Caching Support for Skyline Query Processing with Partially-Ordered Domains

Yu-Ling Hsueh[†] Roger Zimmermann[‡] Wei-Shinn Ku[§]
[†]Dept. of Computer Science & Information Engineering, National Chung Cheng University, Taiwan
[‡]Computer Science Department, National University of Singapore, Singapore
[§]Dept. of Computer Science and Software Engineering, Auburn University, USA
{hsueh@cs.ccu.edu.tw, 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 *CSS*, 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 the level of a relation of a new query to each of the previously cached queries and profiles. The similarity measure allows the cached entries to be effectively ordered according to descending values; 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. Extensive experiments are presented to demonstrate the performance and utility of our novel approach.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Metrics—*Spatial databases and GIS*

General Terms

Algorithms, Performance

Keywords

Skyline Query Processing over Partially-Ordered Domains, Caching techniques, Spatiotemporal databases

1. INTRODUCTION

The need of 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 intensively in the context of spatiotemporal databases is one core technique to assist such multi-criteria applications. Skyline queries have been defined as retrieving a set of points which 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 all dimensions. In many applications, some data dimensions (for example, in the form of hierarchies, intervals and preferences) are *partially-ordered* (PO).

The traditional methods to execute queries over totally-ordered domains cannot efficiently handle data sets with partially-ordered domains. Related solutions ([2, 5]) 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* [5] 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 either 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 *C̲aching S̲upport for S̲kyline Computations* (*CSS*, for short). The main contribution of *CSS* 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.

2. RELATED WORK

Numerous secondary storage based algorithms for computing skylines have been proposed before. Börzsönyi*et al.* [1] introduced the non-progressive *Block-Nested-Loop* and *Divide-*

and-Conquer 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 data set 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.* [6] 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 [3] indexes the data set 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 [4] 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.

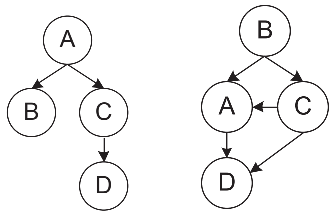
The methods proposed in [2, 7, 5, 8] are the most relevant to our work. Chan *et al.* [2] presented three algorithms for evaluating skyline queries with partially ordered attributes. Their solution is to transform each partially ordered attribute into a two-integer domain, which allows users to utilize index-based algorithms to compute skyline queries in the transformed space. However, all the techniques proposed in [2] have limited progressiveness and pruning abilities. In real applications, dynamic preferences on categorical attributes are more common than a fixed ordering for skyline query evaluation. One straightforward solution is to enumerate all possible preferences and materialize all results of the preferences; however, the costs of a full materialization are usually prohibitive. Therefore, Wong *et al.* [7] proposed a semi-materialization method named the IPO-tree Search, which stores partial useful results only. With these partial results, the outcome of each possible preference can be returned efficiently. Sacharidis *et al.* designed a topological sort-based mechanism named *Topologically-sorted Skylines* (TSS) [5] 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. Because existing methods for skyline with partially ordered domains either use stronger notions of dominance, which generate false positives, or require expensive dominance checks, Zhang *et al.* [8] introduced two methods, which do not have these drawbacks. The first mechanism employs an appropriate mapping of a partial order to a total order, inspired by the lattice theorem and an off-the-shelf skyline algorithm. In addition, the second technique utilizes an appropriate storage and indexing approach, inspired by column stores, which enables efficient verification of whether a pair of objects are incompatible. Nevertheless, all the aforementioned methods do not consider the utilization of previously cached query results to further improve the query evaluation performance.

3. CACHING SKYLINES FOR EFFICIENT SKYLINE COMPUTATIONS

Skyline query results vary with different user preferences 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. The design of the similarity measure, the method for the candidate cached query selection, and the details of handling unanswerable queries are described in the following sections.

User Preference Profile Similarity Measure

A user preference profile can be represented by a *directed acyclic graph* (*DAG*). 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*. A preference or relation is denoted by $v_i \rightarrow v_j$, which consists of two nodes (v_i and v_j) and one edge e.



(a) g (b) g'

Figure 1: DAG examples.

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. For example, in Figures 1(a) and 1(b), $g.C \rightarrow g.D$ matches $g'.C \rightarrow g'.D$; $g.A \rightarrow g.B$ violates $g'.B \rightarrow g'.A$. The similarity function $S(g, g')$ returns a real number that aggregates the matching contributions and is computed as shown in Equation 1.

$$S(g, g') = \frac{\sum Q_{g,g'}(v_i, v_j)}{|E_g^+|}, \qquad (1)$$

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

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 an intermediate node in g', and v_j is a specified node in 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 $|E_g^+|$ in the case when there are matches for all the comparisons.

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 violation), 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 have 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'.

The algorithm of finding a candidate result set for g is outlined in Algorithm 1. In Line 2, D is a candidate result set which is initialed to the result tuples of q_{base} (*i.e.*, q_1). In Line 3, *vioEdges* is a container which stores the violation edges (with respect to g) returned by the *findVioEdges* function. Line 3 checks the baseline query for any violated relations. In Line 5, if *vioEdges* is not empty, q is not an answerable query from the current selected cached queries. If this is the case and the number of the current candidate result set is less than a threshold (δ), 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 regards to $\mathbb{E}$. The set S contains the result of the corresponding q_i using g_i. In Lines 8–9, by using the preference filtering function $T(S, N)$, only the relevant missing tuples 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 1 *FindCandidateSet*(G, δ)

```
1: let $Q = \{q_1, q_2, ..., q_m\}$ be the sorted cached query list
   and $G = \{g_1, g_2, ..., g_m\}$ be the their corresponding
   sorted user profile list in ascending order of the simi-
   larity value with respect to $g$
2: let $D = q_1.result$ be the initial candidate data set.
3: $vioEdges = findVioEdges(g,g_1)$
4: $i = 2$ /* index of the user profiles in $G$ */
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 ($vioEdges$,$D$)
```

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, *CSS* 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* [5] to evaluate a skyline query. 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 $S_i = SkylineQuery(T_i, g_i)$, where T_i = Select ALL from dataTable where $PO_k = v_i$.

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

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

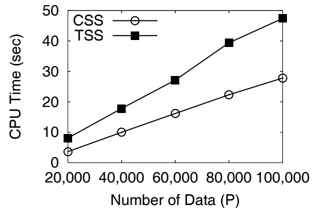
Step 5: Return $\mathbb{R} = T_{ij} - S_{ij}$

4. EXPERIMENTAL EVALUATION

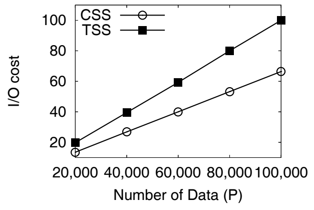
We evaluated the performance of the *CSS* algorithm by comparing it with the *TSS* approach [5], which handles partially-ordered domains. Unlike *CSS*, *TSS* consults the entire data set whenever it executes a new skyline query request. *CSS* adopts *TSS* as the baseline algorithm to evaluate the skyline 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. Our data set for 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 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.

4.1 Data Cardinality

Figures 2 (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. *CSS* achieves a significant reduction in terms of the CPU time compared with *TSS*. This is indicative of how *CSS* 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 significantly high with a large data set, especially as a result of the R-tree constructions. In *CSS*, 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 *CSS* approach that adopts caching and therefore achieves better CPU performance and lower I/O cost than the *TSS* technique.



(a) CPU time

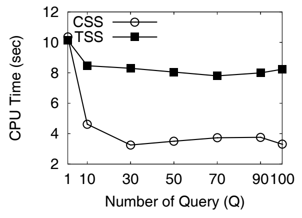


(b) I/O cost

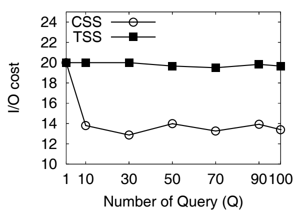
Figure 2: Performance as a function of data cardinality.

4.2 Query Cardinality

Next, we report on the impact of the query cardinality on the performance of the two approaches. Figures 3 (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 *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 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 *CSS* approach slows as the system handles more cached queries and performs more similarity comparisons. However, overall we can see that *CSS* still outperforms *TSS* in terms of the CPU time and I/O cost.



(a) CPU time

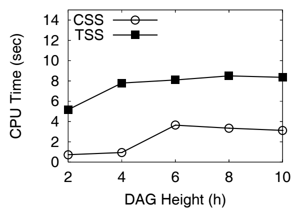


(b) I/O cost

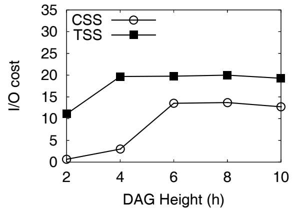
Figure 3: Performance as a function of query cardinality.

4.3 User Profile Cardinality

In this experiment we investigate the effect of the *DAG* height associated with the *PO* domains. In Figures 4 (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, *CSS* 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, 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 is degraded. The performance of the *TSS* approach remains relatively stable, albeit at a worse level than *CSS*.



(a) CPU time



(b) I/O cost

Figure 4: Performance as a function of the *DAG* height.

5. CONCLUSIONS

We have introduced a novel approach, termed *CSS*, 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. Our experimental evaluation demonstrates that *CSS* improves existing methods and is especially suited for interactive applications that require a fast response time.

Acknowledgments

This research has been funded in part by the National Science Council under the Grants NSC101-2221-E-194-054, the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and the National Science Foundation (NSF) Grants CNS-0831502 (CT) and CNS-0855251 (CRI).

References

[1] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, 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] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*, pages 275–286, 2002.
[4] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *SIGMOD Conference*, pages 467–478, 2003.
[5] D. Sacharidis, S. Papadopoulos, and D. Papadias. Topologically Sorted Skylines for Partially Ordered Domains. In *ICDE*, pages 1072–1083, 2009.
[6] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *VLDB*, pages 301–310, 2001.
[7] 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. *PVLDB*, 1(1):1032–1043, 2008.
[8] S. Zhang, N. Mamoulis, B. Kao, and D. W.-L. Cheung. Efficient Skyline Evaluation over Partially Ordered Domains. *PVLDB*, 3(1):1255–1266, 2010.