8-2009

A Fair Assignment Algorithm for Multiple Preference Queries

Leong Hou U
University of Hong Kong

Nikos Mamoulis
University of Hong Kong

Kyriakos Mouratidis
Singapore Management University, kyriakos@smu.edu.sg

DOI: https://doi.org/10.14778/1687627.1687746

A Fair Assignment Algorithm for Multiple Preference Queries

Leong Hou U
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
hleongu@cs.hku.hk

Nikos Mamoulis
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
nikos@cs.hku.hk

Kyriakos Mouratidis
School of Information Systems
Singapore Management University
Singapore 178902
kyriakos@smu.edu.sg

ABSTRACT

Consider an internship assignment system, where at the end of each academic year, interested university students search and apply for available positions, based on their preferences (e.g., nature of the job, salary, office location, etc). In a variety of facility, task or position assignment contexts, users have personal 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 simultaneous requests, 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 the objects. We model this as a stable-marriage problem and propose an efficient method for its processing. Our algorithm iteratively finds stable query-object pairs and removes them from the problem. At its core lies a novel skyline maintenance technique, which we prove to be I/O optimal. We conduct an extensive experimental evaluation using real and synthetic data, which demonstrates that our approach outperforms adaptations of previous methods by several orders of magnitude.

1 Introduction

Consider a system, where users (e.g., students) search and reserve objects or services (e.g., internship positions), based on preference functions. Typically, different users have different preferences expressed by different weights on the attributes of the searched objects (e.g., nature of the job, salary, office location, etc). For a single user, the system returns a set of top-k results with respect to his/her preference function. In this paper we study the problem where multiple preference queries are issued simultaneously. In this case, different users may compete for the same objects. For example, an available internship position could be the top-1 choice of many interested students, 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 users and the objects.

As another example, consider a classroom allocation system, where at the beginning of each teaching semester (or before the exam period) the various instructors declare their classroom preferences in terms of the capacity of the room, the location, available equipment, etc. A central system is then called to make a fair assignment of classrooms to instructors according to their preferences. Instances of this problem also arise in house allocation scenarios [12, 28]. Consider, for example, that every year a government releases new public housing apartments. The attractiveness of each housing option varies from person to person. Thus, interested applicants specify their preferences and a fair 1-1 assignment needs to be made by the government. Such situations are common in countries with large government-owned estates. In China, for instance, this problem is of great significance [27]. Other applications include assignment of students to special-interest high schools or colleges, allocation of offices to employees, placement of legislators to committees, etc.

Fair 1-1 assignments can be based on the classic Stable Marriage Problem (SMP) [9, 11]. To compute a fair assignment between a set of preference functions F 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 [25, 21].

Figure 1 illustrates an example with three linear preference functions $F = \{f_1, f_2, f_3\}$ (by three users), which prioritize a set of four internship positions $O = \{a, b, c, d\}$ with respect to the offered salary (X) and the company's standing (Y). The function coefficients are normalized (i.e., they sum to 1, in order not to favor any user) and they express weights of importance on the different attributes. For example, user f_1 prefers an internship of high salary over one at a company with high standing. In a stable 1-1 matching, position (i.e., object) c is assigned to preference function f_1 since $f_1(c) = 0.68$ has the highest aggregate value among all function-object pairs. Subsequently, f_1 and c are removed from F and O respectively. Next, object b is assigned to f_2, and, finally, user f_3 takes object a.

In practice, the functions can be computed by a real system that asks the users to input their preferences over the different search attributes. Table 1 illustrates an exemplary input form. After a user has expressed his/her preferences, the system translates them to a function; the marked preferences in the example translate to the preference function $f_1 = 0.8X + 0.2Y$ of Figure 1, as Salary has weight $4/5$ and Standing has weight $1/5$.

The spatial assignment algorithms proposed in [25] and [21] can be adapted to solve this problem. Both methods work progressively, by finding a pair that is guaranteed to be in the stable assignment, reporting it, and iteratively finding the next one, until

VLDB '09, August 24-28, 2009, Lyon, France

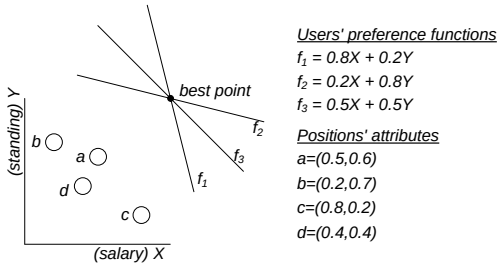


Figure 1: Internship Assignment Example

Table 1: Example of a Preference Input Form

User 1	Lowest				Highest
	1	2	3	4	5
Salary				X	
Company Standing	X				

the complete stable 1-1 assignment is established. These methods can solve our problem by incremental top-k searches, one for each function (e.g., using the algorithm of [19]). As we elaborate in more detail later, their burden is the large number of top-k queries performed on the complete set of objects.

We propose a specialized technique, which greatly outperforms adaptations of spatial assignment algorithms. Our solution is 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. In the internship assignment example (Figure 1), note that object d is not necessary to be retrieved before the assignment of object a (assuming that all preference functions are monotone). Thus, we can avoid accessing and examining objects unnecessarily by maintaining the skyline of O. As modules of our technique, we provide an efficient skyline maintenance algorithm and a fast method for identifying matching pairs between the skyline of O and F. A short outline of our methodology appears in [20]; in this paper, the search modules are optimized, there is an optimality proof for the skyline maintenance algorithm, we include and evaluate variants of the basic problem, and study the case where F is larger than O.

The rest of the paper is organized as follows. Section 2 reviews related work, and Section 3 formalizes the problem under investigation. Section 4 presents the basic idea of our approach, while Section 5 describes a number of optimizations. Section 6 studies two problem variations: assignment where objects may have capacities (e.g., a company offers 10 identical positions with the same salary) and de-normalized functions (e.g., problems where users have different priority). Section 7 empirically evaluates our solution in various settings. Section 8 concludes the paper with directions for future work.

2 Related Work

Our problem is closely related to three types of search: spatial assignment problems, skyline retrieval, and top-k queries. In this section, we review algorithms for these problems that are most related to our approach.

2.1 Spatial Assignment Problems

The Stable Marriage Problem (SMP) is a common 1-1 assignment problem between objects from two different datasets. The goal of SMP is to find a stable matching M, as in Definition 1.

DEFINITION 1. *Given two datasets A and B, a 1-1 matching M is stable if there are no two pairs (a, b) and (a', b') in M, such that a prefers b' to b, and b' prefers a to a' (where $a, a' \in A$ and $b, b' \in B$).*

[11] describes SMP and gives detailed solutions to its variants. SMP has been recently considered as a model for fair spatial assignments [21, 25]. Given two sets of spatial objects, the objective is to find a 1-1 stable matching, considering Euclidean distance as the preference function. That is, a prefers b' to b if and only if $dist(a, b') < dist(a, b)$. In [21], this spatial SMP is referred to as exclusive closest pairs (ECP) join and is treated as a variant of the closest pairs problem [7]. ECP searches for the closest pair of objects in the cross product of the two datasets. Once the closest pair is found, it is output, and the corresponding objects are removed from their respective datasets. The process is repeated until one set becomes empty. Each computed pair is proven to be stable. [21] and [25] use Property 1 to solve the problem. Assuming that pairs found to be stable are removed from the problem, at any point during the execution of the assignment algorithm it holds that:

PROPERTY 1. *A pair (a, b) is stable if and only if a's closest object is b and b's closest object is a, where a and b are among the unassigned (remaining) objects in A and B respectively.*

Based on this property, [25] proposes the Chain algorithm, which solves the problem by executing at most $3 \cdot \min\{|A|, |B|\}$ nearest neighbor queries and $2 \cdot \min\{|A|, |B|\}$ deletions, assuming that the input datasets A and B are indexed by two main-memory R-trees. Chain first picks a random object a from dataset A and finds its NN (nearest neighbor) a_{NN} in B, using B's index. Then, Chain finds the NN a' of a_{NN} in A (using A's index). If $a' \neq a$, then a_{NN} is pushed into a queue Q. Otherwise (i.e., if $a' = a$), pair (a, a_{NN}) is output as a result pair and a, a_{NN} are removed from A and B respectively (by deletions performed on the corresponding spatial indexes). The algorithm continues by de-queuing the next object x from Q (or picks a random object from A if Q is empty) and testing if x's NN in the other dataset has x as its nearest neighbor (i.e., the same test described above). Depending on the result of this test, the corresponding ECP pair is output or x's NN is pushed into Q. Eventually, Chain terminates after all ECP pairs have been identified this way.

[22] solves a related spatial assignment problem, where the objective is to minimize the average distance in the assigned pairs. This problem is significantly more complex than the stable spatial assignment described above, as algorithms that derive an exact solution have $O(n^3)$ cost, if both datasets have $O(n)$ size. The methods provided in [22] exploit spatial indexes and apply approximation techniques to prune the search space and to reduce the computation cost. Still, as shown in [21], the stable assignment achieves similar quality in terms of the average distance measure of [22], although designed for a different problem. Given this fact, and considering the lower complexity of the stable marriage problem, we choose to adapt Definition 1 to our function-object assignment problem. We stress that our solution is fundamentally different from [21, 25, 22], as (i) we compute an assignment between elements of different nature (i.e., preference functions and multidimensional feature vectors) and (ii) we rely on the computation and maintenance of a skyline, a notion that is not meaningful/applicable in the spatial assignment problems of [21, 25, 22].

2.2 Skyline Queries

Consider a set O of D-dimensional points. Point $o \in O$ is said to dominate point $o' \in O$, if for all dimensions i, $1 \leq i \leq D$, o_i (i.e., the value of o in dimension i), is greater than or equal to o'_i (the value of o' in dimension i), and the two points do not coincide.

The skyline of O consists of all points $o \in O$ that are not dominated by any other point in O.

[4] is the first work on skyline computation in secondary storage, proposing two algorithms: BNL (block nested loops) and DC (divide and conquer). BNL scans the dataset once, while keeping all non-dominated objects in the memory buffer. If during this process the buffer overflows, objects in it are flushed to a temporary file. After the scan, some objects in the buffer are guaranteed to be in the skyline and output, while the rest remain in the buffer and the process is repeated (as many times as necessary) taking the temporary file as input. DC computes local skylines in partitions of the space, and then merges these skylines recursively. LESS (linear elimination sort for skyline) [10] is an adaptation of BNL that reduces the average-case running time by topologically sorting the points before evaluation. SaLSa [3] adopts the idea of LESS to presort the input data, but uses an appropriate ordering and a termination condition that avoids scanning the complete ordered input.

Skyline computation is faster if the objects are indexed. [18, 15, 17] propose algorithms that rely on indexing. Branch and Bound Skyline (BBS) [17] assumes that O is indexed by an R-tree and computes the skyline of O by accessing the minimum number of R-tree nodes (i.e., it is I/O optimal). BBS accesses the nodes of the tree in ascending distance order from the *sky point*; that is the corner of the space with the largest attribute values in all dimensions, and corresponds to the (imaginary) most preferable object possible. Once a data object is found, it is added to the skyline and all R-tree nodes/subtrees dominated by it are directly pruned.

We illustrate the BBS algorithm in Figure 2, where O contains 13 objects indexed by the depicted R-tree. BBS executes an incremental nearest neighbor search (INN) from the sky point. The first NN is object e; the right part of the figure shows the contents of the INN search heap at the stage when e is confirmed to be NN; the R-tree nodes accessed so far are drawn with bold contour. Object e is guaranteed to be in the skyline and is placed into set O_{sky}. e dominates all objects falling in the shaded area; therefore, heap elements d, m_1, i, and c are pruned as they are de-heaped. The next NN found is a, which is also inserted into O_{sky}. Finally, heap entries M_2 and M_3 are dominated by e and a, respectively, and pruned. BBS terminates at this point (since the INN heap becomes empty), with $O_{sky} = \{e, a\}$.

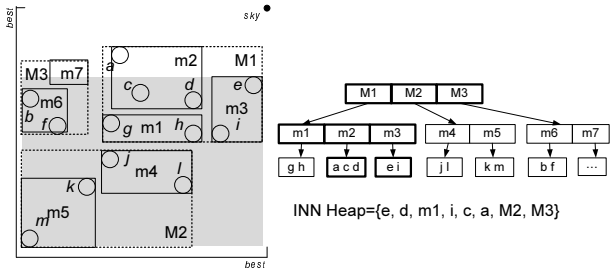
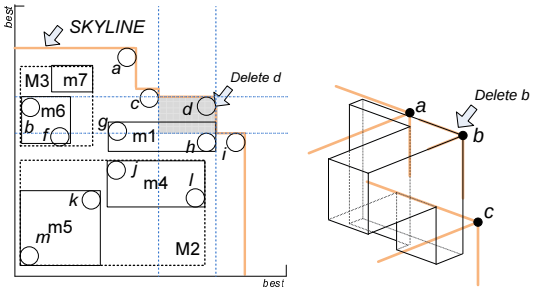


Figure 2: Example of Branch-and-Bound Skyline

For dynamic datasets, the skyline can be maintained as follows. Each inserted object o_{ins} is compared against the current skyline. If it is not dominated by any object in O_{sky}, it is included in it. If it dominates some skyline objects, they are removed from O_{sky}. Deletions are more complex to handle. Once a skyline object o_{del} is deleted, the skyline is updated by considering only objects in the region exclusively dominated by o_{del}, called *exclusive dominance region* (EDR). We illustrate this by an example in Figure 3(a). If object d is removed, the skyline is updated by inserting into O_{sky} the skyline of the EDR (the shaded region). A constrained version of BBS can be used, which accesses only the entries whose MBRs intersect the EDR (e.g., m_1). In this example, no new objects are added to the skyline, since all objects in m_1 are outside the EDR. As a result, the updated skyline is $O_{sky} = \{a, c, i\}$. Note that the EDR is not a simple hyper-rectangle when the dimensionality exceeds 2. Figure 3(b) shows a 3-dimensional EDR example; the EDR of object b is formed by several hyper-rectangles.



(a) 2-dimensional EDR (b) 3-dimensional EDR

Figure 3: Examples of Exclusive Dominance Region

BBS requires computing the EDR in response to skyline removals. [26] shows that if we use a systematic way to decompose a D-dimensional EDR into a set of hyper-rectangles, then the number of rectangles is $|O_{sky}|^D$, where D is the dimensionality. The worst case of each intersection check becomes $O(|O_{sky}|^D)$ (if the MBR entry only overlaps with the last rectangle). In view of this potentially huge number of computations, an algorithm called DeltaSky [26] is proposed, which determines the intersection between an MBR and the EDR without explicitly calculating the EDR itself. The worst case complexity of the intersection check is reduced to $O(|O_{sky}| \cdot D)$. Note that, for each deletion in O_{sky}, BBS and DeltaSky traverse the R-tree once to process an EDR-constrained query. If more than one skyline objects are removed at the same time, these methods may read the same R-tree nodes multiple times, incurring high I/O cost. As a building block of our assignment algorithm, we propose a skyline maintenance technique for situations where only deletions happen. Our method outperforms DeltaSky and is I/O optimal.

A recent work by Wong et al. [24] studies the efficient computation of skylines on datasets with dynamic preferences on nominal attributes. Nominal attributes do not have a pre-defined (i.e., objective) order, but a custom-based preference for their values can be set. The skyline for such data can be computed by an index-independent method (e.g., [3]), but index-based methods cannot be applied because it is infeasible to maintain an index for each of the (exponentially many) possible orderings. In view of this, [24] proposes a technique, which precomputes and materializes the skylines for a subset of the possible orderings of the nominal attributes. Given a skyline query with arbitrary user preferences on the nominal attributes, the technique computes the result efficiently from the materialized skylines. Computing a matching in object databases with nominal attributes is beyond the scope of our work; we only consider orderings implicitly defined by preference functions expressed by weights on the different attributes.

2.3 Top-k Search

Let O be a collection of n objects and S_1, S_2, ..., S_D be a set of D sorted lists of these objects, based on their atomic scores on different features (i.e., dimensions). Consider an aggregate function f, which takes as input an object $o \in O$, and applies to the D atomic scores o_1, o_2, ..., o_D of o. A top-k query, based on f, retrieves a k-subset O_{topk} of O ($k < n$), such that $f(o) \geq f(o')$, $\forall o \in O_{topk}, o' \in O - O_{topk}$.

Fagin et al. [8] propose a set of algorithms for top-k queries with monotone functions f, among which the threshold algorithm (TA) prevails. The main idea of TA is to pop objects from the sorted lists in a round-robin fashion. For each encountered object o, a random access is performed to retrieve all its atomic scores and compute $f(o)$. The set of the k objects with the highest aggregate scores found so far is maintained. The search terminates when the k-th score in this set becomes greater than or equal to threshold $T = f(l_1, l_2, ..., l_D)$, where l_i is the last atomic value drawn in sorted order from list S_i.

BRS (branch-and-bound ranked search) [19] is an I/O optimal top-k algorithm for datasets indexed by an R-tree. BRS visits the R-tree nodes in an order determined by the preference function f. Given an MBR M, $maxscore(M)$ is defined as the score of its best corner and is an upper bound of the score for any object inside M. BRS considers R-tree nodes in descending $maxscore$ order, and terminates when the score of the k-th best object encountered is no smaller than the $maxscore$ of the next R-tree node in this order.

Onion [5] is a precomputation-based method for top-k queries with linear aggregate functions, which relies on convex hull layers. Onion computes the convex hull of the data objects and sets it as the first layer. Then, it removes the hull objects and repeats the process to construct the next (deeper) layers, until all data are exhausted. Onion computes top-k results of an aggregate function by expanding the convex hull layers progressively, starting from the first one and moving inwards. The main problems of Onion are (i) it may expand all layers if k is large and (ii) the complexity $O(n^{D/2})$ of convex hull computations for n data objects becomes very high if the dimensionality D increases.

As in our method, skyline processing has been used in the past to facilitate top-k queries; however, our work is the first to target a matching problem. [16] studies the continuous monitoring of top-k queries over streaming multidimensional tuples in a fixed-size sliding window. It reduces the problem to a k-skyband [17] maintenance problem, considering the data attributes and their expiration times as skyline dimensions. The k-skyband contains the objects that are dominated by at most $k-1$ others. Thus, for any monotone preference function, the top-k results are contained in the k-skyband. [23] utilizes k-skyband computation techniques to evaluate top-k queries over peer-to-peer networks.

3 Problem Statement

Our model includes 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:

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

Our goal is to find a *stable* 1-1 matching between F and O. The desired matching is described by Definition 1, 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 that:

PROPERTY 2. *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.*

4 Algorithms

In this section, we describe a brute force 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) is indexed by an R-tree on the disk. The main concepts of our approach, however, apply to other indexes and alternative storage settings (discussed in Section 7).

4.1 Brute Force Search

Our assignment problem can be solved by iterative stable pair identification and removal, according to Property 2. However, unlike finding closest pairs in the spatial version of SMP (as in [21, 25]), 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 [19] and discussed in Section 2.3. In addition, after the pair (f, o) with the highest $f(o)$ value is added to 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.

The performance of the algorithm can be improved if we maintain the search heap for each top-1 query. In this case, if the top-1 object of a function f' is assigned to another function f (because $f(o) > f'(o)$), then the search for f' can *resume*. This is possible, if an incremental top-k algorithm is used (e.g., the algorithm of [19]). On the other hand, this solution requires a large amount of memory, as one priority queue must be maintained for each function. We now describe a more efficient algorithm for this function-object assignment problem.

4.2 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 [17], described in Section 2.2). Then, while there

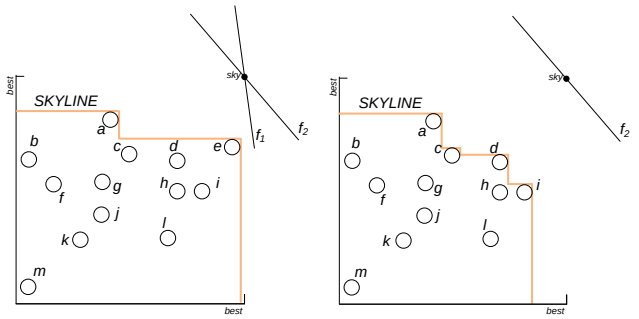
are unassigned functions, the function-object pair (f, o) with the highest $f(o)$ score is found, f and o are removed from F and O respectively, and O_{sky} is updated by considering $O - o$ only.

Algorithm 1 Skyline-Based Stable Assignment

```
   SB(set F, R-tree R_O)
1: O_sky:=∅
2: while |F| > 0 do                          ▷ more unassigned functions
3:     if O_sky =∅ 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 4(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, 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 4(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

Figure 4: 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.

5 Implementing SB Efficiently

Section 5.1 describes techniques that reduce the CPU time required to find the object-function pair with the highest score. Section 5.2 presents an efficient skyline maintenance algorithm, and proves that it is I/O optimal, i.e., that it accesses the minimum possible number of disk pages throughout the SB execution. Section 5.3 discusses how multiple stable pairs can be output in a single SB iteration, leading to earlier termination.

5.1 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}. Then, we can either (i) seek for every function in F the best object in O_{sky} after having indexed O_{sky}, or (ii) seek for every object in O_{sky} the best function in F after having indexed F.

The indexing of O_{sky} is not practical for two reasons. First, the number of updates in O_{sky} at each loop can be large, since many new objects may enter the skyline after the removal of an assigned object. Second, objects in O_{sky} are anti-correlated, so a multidimensional index for them (e.g., R-tree) is not expected to be effective.

Instead, we choose to index F, since only one deletion is performed in it at each loop. This set is also 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) [8]. 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.

Recall that TA, given a classic top-1 search problem, accesses the sorted lists and performs random accesses for the encountered objects to find their aggregate scores. If at some stage the threshold T, computed by aggregating the last values seen in each list in sorted order, is lower than the best aggregate object score found so far, then the algorithm terminates, as it is impossible for any non-encountered object to be better than the best already found.

A similar process can be applied to find the best preference function for an object $o \in O_{sky}$. Assume that we access the sorted lists in a round-robin fashion and 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 set of coefficients $\beta_i, \forall i \in [1, D]$, such that $\sum_{i=i}^{D} \beta_i = 1$ and $\beta_i \leq l_i, \forall i \in [1, D]$, which maximize the quantity $\sum_{i=1}^{D} \beta_i \cdot o_i$.

This is a *knapsack* combinatorial optimization problem. The quantity β_i of each item i to put in the knapsack is a real number in our setting, so the problem is an instance of the *fractional knapsack* combinatorial optimization problem [6], which can be solved using the following greedy algorithm.

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. The $T_{tight} = \sum_{i=1}^{D} \beta_i \cdot o_i$ threshold derived by the above β_i coefficients is a tight upper bound of the score for all functions that have not been encountered in any sorted list.

The table in Figure 5 illustrates an example of three sorted preference lists (L) for five 3-dimensional preference functions (f_a to f_e), shown on the right of the figure. Note that for each function, the sum of coefficients is 1, for example $f_a = 0.8x + 0.1y + 0.1z$. Consider object $o = (10, 6, 8)$. Assume that TA is being executed and it is accessing functions from the lists in a round-robin fashion. First, f_a is accessed from the first list; two random accesses to the other two lists retrieve the complete set of f_a's coefficients and $f_a(o) = 9.4$ is computed. Similarly, f_b and f_d are accessed from

the 2nd and 3rd lists respectively, and $f_b(o) = 6.8$, $f_d(o) = 7.8$ are computed. So far, $f_{best} = f_a$. After these three accesses, we can compute the tight threshold T_{tight} for any unvisited function as follows. We rank the last seen values at each list (i.e., l_1=0.8, l_2=0.8, and l_3=0.9) based on the values of o in the corresponding dimensions (i.e., 10, 6, and 8). Therefore, the order is l_1, l_3, l_2. We initialize $B = 1$ and assign to the first dimension in this order (i.e., dimension 1) coefficient $\beta_1 = \min\{B, l_1\} = 0.8$. Then we update $B = B - 0.8 = 0.2$. Now the second coefficient is $\beta_3 = \min\{B, l_3\} = 0.2$ and B is set to 0. Therefore, we have $\beta_1 = 0.8, \beta_2 = 0, \beta_3 = 0.2$, and $T_{tight} = \sum_{i=1}^{D} \beta_i \cdot o_i = 9.6$. Since T_{tight} is greater than $f_{best}(o)$, we continue and access the next element in the sorted lists, which is function f_c in the first list. After computing $f_c(o) = 8.2$, f_{best} is still f_a. We update T_{tight} based on the revised $l_1 = 0.5$ as $T_{tight} = 0.5{\cdot}10{+}0{\cdot}6{+}0.5{\cdot}8 = 9$, which is now smaller than $f_{best}(o) = f_a(o) = 9.4$. Therefore, there cannot be any function f with $f(o) > f_a(o)$, and TA terminates reporting (f_a, o).

L_1	L_2	L_3
f_a (0.8)	f_b (0.8)	f_d (0.9)
f_c (0.5)	f_e (0.4)	f_e (0.4)
f_e (0.2)	f_c (0.4)	f_c (0.1)
f_b (0.2)	f_d (0.1)	f_a (0.1)
f_d (0.0)	f_a (0.1)	f_b (0.0)

$f_a = 0.8X + 0.1Y + 0.1Z$
$f_b = 0.2X + 0.8Y$
$f_c = 0.5X + 0.4Y + 0.1Z$
$f_d = 0.1Y + 0.9Z$
$f_e = 0.2X + 0.4Y + 0.4Z$

Figure 5: Example of Threshold Calculation

We now discuss some techniques that further optimize the process of finding the best pair.

TA access order: First, TA can be accelerated if instead of accessing the lists in a round-robin fashion, we access the L_i with the highest $l_i \cdot o_i$ value (where l_i is the last coefficient seen in L_i). This biased list probing greedily decreases T, leading to earlier TA termination. For example, for $o = (10, 6, 8)$, the algorithm first accesses L_1 (initially $l_i = 1, \forall i \in [1, D]$) and encounters f_a (0.8). Then, $l_1 = 0.8$ and the list with the largest $l_i \cdot o_i$ value is still L_1. Therefore, f_c (0.5) is accessed. Now $T_{tight} = 0.5 \cdot 10 + 0 \cdot 6 + 0.5 \cdot 8 = 9$, and TA terminates after accessing 2 functions (instead of 4 for a round-robin order).

Resuming search: Every time we need to find the best function for a given object o, we execute TA from scratch. However, a certain object may have to seek for its best function multiple times, if its top choice is assigned to another object (e.g., recall the example above with c). In order to avoid repetitive searches for the same object, we store the state of the previous applied search for the objects in O_{sky} and resume it if necessary. Specifically, for each such object o we maintain in a heap $o.heap$ the scores $f(o)$ for all functions f that have been examined by TA for o. Like before, TA search terminates when the threshold is not greater than the best function in this queue. In the next loop, if search is required again for o (because the top function in its queue has been assigned to another object), the next function in the heap is considered as the currently best one and compared with the threshold to verify whether search has to resume. In the latter case, search in the lists continues from the previous state.

The drawback of this method is the extra memory required for the queue and for keeping the previous state of TA search for all objects in O_{sky} (this is $O(|O_{sky}| \cdot |F|)$ in the worst case). Note that each skyline object only executes a small number of top-1 searches before it forms a stable pair, and this number is much smaller than the total number of functions $|F|$. Therefore, the queue needs not store all seen functions. Motivated by this observation, we develop an *iterative* solution to avoid high memory usage.

First, the queue's maximum capacity is set to Ω, where $\Omega = \omega \cdot |F|$ and ω is a parameter (e.g., $\omega = 5\%$). This means that the queue only stores the top-Ω functions encountered during TA execution. TA proceeds in the same way as the basic resuming search. The only difference is that we have to decrease Ω by 1 when an element is popped from the queue. This is necessary to ensure correctness; the queue can only guarantee top-(Ω-m) retrieval after popped m times. When Ω reaches 0, we need to re-run TA from scratch and reset Ω to $\omega \cdot |F|$. This technique provides control over the tradeoff between execution time and memory usage via parameter ω.

5.2 Incremental Skyline Maintenance

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. In other words, every time an entry E is pruned during the first run of the skyline algorithm (because E is dominated by a skyline object o), E is added to 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). Consider, for example, the skyline in Figure 2; pruned entries d, m_1, i, c, M_2 are inserted into $e.plist$, while M_3 is included in $a.plist$.

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 (i.e., they fall in/overlap with its EDR, as defined in Section 2.2). 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 [17] is then applied, taking as input S_{cand} and the existing skyline objects.

Algorithm 2 is a pseudocode of our incremental UpdateSkyline technique. Figure 6 illustrates the algorithm. Assume that the current skyline is $\{a, e\}$ and e is assigned to a function and removed from the skyline. Suppose that $e.plist = \{d, m_1, i, c, M_2\}$. If any of these entries was dominated by the existing skyline object a, it would be moved to $a.plist$. None is, so the entire $e.plist$ is placed into S_{cand}. Then, the skyline algorithm resumes taking S_{cand} as its input heap; entries therein are examined in ascending distance order from the best point (i.e., upper-right corner of the space). That is, d is examined first, which is found to dominate entries $\{m_1, M_2\}$ in S_{cand}; these entries are added to $d.plist$. The next entries are i and c, which are skyline objects. Thus, O_{sky} is updated to $\{a, c, d, i\}$.

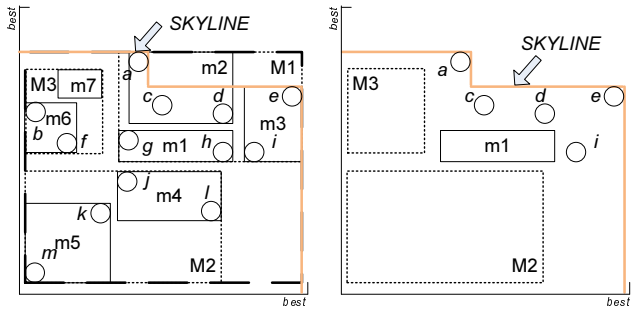
Analytical Study: The following theorem shows that UpdateSkyline (Algorithm 2) is I/O optimal, in the sense that (i) it visits only nodes that intersect the EDR of the removed object, and (ii) it does not access the same R-tree node twice during the entire stable assignment computation.

THEOREM 1. *UpdateSkyline accesses the minimal number of R-tree nodes for stable assignment computation.*

PROOF. Each time UpdateSkyline is invoked, it only accesses entries that are not dominated by any object in the current skyline. Thus, each individual skyline maintenance is performed I/O optimally. It remains to show that no node is accessed more than once during the entire stable assignment or, equivalently, that each call of UpdateSkyline does not access previously visited nodes. This

Algorithm 2 Incremental Branch and Bound Skyline

```
    S_cand := ∅
    algorithm UpdateSkyline(set O_sky, object o, R-tree R_O)
1:  S_cand:={E|E ∈ o.plist, E ∉ o'.plist, ∀o' ∈ O_sky}
2:  new O_sky:=ResumeSkyline(S_cand, O_sky)
    algorithm ResumeSkyline(set S_cand, set O_sky)
1:  push all elements of S_cand into Q
2:  while Q is not empty do
3:      de-heap top entry E of Q
4:      if E is dominated by any o ∈ O_sky then
5:          add E to o.plist
6:      else                          ▷ not dominated by any skyline object
7:          if E is non-leaf entry then
8:              visit node N pointed by E
9:              for all entries E' ∈ N do
10:                 push E' into Q
11:         else
12:             O_sky:=O_sky ∪ E
```



(a) Indexed Objects (b) Stored Elements

Figure 6: UpdateSkyline Example

can be seen easily; once a node is visited (Line 8 in Algorithm 2), it is no longer in S_{cand} nor in the $plist$ of any skyline object. □

Compared to DeltaSky [26] (i.e., the state-of-the-art skyline maintenance algorithm), UpdateSkyline performs fewer node accesses. DeltaSky accesses at most $|\triangle O_{sky}| \cdot h$ nodes, where $\triangle O_{sky}$ is the set of new skyline objects after a removal and h is the height of the R-tree [17]. This bound assumes that the BBS component of DeltaSky has to access a complete path in the tree for each deleted skyline object. UpdateSkyline performs only one complete tree traversal, because it stores the pruned entries in the $plist$ of existing skyline objects. Thus, the accessed path for each new skyline object o is only the path from the topmost MBR in the $plist$ that includes o, with length h', where $h' \leq h$. Thus, the node accesses are reduced to $|\triangle O_{sky}| \cdot h'$, indicating that UpdateSkyline never accesses more nodes than DeltaSky. As we show experimentally, this leads to a significant performance boost.

5.3 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 2; 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 2. 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; 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 Algorithm 2. The above enhanced version of SB is summarized in Algorithm 3.

Algorithm 3 Optimized Skyline-Based Stable Assignment

```
    SB(set F, R-tree R_O)
1:  O_sky:=∅; O_del := ∅
2:  while |F| > 0 do                  ▷ more unassigned functions
3:      if O_sky =∅ then
4:          O_sky:=ComputeSkyline(R_O)
5:      else
6:          UpdateSkyline(O_sky, O_del, R_O)
7:          O_del := ∅
8:      F_best:=∅
9:      for all o ∈ O_sky do
10:         find function o.f_best ∈ F that maximizes f(o)
11:         F_best:=F_best ∪ o.f_best
12:     for all f ∈ F_best do
13:         find object f.o_best ∈ O_sky that maximizes f(o)
14:     for all f ∈ F_best do
15:         if (f.o_best).f_best = f then
16:             F := F − f; O := O − f.o_best
17:             O_sky := O_sky − f.o_best; O_del := O_del ∪ f.o_best
```

6 Problem Variants

In this section we consider variations of our assignment problem and the corresponding adaptations of SB. In particular, Sections 6.1 and 6.2 consider its capacitated and prioritized versions respectively.

6.1 Objects and Functions with Capacities

So far, we have considered sets of distinct functions and objects. In practice, multiple objects may share the same features (e.g., when a company has many identical internship positions), and multiple users may have the same preferences. The algorithms proposed in this paper can be directly applied in such cases, as they do not make any assumptions about the distinctiveness of the objects or functions. Still, further optimizations are possible.

Specifically, our algorithms run faster if we replace multiple identical objects/functions by a single one, having a *capacity* value. For example, 10 identical internship positions can be replaced by a single one o with capacity 10. Similarly, multiple identical functions are replaced by a single function augmented with a capacity. Then, the problem is solved on the distinct sets which are much smaller than the original ones.

The necessary modifications to our solution regard capacity handling. In Algorithm 3, Lines 15–17 are revised. Once a stable pair (f, o) is identified (Line 15), the capacities of f and o are reduced by 1. Function f and object o are only removed from F and O respectively if their capacity reaches zero.

6.2 Functions with Different Priorities

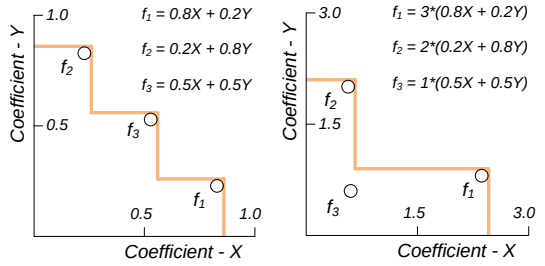
Consider a booking system, where different membership levels have different priorities. In our exemplary internship assignment system, assume that students have different priorities depending on their seniority, e.g., a third year student is preferred over a second year one

when considered for the same position. To accommodate this rule, the output of a function f applied on an object o (Equation 1) can be changed to:

$$f(o) = f.\gamma \cdot \sum_{i=1}^{D} f.\alpha_i \cdot o_i, \quad (2)$$

where $f.\gamma$ is the priority of function f (set according to its user's priority). Our solution works smoothly for this extended form of the problem by making some minor changes in the best pair searching (described in Section 5.1). First, all $f.\alpha_i$ are replaced by $f.\alpha'_i$, where $f.\alpha'_i = f.\alpha_i \cdot f.\gamma$. Then, to adapt the process in Section 5.1 for T_{tight} calculation, the initial value of B is changed to $\max_{f \in F}\{f.\gamma\}$. For example, in a 4-year undergraduate programme, where students have $\gamma = \{4, 3, 2, 1\}$ according to their year of study, B is initialized to 4. After these changes, our assignment algorithm can be directly applied. Nevertheless, SB is not expected to be as efficient as for the case where $\gamma = 1, \forall f \in F$. The reason is that B in the prioritized case leads to a threshold T_{tight} that may not be tight for some functions, thus increasing the number of TA iterations.

We can do better if a skyline F_{sky} is built on the functions, using their modified coefficients ($f.\alpha'_i$). Once the function and object skylines (F_{sky} and O_{sky}) have been computed, the best pair(s) should be between elements of these two skylines. We illustrate this technique using the example in Figure 1. First, we know that $O_{sky} = \{a, b, c\}$. If all functions have the same γ value, then F_{sky} contains all functions in F (as shown in Figure 7(a)) since they all have the same sum of coefficients. In Figure 7(b), where functions f_1, f_2, and f_3 have γ values 3, 2, and 1 respectively, F_{sky} only contains $\{f_1, f_2\}$ as there can be no object o with $f_3(o) \geq f_1(o)$. Thus, the best pair computation needs only be applied between F_{sky} and O_{sky}. Using this technique, it is faster to exhaustively search for the best function for some object than to keep the functions indexed and execute TA. The reason is that F_{sky} is relatively small, but most importantly, there are frequent updates in F_{sky} (deletions and insertions are possible, while in F there were only deletions) and maintaining the objects' TA states would be costly.



(a) Same Priority (b) Different Priorities

Figure 7: Effect of Function Priorities

7 Experiments

In this section we empirically evaluate the performance of our algorithm. We generated three types of synthetic datasets according to the methodology in [4]. In *independent* datasets the feature values are generated uniformly and independently. *Correlated* datasets contain objects whose values are close in all dimensions, i.e., if an object is good in one dimension, it is likely to be good in the remaining ones too. On the contrary, in *anti-correlated* datasets, objects that are good in one dimension tend to be poor in the remaining ones. The above three types of data are common benchmarks for preference-based queries [4, 17]. Our dataspace contains D dimensions (in the range from 3 to 6). Additionally, we experiment with two real datasets, Zillow [1] and NBA [2], described in Section 7.5. 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, except in experiments that study specifically the effect of weight distribution.

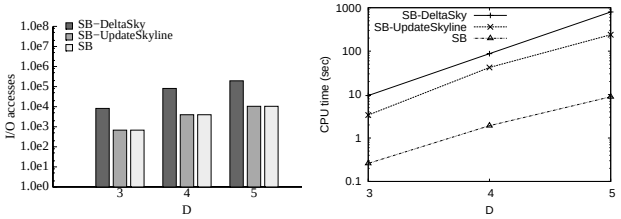
We compare our SB assignment algorithm (after tuning $\Omega = 2.5\% \cdot |F|$) against Brute Force and Chain. Brute Force is described in Section 4.1. Chain is an adaptation of [25] (presented in Section 2.1), where the functions are indexed by a main memory R-tree (built on their weights), and the nearest neighbor module to either O or F is replaced by top-1 search in the corresponding R-tree using BRS [19]. In our SB assignment algorithm, BBS [17] is used to compute the initial skyline, modified to keep track of pruned entries and objects, as described in Section 5.2. All methods were implemented in C++ and experiments were performed on an Intel Core2Duo 2.66GHz CPU machine with 4 GBytes memory, running on Fedora 8. Table 2 shows the ranges of the investigated parameters, and their default values (in bold). In each experiment, we vary a single parameter while setting the remaining ones to their default values. We evaluate the algorithms by three factors; (i) their I/O cost, (ii) their CPU cost, and (iii) the maximum memory consumed by their search structures (i.e., priority queues and pruned lists of skyline objects) during their execution. The CPU cost includes the construction cost of any main-memory indexes (i.e., indexing the function coefficients).

Table 2: Ranges of Parameter Values

Parameter	Values
Function set size, $\|F\|$ (in thousands)	1, 2.5, **5**, 10, 20
Object set size, $\|O\|$ (in thousands)	10, 50, **100**, 200, 400
Dimensionality, D	3, **4**, 5, 6
Capacity value, k	**1**, 2, 4, 8, 16
Maximum function priority, γ	**1**, 2, 4, 8, 16
Buffer size	0%, 1%, **2%**, 5%, 10%

7.1 Effectiveness of Optimizations

Before considering the Brute Force and Chain competitors, we first evaluate the effectiveness of the optimizations proposed in Section 5 within SB. We compare our fully optimized algorithm (SB) against SB-DeltaSky and SB-UpdateSkyline. SB-DeltaSky is Algorithm 1 using DeltaSky [26] for skyline maintenance. SB-UpdateSkyline is Algorithm 1 using our UpdateSkyline technique described in Section 5.2, but not the other two optimizations mentioned in Sections 5.1 and 5.3.

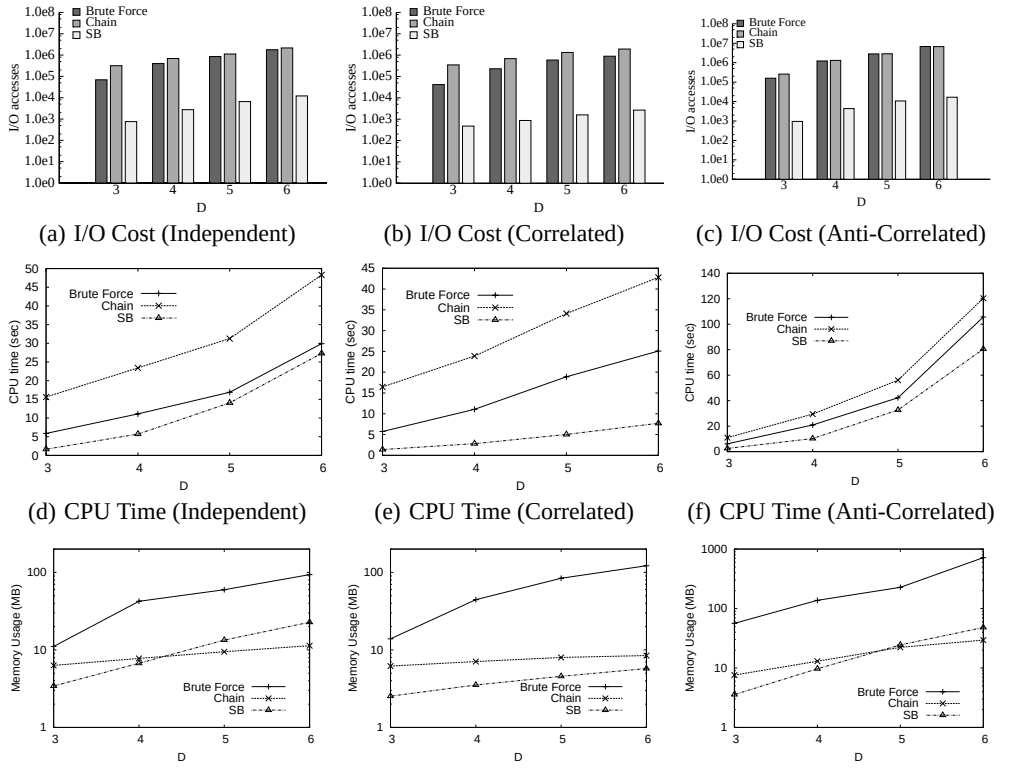


(a) I/O Cost (b) CPU Time

Figure 8: Effect of Optimization Techniques (Anti-Correlated, $|F| = 1000$)

In Figure 8 we show the I/O cost and the CPU time of the above SB variants for different dimensionality. We use anti-correlated

[1] Available at $www.zillow.com$.
[2] NBA Statistics v2.1. *http://basketballreference.com/stats_download.htm*.



(a) I/O Cost (Independent) (b) I/O Cost (Correlated) (c) I/O Cost (Anti-Correlated)

(d) CPU Time (Independent) (e) CPU Time (Correlated) (f) CPU Time (Anti-Correlated)

(g) Memory Usage (Independent) (h) Memory Usage (Correlated) (i) Mem. Usage (Anti-Correlated)

Figure 9: Effect of Dimensionality D

data; the relative performance of the algorithms is similar for independent and correlated ones. DeltaSky is too slow for high D, so we set $|F|$ to 1000 and limit the examined D range to [3..5] (the remaining parameters are set to their defaults).

SB-UpdateSkyline incurs an order of magnitude lower I/O cost than SB-DeltaSky (I/O is the dominant performance factor), verifying the efficiency of our incremental skyline approach. It is also 3 times faster in terms of CPU cost, while requiring roughly the same amount memory (SB needs at most 25% more memory than SB-DeltaSky; the worst case is for $D = 5$ and anti-correlated data).

The comparison of SB with the runner-up (SB-UpdateSkyline) confirms the effectiveness of the best pair search enhancements and of making multiple assignments per loop; there is a 13 to 27 times improvement in CPU time (Figure 8(b)). Note that the above two optimizations are targeted exclusively at reducing the CPU time, thus SB and SB-UpdateSkyline have the same I/O cost (Figure 8(a)). To summarize, the results in Figure 8 verify that the optimizations in Section 5 yield significant performance improvements.

7.2 Standard Preference Queries

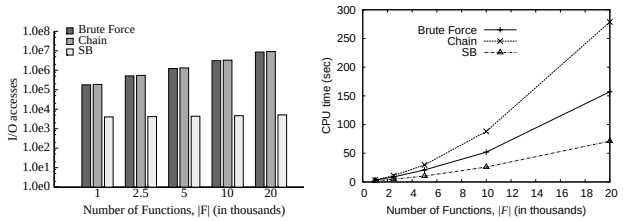
We now compare SB with Brute Force and Chain. We vary the parameters shown in Table 2 and measure the I/O cost, the CPU time, and the memory requirements of the three algorithms.

In Figure 9 we study the effect of dimensionality D, using all three types of synthetic data. Figures 9(a), 9(b), and 9(c) show the I/O cost. 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 I/O optimality 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, the reason being its resuming search feature (explained in Section 4.1). 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 [14]).

Figures 9(d), 9(e), and 9(f) show the CPU cost as a function of D. SB outperforms its competitors in this aspect too, due to their numerous top-1 searches. Chain is the slowest method because it performs even more top-1 searches than Brute Force, while the efficiency of its function R-tree is limited, as their weights sum to 1 and are thus anti-correlated.

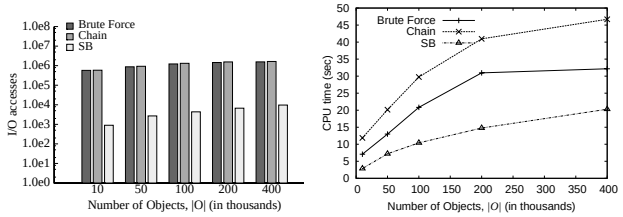
Figures 9(g), 9(h), and 9(i) plot the memory usage versus D. Brute Force consumes several times the space of the other methods, because it maintains a top-1 search heap for each $f \in F$; this is the sacrifice for its ability to resume searches. SB usually requires less memory than Chain, or slightly higher is some cases. The latter cases are in $D = 5$ and $D = 6$; O_{sky} contains more objects in high dimensions, requiring storage of many object TA states. Due to lack of space, we skip the memory usage charts for the remaining experiments; the observed trends are similar to Figures 9(g), 9(h), and 9(i). Also, we provide results for anti-correlated object sets only, as (i) they capture most real scenarios (e.g., a high quality apartment is usually expensive), and (ii) the relative performance of the methods is similar for all three types of synthetic data.

In Figure 10 we study the effect of $|F|$ (the number of functions). The costs for all methods increase with $|F|$, because more stable pairs need to be computed. However, SB scales much better. Especially its I/O cost increases only slightly (from 4030 to 5135 disk accesses for the smallest and largest F respectively), while deterioration is significant for the two competitors. The reason is the very skyline-based processing of SB. Only a few objects are fetched into O_{sky}, and most of them successfully form a stable pair with some function. Therefore, only a few accesses are performed on



(a) I/O Cost (b) CPU Time

Figure 10: Effect of Function Cardinality $|F|$ (Anti-Correlated)



(a) I/O Cost (b) CPU Time

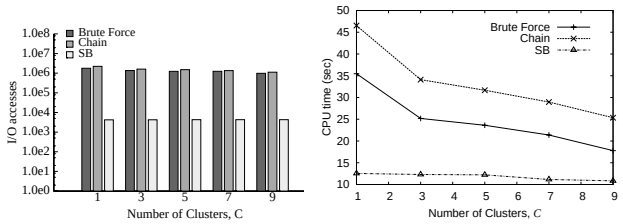
Figure 11: Effect of Object Cardinality $|O|$ (Anti-Correlated)

the object R-tree.

In Figure 11 we investigate the effect of the object cardinality $|O|$. Both the I/O cost and the CPU time increase, as top-1 and skyline searches become more costly. In all cases, SB incurs 2 orders of magnitude fewer I/Os than the runner-up (Brute Force), and its CPU time is several times smaller.

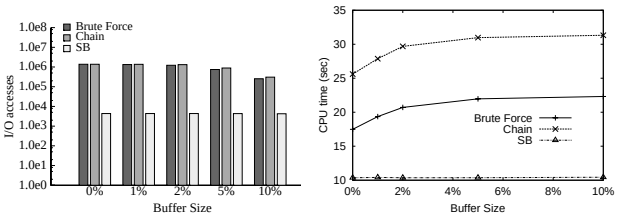
Next, we experiment with the preference weight distribution. We randomly select C independent vectors (comprising $D = 4$ weights each), and treat them as cluster centers. Each function $f \in F$ randomly chooses one of these centers, and its weights are generated by a Gaussian distribution with mean at the selected center and standard deviation equal to 0.05. In Figure 12 we plot the I/O and CPU costs of all methods as C varies between 1 and 9. In all cases, SB incurs 2 orders of magnitude fewer I/Os than its competitors, and fewer computations. The most CPU-intensive case is when $C = 1$, because F is essentially more skewed; this leads to more conflicts among different functions and, thus, longer time to compute stable pairs.

In Figure 13 we examine the effect of the buffer size, varying it from 0% to 10% of the object R-tree size. Brute Force and Chain incur fewer I/Os for a larger buffer; they access R_O nodes multiple times, and a larger buffer suppresses a higher number of these accesses. In contrast, the I/O cost of SB is stable, because of its I/O optimal skyline maintenance. Even for a 10% buffer size, SB is over 60 times more efficient than either competitor.



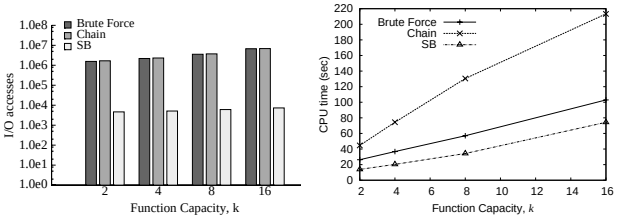
(a) I/O Cost (b) CPU Time

Figure 12: Effect of Function Distribution (Anti-Correlated)

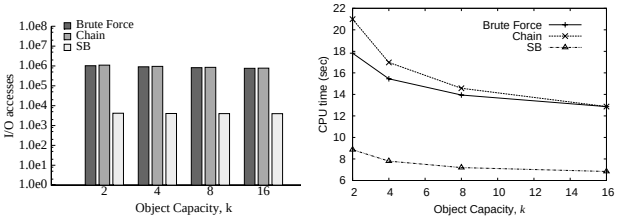


(a) I/O Cost (b) CPU Time

Figure 13: Effect of Buffer Size (Anti-Correlated)



(a) I/O Cost vs. Function Capacity (b) CPU Time vs. Function Capacity



(c) I/O Cost vs. Object Capacity (d) CPU Time vs. Object Capacity

Figure 14: Effect of Function/Object Capacity (Anti-Correlated)

7.3 Queries and Objects with Capacities

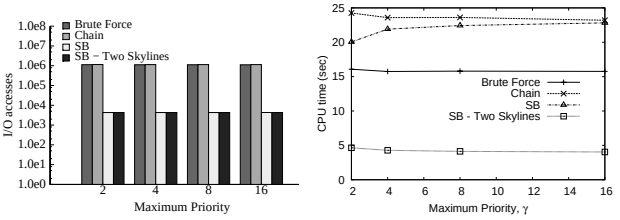
In Figures 14(a) and 14(b) we process functions with capacity k between 2 and 16, setting the remaining parameters to their defaults. When k increases, both the I/O and the CPU costs increase because more stable pairs need to be computed (i.e., $k \cdot |F|$); essentially, the problem size grows.

In Figures 14(c) and 14(d), on the other hand, we use objects with capacities. As the object capacity increases, all methods slightly improve, because fewer top-1 searches and skyline updates are performed. Specifically, if an object ranks high for multiple functions, then its multiple instances (capacity) allow its output in multiple pairs without further object search. The results in Figure 14 verify that SB outperforms its competitors for capacitated assignments too, achieving improvements of similar magnitude to the non-capacitated case.

7.4 Preference Queries with Priorities

Next, we assign priorities to the functions, randomly chosen from the range $[1..\gamma]$. In Figure 15 we study the effect of γ, while including in the charts the two-skyline version of SB, as described in Section 6.2. The I/O cost of the algorithms is practically independent of γ, and the disk accesses of the two SB versions are identical. The effectiveness of the two-skyline technique becomes clear in Figure 15(b), where its CPU time is more than 3 times shorter than any other method. The standard SB performs more computations for larger γ, because its TA threshold becomes looser, as anticipated in

Section 6.2. We note that the memory usage of the two-skyline SB is lower than all other methods; it only maintains the two skylines, skipping, for example, any book-keeping information for resuming search.



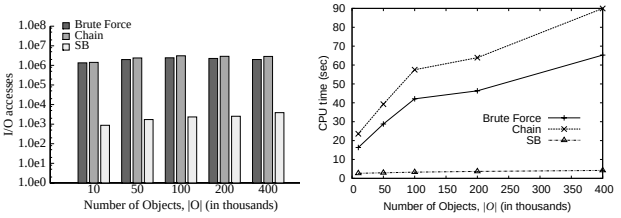
(a) I/O Cost (b) CPU Time

Figure 15: Effect of Function Priorities (Anti-Correlated)

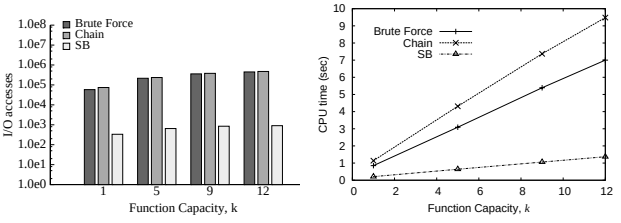
7.5 Experiments with Real Data

In addition to synthetic datasets, we experimented with two real ones: Zillow and NBA. Zillow is a website with real estate information, containing 2M records with five attributes: number of bathrooms, number of bedrooms, living area, price, and lot area. NBA includes statistics about 12278 NBA players since 1973. We selected 5 important attributes in NBA: points, rebounds, assists, steals, and blocks.

In Figures 16(a) and 16(b) we use as object sets O random subsets of Zillow with varying cardinality $|O|$ between 10K and 400K, and set the remaining parameters to their default values. The I/O cost results are similar to Figure 11, verifying the generality of SB. Interestingly, the improvements in CPU time are even larger; 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 (Zillow) (b) CPU Time (Zillow)



(c) I/O Cost (NBA) (d) CPU Time (NBA)

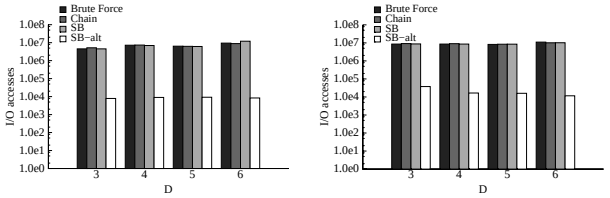
Figure 16: Results with Real Datasets

In Figures 16(c) and 16(d) we use NBA as the object set and perform a capacitated assignment. We generated $|F| = 1000$ functions with capacity $k \in \{1, 5, 9, 12\}$. The results are similar to Figures 14(a) and 14(b), with SB incurring 2 orders of magnitude fewer I/Os than Brute Force and Chain, and requiring only a fraction of their CPU time.

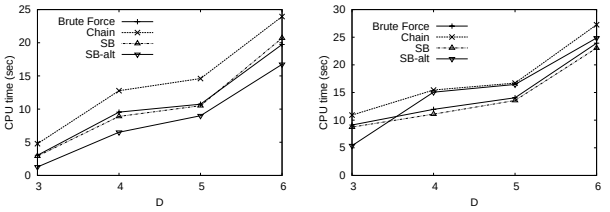
7.6 Different Storage Settings

So far, we have assumed that the set of objects O is persistent and larger than the set of preference functions F, and is thus stored on the disk. We now briefly discuss how our technique can be applied in other cases. If both sets fit in memory our algorithm can directly be applied, if we index O with the help of a main memory R-tree. Its performance gains over Brute Force and Chain can be derived from the CPU-comparison graphs in the experiments already presented.

If the set of functions F does not fit in main memory, we can still apply our method if we materialize on disk the D sorted lists holding the function coefficients. Our SB algorithm can still be applied. However, it is expected to be expensive since each object in O_{sky} executes (or resumes) an individual TA-based search; thus the lists may have to be scanned (and accessed randomly) multiple times at each iteration of the algorithm. To remedy this problem, we can execute all TA searches for the current skyline O_{sky} in batch as follows. We access the lists in a round-robin fashion — one block at a time. For each function f that we find in list L_i, we collect f's remaining coefficients by applying random access on the remaining lists $L_j, j \neq i$. Next, we compute f's aggregate score to all objects in O_{sky} and update their thresholds. If the aggregate score $f(o)$ for an object o is higher than its threshold, then we skip this object in the following iterations. This process is repeated until all objects have found their best function. This method is expected to have good I/O performance, since the number of iterations over the sorted lists is not directly dependent on the number of objects in O_{sky}; i.e., each function coefficient is accessed randomly at most once for the current skyline. Note that this technique is applicable in the case where neither F nor O fits in memory, assuming that the current skyline does. (If O_{sky} does not fit in memory, we can split the set into small enough partitions and apply the best function search in a batch manner for each partition.)



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



(c) CPU Cost (Independent) (d) CPU Time (Anti-Correlated)

Figure 17: Effect of Dimensionality D (F on disk)

Figure 17 evaluates this batch best pair search approach for disk-resident functions using synthetic datasets. We swap the cardinality of functions and objects, and set all remaining parameters to their defaults (the buffer size is now 2% of $|F|$), except from the dimensionality D, which we vary. Apart from the three techniques evaluated in the previous experiments, we also include the alternative of SB, denoted by SB-alt, which applies batch best pair search on the function coefficient lists. Note that SB-alt saves a significant number of I/O accesses. Since now O fits in memory, there are no I/O savings by SB over Brute Force and Chain, but only a computational cost advantage. Still, for independently distributed data SB-alt is even faster than SB, because SB (and Brute Force) has to maintain the heaps of individual objects in order to resume

search (note that search resumption is not applied by SB-alt, as the best functions are identified from scratch for each version of the skyline). If the data are anti-correlated, SB-alt is slower than SB because the skyline is large and the algorithm has to go deep into the sorted lists at each iteration (SB saves effort, because of its ability to resume search). Overall, SB-alt is the best choice for disk-resident functions because of its huge I/O savings, while having similar computational cost to other alternatives.

8 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. The problem arises in a variety of profile-matching applications, such as facility allocation systems and task distribution applications. Our solution is based on the observation that the stable pairs may include only objects that belong to the skyline O. When some of these objects are assigned to a function, they are removed from O and its skyline needs to be updated. To achieve this, we propose an incremental skyline maintenance technique that is proven to be I/O optimal. Additionally, we describe mechanisms that reduce the CPU time by accelerating the matching between functions and skyline objects, and identifying multiple stable pairs in each iteration of the algorithm. Moreover, we extend our algorithm to capacitated and prioritized assignments. An extensive empirical evaluation with synthetic and real datasets shows that our approach outperforms adaptations of existing methods by orders of magnitude in terms of I/O cost (typically 2 or 3), while having several times lower CPU cost.

Besides finding a stable matching or an optimal assignment [22, 2], other definitions for 1-1 fair assignments between functions and objects include the Rank-Maximal Matching [13] and the Maximum Pareto Optimal Matching [1]. A matching is rank-maximal [13] if the maximum number of functions are matched to their first-choice object, and subject to this condition, the maximum number of users are matched to their second-choice object, and so on. This problem can be solved in $O(\min\{n + C, C\sqrt{n}\} \cdot m)$ time, where $n = |F| + |O|$, $m = |F| \cdot |O|$, and C is the maximum c such that some function is assigned to its c^{th}-choice object in M. A matching M is Pareto optimal [1] if there is no other matching M' such that some function gets a better object in M' than in M, while no user gets a worse object in M'. A maximum Pareto optimal matching is a Pareto optimal matching with maximum size; the complexity of finding such an assignment is $O(m\sqrt{n})$. Note that a stable marriage matching is a Pareto optimal matching, but not vice-versa. Given the high complexity of these problems, compared to the $O(m)$ cost of stable matching, and the subjectiveness of fairness in general (i.e., there is no strong evidence that alternative definitions produce fairer assignments than the stable matching in practice), we opted to solve our problem by finding a stable matching. Nevertheless, our solution can be integrated with matching methods that rely on incremental top-k searches. Exploration of this potential is left as a subject for future work. In addition to this, we plan to study issues such as the maintenance of a fair matching in a system, where objects are dynamically allocated/freed.

Acknowledgment

Work supported by grant HKU 714907E from Hong Kong RGC.

9 References

[1] D. J. Abraham, K. Cechlárová, D. Manlove, and K. Mehlhorn. Pareto optimality in house allocation problems. In *ISAAC*, pages 1163–1175, 2005.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows : Theory, Algorithms, and Applications*. Prentice Hall, first edition, 1993.

[3] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *ACM Trans. Database Syst.*, 33(4):1–49, 2008.

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

[5] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In *SIGMOD Conference*, pages 391–402, 2000.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.

[7] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD Conference*, 2000.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

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

[10] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, pages 229–240, 2005.

[11] D. Gusfield and R. W. Irving. *The Stable Marriage Problem, Structure and Algorithms*. MIT Press, 1989.

[12] A. Hylland and R. Zeckhauser. The efficient allocation of individuals to positions. *Journal of Political Economy*, 87(2):293–314, 1979.

[13] R. W. Irving, T. Kavitha, K. Mehlhorn, D. Michail, and K. E. Paluch. Rank-maximal matchings. *ACM Transactions on Algorithms*, 2(4):602–610, 2006.

[14] F. Korn, B.-U. Pagel, and C. Faloutsos. On the 'dimensionality curse' and the 'self-similarity blessing'. *IEEE Trans. Knowl. Data Eng.*, 13(1):96–111, 2001.

[15] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[16] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD Conference*, pages 635–646, 2006.

[17] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[18] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.

[19] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-bound processing of ranked queries. *Information Systems*, 32(3):424–445, 2007.

[20] L. H. U, N. Mamoulis, and K. Mouratidis. Efficient evaluation of multiple preference queries. In *ICDE*, pages 1251–1254, 2009.

[21] L. H. U, N. Mamoulis, and M. L. Yiu. Computation and Monitoring of Exclusive Closest Pairs. *IEEE Trans. Knowl. Data Eng.*, to appear.

[22] L. H. U, M. L. Yiu, K. Mouratidis, and N. Mamoulis. Capacity constrained assignment in spatial databases. In *SIGMOD Conference*, pages 15–28, 2008.

[23] A. Vlachou, C. Doulkeridis, K. Nørvåg, and M. Vazirgiannis. Skyline-based peer-to-peer top-k query processing. In *ICDE*, pages 1421–1423, 2008.

[24] 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.

[25] R. C.-W. Wong, Y. Tao, A. W.-C. Fu, and X. Xiao. On efficient spatial matching. In *VLDB*, pages 579–590, 2007.

[26] P. Wu, D. Agrawal, Ö. Egecioglu, and A. E. Abbadi. Deltasky: Optimal maintenance of skyline deletions without exclusive dominance region generation. In *ICDE*, pages 486–495, 2007.

[27] Y. Yuan. Residence exchange wanted: A stable residence exchange problem. *European Journal of Operational Research*, 90(3):536–546, May 1996.

[28] L. Zhou. On a conjecture by gale about one-sided matching problems. *Journal of Economic Theory*, 52(1):123–135, 1990.