

# C-SKY: Caching Skylines for Efficient Skyline Computations with Partially-Ordered Domains

Yu-Ling Hsueh<sup>†</sup> Roger Zimmermann<sup>‡</sup> Wei-Shinn Ku<sup>§</sup>

<sup>†</sup>Dept. of Computer Science, University of Southern California, USA

<sup>‡</sup>Computer Science Department, National University of Singapore, Singapore

<sup>§</sup>Dept. of Computer Science and Software Engineering, Auburn University, USA

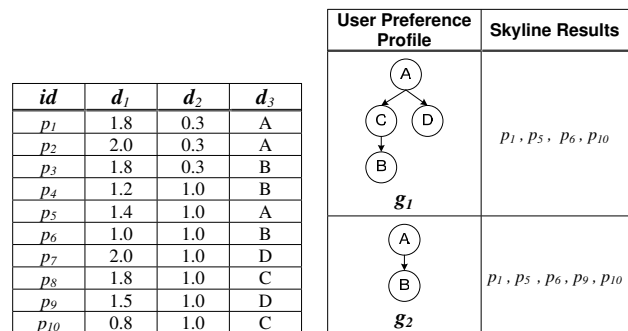
{hsueh@usc.edu, rogerz@comp.nus.edu.sg, weishinn@auburn.edu}

**Abstract**—The results of skyline queries performed on data sets with partially-ordered domains vary depending on users’ preference profiles specified for the partially-ordered domains. Existing work has addressed the issue of handling each individual query with some efficiency. However, processing large volumes of such queries for online applications with low response time is still very challenging. In this paper, we introduce a novel approach, termed *C-SKY*, to reduce the latency by caching query results with their unique user preferences. Of paramount importance in this case is that cached queries with *compatible* preference profiles need to be utilized. For this purpose, we introduce a similarity measure that establishes how related a new query is to each of the previously cached queries and profiles. The similarity measure allows the cached entries to be effectively ordered according to descending values and hence query processing can start with the most promising candidates. If a new query is only partially answerable from the cache, the proposed method pursues a second optimization step. The query processor utilizes the partial result sets and augments them by performing less expensive constraint skyline queries guided by constraint violations between different query preference profiles. Furthermore, to lower the space overhead, we propose a cache management scheme where only the most popular preferences are preserved. Extensive experiments are presented to demonstrate the performance and utility of our novel approach.

## I. INTRODUCTION

Skyline query computations are important for multi-criteria decision making applications and they have been studied intensively in the context of spatio-temporal databases. Skyline queries have been defined as retrieving in multi-dimensional space a set of points, which are not dominated by any other points. An object  $p$  dominates  $p'$ , if  $p$  has more favorable values than  $p'$  in all dimensions. In many applications, some data dimensions – for example in the form of hierarchies, intervals and preferences – are *partially-ordered* (*PO*), where some data lack preferences (e.g., non-specified data nodes). In Figure 1 (a), domains  $d_1$  and  $d_2$  are totally-ordered whereas domain  $d_3$ , with the options (or values) of  $A$ ,  $B$ ,  $C$ , and  $D$ , is partially-ordered. On each partially-ordered domain, every user (*i.e.*, a query) may declare a *user preference profile* which describes the preference order among the options. Figure 1 (b) shows the user preference profiles  $g_1$  and  $g_2$  of the corresponding queries  $q_1$  and  $q_2$ . Such an ordering may, for example, represent the preferences that a frequent traveller has with regards to flying with different airlines. We use a *directed acyclic graph* (*DAG*) to represent a user preference profile and each node in the graph denotes a value

in a partially-ordered domain. Profile  $g_1$  declares a preference on all the options in the domain, while the profile  $g_2$  specifies an ordering only on options  $A$  and  $B$  (*i.e.*, the associated query has no preference on options  $C$  and  $D$ ). Nodes  $C$  and  $D$  are *unspecified nodes*. With different user preference profiles, the results of the skyline queries are different. For preference profile  $g_1$ , point  $p_5$  dominates  $p_8$ , because the dimensional data in both the totally-ordered and the partially-ordered domains for  $p_8$  are equal or worse than for  $p_5$  (recall that query  $q_1$  prefers  $A$  to  $C$ ). Note that  $p_6$  cannot dominate  $p_9$  (although all the *TO* attributes of  $p_6$  are better than the *TO* attributes of  $p_9$ ), since the profile has an equivalent preference between  $B$  and  $D$  (*i.e.*,  $B$  and  $D$  are sibling nodes). For  $g_2$ , since the user does not specify any preferences for the nodes  $C$  and  $D$ , the query processor allows the dominance of two data tuples with the same *PO* values. For example,  $p_{10}$  dominates  $p_8$ , and  $p_9$  dominates  $p_7$ . Therefore, the rest of the non-dominated data tuples with *PO* values equal to  $C$  or  $D$  must be preserved as skyline points.



(a) Sample data set.

(b) DAGs and query results.

Fig. 1. Partially-ordered skyline query example.

The traditional methods to execute queries over totally-ordered domains cannot efficiently handle data sets with partially-ordered domains. Current solutions ([2], [13]) convert each partially-ordered domain data column into integer intervals that enable the traditional index-based skyline algorithms (e.g., *BBS*) to handle such queries. The *TSS* [13] method enhances the pruning ability and progressiveness of this idea further by applying topological sorts on the user preference profiles.

Skyline query computations with partially-ordered domains are very computationally complex in higher dimensions. The cost of the query evaluation process increases as both the number of options for a partially-ordered domain or the number of partially-ordered domains increase. Therefore, existing systems are often unable to provide up-to-date query results with quick response times. To address this challenge we propose a novel approach termed *Caching Skylines for Efficient Skyline Computations (C-SKY, for short)*. The main contribution of *C-SKY* is that it caches previous queries with both their results and user preference profiles such that the query processor can rapidly retrieve a skyline result set for a new query from a set of existing candidate queries with *compatible* user preference profiles. One of the innovations of the approach lies in our proposed similarity function that measures the degree of closeness between two user preference profiles. Since the query processor directly accesses a relatively small candidate result set to retrieve the skyline points for a new query, the response time of the skyline computation can be greatly reduced. To respect the generally limited cache space, we also propose a novel cache management approach that only reserves the most popular user profiles and reduces the number of false hits.

The remainder of this paper is organized as follows. The concept of similarity measure and the design of the similarity function are described in Sections III-A and III-B, respectively. The method for the candidate cached query selection is described in Section III-C, and the details of handling unanswerable queries is outlined in Section III-D. Section IV presents and details our cache management design. Finally, we describe the overall *C-SKY* algorithm in Section V and we verify the performance of our techniques in Section VI.

## II. RELATED WORK

Numerous secondary storage based algorithms for computing skylines have been proposed before. 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 later. BNL does not require data indexing and sorting, however its performance is influenced by the main memory size. The D&C technique divides the dataset into several partitions and computes the partial skyline of the points in every partition. By merging the partial skylines, the final skyline can be obtained. Both algorithms may incur many iterations and are inadequate for on-line processing. Tan *et al.* [15] presented two progressive skyline processing algorithms: the *bitmap* approach and the *index* method. *Bitmap* 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 the minimum coordinate value. The index scans the lists synchronously from the first entry to the last. With pruning strategies, the search space can be reduced. The *nearest neighbor* (NN) method [5] indexes the dataset with an R-tree and utilizes

a nearest neighbor search to find the skyline results. The approach repeats the query-and-divide procedure and inserts the 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 [12] traverses an R-tree to find the set of skyline points. BBS recursively performs a nearest neighbor search to compute intermediate/leaf nodes which 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 returned to users immediately. Additionally, many of the recent techniques are aimed at continuous skyline support for moving objects and data streams. Lin *et al.* [8] utilize  $n$ -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.* [10] illustrated a scalable *LookOut* algorithm for updating a continuous time-interval skyline efficiently. Sharifzadeh and Shahabi [14] introduced the concept of *Spatial Skyline Queries* (SSQ). Given a set of data points  $P$  and a set of query points  $Q$ , SSQ retrieves those points of  $P$  which are not dominated by any other point in  $P$  while considering their derived spatial attributes with respect to query points in  $Q$ . For moving query points, a continuous skyline query processing strategy 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 [6]. 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 [9], [11], [16], [17]. However, all the aforementioned studies differ from the main goal of this research which is to support efficient evaluation for skyline queries with partially-ordered domains.

Two groups of prior methods are the most relevant to our work [2], [13]. Chan *et al.* [2] presented three algorithms for evaluating skyline queries with partially-ordered attributes. Their solution is to transform each partially-ordered attribute into a two-integer domain which allows users to utilize index-based algorithms to compute skyline queries in the transformed space. However, all the techniques proposed by Chan *et al.* have limited progressiveness and pruning abilities. Sacharidis *et al.* designed a topological sort based mechanism named *Topologically-sorted Skylines* (TSS) [13] which is both progressive and exact. TSS introduces a novel dominance check function which eliminates false hits and misses. In addition, TSS is able to handle dynamic skyline queries. Nevertheless, the research did not consider the utilization of previously cached query results to further improve the query evaluation performance.

## III. CACHING SKYLINES FOR EFFICIENT SKYLINE COMPUTATIONS

Consider a partially-ordered domain where a user  $q$  (note that we will use the terms *user* and *query* interchangeably

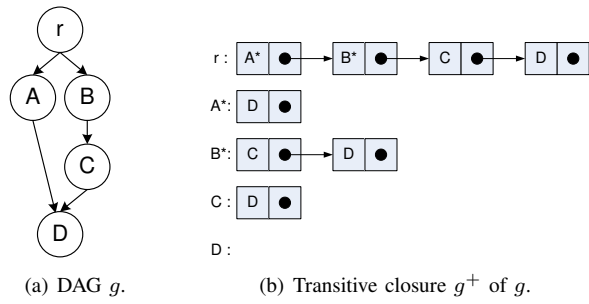


Fig. 2. Sample user preference DAG and its transitive closure.

in this study) declares specific preferences for some data dimensions. Skyline query results vary with different user preferences (as is illustrated in the examples of Figure 1) and the computation is very costly. Our conjecture is that query results that were previously obtained with a user preference profile similar to the profile of the query currently under consideration may contribute useful candidate result points. We will first introduce some terminology to help formally describe the problem. A user preference profile (denoted by  $g = (V, E)$ ) can be represented by a *directed acyclic graph* (DAG), which consists of a set  $V$  of option nodes and a set  $E$  of edges. The node set  $V$  includes a unique artificial root node  $r$  with no predecessor together with the actual entry node(s) as successor(s). A *primitive relation* (or preference) 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  $g$ , and hence there is at least one additional node between nodes  $v_i$  and  $v_j$ . When  $v_i$  and  $v_j$  have equivalent preferences, 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 the aggregate contribution of the preference relations that compare pairs of nodes from both DAGs. We adopt an adjacency list to represent a DAG  $g$  and then compute  $g$ 's *transitive closure*  $g^+ = (V, E^+)$  consisting of all 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  $E^+$ . A transitive closure list contains at most  $n$  sub-lists ( $n$  equals the maximum number of options allowed for a user preference profile), each of which starts with an intermediate node in the DAG. An actual entry node of a DAG is marked with an asterisk (\*) to distinguish it from other intermediate and leaf nodes. Figures 2 (a) and (b) illustrate a DAG  $g$  and its corresponding transitive closure. The artificial root node is  $r$ , whereas  $A^*$  and  $B^*$  with a asterisk mark each are the actual entry nodes.

Figure 3 shows the overall framework of the C-SKY system. The query processor initially computes the skyline query results for a query request (A) and caches the result set with the associated preference profiles (C). When a new query request  $q$  enters the system, Task (B) performs the similarity

measure by computing the similarity values between the new query  $q$  and the cached queries. Upon its completion, Task (B) forwards a sorted list of candidate queries to Task (D), which in turn selects a set of candidates from the list. Next, Task (F) computes  $q$ 's skyline results based on the result sets of these candidate queries. If the new query  $q$  is not answerable from the cache, the Data Restoration component accesses the whole data set (E) to perform less expensive constraint queries to restore all of the possibly missing answer points. Finally, the query processor evaluates the result based on the preference profiles of the new query to refine the final answer. Since cache space is limited, Task (G) purges the cache by preserving the most popular preferences; *i.e.*, the strategy is to eliminate queries with the least recently used preference profiles from the cache.

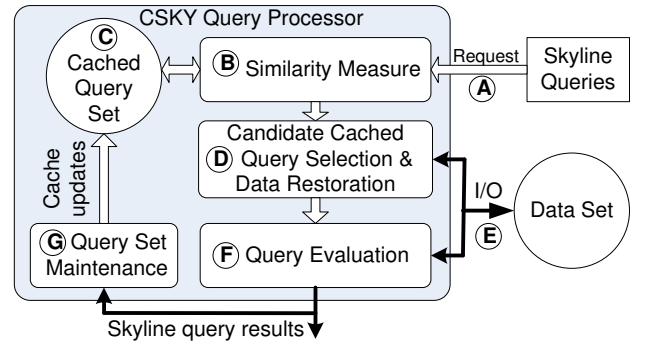


Fig. 3. C-SKY system framework.

Before we describe each task in detail we explain the main intuition behind our work. The key concept of the C-SKY algorithm involves *preference filtering*. Let  $\bar{V}_k$  be the unspecified preference nodes for a partially-ordered domain in dimension  $k$  ( $PO_k$ ), where  $\bar{V}_k \subseteq \mathbb{V}_k$  of  $g_k$  is used by a query  $q$ .  $\mathbb{V}_k$  is a node set of all the possible node values allowed in the system for the  $PO_k$  dimension.  $\mathbb{V}_k$  consists of the unspecified node set  $\bar{V}_k$  and specified node set  $V_k$  indicated in a user profile DAG. We define a data tuple set  $T(P, \bar{V})$ , where each tuple contains at least one unspecified preference node in one of the  $PO$  dimensions.  $T(P, \bar{V})$  is a potential skyline point set for  $q$ . For the data tuples with unspecified preference values, a data point  $p_i = (TO_1, \dots, TO_d, PO_1, \dots, PO_n)$  can dominate  $p_j = (TO_1, \dots, TO_d, PO_1, \dots, PO_n)$ , if  $p_i.TO_k < p_j.TO_k$  and  $p_i.PO_{k'} = p_j.PO_{k'}$ , where  $k = 1 \dots d$  and  $k' = 1 \dots n$ . The query processor directly retrieves the actual skyline points by using the traditional skyline computation without considering the user preference profiles (as the preference example  $g_2$  shows in Figures 1 (a) and (b)). The following two equations describe the key intuition in this paper.

$$P = T(P, \bar{V}) + T(P, V) \quad (1)$$

$$P' = T(P, \bar{V}) + T(D, V) \quad (2)$$

In this paper, to retrieve a complete result set for  $q$ , the query processor focuses on the data tuples  $T(P, V) = P - T(P, \bar{V})$ ,

where  $P$  is the entire data set in the system (Eqn. 1). In this study, the  $C$ - $SKY$  algorithm handles a small data set  $P' \subseteq P$ , where  $D$  is a candidate result set obtained from the results of the cached queries (Eqn. 2). Therefore, the preference filtering operation enables  $C$ - $SKY$  to (a) efficiently retrieve the skyline points with partially-ordered domains by reducing the search space to the specified data tuples only, and (b) ease the complexity of the similarity measure, which targets the specified nodes only for the similarity comparisons. The details of each task in Figure 3 are described in the following sections. Table I summarizes the symbols and functions we use throughout the following sections.

Symbol	Description
$q$	A user query with a user profile set $g = \{g_1, g_2, \dots, g_n\}$ , where $n$ is the number of the partially-ordered domains.
$P$	The entire data set in the system.
$D$	The candidate result set obtained from the results of the cached queries.
$\mathbb{V}_k = V_k + \bar{V}_k$	The maximum option nodes (or values) of the partially-ordered domain $PO_k$ .
$V_k (\bar{V}_k)$	The specified (unspecified) option nodes of the partially-ordered domain $PO_k$ .
$g_k = (V_k, E_k)$	A user preference DAG comprising a set $V_k$ of the specified option nodes with a set $E_k$ of edges between nodes for the partially-ordered dimension $k$ .
$g = \{g_1, g_2, \dots, g_n\}$	The user preference profile set for all partially-ordered domains.
$G_k$	The cached preference DAGs maintained by the system for the partially-ordered dimension $k$ .
$\hat{G}_k$	The cached preference DAGs sorted in descending order of the similarity values for the partially-ordered dimension $k$ .
$T(P, \bar{V})$	The data set with at least one of the unspecified preference nodes in $\bar{V}$
$A \rightarrow B$	A primitive preference where node $A$ is directly connected to $B$
$A \dashrightarrow B$	A transitive preference where node $A$ is indirectly connected to $B$
$A \leftrightarrow B$	A user has an equivalent preference for $A$ and $B$
$p \vdash p'$	$p$ dominates $p'$
$S(g_k, g'_k)$	Similarity function that returns a numeric number to measure the similarity between $g_k$ and $g'_k$ in the partially-ordered domain $PO_k$

TABLE I  
SYMBOLS AND FUNCTIONS FOR THE  $C$ - $SKY$  APPROACH.

### A. User Preference Profile Similarity Measure

To enable a quantitative comparison among preferences, we define a similarity function that returns the 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 value ranging from a negative value up to 1, e.g., when  $g$  and  $g'$  are identical, the similarity value equals 1.  $S(g, G)$  returns the sorted query list  $\hat{G}$  in descending order of the numeric similarity values for  $g$  with respect to all DAGs in  $G$ . In the best case, the system will find a *perfectly similar preference* DAG  $g^* \in G$ . In that

scenario, the corresponding query associated with  $g^*$  contains the complete skyline results for  $q$ . Therefore, the system does not have to search further to retrieve the final query result for  $q$ . A perfectly similar preference DAG has the following 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 will give a more formal definition below. Consider the examples in Figure 4, 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 (i.e.,  $g_1.A$  is a sibling node of  $g_1.B$ ). Therefore, there is no preference violation and, 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 can be stated as follows:

**Definition 1: (Violation):** Given  $g$  and  $g'$ , a pair of preference nodes are in violation with 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,  $\forall i, j < n$ . Additionally, if  $g.v_i \leftrightarrow g.v_j$  is true, a violation occurs when  $g'.v_i \leftrightarrow g'.v_j$  holds.

From this definition we can clearly observe that if  $g'.v_i \rightarrow g'.v_j$  ( $g'.v_i \dashrightarrow g'.v_j$ ), a *match* occurs with  $g.v_i \rightarrow g.v_j$ . A match holds when both  $g$  and  $g'$  have an identical relation from  $v_i$  to  $v_j$  (either a primitive or a transitive relation), or when one has a primitive and the other has a transitive relation. A special case of a match exists when  $g$  has a primitive or transitive relation from  $v_i$  and  $v_j$  and  $g'$  has an equivalent preference to  $v_i$  and  $v_i$ . We can restate *Definition 1* and say that a violation condition does not hold if a match is true. Since  $g'.v_j \rightarrow g'.v_i$  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. This is illustrated in the examples (a) and (b) in Figure 4, where  $g_1.B \rightarrow g_1.C$  violates  $g.B \leftrightarrow g.C$ .

**Property 2: (Inclusion):** The query result of a perfectly similar preference DAG  $g^*$  with respect to  $g$  is a minimal super query result set of query  $q$  using  $g$ .

Let  $R^*$  be the query result of  $q^*$  using  $g^*$ . Since a perfectly similar preference profile is free of violations,  $R^*$  must be an inclusive super result set of  $q$  using  $g$ . For example, in Figure 1, the skyline result of the query  $q_2$  using  $g_2$  is a super result set of the query  $q_1$  using  $g_1$ , since  $g_2$  is a user profile that is compatible with  $g_1$ . We can retrieve a complete result set for  $q_1$  from the result set of  $q_2$ . We earlier described the definition of a violation, and we now provide proof for its properties. Given  $g$  and  $g'$ , a violation occurs when a more constraint preference is used in  $g'$  compared to the

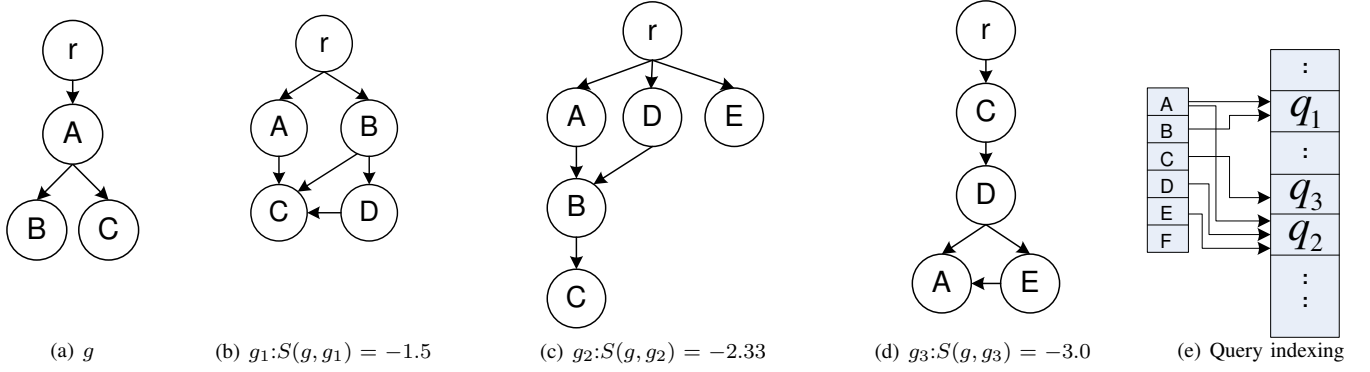


Fig. 4. Example DAGs and query indexing structure.

corresponding preference used in  $g$ . The query results of  $q'$  using  $g'$  may exclude some data points which 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, which would affect the performance of the query evaluation. An optimal perfectly similar preference DAG  $g^*$  of  $g$  must share identical preferences. However, in reality, such an optimal perfectly similar preference DAG rarely occurs, hence the system needs to choose an existing available DAG with a high degree of similarity to be as close as possible to an optimal selection. To enable quantitative measurements for the degree of similarity, we define a similarity function in the following section.

### B. Similarity Functions

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

$$S(g, g') = \frac{\sum Q_{g,g'}(v_i, v_j)}{|E_g^+|}, \quad (3)$$

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

Here  $|E_g^+|$  denotes the total number of edges of the transitive closure  $g^+$ . For all valid relations  $(v_i, v_j)$  in  $g'$ ,  $v_i$  is a specified, intermediate node in  $g'$ , and  $v_j$  is a specified node in  $g$ . Since the unspecified data tuples are processed separately in another procedure from the regular skyline computations for the specified data set, the similarity measurement ignores

the relation  $(v_i, v_j)$  if  $v_j$  is an unspecified node in  $g$ . However, if  $v_i$  is an unspecified node in  $g$ , the measurement still counts such a relation as if it was a violation with  $g$ .  $Q_{g,g'}(v_i, v_j)$  returns a similarity value 1 for a match (case (i)). A violation incurs  $-|E_g^+|$  as a penalty (case (ii)), which reduces the accumulation. The maximum similarity value is 1 in the case when there are matches for all the comparisons. For example, in Figure 4 (c),  $Q_{g,g_2}(A, C)$  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 value of  $-|E_g^+|$  to cause the current summation to become negative. Consequently, user preference profiles that are free of violations are always ranked higher than user preference profiles with violations. For example,  $g_1.D \rightarrow g_1.C$  violates  $g$  because  $C$  is a specified node in  $g$  and there is no relation among  $(g.D, g.C)$  defined in  $g$ . Therefore, the similarity value is negative if at least one violation occurs. The relation of  $(g_1.B, g_1.D)$  is ignored during the measurement process since  $D$  is not a specified node in  $g$ . Since there is one match ( $Q_{g,g_1}(A, C)$ ), and two violations ( $Q_{g,g_1}(B, C)$  and  $Q_{g,g_1}(D, C)$ ), the similarity value equals -1.5. The overall similarity algorithm is outlined in Algorithm 1.

### C. Cached Query Selection

A perfectly similar preference profile can rarely be found among the cached queries, especially as the maximum number of options allowed per user preference profile increases and the users are more likely to specify very different preference profiles. For example, if the query processor accesses the top cached query and it produces a negative score (*i.e.*, indicating a preference violations), this would imply that the system cannot retrieve a complete result set for the new query from the existing cached queries (since all of them will have equal or worse negative scores). To address this challenge, we introduce a novel approach in this study to select a minimum set of queries  $G'$ , from which the query processor can find a complete set of skyline results by combining the results of each query in  $G'$ .

*Lemma 1:* Given are a new query  $q$  and two cached queries  $q_i$  and  $q_j$ , both with the user preference profiles  $g_k$ , where

---

**Algorithm 1**  $S(g, g')$ 

---

```
1: let  $V$  and  $V'$  be the specified node sets of  $g_k$  (used by a
   new query) and  $g'_k$  (used by a cached query), respectively.
2: let  $n$  be the number of the partially-ordered domains
3: FinalScores = 0;
4: for  $k = 1$  to  $n$  do
5:   scores =  $\phi$ ;
6:   for (every  $v_i \in V'$ ) do
7:     for (every node  $v_j \in V, v_i \neq v_j$ ) do
8:       if ( $g'_k.hasEdge(v_i, v_j)$ ) then /*A valid relation in
           $g'_k$ */
9:         if ( $g_k.hasEdge(v_i, v_j)$ ) then /*A match*/
10:          scores += 1;
11:        else
12:          scores +=  $-|E_{g_k}^+|$ ;
13:        end for
14:      end for
15:    FinalScores += scores/ $|E_{g_k}^+|$ 
16:  end for
17: return FinalScores; /*return final scores*/
```

---

$k = 1 \dots n$  for all partially-ordered domains. If all preference relations in  $g_k$  of  $q$ ,  $\forall k = 1 \dots n$ , are compatible with the relations defined in the corresponding  $PO$  dimension of either  $q_i$  or  $q_j$ , the results of  $q$  are completely retrievable from the candidate result set  $D = \{q_i.result \cup q_j.result\}$ . The query list (in this case  $q_i$  and  $q_j$ ) that produces the candidate result set  $D$  is referred to as a *complementary query list*.

**Proof:** By definition. Since a complete result set for  $q$  is computed, guided by the preference relations in  $g$ , and since the cached queries  $q_i$  and  $q_j$  consist of all the relations declared in  $g$ , the result set union 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 value. 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 ( $\delta$ ) is introduced to stop the operation of continuously merging query results. Furthermore, since the top-ranked cached query  $q_{base}$  has the fewest violations with the new query, we adopt it as a *baseline query* when searching for violated relations  $\mathbb{E}$  with respect to the new query. We then select the cached queries  $q'$  for the complementary query list, if  $q'$  has compatible relations with a corresponding violated edge  $\mathbb{E}$  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:* Given is  $e = (v_i \dashrightarrow v_j)$ , a violated relation in  $\mathbb{E}$  for the partially-ordered domain  $PO_k$  with the baseline query, that is the top-ranked cached query. Let  $q'$  be a selected query for the complementary query list. Furthermore,  $g'_k$  has

no violated relations  $e' = (v_s \dashrightarrow v_j)$  ( $v_s$  is an any specified node, including  $v_i$  in  $g'_k$ ). In other words,  $g'_k$  has compatible relations with regards to  $e$  in dimension  $PO_k$ . Then 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 for which the dimensional data value in  $PO_k$  is equal to  $v_i$  and  $v_j$ , respectively. Let all the  $TO_w$  attribute values of  $P_1$ ,  $\forall w = 1 \dots d$  be better than the  $TO_w$  attribute values of  $P_2$  and the rest of  $PO_c$  attribute values of  $P_1$ ,  $\forall c = 1 \dots n, c \neq k$  be better than the  $PO_k$  attribute values of  $P_2$ . Assume that both the user profiles  $g_k$  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  $v_t \neq v_s$ . Since  $g'_k$  has compatible relations in  $PO_k$  (no violated ( $v_s \dashrightarrow v_j$ ) relations),  $P_1$  cannot dominate  $P_2$ . Furthermore, the eliminated  $P_2$  data tuples, due to the ( $v_t \dashrightarrow v_j$ ) relation for  $g'_k$ , must also be eliminated for  $g$ . Therefore, all the  $P_2$  tuples that are considered as skyline points for the new query, must be preserved in the results of  $g'_k$ . ■

To obtain a candidate result set for the new query  $q$ , we first insert the result tuples with node values defined in the user preference profiles of  $q$ . For each selected query for the complementary query list, we only combine 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 as all the result tuples in  $q'.result$  are the missing data tuples for the result of the new query. For example, in Figure 4,  $g_1$  is the baseline cached query, since it has the highest similarity value. The algorithm starts to search for the violated edges, if any exist. Profile  $g_1$  has two violated edges, ( $g_1.B \rightarrow g_1.C$ ) and ( $g_1.D \rightarrow g_1.C$ ) with respect to  $g$ . Since the result set of  $g_1$  does not contain a complete result set for  $q$ , the system searches for the next cached query. In profile  $g_2$ , the user does not specify any preference over  $C$  except for a compatible relation ( $g_2.A \rightarrow g_2.C$  matches  $g.A \rightarrow g.C$ ). The complementary query list of  $g$  must contain both  $g_1$  and  $g_2$ , since the violations ( $g_1.B \rightarrow g_1.C$  and  $g_1.D \rightarrow g_1.C$ ) do not hold in  $g_2$ . Therefore, the skyline query result can be found from the results of the corresponding queries using  $g_1$  and  $g_2$  without accessing the entire data set.

The algorithm of finding a candidate result set for  $g$  is outlined in Algorithm 2. In Line 2,  $D$  is a candidate result set. Initially, the result tuples with specified node values in  $V$  of  $g$  used by the new query are inserted into  $D$ . In Line 3,  $vioEdges$  is a container which stores the violation edges (with respect to  $g$ ) returned by the *findViolatedEdges* function. Line 3 checks the baseline query ( $g_1$ , the first query with highest similarity value in list  $G$ ) for any violated relations. Line 5 mainly checks whether the current candidate result set is larger than a threshold ( $\delta$ ) or whether  $vioEdges$  set is empty. Line 6 performs a *removeViolatedEdges* function, which deletes the violated relation(s)  $\mathbb{E}$  in  $vioEdges$ , if  $g_i$  has compatible relations with regards to  $\mathbb{E}$ . The set  $s$  contains the result of the corresponding  $q_i$  using  $g_i$ . In Lines 5–12, if  $vioEdges$  is

not empty after checking the user preference profiles in the cache,  $q$  is not an answerable query by the current selected cached queries. In this case, the query processor performs constraint queries to restore the missing data points. In Lines 8–9, by using the preference filing function, only the relevant missing tuples (with the corresponding  $PO$  attributes equal to any node value in  $N$ ) of the new query result are inserted into the candidate result set. The details of handling unanswerable queries are discussed in the next section.

---

**Algorithm 2** *FindCandidateResultSet( $G, \delta$ )*

---

```

1: let  $Q = \{q_1, q_2, \dots, q_n\}$  be the sorted cached query list and
    $G = \{g_1, g_2, \dots, g_n\}$  be the their corresponding sorted user
   profile list in ascending order of the similarity value with
   respect to  $g$ 
2: let  $D = T(q_1.result, g.V_1)$  be the initial candidate data set.
3:  $vioEdges = findViolatedEdges(g, g_1)$ 
4:  $i = 2$  /* index of the user profiles in  $G$  */
5: while ( $vioEdges \neq \phi$  AND  $D < \delta$  AND  $i < n$ ) do
6:   ( $s, \mathbb{E}$ ) =  $removeViolatedEdges(vioEdges, g_i)$ ;
7:   if ( $s$  is not empty) then
8:     let  $N$  be the node set of  $v_j$ , from the 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 ( $vioEdges, D$ )

```

---

#### D. Unanswerable Queries

A new query  $q$  is termed *unanswerable* if the selected cached queries do not contain a complete result set for  $q$ . This may occur when all the relations of the cached user preference profiles 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, C-SKY performs less expensive constraint queries to restore the missing data tuples which were eliminated because of the violated relations of the cached queries. Let *SkylineQuery* be a function that embodies the non-caching algorithm TSS [13] to evaluate a skyline query.

In Section III-C, we describe the *removeViolatedEdges* 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 unremovable violated relations (when *vioEdges* is not an empty set in the Algorithm 2, Line 13), the following operations are necessary to restore the missing data tuples which might have been eliminated by such violated relations. The first step is to create a new DAG  $g_k$  with only the violated relations  $e$  in the  $PO_k$  domain, if  $e$  originally violated  $g_k$  in the  $PO_k$  domain. We use the same user profiles defined in the selected query – which contain the violated relations  $e$  with the new query – for the rest of the  $PO$  domains, such that *SkylineQuery* can

restore only the data tuples which were eliminated through the violated relations of their own partially-ordered domain. The following lemma and proof of correctness underlie the process.

*Lemma 3:* Given is  $e = (v_i \dashrightarrow v_j)$ , an unremovable violated relation in  $\mathbb{E}$  for the partially-ordered domain  $PO_k$ . Create a new  $g = \{g_1, g_2, \dots, g_n\}$  and let  $g_k$  be one of the user profiles with only one relation  $e$ , where  $1 \leq k \leq n$ . 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  $P'_2 \subseteq P_2$  be the eliminated data tuples during the skyline evaluation. We can conclude that  $P'_2$  must contain all the missing data due to the violated relation  $e$ .

**Proof:** By definition. Since the set  $P'_2$  contains the eliminated data tuples based on the new user profile  $g^*$ , all the attributes of each data tuple in  $P'_2$  must be worse than all the attributes 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  $TO$  attribute that is not worse than the corresponding attribute of all  $P_1$  tuples. 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 exactly 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  $(v_i \rightarrow v_j)$  of a partially-ordered domain  $PO_k$ .

- Step 1: Let  $g_i$  contain relation  $(v_i \rightarrow v_i)$ ,  $g_j$  contain relation  $(v_j \rightarrow v_j)$ , and  $g_{ij}$  contain relation  $(v_i \rightarrow v_j)$  in the  $PO_k$  domain, respectively.
- Step 2: Let  $S_i = \text{SkylineQuery}(T_i, g_i)$ , where  $T_i = \text{Select ALL from dataTable where } PO_k = v_i$ .
- Step 3: Let  $S_j = \text{SkylineQuery}(T_j, g_j)$ , where  $T_j = \text{Select ALL from dataTable where } PO_k = v_j$ .
- Step 4: Let  $S_{ij} = \text{SkylineQuery}(T_{ij}, g_{ij})$ , where  $T_{ij} = S_i \cup S_j$ .
- Step 5: Return  $T_{ij} - S_{ij}$

Step 2 (and also Step 3) finds a skyline result set, which excludes the data tuples eliminated by the constraint  $v_i \rightarrow v_i$  (respectively  $v_j \rightarrow v_j$ ). Thus, if a data tuple  $p_1$  dominates  $p_2$  in all totally-ordered domains ( $p_1.TO_x < p_2.TO_x, \forall x = 1 \dots d$ ), and the partially-ordered domains ( $p_1.PO_y$  is preferable to  $p_2.PO_y, \forall y = 1 \dots n$ , except for  $k$  ( $1 \leq k \leq n$ ), we can eliminate  $p_2$ , since  $p_1.PO_k$  equals  $p_2.PO_k$ . Next, in Step 4,  $S_{ij}$  is the skyline point set retrieved from  $S_i \cup S_j$ .  $S_{ij}$  only contains data tuples where  $PD_k$  equals  $v_i$  or  $v_j$ , which are not eliminated in Steps 2 and 3. Finally, Step 5 retrieves the data tuples that are only eliminated due to the  $v_i \rightarrow v_j$  constraint. Let us continue the example in Figure 4 and consider the data set sample in Figure 1 (a). Assume that  $g_2$  and  $g_3$  do not exist in the cache. Since the only cached user preference profile has two violated relations ( $B \rightarrow C$  and  $B \rightarrow D$ ), the query processor cannot retrieve a complete skyline result for  $q$  using  $g$ . For example, for violated edge

$B \rightarrow D$ , we first perform  $SkylineQuery(T_B, B \rightarrow B)$ , where  $T_B = \{p_3, p_4, p_6\}$  and  $S_B = \{p_3, p_6\}$ . The approach performs the second  $SkylineQuery(T_D, D \rightarrow D)$  operation, where  $T_D = \{p_7, p_9\}$  and  $S_D = \{p_9\}$ .  $SkylineQuery(T_{BD}, B \rightarrow D)$  returns  $S_{BD} = \{p_3, p_6\}$ , where  $T_{BD} = S_B \cup S_D = \{p_3, p_6, p_9\}$ . Now, the missing data eliminated by the  $B \rightarrow D$  constraint is  $T_{BD} - S_{BD} = \{p_9\}$ .

Considering the data sets  $S_i$  and  $S_j$  in Steps 1 and 2, we can observe that if a cached query  $q'$  with  $v_i$  as the entry node is defined in the  $PO_k$  domain,  $T(q.result, v_i)$  must be an identical set to  $S_i$ . Because  $v_i$  as an entry node cannot be dominated by any other node, the result tuples of  $q'$  must contain all the data tuples with the attribute value equal to  $v_i$  in  $PO_k$ . Hence, the performance of the data restoration process through constraint skyline query computations is improved as compared to using the whole data set.

#### IV. CACHE MANAGEMENT AND REPLACEMENT

Naturally, the number of cached queries progressively increases after system startup until the cache is full. Since the cache space is limited, efficient cache management and replacement strategies are needed to enable long-term operations. In Section III-C, a threshold  $\delta$  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 (e.g., the time reduction of the skyline evaluation might not be significant when compared with the computation time of a non-cached skyline evaluation). Furthermore, using a threshold excludes such unhelpful queries from being cached, which 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 measurements and query selection. Accessing all the cached queries is time consuming. We provide a query indexing structure to effectively locate a set of cached queries relevant for the new query. The goal of the replacement algorithm is to keep track of usage information in order to improve the “hit rate.” For this we adopt the concept of the *least frequently used (LFU)* cache replacement algorithm. The details regarding the query indexing and replacement algorithms are described in the next Sections IV-A and IV-B.

##### A. Query Indexing

We propose a query indexing structure for the query processor to effectively locate a set of relevant cached queries. The *C-SKY* algorithm uses this set to perform similarity measurement and query selection operations to determine a candidate result set for the query processor. It is advantageous to determine a high-level similarity to the new query before performing actual similarity measurements and query selection on the cached queries, because these two operations are time-consuming. For example, a  $PO$  domain with  $\mathbb{V} = 10$  distinct nodes has at most  $2^{10} = 1024$  lattices corresponding to all possible nodes. We use the entry node(s) of a cached query  $q'$  as the key(s) and store each group (at most  $h$  groups, where  $h$  is the maximum

size of a  $PO$  domain), because if a cached query  $q'$  and the new query  $q$  have the same entry node sets, the violations are possibly fewest. Therefore, the relevance of two queries are approximately measured by the entry node(s). For example, in Figures 4 (b), (c), and (d), the keys of  $g_1$  used by  $q_1$  are  $A$  and  $B$ ; the keys of  $g_2$  used by  $q_1$  are  $A$ ,  $D$ , and  $E$ ; 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  $q_2$  as the relevant queries to  $q$ . Hence,  $p_3$  is ignored since it is less relevant to  $q$ . Figure 4 (e) shows the indexing structure. An additional table is used and each key points to the corresponding cached queries with the key as the entry node.

##### B. Cache Replacement

In *C-SKY*, we preserve the queries with the most popular user preference profiles since they are more likely to be used later. We assign each cached query a counter, which describes how often the cached query is utilized. If a cached query contributes its total (or partial) results during an execution run, we increment the counter by one. Once the cache is full, the system must discard some cached queries to free up space for newly arriving queries. We adopt a similar concept to the least recently used (*LRU*) algorithm. The least used query (or queries) are replaced by the new query. Figure 5 illustrate the cache structure. Each cached query, which is directly mapped by the corresponding indexes in the main memory, is assigned a counter. The index of a replaced query is removed from both the main memory and the cache memory.

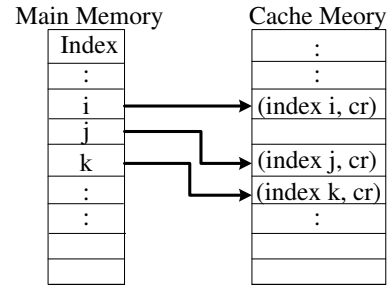


Fig. 5. Cache structure

#### V. DESCRIPTION OF THE *C-SKY* ALGORITHM

Finally, the overall algorithm of *C-SKY* 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., *FindCandidateResultSet*) to find a candidate result set for the new query.

#### VI. EXPERIMENTAL EVALUATION

We evaluated the performance of the *C-SKY* algorithm by comparing it with the state-of-the-art *TSS* approach [13],

---

**Algorithm 3** *CSKY*( $q$ )

---

```
1: let  $g = \{g_1, g_2, \dots, g_n\}$  be the user preference profiles used
   by a new query  $q$ ;
2: let  $G$  be a set of cached queries with the same entry nodes
   as  $g$  for all  $PO$  domains
3: let  $R$  be a container for the result set
4: if ( $G$  is empty) then
5:   return  $SkylineQuery(T(P, g.V), g)$ , where  $P$  is the entire
   data set and  $V$  is the specified node set of  $g$ 
6: else
7:   let  $\hat{G}$  be a sorted query 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
   value  $> 0$ ) then
9:      $R = SkylineQuery(T(q'.result, g.V), g)$ 
10:   else
11:      $(vioEdges, D) = FindCandidateResultSet(G, \delta)$ 
12:     if ( $vioEdges$  is not empty) then
13:        $R = SkylineQuery(D, g)$  /*  $q$  is answerable */
14:     else
15:       perform constraint queries to restore eliminated
       data tuples  $\mathbb{R}$  by each violated relation in  $vioEdges$ ;
16:        $R = SkylineQuery(D \cup \mathbb{R}, g)$ 
17:     end if
18:   end if
19: end if
20: return  $R$  /* the skyline points */
```

---

which handles partially-ordered domains. Unlike *C-SKY*, *TSS* consults the entire data set whenever it executes a new skyline query request. *C-SKY* adopts *TSS* as the baseline algorithm to evaluate the skyline results for partially-ordered domains and adds its own caching mechanisms. Therefore, the CPU execution 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 for indexing the data and skyline points. Specifically, we use the Spatial Index Library [7] for the R-tree index. A page size of 4 KBytes is deployed, resulting in node capacities between 78 ( $d = 6$ ) and 204 ( $d = 2$ ). The skyline result points are indexed by a main-memory R-tree to improve the performance of the dominance checks. Our data set for the a totally-ordered domain is in the range of  $[0, 1000)$  and we generated up to 100,000 normal distributed data points with dimensions in the range of 2 to 4. For a partially-ordered domains, we generate a *PO* value for each data dimension from 2 to 10, which is the maximal number of distinct options for a user preference profile 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 0.6. We set the threshold  $\delta$  as a percentage of the data set.  $\delta$  is used for the query selection operation, which avoids caching a query with a result set size larger than  $\delta$ . The cache size  $\xi$ , on the other hand, is the percentage of the maximum

result size for all the queries (equals  $\delta \times$  the number of queries). Experiments are conducted with a Pentium 3.20 GHz CPU and 1 GByte of memory. The query results are evaluated with an event-driven approach. The main measurement in the following simulations is the CPU time (for evaluating each skyline query) and the I/O cost. Our experiments use several metrics to compare these algorithms. Table II summarizes the default parameter settings used in the following simulations.

Parameter	Default	Range
Data cardinality ( $P$ )	20,000	20,000, 40,000, 60,000, 80,000, 100,000
Query cardinality ( $Q$ )	100	1 to 100
Number of <i>TO</i> domains ( $ TO $ )	2	2, 3, 4
Number of <i>PO</i> domains ( $ PO $ )	2	1, 2
<i>DAG</i> height ( $h$ )	6	2, 4, 6, 8, 10
<i>DAG</i> density ( $d$ )	0.6	-
Cache threshold ( $\delta$ )	0.8%	0.05%, 0.1%, 0.2%, 0.4%, 0.8%, 1.6%
Cache size ( $\xi$ )	3%	1%, 2%, 3%, 4%, 5%

TABLE II

SIMULATION PARAMETERS FOR THE *C-SKY* APPROACH.

#### A. Cache Threshold ( $\delta$ )

First, we measured the CPU execution time by varying the cache threshold size. The choice of a threshold size is critical for the performance of the system. If the threshold is too small, more queries with small result sets are cached. Such queries with small result sets often have intricate user preference profiles. The system might busily perform constraint skyline queries to restored missing data tuples due to a large number of violations. Furthermore, less queries would be qualified to be cached in the system, which results in less chances to utilize the cached queries with compatible user profiles. Therefore, the performance degrades. If the threshold is too high, more queries with a large result set are cached. Consequently, more cache space is occupied and the cache is more likely to be full. Hence, the system has to perform cache replacement operations more often. However, since the *C-SKY* algorithm involves preference filtering to facilitate the retrieval of 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 size of the candidate result set results in an efficient skyline evaluation time, which is the most time-consuming portion of the overall performance. Figures 6 (a), (b) and (c) show the CPU overhead and I/O cost as a function of the threshold size ranging from 0.05% to 1.6% of the entire data set. When the threshold size is set to 0.05% or below, the performance of *C-SKY* is degraded in terms of the CPU execution time and I/O cost. A threshold size  $> 0.8\%$  provides a better performance in terms of the CPU time and I/O cost. Therefore, we chose 0.8% as the default threshold size for the rest of our experiments.

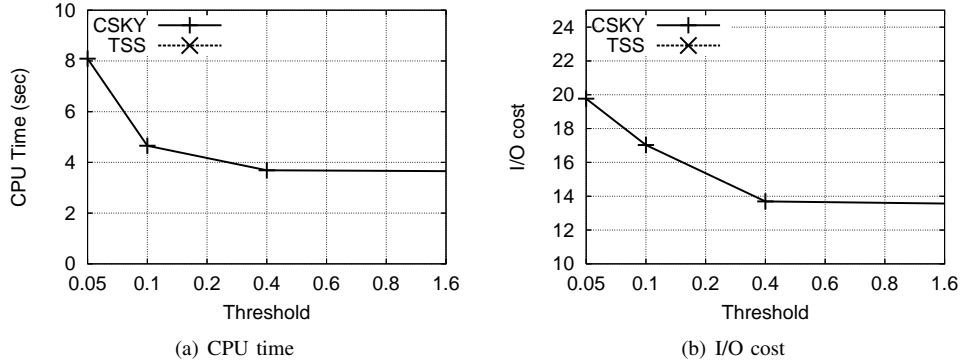


Fig. 6. Performance as a function of the cache threshold  $\delta$ .

### B. Data Cardinality

Figures 7 (a) and (b) show the CPU execution time and I/O cost as a function of the number of data points, respectively. Overall, the CPU overhead increases with the number of data points. *C-SKY* achieves a significant reduction in terms of the CPU time compared with *TSS*. This is indicative of how *C-SKY* takes advantage of the results of a set of cached queries with compatible user preference profiles. Since the *TSS* approach considers the entire data set when evaluating the skyline result for each new query, the CPU overhead is significant with a large data set, especially as a result of the R-tree constructions. In *C-SKY*, since the system only has to construct R-trees on a small candidate result set, the overall CPU time is reduced. The experimental results confirm the benefits of the *C-SKY* approach that adopts caching and therefore achieves better CPU performance and lower I/O cost than the *TSS* technique.

### C. Query Cardinality

Next, we report on the impact of the query cardinality on the performance of the two approaches. Figures 8 (a) and (b) show the CPU overhead and I/O cost versus the query cardinality as it ranges from 1 to 100, respectively. When starting the system, the CPU overheads of both approaches for evaluating the first skyline query are identical. As time progresses, the *C-SKY* system caches more queries and hence the algorithm can utilize and retrieve a candidate result set that is a subset of the entire data set. The CPU performance is improved as more relevant queries are accessed by new queries. However, as the number of queries increases (the cache is likely full), the improvement of the *C-SKY* approach slows as the system handles more cached queries and more similarity comparisons are performed. For cache management, since the cache is more likely full, replacement operations are executed more frequently. However, overall we can see that *C-SKY* still outperforms *TSS* in terms of the CPU time and I/O cost.

### D. User Profile Cardinality

In this experiment we investigate the effect of the *DAG* height associated with the *PO* domains. In Figures 9 (a) and

(b), we vary the *DAG* height from 2 to 10. Both algorithms incur an increasing CPU load and I/O cost as the *DAG* height increases. When the total number of lattice nodes of a *DAG* increases, *C-SKY* mainly suffers from higher computation costs of the similarity measurements, since the system has to check a large number of lattice nodes (or relations) for similarity comparisons. Furthermore, *t-dominance* operations are performed intensively, because the query processor might access intricate user profiles composed of more lattice nodes. Consequently, the skyline result points are often large such that the performance of the dominance checks is degraded. The performance of the *TSS* approach remains relatively stable, albeit at a worse level than *C-SKY*.

### E. Dimensionality

Next, we investigate the impact of the dimensionality on the performance of the *TSS* and *C-SKY* techniques. Figures 10 (a), (b) and (c) illustrate the CPU overhead and I/O cost versus the *PO* and *TO* dimensionality in pairs of (size of *PO*, size of *TO*), ranging from 2 to 4 for the *TO* domains and 1 to 2 for the *PO* domains, respectively. When the dimensionality increases, the performance of all methods is degraded because the R-trees fail to filter out irrelevant data entries in higher dimensions. The system possibly outputs more skyline points that in turn incur more dominance checks. From all the figures we can see that *C-SKY* outperforms *TSS* slightly when the dimensionality is high. Because the skyline result sets are often large and significant dominance checks are required, the cached queries cannot contribute much in this case.

### F. Cache Size

We investigate the effect of the cache size in Figure 11. 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 less useful queries are cached. On the other hand, if the cache size is too large, the *C-SKY* algorithm must process a large set of relevant cached queries with respect to each new query. Specifically, many similarity measurement operations need to be executed to retrieve the candidate data set. Figure 11 nicely

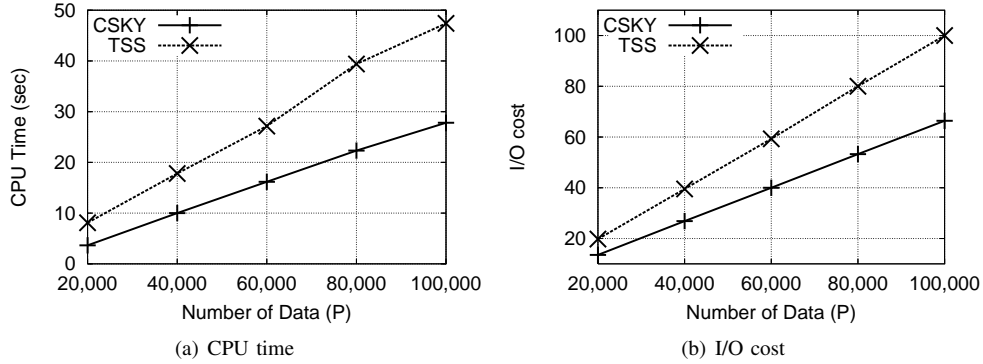


Fig. 7. Performance as a function of data cardinality.

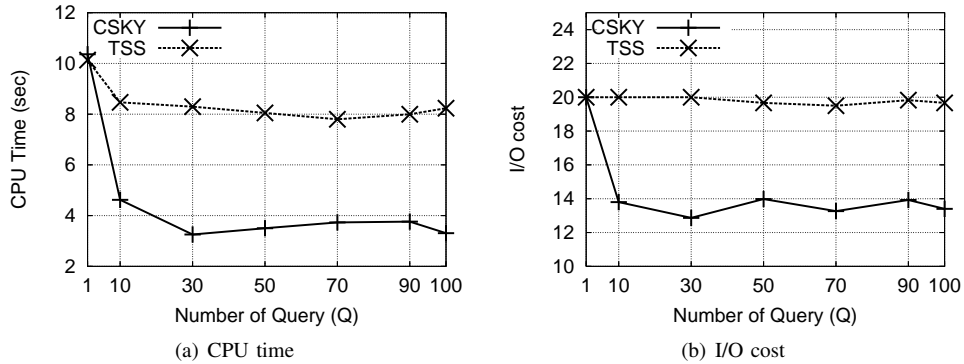


Fig. 8. Performance as a function of query cardinality.

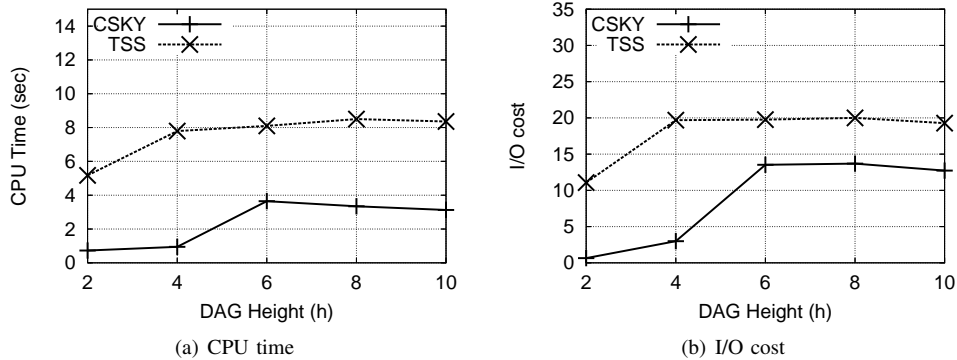


Fig. 9. Performance as a function of the DAG height.

illustrates the tradeoff as a cache size of between 3% to 4% seems to result in optimal performance.

## VII. CONCLUSIONS

We have introduced a novel approach, termed *C-SKY*, to process skyline queries with partially-ordered domains by caching the query results with their unique user preference profiles. The query response time of a new query is significantly reduced by retrieving its result from the cached result sets with compatible specifications. Our similarity measure enables the query processor to find the minimum set among the candidate

results. In case 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, to lower the space overhead, we propose a cache management scheme where only the most popular specifications are preserved. Our experimental evaluation demonstrates that *C-SKY* improves existing methods.

## VIII. ACKNOWLEDGMENTS

This research has been funded in part by NSF grant IIS-0534761, NUS AcRF grant WBS R-252-050-280-101/133 and

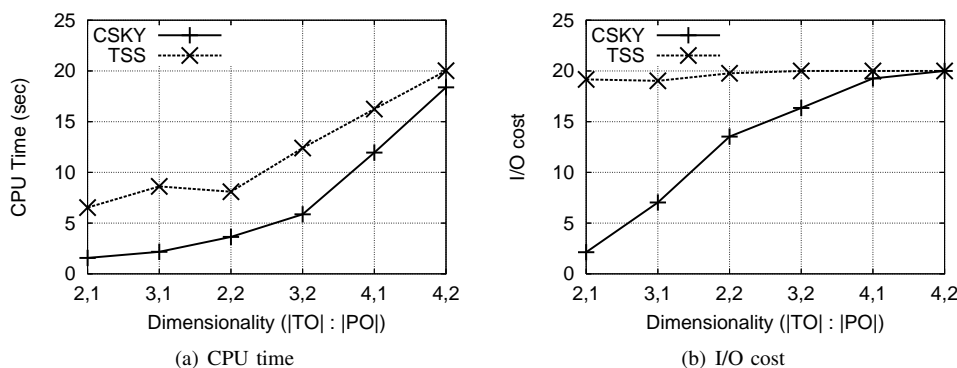


Fig. 10. Performance as a function of the dimensionality.

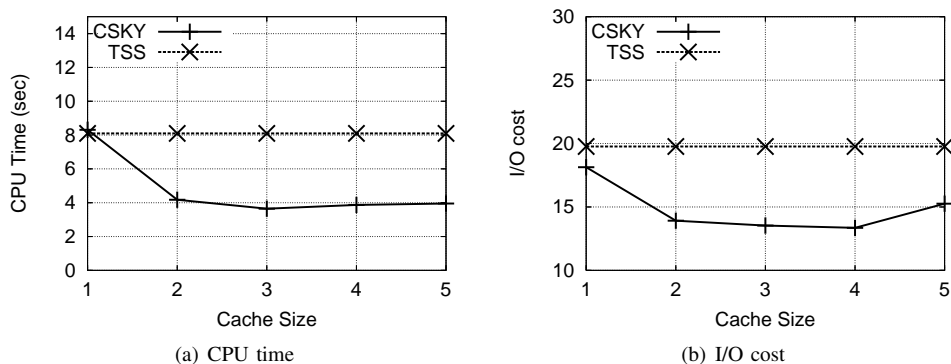


Fig. 11. Performance as a function of the DAG density.

equipment gifts from the Intel Corporation, Hewlett-Packard, Sun Microsystems and Raptor Networks Technology. We also acknowledge the support of the NUS Interactive and Digital Media Institute (IDMI).

## REFERENCES

- [1] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proceedings of the 17th International Conference on Data Engineering (ICDE), Heidelberg, Germany*, pages 421–430, 2001.
- [2] C. Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *SIGMOD Conference*, pages 203–214, 2005.
- [3] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.
- [4] Z. Huang, H. Lu, B. C. Ooi, and A. K. H. Tung. Continuous Skyline Queries for Moving Objects. *IEEE Trans. Knowl. Data Eng.*, 18(12):1645–1658, 2006.
- [5] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB), Hong Kong, China*, pages 275–286, 2002.
- [6] K. C. K. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the Skyline in Z Order. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB), University of Vienna, Austria*, pages 279–290, 2007.
- [7] S. I. Library. <http://www.research.att.com/marioh/spatialindex/index.html>.
- [8] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *Proceedings of the 21st International Conference on Data Engineering (ICDE), Tokyo, Japan*, pages 502–513, 2005.
- [9] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting Stars: The k Most Representative Skyline Operator. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE), Istanbul, Turkey*, pages 86–95, 2007.
- [10] M. D. Morse, J. M. Patel, and W. I. Grosky. Efficient Continuous Skyline Computation. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE), Atlanta, GA, USA*, page 108, 2006.
- [11] M. D. Morse, J. M. Patel, and H. V. Jagadish. Efficient Skyline Computation over Low-Cardinality Domains. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB), University of Vienna, Austria*, pages 267–278, 2007.
- [12] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 467–478, New York, NY, USA, 2003.
- [13] D. Sacharidis, S. Papadopoulos, and D. Papadias. Topologically sorted skylines for partially ordered domains. In *Proceedings of the 25th International Conference on Data Engineering (ICDE), Shanghai, China*, 2009.
- [14] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB), Seoul, Korea*, pages 751–762, 2006.
- [15] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.
- [16] L. Tian, L. Wang, P. Zou, Y. Jia, and A. Li. Continuous Monitoring of Skyline Query over Highly Dynamic Moving Objects. In *Sixth ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE), Beijing, China*, pages 59–66, 2007.
- [17] P. Wu, D. Agrawal, Ö. Egecioglu, and A. E. Abbadi. Deltasky: Optimal Maintenance of Skyline Deletions without Exclusive Dominance Region Generation. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE), The Marmara Hotel, Istanbul, Turkey*, pages 486–495, 2007.