Efficient Evaluation of Multiple Preference Queries

Leong Hou U #1, Nikos Mamoulis #2, Kyriakos Mouratidis *3

#*Department of Computer Science, University of Hong Kong*
Pokfulam Road, Hong Kong
1hleongu@cs.hku.hk
2nikos@cs.hku.hk

**School of Information Systems, Singapore Management University*
80 Stamford Road, Singapore 178902
3kyriakos@smu.edu.sg

***Abstract*— Consider multiple users searching for a hotel room, based on size, cost, distance to the beach, etc. Users may have variable preferences expressed by different weights on the attributes of the searched objects. Although individual preference queries can be evaluated by selecting the object in the database with the highest aggregate score, in the case of multiple requests at the same time, a single object cannot be assigned to more than one users. The challenge is to compute a fair 1-1 matching between the queries and a subset of the objects. We model this as a stable-marriage problem and propose an efficient technique for its evaluation. Our algorithm is an iterative process, which finds at each step the query-object pair with the highest score and removes it from the problem. This is done efficiently by maintaining and matching the skyline of the remaining objects with the remaining queries at each step. An experimental evaluation with synthetic and real data confirms the effectiveness of our method.**

I. Introduction

Consider a booking system, where users search and reserve objects or services (e.g., hotel rooms), based on preference functions. Typically, different users have variable preferences expressed by different weights on the attributes of the searched objects. For a single user, the system returns the best objects with respect to his/her preference function. In this paper we study the problem where multiple preference queries are issued simultaneously (e.g., at a popular online hotel reservation site). In this case, different users may compete for the same objects. For example, a given hotel room could be the top-1 choice of many users, while it can only be assigned to one of them. As a result, the system must look for a fair 1-1 matching between the queries and a subset of the objects.

Fair 1-1 assignments can be based on the classic stable marriage problem (SMP) [1]. To compute a fair assignment between a set of preference functions F (i.e., queries) and a set of objects O, the pair (f, o) in $F \times O$ with the largest $f(o)$ value is found and established (i.e., the user corresponding to f is assigned to o). Then, f and o are removed from F and O respectively, and the process is iteratively repeated until either F or O becomes empty. This 1-1 matching model based on stable pairs has been also adopted by previous work on spatial assignment problems [2]. Accordingly, the algorithm proposed in [2] can be adapted to solve our matching problem by replacing the progressive NN search by incremental top-k search (e.g., using the method of [3]). Specifically, assuming that the set of objects O is indexed by an R-tree (using the object attributes as dimensions), we can apply an incremental top-k search for each function in F to retrieve their best (i.e., most preferred) objects. Then, the function-object pair (f, o) with the largest aggregate score is guaranteed to be stable. Although this method is a possible solution to our problem, it is expected to suffer by the large number of top-k queries performed on the complete set of objects.

We propose an alternative approach based on the observation that only objects in the skyline [4] of O need to be considered at each step of the assignment process. The skyline of O contains all objects in O, for which there does not exist an equal or better object in O with respect to all attributes. Thus, we can avoid accessing and examining objects unnecessarily by maintaining the skyline of O and iteratively matching it with the function set F. Our method includes an efficient skyline maintenance module and a fast method for identifying matching pairs between the skyline of O and F.

II. Problem Statement

We consider a set of user preference functions F over a set of multidimensional objects O. Each object $o \in O$ is represented by D feature values $o_1 \dots o_D$. Every function $f \in F$ is defined over these D values and maps object $o \in O$ to a numeric score $f(o)$. F may contain any *monotone* function; i.e., if for two objects $o, o' \in O$, $o_i \geq o'_i, \forall i \in [1, D]$, then $f(o) \geq f(o'), \forall f \in F$. For ease of presentation, however, we focus on *linear* functions; i.e., each function specifies D *weights* $f.\alpha_1 \dots f.\alpha_D$, one for each dimension. The weights are normalized, such that $\sum_{i=1}^{D} f.\alpha_i$ equals 1. This assures that no function is favored over another. Given an object $o \in O$, its score with respect to an $f \in F$ is:

$$f(o) = \sum_{i=1}^{D} f.\alpha_i \cdot o_i, \qquad (1)$$

Our goal is to find a *stable* 1-1 matching [1] between F and O, subject to the convention that function f prefers o to o' if $f(o) > f(o')$ and, symmetrically, object o prefers f to f', if $f(o) > f'(o)$.

Similar to SMP, the matching can be computed by iteratively reporting the (f, o) pair with the highest score in $F \times O$, and removing f and o from F and O, respectively. During any process that outputs matching pairs in this order, it holds:

Property 1: A function-object pair (f, o) in $F \times O$ is stable, if there is no function $f' \in F, f' \neq f, f'(o) > f(o)$ and there is no object $o' \in O, o' \neq o, f(o') > f(o)$, where F and O are the sets of the unassigned (remaining) functions and objects.

III. ALGORITHMS

In this section, we describe a naïve solution and then sketch our proposed approach. Both techniques are *progressive*, i.e., stable function-object pairs are output as soon as they are identified. We assume that F is kept in memory while O (which is typically persistent and much larger than F) is indexed by an R-tree on the disk. The main concepts of our approach, however, apply to other indexes and alternative storage configurations.

A. Brute Force Search

Our assignment problem can be solved by iterative stable pair identification and removal, according to Property 1. However, unlike finding closest pairs in the spatial version of SMP (as in [2]), identifying stable function-object pairs may require substantial effort. A brute force approach is to issue top-1 queries against O, one for every function in F. This will produce $|F|$ pairs. The pair (f, o) with the highest $f(o)$ value should be stable, because (i) o is the top-1 preference of f and (ii) $f'(o)$ cannot be greater than $f(o)$ for any function $f' \neq f$ (since (f, o) is the pair with the highest score).

This method requires numerous top-1 queries to be initiated; one for each function in F. Assuming that O is indexed by an R-tree R_O, these queries can be implemented similarly to NN queries, as shown in [3]. In addition, after the pair (f, o) with the highest $f(o)$ value is added in the query result, o must be removed from R_O, and if o was the top-1 object for some other function $f' \neq f$, top-1 search must be re-applied for f'. In the worst-case, where top-1 search must be re-applied for all remaining functions after the identification of each stable pair, this algorithm requires O($|F|$) deletions from R_O and O($|F|^2$) top-1 searches in R_O. Deletions and top-1 searches have logarithmic costs. We now describe a more efficient algorithm for this function-object assignment problem.

B. Skyline-based Search

An important observation is that, if F contains only monotone functions, then the top-1 objects of all preference functions should be in the skyline of O. Recall that the skyline O_{sky} of O is the maximum subset of O, which contains only objects that are not dominated by any other object. In other words, for any $o \in O$, if o is not in the skyline, then there exists an object o' in O_{sky}, such that any function $f \in F$ would prefer o' over o.

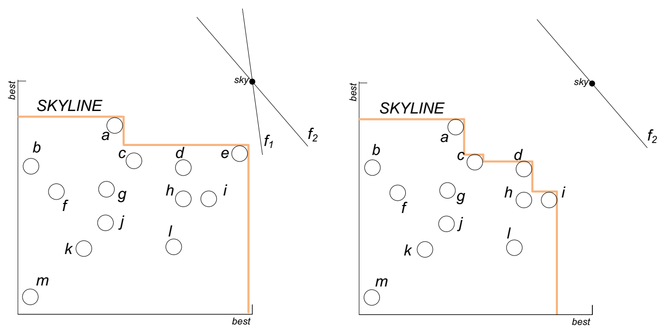
Based on this observation, we propose an algorithm, which computes and maintains the skyline O_{sky}, while stable function-object pairs between O_{sky} and F are found and reported. Algorithm 1 is a high-level pseudocode for this *skyline-based* (SB) approach. First, we compute the skyline O_{sky} of the complete set O (e.g., using the algorithm of [5]). Then, while there are unassigned functions, the function-object pair (f, o) with the highest $f(o)$ score is found, f and o are removed from F and O respectively, and O_{sky} is updated by considering $O - o$ only.

Algorithm 1 Skyline-Based Stable Assignment

```
   SB(set $F$, R-tree $R_O$)
1: $O_{sky}$:=$\emptyset$
2: while $|F| > 0$ do                          ▷ more unassigned functions
3:     if $O_{sky}$ =$\emptyset$ then
4:         $O_{sky}$:=ComputeSkyline($R_O$)
5:     else
6:         UpdateSkyline($O_{sky}, o, R_O$)      ▷ $o$ = last deleted object
7:     $(f, o)$:= BestPair($F, O_{sky}$)
8:     Output $(f, o)$
9:     $F := F - f$; $O := O - o$; $O_{sky} := O_{sky} - o$
```

We illustrate the SB algorithm using an example. In Figure 1(a), we have 2 linear preference functions (shown as lines) and 13 objects (shown as 2-dimensional points). The top-1 object of each function is the first one to be met if we sweep the corresponding line from the best possible object (top-right corner of the space) towards the worst possible (origin of the axes). In the figure, we can observe that e is the top-1 object for both functions.

SB first computes the skyline of O: $O_{sky} = \{a, e\}$. From this fact, we know that only a and e may be the top-1 objects for f_1 and f_2. Therefore, it is only necessary to compare 4 object-function pairs (instead of $13 \cdot 2 = 26$) in order to find the highest $f(o)$ score. In this example, pair (f_1, e) is the first stable pair output by the algorithm. O_{sky} is then updated to $O_{sky} = \{a, c, d, i\}$, as shown in Figure 1(b), and Lines 7-9 are repeated to identify the next highest score pair (f_2, d); this pair is reported as stable and SB terminates.



(a) Original Skyline (b) Updated Skyline

Fig. 1. Example of Skyline-Based Stable Assignment

The efficiency of SB relies on appropriate implementations of the BestPair and UpdateSkyline functions. In the next section, we propose optimized methods for these modules. In addition, we show how SB can be further enhanced to report more than one stable pairs at each loop.

IV. Implementing SB Efficiently

A. Best Pair Search

At each loop, the SB algorithm seeks for the best pair in the cross product $F \times O_{sky}$. A brute force implementation of this process is not efficient, as it requires $|F| \cdot |O_{sky}|$ comparisons. This number can be reduced by indexing either F or O_{sky}. We choose to index F, since only one deletion is performed in it at each loop (while multiple new objects may enter O_{sky} after an object deletion). This set is anti-correlated, therefore, organizing the function coefficients (i.e., preference weights) with a multidimensional index is inefficient. We propose to index the functions as sorted lists, one for each coefficient. Then, for each object in O_{sky} we can apply a *reverse* top-1 search on the lists, where the roles of objects and functions are swapped, by adapting the threshold algorithm (TA) [6]. Consider D ordered lists $L_1, L_2, \ldots, L_D$ (where D is the dimensionality), such that list L_i holds the $(f.\alpha_i, f)$ pairs of all functions $f \in F$ (where $f.\alpha_i$ is the i-th coefficient of f), sorted on $f.\alpha_i$ in descending order.

Assume that we seek the best function for an object $o \in O_{sky}$, accessing the sorted lists in a round-robin fashion. For each visited function f, we compute $f(o)$, while maintaining the function f_{best} with the highest aggregate score on o. Assume that the last values seen in the lists in sorted order are $\{l_1, l_2, ..., l_D\}$. Then, the threshold T can be calculated as $\sum_{i=1}^{D} l_i \cdot o_i$. Nevertheless, $\sum_{i=1}^{D} l_i$ could be larger than 1, which violates our assumption that the functions should be normalized (the coefficients should sum to 1). Therefore, our goal is to find a tighter threshold T_{tight} given a set of coefficients β such that $\sum_{i=i}^{D} \beta_i = 1$ and $\beta_i \leq l_i, \forall i \in [1, D]$. The threshold is calculated as $T_{tight} = \sum_{i=1}^{D} \beta_i \cdot o_i$. The set of coefficients β (and, concordantly, T_{tight}) is computed as follows.

First, we rank the dimensions in descending order based on o's corresponding values. Next, we consider each dimension i in this order. Starting with $B = 1$, we set $\beta_i = \min\{B, l_i\}$, update $B = B - \beta_i$ and proceed to the next dimension. We continue until all β_i values are set; note that if at some point B drops to 0, we directly set the remaining β_i to 0 and terminate. It can be easily seen that the T_{tight} threshold derived by the above β_i coefficients is a valid upper bound of the score for all functions that have not been encountered in any sorted list.

B. Incremental Skyline Maintenance

Apart from finding the best pair, another costly module of the SB algorithm is the maintenance of the skyline after an object in it has been assigned to a function. Re-computing the skyline from scratch is unacceptably expensive; if $|F|$ pairs are found in total, we would have to execute a skyline algorithm on R_O $|F|$ times.

As suggested in [5], incremental skyline maintenance can be achieved if we run the exact skyline R-tree traversal algorithm, but prune all entries whose MBRs are dominated by current skyline objects. This reduces the maintenance cost to accessing only the fraction of the tree that is not dominated by the current skyline. Still, this approach requires a tree traversal each time we update the skyline. As a result, some of the objects and non-leaf entries may be accessed multiple times.

In order to minimize the tree traversal cost during skyline maintenance, we keep track of the pruned entries and objects during the first run of the skyline computation algorithm of [5]. In other words, for every R-tree entry E pruned during the first run of the skyline algorithm, because E is dominated by a skyline object o, E is added in the *pruned list* $o.plist$ of o. Therefore, after the computation of the skyline, each skyline object may contain a list of entries (non-leaf entries and/or objects) that it dominates. Note that, in order to minimize the required memory, each pruned entry E is kept in the $plist$ of exactly one skyline object o (although E could be dominated by multiple skyline objects).

Skyline maintenance now operates as follows. Once a skyline object o is removed, we scan $o.plist$. For each entry E there, we check whether E is dominated by another skyline object o'; in this case, we *move* E to $o'.plist$. Otherwise, E is moved to a *skyline candidate set* S_{cand}. Note that all objects and non-leaf entries in S_{cand} are exclusively dominated by the removed skyline object o. The entries of S_{cand} are organized in a heap, based on their distance to the best corner of the search space. The algorithm of [5] is then applied, taking as input S_{cand} and the existing skyline objects.

C. Finding Multiple Pairs per Loop

At each loop, SB finds the best function in F for each object in the skyline O_{sky}. After the best object-function pair (f, o) is identified and reported, we remove o from O_{sky}, necessitating skyline maintenance. We can reduce the number of loops required (and, thus, the number of calls to the skyline maintenance module), if we output multiple stable object-function pairs at each loop.

To achieve this, we use Property 1; if for an object o the best function is f and o is the best object for function f, then (f, o) must be stable. We take advantage of this property, as follows. At each loop, let F_{best} be the subset of F that includes for every object $o \in O_{sky}$, the function $o.f_{best}$ that maximizes $f(o)$. For each $f \in F_{best}$, we record the object $f.o_{best} \in O_{sky}$ that maximizes $f(o)$. Then, we identify and report all those pairs that satisfy Property 1. Specifically, we scan F_{best} and for each f therein, we check whether $(f.o_{best}).f_{best} = f$. If so, $(f, f.o_{best})$ is a stable pair and the corresponding function/object are removed from F, O and O_{sky}. Note that at least one pair is guaranteed to be output (i.e., the pair (f, o) in $F \times O_{sky}$ with the highest $f(o)$ score). If more than one pairs are output, then multiple skyline objects are removed from O_{sky}. This does not affect the functionality of the UpdateSkyline module in Algorithm 1; all entries in the $plist$ of these objects are either placed in the $plist$ of a remaining skyline object (if dominated by it) or otherwise en-heaped and processed by the incremental skyline maintenance algorithm discussed in IV-B.

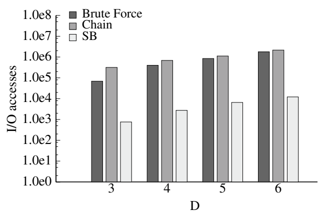
V. Experiments

In this section, we empirically evaluate the performance of our *skyline-based* (SB) algorithm, comparing it with Brute Force and Chain. Brute Force is described in Section III-A. Chain is an adaptation of [2], where the functions are indexed by a main memory R-tree (built on their weights), and the nearest neighbor module to either O or F is replaced by top-1 search in the corresponding R-tree [3]. All methods are implemented in C++ and experiments are performed on an Intel Core2Duo 2.66GHz CPU machine with 4 GBytes memory, running on Fedora 8.

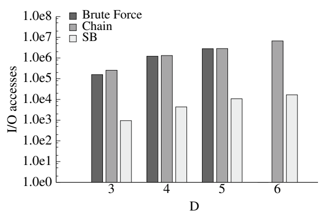
We generated two types of synthetic datasets according to the methodology in [4]. In *independent* datasets the feature values are generated uniformly and independently. In *anti-correlated* datasets, objects that are good in one dimension tend to be poor in the remaining ones. These types of data are common benchmarks for preference-based queries [4], [5]. Our dataspace contains D dimensions (in the range from 3 to 6). We also experimented with a real dataset. Zillow (www.zillow.com) is a web site with real estate information, containing 2M records with five attributes: number of bathrooms, number of bedrooms, living area, price, and lot area. Each dataset is indexed by an R-tree with 4Kbytes page size. We use an LRU memory buffer with default size 2% of the tree size. The preference functions are linear with weights generated independently.

In Figure 2, we compare the algorithms for uniform and anti-correlated synthetic object sets (O) of size 100K, matched with 5K functions, for various values of the problem dimensionality D. SB incurs 2 to 3 orders of magnitude fewer I/Os than the runner-up, i.e., Brute Force. The reason for this vast advantage of SB is the efficiency of its skyline maintenance module (UpdateSkyline), juxtaposed with the huge number of top-1 queries required by its competitors. Brute Force, on the other hand, is more efficient than Chain, as it performs fewer top-1 searches. The I/O cost increases with D for all methods, because the effectiveness of the object R-tree degrades (a fact known as the dimensionality curse). SB outperforms its competitors in terms of CPU cost too. Chain is the slowest method because it performs even more top-1 searches than Brute Force, while the efficiency of the function R-tree it uses is limited, as the functions are anti-correlated. Note that the Brute Force measurements for anti-correlated objects for $D = 6$ are missing because its space requirements exceed the available memory.

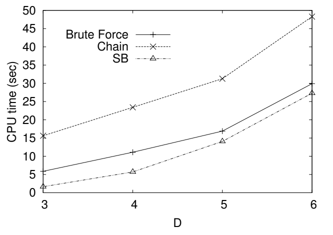
In Figures 3(a) and 3(b) we use as O random subsets of Zillow with varying cardinality from 10K to 400K, and match them with sets of 5K functions. The I/O cost results verify the superiority of SB over alternative approaches. Interestingly, the improvements in CPU time are even larger compared to the synthetic data experiments; Zillow is highly skewed and this worsens the performance of Brute Force and Chain (due to their top-1 searches), but not that of SB (due to its skyline-based nature).



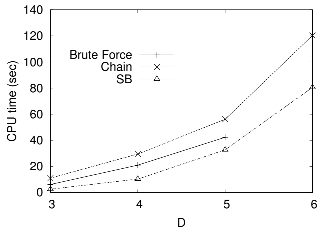
(a) I/O Cost (Independent)



(b) I/O Cost (Anti-Correlated)

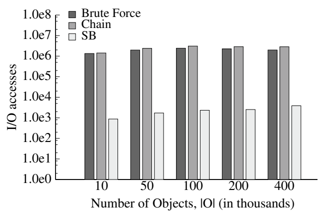


(c) CPU Time (Independent)

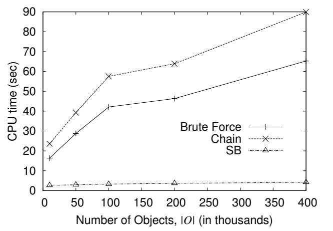


(d) CPU Time (Anti-Correlated)

Fig. 2. Effect of Dimensionality D



(a) I/O Cost (Zillow)



(b) CPU Time (Zillow)

Fig. 3. Results on a Real Dataset

VI. Conclusion

In this paper we address a stable marriage problem between a set of preference functions F and a set of objects O. The functions specify weights defining their requirements from the objects, and our task is to compute a fair 1-1 assignment between functions and objects. Our method progressively forms stable pairs drawing objects from the skyline of O. To efficiently update the skyline, we propose an incremental maintenance technique. Our solution is experimentally shown to outperform adaptations of existing approaches by orders of magnitude.

Acknowledgements

This work was supported by grant HKU 714907E from Hong Kong RGC and by the Research Center, School of Information Systems, Singapore Management University.

References

[1] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *Amer. Math.*, vol. 69, pp. 9–14, 1962.

[2] R. C.-W. Wong, Y. Tao, A. W.-C. Fu, and X. Xiao, "On efficient spatial matching," in *VLDB*, 2007, pp. 579–590.

[3] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou, "Branch-and-bound processing of ranked queries," *Information Systems*, vol. 32, no. 3, pp. 424–445, 2007.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.

[5] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 41–82, 2005.

[6] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 614–656, 2003.