimension. The result of q is completely retrievable from the* candidate result set $D = \{q_1$*.result* $\cup$ q_2*.result* $\cup \ldots \cup$ q_j*.result*$\}$.

Lemma 1. *Assume a new query q and a complementary query list consisting of two cached queries q_i and q_j, both with the user preferences g_k, where $k = 1 \ldots n$ for all partially ordered domains. The results of q are completely retrievable from the candidate result set $D = \{q_i$.result $\cup$ q_j.result$\}$.*

Proof. By definition. Because a complete result set for q is computed and guided by the preference relations in g, and because the cached queries q_i and q_j consist of all of the relations declared in g, the union of the result sets of q_i and q_j must contain the complete set of skyline result points. □

To find the complementary query list for a new query q, we start from the cached query with the highest similarity score. However, in the worst case, such an operation is still expensive when none of the high-ranked cached queries have compatible relations with q. Therefore, a heuristic threshold parameter (δ) is introduced to avoid caching a query that has a large result set, which might not help the query processor to reduce the complexity of the skyline computations. Furthermore, because the top-ranked cached query q^{top} has the fewest violations with the new query, we adopt it as a *baseline query*, through which we obtain a violated relation set $\mathbb{E}$ with respect to the new query. We then select the cached queries Q' as the complementary query list, where each query in Q' contains at least one relations against a corresponding violated edge $e \in \mathbb{E}$ in at least one partially ordered domain. For a violated relation in $\mathbb{E}$, we relax the definition of a relation $(v_i \dashrightarrow v_j)$, which is a broader notation that covers the relation of $(v_i \rightarrow v_j)$. We establish Lemma 2 as follows:

Lemma 2. *Let $e = (v_i \dashrightarrow v_j)$ be a violated relation in $\mathbb{E}$ for the partially ordered domain PO_k. Let q' be a selected query from a complementary query list Q' and let g'_k of q' have NO such violated relations $e' = (v_s \dashrightarrow v_j)$ (v_s is any specified node, including v_i in g'_k). Therefore, the data tuples $T(q'.result, v_j)$ retrieved from the result set of q' must contain the candidate skyline result points for the new query.*

Proof. By definition. Let P_1 and P_2 be the data tuples, in which the dimensional data value in PO_k is equal to v_i and v_j, respectively. Let all of the TO_w attribute values of P_1 be more preferable than the TO_w attribute values of P_2, $\forall w = 1 \ldots d$, and let the remaining PO_c attribute values of P_1 be more preferable than those of P_2, $\forall c = 1 \ldots n$ except for PO_k. Assume that both the user preferences g_k of the new query q and g'_k of the cached query q' in PO_k contain a relation edge $(v_t \dashrightarrow v_j)$, where $v_t \neq v_i$. Hence, P_1 cannot dominate P_2. Furthermore, the eliminated data tuples in P_2, due to the $(v_t \dashrightarrow v_j)$ relation for g'_k, must also be eliminated for g_k. Therefore, all of the P_2 tuples that are considered to be skyline points for the new query q must be preserved in the results of q'. □

To obtain a candidate result set for the new query q, we first insert the resulting tuples with the node values defined in the user preferences of q. For each selected query of the complementary query list, we combine only the missing tuples that might be eliminated by the violated relations. For example, q' is violation-free with respect to the violated relation $(v_i \dashrightarrow v_j)$. We insert $T(q'.result, v_j)$ into the candidate result set only when all of the result tuples in $q'.result$ are the missing data tuples for the result of the new query. For example, in Fig. 3, q_1 using g_1 is the baseline cached query because it has the highest similarity score. The algorithm starts to search for the violated edges, if any exist. Preference g_1 has one violated edge, $(g_1.B \rightarrow g_1.D)$ with respect to g. Because the result set of g_1 does not contain a complete result set for q, the system searches for the next cached query. Consequently, the complementary query list of q must contain both g_1 and g_2 because the violation $(g_1.B \rightarrow g_1.D)$ does not hold in g_2. Therefore, the skyline query result can be found only from the results of the corresponding queries using g_1 and g_2 without accessing the entire data set.

The algorithm for finding a candidate result set for g is outlined in Algorithm 2. In Line 2, D is a candidate result set that is initialized to the resulting tuples of q^{top} (i.e., q_1). In Line 3, *vioEdges* stores the violation edges (with respect to g) that are returned by the *findVioEdges* function. Line 3 checks the baseline query for any violated relations. In Line 5, if *vioEdges* is not empty, then q is not an answerable query when using the current selected cached queries. In this case, if the number of tuples in the current candidate result set is less than a threshold (δ, which is also used to avoid seeking a large candidate result set), then the query processor performs constraint queries to restore the missing data points (Lines 5–12). Line 6 performs a *rmVioEdges* function, which deletes the violated relations $\mathbb{E}$ in *vioEdges*, if g_i has compatible relations with regard to $\mathbb{E}$. The set S contains the result of the corresponding q_i using g_i. In Lines 8–9, by using the filtering function $T(S, N)$, only the relevant missing tuples of the new query result are inserted into the candidate result set.

Algorithm 2 *FindCandidateSet*(G, δ)

```
1: let $Q = \{q_1, q_2, ..., q_m\}$ be the sorted cached query list and let $\hat{G} = \{g_1, g_2, ..., g_m\}$ be their corresponding sorted user preference list in descending order of the similarity score with respect to $g$
2: let $D = q_1$.result be the initial candidate data set.
3: vioEdges = findVioEdges($g$,$g_1$)
4: $i = 2$ //The next index of a user preference in $\hat{G}$
5: while (vioEdges $\neq \phi$ AND $|D| < \delta$ AND $i \leq n$) do
6:     ($S$, $\mathbb{E}$) = rmVioEdges(vioEdges,$g_i$);
7:     if (S is not empty) then
8:         let $N$ be a node set of the sink node $v_j$ from each relation pair of $(v_i, v_j)$ in $\mathbb{E}$
9:         insert $T(S, N)$ into $D$;
10:    end if
11:    $i = i + 1$
12: end while
13: return (vioEdges,$D$)
```

4.4 Unanswerable Queries

A new query q is termed *unanswerable* if the selected cached queries do not contain a complete result set for q. This case can occur when all of the relations of the cached user preferences violate the relations specified in q. However, even in this case, some optimization can be achieved. Instead of accessing the entire data set to retrieve the skyline results, CSS performs less expensive constraint queries to restore the missing data tuples that were eliminated because of the violated relations of the cached queries. Thus, we let *SkylineQuery* be a function that embodies the non-caching algorithm TSS [16] to evaluate a skyline query.

In Section 4.3, we describe the *rmVioEdges* function, which removes the violated relations when at least one of the cached queries (other than the baseline query) has compatible relations in the corresponding PO domains. When the function terminates and there are still non-removable violated relations (when *vioEdges* is not an empty set in Algorithm 2, Line 13), the following operations are necessary to restore the missing data tuples that might have been eliminated by such violated relations. The first step is to create a new DAG with only the violated relation $e \in vioEdges$. We use the same user preferences that were defined in the selected queries (which contain the violated relations in $vioEdges$) for the remaining PO domains, such that *SkylineQuery* can restore only the data tuples that were eliminated due to the violated relations of their corresponding partially ordered domains. The following lemma and proof of correctness support this process.

Lemma 3. *Assume a non-removable violated relation $e = (v_i \dashrightarrow v_j) \in VioEdges$ and assume that e is a relation in the partially ordered domain PO_k, where $1 \leq k \leq n$. We clone the user preferences of q to a new g^{new} and replace the existing preference in the PO_k dimension by g^k with only one relation e. Let P_1 and P_2 be the data tuples in which the dimensional data values in PO_k are equal to v_i and v_j, respectively. Let $P_2' \subseteq P_2$ be the data tuples eliminated during the skyline evaluation. We can conclude that P_2' must contain all of the missing data due to the violated relation e.*

Proof. Because the set P_2' contains the data tuples that were eliminated based on the new user preference g^{new}, all of the PO dimensional values of each data tuple in P_2' must be worse than those of at least one data tuple in P_1. For each non-dominated tuple in $(P_2 - P_2')$, there must exist at least one PO or TO dimensional value that is not worse than the corresponding dimensional value of the data tuples in P_1. The data tuples $(P_2 - P_2')$ are originally preserved in the corresponding query results, which are inserted into the candidate result set. Therefore, P_2' must be the eliminated data set due to $e = (v_i \dashrightarrow v_j)$. □

We summarize the steps below to perform such constraint queries for each violated relation $(v_i \dashrightarrow v_j)$ of the partially ordered domain PO_k.

Step 1: Let g_i contain relation $(v_i \dashrightarrow v_i)$, let g_j contain relation $(v_j \dashrightarrow v_j)$, and let g_{ij} contain relation $(v_i \dashrightarrow v_j)$ in the PO_k domain, respectively.

Step 2: Let $S_i = SkylineQuery(T_i, g_i)$, where $T_i =$ Select ALL from dataTable, where the dimensional value of $PO_k = v_i$.

Step 3: Let $S_j = SkylineQuery(T_j, g_j)$, where $T_j =$ Select ALL from dataTable, where the dimensional value of $PO_k = v_j$.

Step 4: Let $S_{ij} = SkylineQuery\ (T_{ij}, g_{ij})$, where $T_{ij} = S_i \cup S_j$.

Step 5: Return $\mathbb{R} = T_{ij} - S_{ij}$.

Steps 2 and 3 find a skyline result set that excludes the data tuples eliminated by the constraint $v_i \dashrightarrow v_i$ and $v_j \dashrightarrow v_j$, respectively. Thus, if a data tuple p_1 dominates p_2 in all of the totally ordered domains ($p_1.TO_x < p_2.TO_x$, $\forall x = 1 \ldots d$), and the partially ordered domains ($p_1.PO_y$ is preferable to $p_2.PO_y$) for all of the dimensions except PO_k, then we can eliminate p_2 because the dimensional value of $p_1.PO_k$ equals that of $p_2.PO_k$. Next, in Step 4, S_{ij} is the skyline point set retrieved from $S_i \cup S_j$. S_{ij} contains only data tuples in which the dimensional value of PO_k equals v_i or v_j, which are not eliminated in Steps 2 and 3. Finally, Step 5 retrieves only the data tuples that are eliminated due to the $v_i \dashrightarrow v_j$ constraint. Hence, the performance of the data restoration process through the constraint skyline query computations is improved when compared to using the whole data set. Let us continue the example in Fig. 3 and consider the data set sample in Fig. 1a. Assume that g_2 and g_3 do not exist in the cache. Because the only cached user preference g_1 has one violated relation $(B \rightarrow D)$, the query processor cannot retrieve a complete skyline result for q. For example, for the violated edge $B \rightarrow D$, we first perform *SkylineQuery* (T_B, $B \rightarrow B$), where $T_B = \{p_3, p_4, p_6\}$ and $S_B = \{p_3, p_6\}$. This approach performs a second *SkylineQuery* (T_D, $D \rightarrow D$) operation, where $T_D = \{p_7, p_9\}$ and $S_D = \{p_7, p_9\}$. *SkylineQuery* (T_{BD}, $B \rightarrow D$) returns $S_{BD} = \{p_3, p_6\}$, where $T_{BD} = S_B \cup S_D = \{p_3, p_6, p_7, p_9\}$. Now, the missing data eliminated by the $B \rightarrow D$ constraint are $T_{BD} - S_{BD} = \{p_7, p_9\}$.

4.5 The CSS Framework

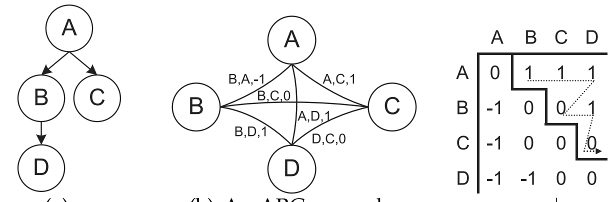
Finally, the main algorithm for the CSS framework is illustrated in Algorithm 3. Lines 4–5 are executed when the first skyline query is performed or when no relevant cached queries are selected. In Lines 8–9, the highest-ranked cached query has no violations with the new query and contains a complete result set for the new query. Hence, the system directly performs the skyline query based on this result set. Otherwise, the algorithm performs the alternate list of operations (e.g., *FindCandidateSet*) to find a candidate result set D for the new query. Line 15 restores the eliminated data tuples using the steps described in Section 4.4.

Algorithm 3 CSS(q)

```
1: let $g$ be the user preference used by a new query $q$
2: let $G$ be a set of user preferences of cached queries
3: let $R$ be a storage for storing the result set
4: if ($G$ is empty) then
5:    return SkylineQuery($P$,$g$), where $P$ is the entire data set
6: else
7:    let $\hat{G}$ be a sorted user preference list in descending order of
      $S(g, g_i), \forall g_i$ in $G$
8:    if (the top element $g' \in \hat{G}$ used by $q'$ has a similarity score $> 0$) then
9:       $R$ = SkylineQuery($q'.result$, $g$)
10:   else
11:      (vioEdges,$D$) = FindCandidateSet($\hat{G}$, $\delta$)
12:      if (vioEdges is empty) then
13:         $R$ = SkylineQuery($D$, $g$)
14:      else
15:         /* $q$ is un-answerable */
16:         perform constraint queries to restore the eliminated data tuples $\mathbb{R}$
            by checking each violated relation in vioEdges
17:         $R$ = SkylineQuery($D \cup \mathbb{R}$, $g$)
18:      end if
19:   end if
20: end if
21: return $R$ /* the skyline points */
```

5 Query Indexing and Maintenance

The cache-based CSS approach leads to several challenges. One challenge related to caching answered queries is the data modeling issue for partially ordered domain data. To access relevant queries that are stored in the cache without stepping through the entire data set, we must index the cached queries by their user preferences. However, a user preference cannot be quantified without losing some preference information. Existing graph indexing approaches [21] generally focus on how to store graph data so as to facilitate similarity searches. The similarity referred to similar searches for graphs is to measure the similarity between two graphs in terms of their morphisms. A well-known technique to identify two similar graphs is graph edit distance (GED), which is the minimum number of edit operations needed to transform one graph into another. In other words, if GED equals 0, two graphs are identical. However, the concept of similarity here is different from what we define in our work. We consider the



(a) g (b) An ARG example. (c) A_g^+

Fig. 4. An example of a DAG transformation.

violations and matches in our similarity measures to facilitate the cache-based skyline query search over partially ordered domains. Therefore, using an existing graph indexing approach that approximates a similar set before carrying the computations to retrieve the final results may incur significant false negatives. In this study, we propose two query index structures for the query processor to effectively locate a set of relevant cached queries. The CSS framework conducts the query selection operation to determine a set of candidate queries. It is advantageous to determine high-level query similarity measures (as a filter step) before performing the query selection operation (as a refinement step), which is more time-consuming. We propose the query index structures described in the following sections.

5.1 Basic Query Indexing

We first propose a basic index structure that uses the entry node(s) of a cached query q' as the key(s) and stores each group (at most h groups, where h is the maximum number of option nodes in a PO domain) because if a cached query q' and the new query q have the same entry node sets (the nodes in the first level, not including the artificial node r), the fewest violations may arise. Therefore, the relevance of two queries is approximately measured by the entry node (s). For example, in Figs. 3b, 3c, and 3d, the keys of g_1 used by q_1 are A and B; the keys of g_2 used by q_2 are A and D; and the key of g_3 used by q_3 is C. When a new query q using g is requested, the system finds q_1 and q_2 to be the relevant queries for q. Hence, p_3 is ignored because it is less relevant to q. An additional table is used, and each key points to the corresponding cached queries that have a key associated with the entry node(s).

5.2 Advanced Query Indexing

The basic index structure described in Section 5.1 simply uses the entry node(s) of a cached query as the key(s) to index and store each cached query. In such an index structure, the relevance of two queries is only approximately measured by the entry node(s). However, because a user preference can become quite complex when the number of option nodes increases, the basic index structure does not allow us to capture a high similarity for the system to efficiently compute the search results. In this section, we describe an advanced index structure that is similar to the approach used in object decomposition techniques for image processing. A DAG can be conceptually mapped to a corresponding *attribute relational graph* (ARG), which is essentially used to describe the features of an image for content retrieval. For example, the user preference g and its corresponding ARG are shown in Figs. 4a and 4b, respectively. Therefore, a DAG is mapped directly onto an n-dimensional *feature point*. We then build an R-tree to store each feature point, with a link to its corresponding query. The main idea here is to store similar user preferences that are close to each other with the use of R-trees to rapidly retrieve similar preferences while preserving all of the relations in the DAG. To search for sufficiently similar queries to support our CSS framework, we find the k nearest feature points from the index for each PO domain according to the feature point transformed from the DAG of a user preference. In this section, we present how we transform a DAG to a feature point and how we maintain indexes with regard to the cache size limitation.

PO Domain 1	PO Domain 2	Entire Query Preference Feature Point
[1][1][1][0][1][0]	[1][0][-1][1][0][0]	[1][1][1][0][1][0][1][0][-1][1][0][0]

Fig. 5. Feature point concatenation.

5.2.1 DAG Transformation and Feature Points

Because the feature point is to be stored in an R-tree, we must keep its dimensionality as low as possible in order to reduce the computational time used for selecting relevant queries, while preserving all of the information contained in the DAG. Conventionally, the nodes of an ARG are annotated with object labels, and the edges between two nodes are labelled with information about the relationships as shown in Fig. 4b. To transform a DAG g with h nodes, we represent the DAG by an adjacency matrix A_g and compute its transitive closure A_g^+, as shown in Fig. 4c. Note that we assign values of -1 to the entries that are dominated by the counterparts. Because the matrix A_g^+ is symmetrical and anti-reflexive, we omit the entries in the bottom-left corner and the diagonal entries of the matrix, to avoid redundant entries that do not need to be preserved for the feature point transformation. Let f_g denote a feature point of $\frac{h^2-h}{2}$ dimensions, which corresponds to the top-right corner of the matrix A_g^+. Consider $a_{ij} \in A_g^+, h \geq j > i$ and $j > 1$. We define a_{ij} as the value in the kth dimension of fp_g, if and only if k, i and j satisfy the following equation:

$$k = \frac{i}{2}[(h-1)+(h-i)] - (h-j).$$

Taking Fig. 4c as an example, the a_{ij} values along the dashed line are chosen to form the feature point f_g, which is (1, 1, 1, 0, 1, 0). Furthermore, this approach can be easily extended to adapt multiple *PO* domains by concatenating the sub-domain feature points of different *PO* domains into a single domain, as shown in Fig. 5.

5.2.2 Query Retrievals

To index feature points in the cache, we use a spatial R-tree index structure that preserves the closeness of the feature points to efficiently access a set of similar cached queries with respect to the new query. Each data point in the form of a feature point indexed by an R-tree corresponds to a cached query. We use the nearest neighbor function, which returns the K nearest points (in the sense of similarity) of the new query to obtain a relevant cached query set. The extensive tests presented in Section 7 were conducted to estimate the optimal node capacity parameter of an R-tree and the K parameter, both of which influence the efficiency of our method.

5.3 Query Maintenance

Naturally, the number of cached queries progressively increases after the system startup until the cache is full. Because the cache space is limited, efficient strategies for query set management are needed to enable long-term operations. In Section 4.3, a threshold δ is used to avoid caching a query with a large result set which might not help the query processor to reduce the complexity of the skyline computations, because the time reduction of the skyline evaluation might not be significant when compared with the computational time of a non-cached skyline evaluation. Furthermore, the use of a threshold prevents unhelpful queries from being cached and results in more free space for other queries. As the number of cached queries increases, the query processor requires more time to find a relevant set of cached queries with respect to the new query for similarity evaluation and query selection. Therefore, our proposed query index structures play an important role in effectively locating a set of relevant cached queries for the new query. Furthermore, to keep track of the cached information in order to improve the “hit rate”, we adopt the concept of the *least frequently used* (*LFU*) cache replacement algorithm.

6 Time and Space Complexity Analysis

The main algorithm for the CSS framework is bounded by the skyline query evaluation, which has a complexity of $O(log|P|)$, where $|P|$ is the number of data points, because the skyline query evaluation is based on the R-tree index structure. However, if the cached results are fully utilized, the time complexity can be reduced to $O(log|D| + h* n * |C|)$, where $|D|$ is the candidate result set obtained from the results of the cached queries, $|C|$ is the number of cached queries, h is the maximum number of option nodes in a PO domain, and n is the number of PO domains. The cost of $(h * n * |C|)$ comes from the similarity measures. The space complexity is $O(|P| + |C|)$, where $|P|$ is the number of data points and $|C|$ is the number of cached queries.

The time complexity for the basic index structure is $O(n * |P|)$ and the space complexity is $O(h^2 * n * |P|)$, where $|P|$ is the number of data points, h is the maximum number of option nodes in a PO domain, and n is the number of PO domains. Because the basic index structure indexes each DAG based on its entry node, there are at most h groups and h nodes in each DAG. There are $(h^2 * |P|)$ for each group, so the time complexity is $(h^2 * n * |P|)$ for all PO domains. The time complexity for the advanced index structure is $O(n * log|P|)$, because the feature points for a PO domain are indexed by one R-tree. The space complexity is $O(h^2 * n * |P|)$, because a feature point has at most $(h^2\text{-}h)/2$ dimensions, and there are $n * (h^2\text{-}h)/2$ for all dimensions for each data point in terms of the PO domains. Therefore, the space complexity is $O(h^2 * n * |P|)$ for the entire data set.

TABLE 2
Simulation Parameters

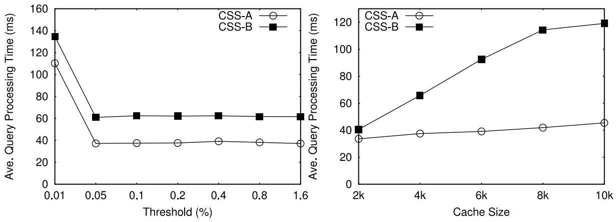
Parameter	Default	Range
Data cardinality (P)	$128k$	$64k$, $128k$, $256k$, $512k$, $1024k$
Query cardinality (Q)	100 and $10k$	$1k$, $5k$, $10k$, $15k$, $20k$
Number of TO domains ($\|TO\|$)	2	2, 3, 4
Number of PO domains ($\|PO\|$)	2	1, 2, 3, 4, 5, 6
Cache threshold (δ)	0.8%	0.01%, 0.05%, 0.1%, 0.2%, 0.4%, 0.8%, 1.6%
Cache size (ξ)	$4k$	$2k$, $4k$, $6k$, $8k$, $10k$
DAG height (h)	6	2, 4, 6, 8, 10
DAG density (d)	0.6	-

7 Experimental Evaluation

We evaluate the performance of the CSS algorithm using the advanced index structures (CSS-A) and basic index structures (CSS-B) by comparing it with the TSS approach [16], which handles partially ordered domains. Unlike CSS, TSS consults the entire data set whenever it executes a new skyline query request. CSS adopts TSS as the underlying algorithm to evaluate the skyline results for partially ordered domains, and adds its own caching and query indexing mechanisms. Therefore, the query processing time for the first query is identical to the TSS approach. Subsequently, as the cache takes effect, performance gains are achieved. We utilize R-trees as the underlying structure to index the data and skyline points. Specifically, we use the Spatial Index Library [8] for the R-tree index. Our data set for the totally ordered domain is in the range of $[0, 1000)$, and we generate up to 1,024,000 normally distributed data points with dimensions in the range of 3 to 10. For partially ordered domains, we generate a PO value of 2 to 10 for each data dimension, where 10 is the maximal number of distinct options for a user preference in the system. The *height* of a DAG is the maximum length of any path in the graph. The lattice node size for a DAG is determined by a *height* from 2^2 to 2^{10} and a *density* ratio of 0.6. We set the threshold δ to be a percentage of the data set. Here, δ is used to avoid caching a query with a large result set. Furthermore, δ is also used for the query selection operation, which avoids caching a query with a result set size that is larger than δ. We experiment with a small and a large query set, which is set to 100 and $10k$, respectively, and we obtain the average query processing time for comparison. TSS and CSS-A and CSS-B are compared only over a small query set because TSS incurs significant overhead. In addition, as TSS is a scalable non-cache-based approach, the average query processing time is fairly stable over either a small or a large query set. The main measurement in the following simulations is the CPU time (for evaluating each skyline query). Our experiments use several metrics to compare these algorithms. Table 2 summarizes the default parameter settings used in the following simulations.

7.1 Cache Threshold (δ) and Cache Size (ξ)

First, we evaluate the average query processing time by varying the cache threshold size. The choice of a



(a) Threshold (b) Cache size

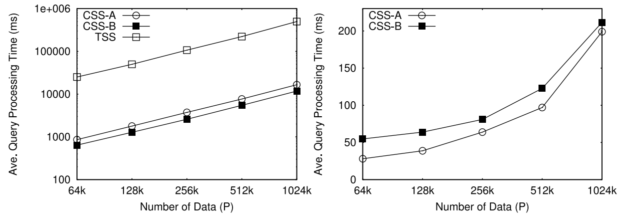
Fig. 6. Performance as a function of cache threshold and cache size.

threshold size is critical for the performance of our system. If the threshold is too small, then more queries with small result sets are cached. Such queries with small result sets often have intricate user preferences; therefore, the system might perform numerous constraint skyline queries to restore missing data tuples due to a large number of violations. Furthermore, fewer queries would be qualified for caching in the system, which results in fewer chances to utilize the cached queries with compatible user preferences. As a result, the performance is degraded. If the threshold is too high, then more queries with large result sets are cached. Consequently, more cache space is occupied, and the cache is more likely to be full. Hence, the system must perform cache replacement operations more often. However, because the CSS algorithm retrieves a small candidate result set, a cached query with a large result set does not necessarily reduce the gain for the skyline evaluation time. A small candidate result set requires less time for the skyline evaluation, which is the most time-consuming portion of the overall operation. Fig. 6a shows the average query processing time for threshold sizes ranging from 0.01 to 1.6 percent of the entire data set over a large query set (i.e., 10k) for the basic and the advanced index structures. When the threshold size is set to 0.01 percent or lower, the performance of the CSS is degraded in terms of the average query processing time. A threshold size greater than 0.05 percent provides stable performance. Conventionally, we chose a mean number 0.8 percent as the default threshold size for the remaining experiments.

Next, the size of the cache is important in terms of its overall impact on improving the performance of the system. If the cache size is too small, the system suffers from more disk I/Os because fewer useful queries are cached. However, if the cache size is too large, the CSS algorithm must process a large set of cached queries with respect to each new query. Specifically, many similarity measurement operations must be executed to retrieve the candidate data set. Fig. 6b illustrates the tradeoff in which a cache size of 2 to 10k appears to result in optimal performance. We chose 4k as the default cache size for both the CSS-A and CSS-B approaches.

7.2 Data Cardinality

Figs. 7a and 7b show the average query processing time as a function of the number of data points over a small and a large query set, respectively. Overall, the query processing time increases with the number of data points. CSS-A and CSS-B achieve a significant reduction in terms of the average query processing time compared with TSS. This finding is indicative of how CSS takes advantage of the results of a set of cached queries with compatible user preferences. Because the TSS approach considers the entire data set when evaluating the skyline result for each new query, the average query processing time is significantly high with a large data set, especially as a result of the R-tree constructions. In CSS, because the system is only required to construct R-trees for a small candidate result set, the overall query processing time is reduced. For a small query set (see Fig. 7a), CSS-B outperforms CSS-A because the starting overhead (e.g., DAG transformation) of CSS-A is high. However, CSS-A is more efficient in handling a large query set as shown in Fig. 7b, because CSS-A determines better query similarity measures before performing the time-consuming query selection operation. The overall query processing time of both index structures is significantly reduced when compared to the performance of running a small query set. Furthermore, the percentage of the number of accessed data points using CSS-A over TSS is only up to 3.65 percent, which is a very small portion of data that are accessed for the skyline computations. The experimental results confirm the benefits of the CSS approach to caching, which achieves better CPU performance than the TSS technique.

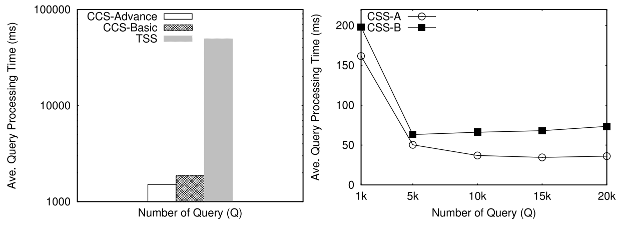


(a) Small query set (b) Large query set

Fig. 7. Performance as a function of data cardinality.

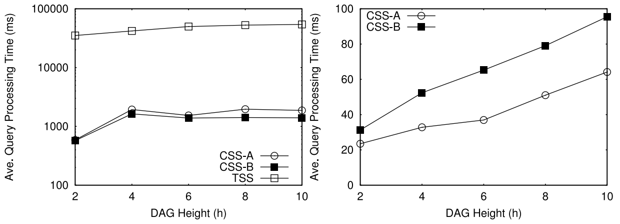
7.3 Query Cardinality

Next, we report on the impact of the query cardinality on the performance of the approaches. Fig. 8a shows the average query processing time over a small query set (100 queries) and Fig. 8b shows the average query processing time for query cardinality values ranging from 1 to 20k. When the system is started, the query processing time of the tree approaches for evaluating the first skyline query are identical. As time progresses, the CSS system caches more queries; hence, the algorithm can utilize and retrieve a candidate result set that is a subset of the entire data set. The performance (see Fig. 8a) is significantly improved when more relevant queries are accessed by new queries. Both CSS-A and CSS-B outperform TSS dramatically, although



(a) Small query set (b) Varying query sets

Fig. 8. Performance as a function of query cardinality.



(a) Small query set (b) Large query set

Fig. 9. Performance as a function of DAG height.

TABLE 3
Impact of the K Factor versus the Node Capacity on the Average Query Processing Time (ms)

K \ Node Capacity	5	10	20	40	80	160
5	43.17	38.22	36.95	39.15	37.87	38.25
10	43.88	39.51	37.69	39.63	38.67	36.84
20	44.82	39.88	38.80	39.84	38.34	36.97
40	45.48	40.91	38.33	40.12	39.13	37.61
80	47.63	41.82	38.76	40.87	39.70	37.66
160	49.39	42.75	40.55	40.57	39.26	39.25

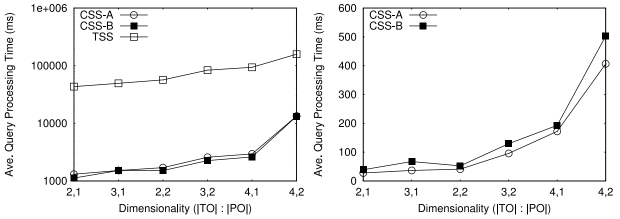
the performance of CSS-A and CSS-B is very competitive. In Fig. 8b, however, as the number of queries increases, CSS-A achieves up to a 50.87 percent improvement on the average query processing time compared with CSS-B. Furthermore, the chance of finding perfectly similar preferences for each skyline evaluation is about 55 percent. Consequently, a new query can fully take advantage of cached queries to retrieve query results.

7.4 User Preference Complexity

In this experiment, we investigate the effect of the DAG height associated with *PO* domains. In Figs. 9a and 9b, we vary the DAG height from 2 to 10 over a small and a large query set, respectively. All algorithms incur an increasing CPU load as the DAG height increases. When the total number of lattice nodes of a DAG increases, CSS suffers mainly from a higher computational cost of the similarity evaluation because the system must check a large number of lattice nodes (or relations) for similarity comparisons. Furthermore, intensive dominance operations are performed because the query processor might access intricate user preferences that consist of more lattice nodes. Consequently, the skyline result points are often large, such that the performance is degraded. The performance of the TSS approach remains relatively stable, albeit at a worse level than the performance of the CSS. However, both CSS-A and CSS-B still outperform TSS greatly as shown in Fig. 9a, while in Fig. 9b, CSS-B fails to filter out irrelevant cached queries because a user preference can become complex when the number of option nodes increases. CSS-A, on the other hand, achieves significant reduction in query processing time.

7.5 Dimensionality

We investigate the impact of dimensionality on the performance of the TSS and CSS techniques. Figs. 10a and 10b illustrate the average query processing time versus the PO and TO dimensions in pairs of (size of PO, size of TO), ranging from 2 to 4 for the TO domains and 1 to 6 for the PO domains over a small and a large query set, respectively. When the dimensionality increases, the performance of all of the methods is degraded because the R-trees fail to filter out irrelevant data entries in higher dimensions. The system may output more skyline points that, in turn, incur more dominance checks. From Fig. 10a, we can see that CSS-A and CSS-B significantly outperform TSS. In Fig. 10b, as a large query set is experimented with, we can see that the average query processing time of CSS-A and CSS-B for all dimensions in pairs is much lower than the average processing time of these two methods over a small query set (see Fig. 10a), because more cached queries can be utilized to reduce the compaction cost. However, CSS-A slightly outperforms CSS-B as a result of high dimensionality.



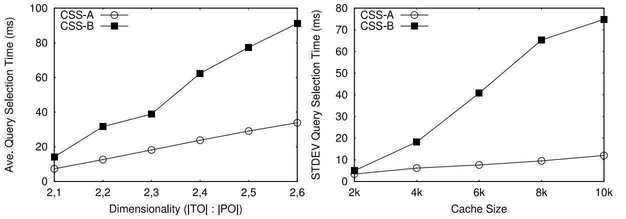
(a) Small query set (b) Large query set

Fig. 10. Performance as a function of dimensionality.

7.6 Index Performance and Stability

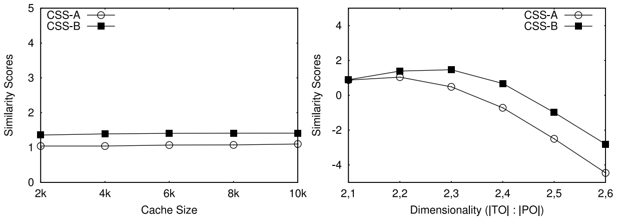
We evaluate the performance of the basic and advanced index structures used in the CSS approach. In this section, we focus on the query efficiency of both index mechanisms in terms of retrieving and scoring a candidate query list from the cache. Therefore, the query selection time measured in this section represents the time required to retrieve queries from the cache through the index structures, and to perform similarity measures on the selected queries. The efficiency of the CSS query processor is affected by several inner and outer parameters. The spatial index library that we use for the R-trees is a generic library whose inner parameters must be adapted to our use. In Table 3, we present the results in terms of the average query selection time, which guide us in selecting the optimal K-factor and the node capacity parameter of the R-tree for indexing the cached queries. Because the two parameters do not appear to be independent, the results in the table represent the average query processing time obtained while testing both of the parameters simultaneously. Finally, we select 5 for K and 20 for the node capacity for the remaining simulations.

We evaluate the impact of the user preference complexity on both of the index structures. The basic index structure used in the CSS-B method considers a cached query to be relevant if the DAG of one of its PO dimensions has at least one root node in common with the currently computed DAG of the user query in the same dimension. The consequence of this step is that the more complex a user preference is, the more root nodes each DAG has. Therefore, more queries are likely to be selected from the cache to perform similarity measures, which are very time consuming. The complexity of a user preference is determined by certain



(a) Impact of dimensionality (b) Stability improvement

Fig. 11. Index performance.



(a) Impact of cache size (b) Impact of dimensionality

Fig. 12. Index result degradation.

factors, such as the number of nodes, the density, and the dimensionality of the PO domains. Here, we focus on the dimensionality, and the experimental results are shown in Fig. 11a. For the advanced index structure used in the CSS-A method, as the dimensionality increases, the dimensions of the feature point increase as well. As a result, the query selection operations incur more computational time because of the distance comparisons used by the nearest neighbor function to select a candidate query list with more dimensions. Additionally, because the R-tree computes intermediately overlapping regions to cluster the feature points, the efficiency is affected by the increasing dimensions of the feature point. In summary, we can see that CSS-A has a better tolerance for complex user preferences than CSS-B.

The basic index structure considers a cached query to be relevant if the DAG of one of its PO dimensions has at least one root node in common with the currently computed DAG of the user query in the same dimension. Therefore, depending on the number of root nodes of the DAG, the number of selected cached queries differs from one query to another. Hence, the computational time required to obtain a candidate query list varies significantly for different queries. If a user presents preferences with many root nodes, the system selects many cached queries that are considered to be relevant and then performs several time-consuming similarity measurements. The following figure represents the standard deviation of the query selection time (labeled STDEV query selection time) of both index structures with respect to the cache size. As shown in Fig. 11b, CSS-A offers a much more stable performance than CSS-B.

7.7 Index Result Degradation

The main goal of using index structures is to reduce the average query processing time required to select appropriate cached queries by reducing the number of cached queries on which the similarity measurement operation is performed without affecting the relevance of the selected queries. The previous sections illustrate the reduction in query processing time obtained by using index structures. In this section, we investigate the relevance of the queries for both indexes. The CSS algorithm uses a similarity measurement function to score the cached queries depending on their level of similarity with respect to the currently computed user query preference (i.e., a new query). The cached query with the highest score is called the baseline query, which is used as a representative in our comparisons regarding the index result degradation. To measure the relevance of the queries selected by both index structures, we compare the score of the baseline queries obtained from both methods. The similarity score obtained by using the basic index structure can be treated as an almost optimal score (i.e., the highest score calculated based on a non-index structure) because significantly relevant or irrelevant cached queries are selected through the use of the basic index structure. Figs. 12a and 12b represent the variance in the baseline scores of both of the index structures with respect to cache size and the number of PO domains, respectively. We can see that the baseline scores of both methods are almost identical. The scores of CSS-A using the advanced index structure are at most 2 percent lower than the scores of CSS-B using the basic index system and yet, the query selection time shows significant improvement. Therefore, the advanced index structure does not incur a high index result degradation.

8 Conclusions and Future Work

We have introduced a novel framework, called CSS, to process skyline queries with partially ordered domains by caching query results with their unique user preferences. The query response time of a new query is significantly reduced by retrieving its result from cached result sets with compatible specifications. Our similarity measures enable the query processor to find a complementary query list, which contains a minimum set of cached queries whose answered results are utilized to answer a new query. If a query result cannot be fully computed from the cache, we propose the use of less expensive constraint skyline queries to restore missing data tuples. Finally, we adopt query set indexing and maintenance schemes to efficiently access the cached queries and to lower the space overhead. Our experimental evaluation demonstrates that CSS improves upon existing methods and is especially well-suited for interactive applications that require a fast response time.

Several issues can be refined in the future. First, currently, CSS adopts a basic cache management strategy (i.e., the least frequently used replacement) to update the cache. However, there could be a better strategy to improve the hit rate such that the overall performance can be improved as well. Second, our index structures can be further improved by integrating with the existing indexing methods [21] for graphs. Third, the CSS approach does not consider dynamic data attributes in TO domains. We plan to investigate how to efficiently update the skyline results with dynamic TO domains.

References

[1] S. Börzsönyi, D. Kossmann, and K. Stocker, "The Skyline Operator," *Proc. 17th Int'l Conf. Data Eng. (ICDE)*, pp. 421-430, 2001.

[2] C.Y. Chan, P.-K. Eng, and K.-L. Tan, "Stratified Computation of Skylines with Partially-Ordered Domains," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pp. 203-214, 2005.

[3] V. Gaede and O. Günther, "Multidimensional Access Methods," *ACM Computing Surveys*, vol. 30, no. 2, pp. 170-231, 1998.

[4] Z. Huang, H. Lu, B.C. Ooi, and A.K.H. Tung, "Continuous Skyline Queries for Moving Objects," *IEEE Trans. Knowledge and Data Eng.*, vol. 18, no. 12, pp. 1645-1658, Dec. 2006.

[5] M.S. Islam, R. Zhou, and C. Liu, "On Answering Why-Not Questions in Reverse Skyline Queries," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 973-984, 2013.

[6] D. Kossmann, F. Ramsak, and S. Rost, "Shooting Stars in the Sky: An Online Algorithm for Skyline Queries," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB)*, pp. 275-286, 2002.

[7] K.C.K. Lee, B. Zheng, H. Li, and W.-C. Lee, "Approaching the Skyline in Z Order," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB)*, pp. 279-290, 2007.

[8] S.I. Library, "http://www.research.att.com/marioh/spatialindex/index.html, 2014..

[9] X. Lin, Y. Yuan, W. Wang, and H. Lu, "Stabbing the Sky: Efficient Skyline Computation over Sliding Windows," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 502-513, 2005.

[10] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang, "Selecting Stars: The k Most Representative Skyline Operator," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 86-95, 2007.

[11] M.D. Morse, J.M. Patel, and W.I. Grosky, "Efficient Continuous Skyline Computation," *Proc. Int'l Conf. Data Eng. (ICDE)*, p. 108, 2006.

[12] M.D. Morse, J.M. Patel, and H.V. Jagadish, "Efficient Skyline Computation over Low-Cardinality Domains," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB)*, pp. 267-278, 2007.

[13] M. Nagendra and K.S. Candan, "Layered Processing of Skyline-Window-Join (swj) Queries Using Iteration-Fabric," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 985-996, 2013.

[14] D. Papadias, G. Fu, J.M. Chase, and B. Seeger, "Progressive Skyline Computation in Database Systems," *ACM Trans. Database Systems*, vol. 30, pp. 41-82, 2005.

[15] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An Optimal and Progressive Algorithm for Skyline Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pp. 467-478, 2003.

[16] D. Sacharidis, S. Papadopoulos, and D. Papadias, "Topologically Sorted Skylines for Partially Ordered Domains," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 1072-1083, 2009.

[17] M. Sharifzadeh and C. Shahabi, "The Spatial Skyline Queries," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB)*, pp. 751-762, 2006.

[18] K.-L. Tan, P.-K. Eng, and B.C. Ooi, "Efficient Progressive Skyline Computation," *Proc. 27th Int'l Conf. Very Large Data Bases (VLDB)*, pp. 301-310, 2001.

[19] L. Tian, L. Wang, P. Zou, Y. Jia, and A. Li, "Continuous Monitoring of Skyline Query over Highly Dynamic Moving Objects," *Proc. ACM Sixth Int'l Workshop Data Eng. for Wireless and Mobile Access (MobiDE)*, pp. 59-66, 2007.

[20] A. Vlachou, C. Doulkeridis, and N. Polyzotis, "Skyline Query Processing Over Joins," *Proc. ACM SIGMOD Int'l Conf Management of Data (SIGMOD)*, pp. 73-84, 2011.

[21] X. Wang, X. Ding, A.K.H. Tung, S. Ying, and H. Jin, "An Efficient Graph Indexing Method," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 210-221, 2012.

[22] R.C.-W. Wong, A.W.-C. Fu, J. Pei, Y.S. Ho, T. Wong, and Y. Liu, "Efficient Skyline Querying with Variable User Preferences on Nominal Attributes," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 1032-1043, 2008.

[23] P. Wu, D. Agrawal, Ö. Egecioglu, and A.E. Abbadi, "Deltasky: Optimal Maintenance of Skyline Deletions without Exclusive Dominance Region Generation," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 486-495, 2007.

[24] S. Zhang, N. Mamoulis, B. Kao, and D.W.-L. Cheung, "Efficient Skyline Evaluation over Partially Ordered Domains," *Proc. VLDB Endowment*, vol. 3, no. 1, pp. 1255-1266, 2010.

Yuling Hsueh received the MS and PhD degrees in computer science from the University of Southern California (USC), in 2003 and 2009, respectively. She has been an assistant professor in the Department of Computer Science and Information Engineering, National Chung Cheng University (CCU), Taiwan, since 2011. Her research interests include temporal/spatial databases, mobile data management, scalable continuous query processing, and spatial data indexing.

Tristan Hascoet is working toward the graduate degree at the National Institute of Applied Science in Lyon (INSA de Lyon). His research interest include data management, information theory and cryptography.