# A Unified Approach for Computing Top-k Pairs in Multidimensional Space


Muhammad Aamir Cheema[†], Xuemin Lin[†], Haixun Wang[‡], Jianmin Wang[*], Wenjie Zhang[†]

[†]*University of New South Wales, Australia*
*{macheema,lxue,zhangw}@cse.unsw.edu.au*

[‡]*Microsoft Research Asia*
*haixunw@microsoft.com*

[*]*School of Software, Tsinghua University*
*Tsinghua National Laboratory for Information Science and Technology, China*
*jimwang@tsinghua.edu.cn*



***Abstract*— Top-$k$ pairs queries have many real applications. $k$ closest pairs queries, $k$ furthest pairs queries and their bichromatic variants are some of the examples of the top-$k$ pairs queries that rank the pairs on distance functions. While these queries have received significant research attention, there does not exist a unified approach that can efficiently answer all these queries. Moreover, there is no existing work that supports top-$k$ pairs queries based on generic scoring functions. In this paper, we present a unified approach that supports a broad class of top-$k$ pairs queries including the queries mentioned above. Our proposed approach allows the users to define a *local* scoring function for each attribute involved in the query and a *global* scoring function that computes the final score of each pair by combining its scores on different attributes. We propose efficient internal and external memory algorithms and our theoretical analysis shows that the expected performance of the algorithms is optimal when two or less attributes are involved. Our approach does not require any pre-built indexes, is easy to implement and has low memory requirement. We conduct extensive experiments to demonstrate the efficiency of our proposed approach.**


## I. Introduction

Given a set of objects $\{o_1, \cdots, o_N\}$ and a ranking function that returns the score of a pair of objects $(o_u, o_v)$, a top-$k$ pairs query returns $k$ pairs with the best scores. An important and well studied special case of the top-$k$ pairs query is the $k$ closest pairs query which returns $k$ pairs with the smallest distances. The $k$ closest pairs queries have been extensively studied in the context of computational geometry (see [1] and references therein).

The database community has also conducted significant research on the $k$ closest (or most similar) pairs queries, $k$ furthest (or most dissimilar) pairs queries and their variants [2], [3], [4], [5]. However, all the existing techniques are developed to solve some specific problems and there does not exist a unified approach that answers different variants of the top-$k$ pairs queries (e.g., different $L_p$ distances). Another interesting variation for which no efficient solution exists is to find the pairs of the objects that are similar to each other in one subspace and dissimilar in another subspace. We are the first to provide a unified framework that supports a broad class of top-$k$ pairs queries including the above mentioned queries.

We present a unified approach to efficiently answer the top-$k$ pairs queries based on generic scoring functions which are not supported by the existing work. Consider a simple example of an insurance company. The manager might want to retrieve two insurance agents who sell very similar amount of policies (i.e., the total premium of their sold policies is similar) but receive very different salaries. Suppose that the relevant information is stored in a table named `agent`. The manager may issue the following query to retrieve the top-$k$ pairs of agents.

```
Q1: select a.id, b.id from agent a, agent b
where a.id < b.id
order by
|a.sold - b.sold| - |a.salary - b.salary|
limit k
```

Here $|x - y|$ denotes the absolute difference of $x$ and $y$. Note that the `order by` clause prefers the pair of agents with larger difference in their salaries and smaller difference in the amount of the policies they sold[1]. The condition $a.id < b.id$ is used to avoid the pair $(a, b)$ being repeated as $(b, a)$.

While the example shows a simple ranking criteria, in the real applications, the users may define more sophisticated scoring functions. Our framework allows the users to define a different scoring function for each attribute involved in the query. Such scoring functions are called *local* scoring functions. The users define a *global* scoring function that computes the final score of a pair by combining its scores on all attributes.

Our framework supports any global scoring function that is *monotonic* and any local scoring function that is *loose monotonic*. A wide range of functions that are used in many real applications are monotonic. Although we define monotonic and loose monotonic scoring functions in Section II-A, we remark here that the loose monotonic functions are more general than the monotonic functions. In the above example, the two local scoring functions are $|a.sold - b.sold|$ and $-|a.salary - b.salary|$, respectively. The global scoring function is the sum of the local scores.

Our framework does not fix the number of attributes involved in the query. In other words, the users can issue a top-$k$ pairs query on any subset of the attributes using a different loose monotonic scoring function for each attribute. This enables us to support many interesting queries (e.g., similarity in one subspace and dissimilarity in another).

[1] Without loss of generality, throughout this paper, we assume that the top-$k$ pairs queries retrieve $k$ pairs with the smallest final scores.

We further generalize the supported top-$k$ pairs queries by classifying them into *chromatic* and *non-chromatic* top-$k$ pairs queries. The *chromatic* queries are further classified into *homochromatic* and *heterochromatic* top-$k$ pairs queries. Suppose that each object in the database has been assigned a color. A homochromatic top-$k$ pairs query returns the top-$k$ pairs among the pairs that contain two objects having the same color. On the other hand, a heterochromatic top-$k$ pairs query considers only the pairs that contain two objects having different colors. A top-$k$ pairs query that does not consider the colors of the objects (i.e., all pairs are considered) is called a non-chromatic top-$k$ pairs query.

In the query Q1, the user may want to consider only the pairs of agents who work under different managers. The user may issue a heterochromatic top-$k$ pairs query by adding the condition `a.manager` $\neq$ `b.manager` in the `where` clause of the query. Note that the heterochromatic queries are more general than the *bichromatic* queries. The bichromatic queries assume that some of the objects are assigned blue color and others are assigned red color. Only the pairs that contain one red object and one blue object are considered. Existing work on $k$ closest pairs queries [2], [3] solve bichromatic queries and the extension to heterochromatic queries is either non-trivial or inefficient.

We further generalize the problem by defining two new types of queries called *skyline pairs* query and *rank-based top-k pairs* query. These queries are useful for the users who may not be able to define a suitable global scoring function (e.g., due to the lack of domain knowledge). For details, we refer the readers to Section V.

Below, we summarize our contributions.

- We are first to provide a unified and efficient approach for a broad class of top-$k$ pairs queries. Our framework does not require any pre-built data structure, has low memory requirement and is easy to implement.
- We theoretically analyse the performance of the proposed algorithms and show that the expected performance is optimal when the number of attributes involved is two or less[2].
- Our extensive experiments demonstrate a significant improvement over the existing best known solution for k closest pairs query. For the more general top-$k$ pairs queries, we compare our algorithm with a naïve algorithm and observe up to three orders of magnitude improvement.
- Due to the generality of the framework, it can support several other interesting queries (e.g., *skyline pairs* and *rank-based* top-$k$ pairs queries). In Section V, we present efficient solutions for these queries and provide a detailed theoretical analysis. We also present experiments to evaluate the efficiency of these algorithms.

[2]When $d$ attributes are involved, the expected time complexity is $O(dV^{\frac{d-1}{d}}k^{\frac{1}{d}}Log\ N)$ and expected IO cost is $O(\frac{d}{B}V^{\frac{d-1}{d}}k^{\frac{1}{d}}(Log_{\frac{M}{B}}\frac{N}{B}))$ where $V$ is the total number of valid pairs, $N$ is the total number of objects, $B$ is the number of pairs that can be stored in one disk block and $M$ is the number of pairs that can be stored in the main memory.

The rest of the paper is organized as follows. In Section II, we formally define the problem and give an overview of the most relevant work. We present our framework and its advantages in Section III. In Section IV, we present our technique to create and maintain internal memory and external memory sources which is the core part of our approach. We present our query processing algorithms in Section V. Experiment results are given in Section VI. Section VII concludes the paper.

## II. PRELIMINARIES

### A. Problem Definition

First, we define *monotonic* and *loose monotonic* scoring functions. A function $f$ is called a monotonic function if it satisfies $f(x_1, \cdots, x_n) \leq f(y_1, \cdots, y_n)$ whenever $x_i \leq y_i$ for every $1 \leq i \leq n$.

Now, we define the loose monotonic functions. Let $s(.,.)$ be a scoring function that takes two values as parameter and returns a score. A function $s(.,.)$ is a loose monotonic function if for every value $x_i$ both of the following are true: i) for a fixed $x_i$ and every $x_j > x_i$, $s(x_i, x_j)$ either monotonically increases or monotonically decreases as $x_j$ increases, and ii) for a fixed $x_i$ and every $x_k < x_i$, $s(x_i, x_k)$ either monotonically increases or monotonically decreases as $x_k$ decreases.

The absolute difference of two values (e.g., $|x_i - x_j|$) is a loose monotonic function. This is because for a fixed $x_i$ and any value $x_j$ larger than it, the absolute difference monotonically increases when $x_j$ increases. Similarly, for any fixed $x_i$ and any value $x_k$ smaller than it, the absolute difference monotonically increases as $x_k$ decreases. Please note that the loose monotonic functions are more general because these require the scores to be monotonic only with respect to every individual $x_i$ and the function may not be monotonic in general. All monotonic functions are loose monotonic functions but the converse may not be true for some functions. For example, the absolute difference of two values is a loose monotonic function but it is not a monotonic function. The average of two values is a loose monotonic function as well as a monotonic function.

For ease of presentation, we classify loose monotonic functions into different categories. A loose monotonic function is called right increasing (resp. decreasing) function if for every $x_j > x_i$ for the fixed $x_i$, $s(x_i, x_j)$ monotonically increases (resp. decreases) as $x_j$ increases. For example, the absolute difference is a right increasing function. A loose monotonic function is called left increasing (resp. decreasing) function if for every $x_k < x_i$ for the fixed $x_i$, $s(x_i, x_k)$ monotonically increases (resp. decreases) as $k$ decreases. For instance, the absolute difference is a left increasing function whereas the average of two values is a left decreasing function.

Let $d$ be the number of attributes specified by the user for a top-$k$ pairs query. For each attribute $i$, the user specifies a loose monotonic scoring function $s_i(.,.)$ that computes the score of a pair on the attribute $i$. Such scoring function is called a local scoring function and the score $s_i(a, b)$ of a pair $(a, b)$ is called its local score. The users are allowed to define a different local scoring function for each attribute. The user defines a

monotonic global scoring function $f$ that takes $d$ local scores as parameter and returns the final score $SCORE(a,b)$ of a pair $(a,b)$ as $f(s_1(a,b),\cdots,s_d(a,b))$.

**Score-based top-$k$ pairs query.** Given a set of objects $O$, a non-chromatic top-$k$ pair query returns a set of pairs $P \subseteq O \times O$ that contains $k$ pairs such that for any pair $(a,b) \in P$ and any pair $(a',b') \notin P$, $SCORE(a,b) \leq SCORE(a',b')$.

**Chromatic queries.** Consider that each object in a set of objects $O$ is assigned a color. A chromatic query is similar to a non-chromatic query except for an additional constraint; that is, only the pairs that meet the color requirement are considered. A homochromatic top-$k$ pairs query considers only the pairs that have two objects having the same color. In contrast, a heterochromatic top-$k$ pair query considers only the pairs that contain objects with different colors.

We define the skyline pairs query and the rank-based top-$k$ pairs query in Section V-B and Section V-C, respectively.

### B. Related Work

*1) k Closest Pairs Queries:* The $k$ closest pairs query is a special case of the score-based top-$k$ pairs queries. The problem of $k$ closest pairs queries has received significant research attention by the computational geometry community (see [1] for a nice survey). Below, we give an overview of the previous work in the context of spatial databases.

Hjaltason et al. [2] are the first to study the problem of closest pairs in the context of spatial databases. They propose incremental distance joins where two datasets are joined and the pairs are output incrementally according to the distances between them. While the proposed solution has a nice feature that it returns the pairs incrementally, its priority queue size may be prohibitively large.

Corral et al. [3] propose several algorithms for k closest pairs queries. Similar to the previous algorithm [2], they also index the datasets by R-trees. They use distance bounds to prune the intermediate node pairs. They observe that the performance of their algorithm largely depends on the overlap factor of the two datasets. It is important to note that although the amount of the memory used by their algorithm is small as compared to the memory usage of the algorithm proposed in [2], there is no guarantee on the amount of the main memory usage (e.g., the size of the heap can be $O(V)$ where $V$ is the total number of possible pairs).

*2) Top-k Query Processing:* Top-$k$ queries retrieve the top-$k$ objects based on a user defined scoring function. The problem has been extensively studied [6], [7], [8], [9]. Ilyas et al. [10] give a comprehensive survey of top-$k$ query processing techniques. We briefly describe some of the top-$k$ processing algorithms that combine multiple ranked sources and return the top-$k$ objects. More specifically, each source $S_i$ contains the objects ranked on their scores according to a preference $i$. Let $x_i$ be the score of an object in a source $S_i$. The final score of the object is computed by using a monotonic function $f(x_1,\cdots,x_d)$ where $d$ is the number of sources. The algorithms report $k$ objects with the smallest final scores.

The top-$k$ algorithms assume that the objects in a source can be accessed in two ways. A *sorted* access on a source reads the next object in the sorted order. A *random* access returns the score of any specified object in a given source. In a random access, the specified object is searched in the source and its score is returned. It is important to note that not all the sources can support both types of accesses (e.g., a search engine provides the sorted access but does not support a random access).

Now, we briefly introduce three well known algorithms.

**Fagin's Algorithm (FA).** FA [11] assumes that the sources support both sorted and random accesses. Let there be $d$ sources $S_i,\cdots,S_d$. FA works as follows.

**1.** Do sorted access in parallel on each of the $d$ sources. Go to step 2 when there are at least $k$ objects that have been returned by *every* source.

**2.** For each object that has been returned by at least one source, do the random accesses on the other sources to retrieve its scores on remaining sources and compute its final score. Return $k$ objects with the smallest final scores.

A major problem with FA is that it uses unbounded buffer (i.e., the number of objects stored in the main memory may be arbitrarily large).

**Threshold Algorithm (TA).** TA (independently proposed in [11], [9], [12]) also assumes that the sources support both sorted and random accesses. TA works as follows.

**1.** Do sorted accesses in parallel on each of the $d$ sources. For each object $o$ returned from a source $S_i$, do the random accesses on every other source to obtain its scores in the other sources. Compute the final score of $o$ using the monotonic function $f$. Maintain a heap that contains $k$ objects with the smallest scores.

**2.** Let $\underline{x}_i$ be the score of the last object returned from the source $S_i$ through a sorted access. After every sorted access, update the *threshold value* as $t = f(\underline{x}_1,\cdots,\underline{x}_d)$. Terminate the algorithm when the heap contains $k$ objects whose scores are at most equal to $t$. Report the objects in the heap as top-$k$ objects.

It has been shown that the number of accesses by TA cannot be larger than the number of accesses by FA. Furthermore, TA is optimal in number of accesses when every source supports both the sorted and random accesses. Moreover, the buffer size of TA is $O(k)$ because at any time it keeps only the best $k$ objects in its buffer.

**No Random Access (NRA) Algorithm.** NRA [11] assumes that the sources do not support the random accesses. The algorithm works as follows.

**1.** Do the sorted accesses in parallel on each of the $d$ sources. For each returned object $o$, compute its best possible score $B(o)$ and its worst possible score $W(o)$ by assuming the best and worst possible scores on the sources that have not yet returned it. Maintain a heap that contains $k$ objects with the smallest worst scores $W(o)$.

**2.** Let $W_k$ be the largest of the worst scores of $k$ objects in the heap. At each sorted access, update $W_k$ and the best possible score $B(o)$ of every seen object $o$. Terminate the algorithm

when $B(o) \geq W_k$ for every seen object $o$. Report the objects in the heap as the top-$k$ objects.

It has been shown that NRA is optimal in the number of accesses when the random access is not supported by the sources. However, like FA, it also requires an unbounded buffer. Moreover, the best possible scores of all seen objects are to be updated whenever an object is returned by a sorted access.

Mamoulis et al. [13] present several interesting observations and propose an algorithm LARA that significantly improves the performance of NRA. Due to the space limitations, we omit the details and refer the readers to [13].

## III. Our Proposed Framework

Let $d$ be the number of local scoring functions involved in the top-$k$ pairs query. We map our problem to the well studied problem of top-$k$ query that combines the scores from different *ranked sources* (see the previous section). More specifically, we maintain $d$ sources (please see Fig. 1) such that each source $S_i$ incrementally returns the pair with the best score according to the $i^{th}$ local scoring function. The existing top-$k$ algorithms (e.g., FA, TA and NRA) view these sources as the ranked inputs and can be used to retrieve the top-$k$ pairs by combining these ranked inputs.

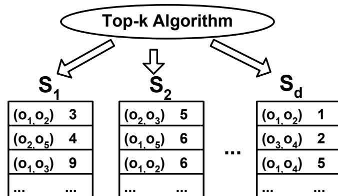



Fig. 1. Our framework

Most of the existing work on the top-$k$ queries can be applied to solve the problem of the top-$k$ pairs queries. However, these algorithms assume that the sources can report the elements in a sorted order. Hence, it is important to develop efficient techniques to create and maintain the sources such that each source can return the pairs of objects in a sorted order. A straight forward solution to create a source $S_i$ is to sort all the possible pairs according to their local scores on the $i^{th}$ attribute. However, this solution requires storing and sorting $O(V)$ pairs where $O(V)$ is the number of valid pairs (this number is $O(N^2)$ for non-chromatic queries if $N$ is the number of objects). Clearly, the time and the space complexity of this straight forward approach may be prohibitive.

In the next section, we present an optimal internal memory algorithm and an optimal external memory algorithm to create and maintain such sources. The internal memory algorithm uses $O(N)$ space and is optimal in time complexity. The external memory algorithm is I/O optimal.

Below we highlight a few advantages of our framework.

**1. No pre-built indexes required.** Our proposed algorithm does not require any pre-built indexes, i.e., there does not exist any index at the time a query is issued. We remark here that the indexes like R-tree usually index all the dimensions (i.e., attributes) of the objects and the queries that involve a subset of these dimensions may not be answered efficiently by these indexes. Moreover, the pruning rules used on these indexes are based on the distance metrics and may not work for the generic scoring functions.

**2. Known memory requirement.** The existing techniques for $k$-closest pairs queries [2], [3] use heap to store the intermediate nodes of the R-trees. The size of the heap may become as large as $O(V)$ and the system may run out of memory. In contrast, our external memory algorithm has a bounded memory requirement (it requires $O(k)$ space in addition to $2d$ buffer pages).

**3. Efficient.** Although our proposed approach supports more general top-$k$ pairs queries and does not require any pre-built indexes, our experimental results demonstrate that the proposed approach is in general more efficient than the existing solutions of $k$ closest pairs queries. We also conduct theoretical analysis and show that the expected cost of our proposed approach is optimal for the queries that involve two or less attributes.

**4. Feasible for implementation in a DBMS.** Unlike the existing techniques that target specific problems, our general algorithmic framework solves a broad class of top-$k$ pairs queries (including all the existing variants) and is easy to implement. Moreover, the proposed technique outperforms existing algorithms both theoretically and experimentally. Hence, it is a good choice to be implemented in any DBMS.

## IV. Maintaining The Sources

### A. Internal Memory Source

First, we define some terminologies. Suppose that all the objects are sorted in ascending order of their attribute values such that $o_1 \leq o_2 \leq \cdots \leq o_N$. For any pair $(o_u, o_v)$, we refer to the first object $o_u$ in the pair as *host* and the second object $o_v$ as *guest*. A pair $(o_u, o_v)$ means that the object $o_u$ is a host to a guest $o_v$.

For the ease of presentation, we assume that the local scoring function $s(.,.)$ satisfies[3] $s(o_u, o_v) = s(o_v, o_u)$. To avoid reporting a pair $(o_u, o_v)$ again as $(o_v, o_u)$, we will consider only the pairs $(o_u, o_v)$ such that $u < v$. This implies that every object $o_u$ can host only the objects that are on the right side of $o_u$ in the sorted list $o_1 \leq o_2 \leq \cdots \leq o_N$. For chromatic queries, only the objects that meet the color requirement and are on the right side of $o_u$ will be considered its guests. Let $o_v$ and $o_{v'}$ be two guests of $o_u$. We say that $o_v$ is a better guest of $o_u$ than $o_{v'}$ if $s(o_u, o_v) < s(o_u, o_{v'})$. An object $o_v$ is called the best guest of a host $o_u$ if for *every* other guest $o_{v'}$ of the host $o_u$, $s(o_u, o_v) \leq s(o_u, o_{v'})$. We say that an object $o_u$ has hosted the object $o_v$, if the pair $(o_u, o_v)$ has been reported to the main algorithm.

Algorithm 1 presents the details of creating and maintaining a source. Initially, all the objects are sorted in ascending order of their attribute values (ties are broken arbitrarily). Then, for

[3]The scoring functions for which $s(o_u, o_v) \neq s(o_v, o_u)$ can be easily handled by joining two sources such that the first source considers only the pairs $(o_u, o_v)$ for every $u < v$ and the second source considers only the pairs $(o_v, o_u)$ for every $u < v$.

**Algorithm 1** Creating and maintaining a source

**InitializeSource()**
1: sort the objects in ascending order of their values
2: **for** each object $o_u$ **do**
3: $o_v \leftarrow$ the best guest of $o_u$
4: insert the pair $(o_u, o_v)$ into heap with score $s(o_u, o_v)$

**getNextBestPair()**
1: get the top pair $(o_u, o_v)$ from the heap
2: **if** next best guest of $o_u$ exists **then**
3: $o_{v'} \leftarrow$ the next best guest of $o_u$
4: insert the pair $(o_u, o_{v'})$ in heap with score $s(o_u, o_{v'})$
5: **return** $(o_u, o_v)$

each object $o_u$, a pair $(o_u, o_v)$ is created such that $o_v$ is the best guest of $o_u$. All these pairs are inserted in a heap.

Whenever a request for the next best pair arrives, the source retrieves the top pair $(o_u, o_v)$ from the heap and reports it to the main algorithm. The next best pair $(o_u, o_{v'})$ is inserted in the heap where $o_{v'}$ is the next best guest of $o_u$. At any stage during the execution, the next best guest of $o_u$ is the best guest among the guests of $o_u$ which has not been hosted by $o_u$ earlier.

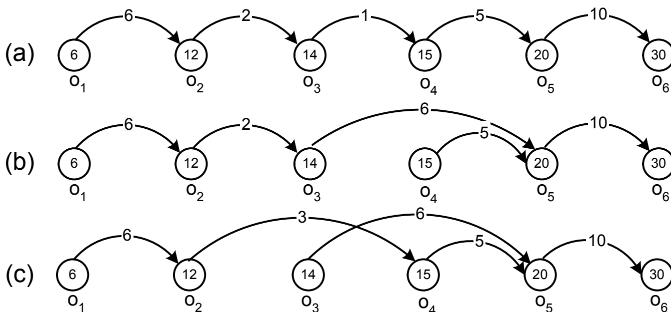



Fig. 2. Illustration of Algorithm 1

EXAMPLE 1 : Consider the example of Fig. 2 which shows six objects $o_1$ to $o_6$ sorted on their attribute values. The values inside the circles are the attribute values. Assume that the scoring function is the absolute difference. A pair $(o_u, o_v)$ is shown by a directed edge from the host $o_u$ to the guest $o_v$. Initially, for each object, a pair with its best guest is created and inserted in the heap. Note that the best guest of an object is its right adjacent object when the function is absolute difference. Fig. 2(a) shows the pairs (see the edges) that are inserted in the heap. The number on an edge corresponds to the score of the pair. The best pair is $(o_3, o_4)$ and its score is 1. When this is retrieved, the algorithm determines that the next best guest of $o_3$ is $o_5$ and inserts $(o_3, o_5)$ in the heap with score 6 (see Fig. 2(b)). Now the top pair of the heap is $(o_2, o_3)$ which is returned when the system requests the next best pair from this source. The next best guest of $o_2$ is $o_4$ so a new pair $(o_2, o_4)$ is inserted in the heap with score 3 (see Fig. 2(c)).

The intuitive justification of the correctness of the algorithm is that at any stage, we keep the best guests (among those that it has not hosted yet) for each object in the heap. This implies that for every pair that does not exist in the heap either there exists a better pair in the heap or the pair has already been reported to the main algorithm. The following lemma proves the correctness of the algorithm.

LEMMA 1 : For any pair $(o_x, o_y)$ that is not present in the heap and has not been reported earlier, there exists at least one pair $(o_u, o_v)$ in the heap such that $s(o_u, o_v) \leq s(o_x, o_y)$.

*Proof:* First we prove it for the case when $x < y$. For each object $o_x$, we always have one object $o_v$ in the heap (if $o_x$ has not already hosted all valid guests) such that $o_v$ is its best guest among the objects that it has not hosted yet. If $o_x$ has hosted all valid guests, this implies that the pair $(o_x, o_y)$ has been hosted. Otherwise, there must be at least one pair $(o_x, o_v)$ in the heap such that $s(o_x, o_v) \leq s(o_x, o_y)$. This is because an object $o_x$ will not host $o_y$ unless it has hosted all the guests that are better than $o_y$.

Now, assume $x > y$. Following the similar argument as above, if the pair $(o_y, o_x)$ has not been reported then there exists at least one pair $(o_y, o_v)$ in the heap such that $s(o_y, o_v) \leq s(o_y, o_x)$. ∎

In order to achieve the optimal complexity, the algorithm must find the best guests for all N objects in O(N). Moreover, the algorithm must find the next best guest of any object $o_u$ in O(1).

Before we show the details of how to do these operations with required complexity, we introduce the concept of *left adjacent* and *right adjacent* objects. A left (resp. right) adjacent object of $o_u$ is the first object $o_x$ on the left (resp. right) side of $o_u$ in the sorted list $o_1 \leq o_2 \leq \cdots \leq o_N$ such that the pair $(o_u, o_x)$ satisfies the color requirement.

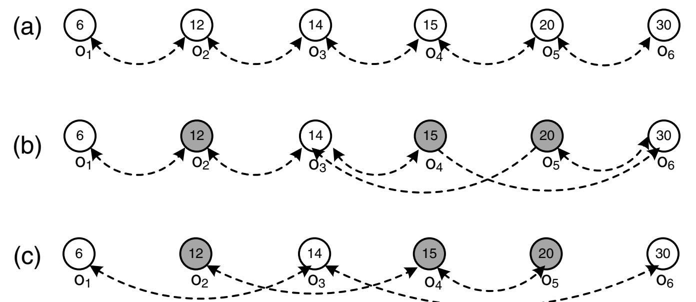



Fig. 3. Adjacent objects (a) Non-chromatic (b) Heterochromatic (c) Homochromatic

Fig. 3 shows an example where the objects $o_1$ to $o_6$ are shown. Some objects are shaded ($o_2, o_4$ and $o_5$) and others are white ($o_1, o_3$ and $o_6$). Fig. 3(a), (b) and (c) show the adjacent objects for non-chromatic queries, heterochromatic queries and homochromatic queries, respectively. The adjacent objects are shown with broken lines. An arrow from an object $o_x$ to $o_y$ indicates that $o_y$ is the adjacent object of $o_x$ in that direction.

Later in this section, we show that the left and the right adjacent objects of all the objects can be determined in O(N).

*1) Finding the best guest for each object $o_u$:* Below, we describe the procedure for the right increasing and the right decreasing functions (see Section II-A for the definitions).

**For right increasing functions.** Recall that if the scoring function is right increasing then the score $s(o_u, o_v) \leq s(o_u, o_{v'})$ if $v < v'$ (i.e., $o_{v'}$ is on the right side of $o_v$ in the sorted list). Hence, for any object $o_u$, its best guest is its right adjacent object. For example, in Fig. 3(c), $o_3$ is the best guest of $o_1$ if the scoring function is right increasing function (e.g., absolute difference).

**For right decreasing functions.** For any object $o_u$, the best guest in this case is the right most object $o_v$ such that the pair $(o_u, o_v)$ meets the color requirement. More specifically, for non-chromatic queries, the best guest of any object $o_u$ is $o_N$. For example, in Fig. 3(a) the best guest of every object is $o_6$ if the scoring function is a right decreasing function (e.g., $s(o_u, o_v) = -(o_u + o_v)$).

For the heterochromatic queries, if $o_N$ has a color different than $o_u$ then $o_N$ is the best guest of $o_u$. Otherwise the left adjacent object of $o_N$ is the best guest of $o_u$ because it is guaranteed to have a color different than $o_u$. In the example of Fig. 3(b), $o_6$ is the best guest of $o_2, o_4$ and $o_5$ whereas $o_5$ is the best guest of $o_1$ and $o_3$.

For the homochromatic queries, we scan the sorted list $o_1 \leq \cdots \leq o_N$ once and maintain the right most object of each color. For each object $o_u$, its best guest is the right most object of the same color. In the example of Fig. 3(c), $o_6$ is the best guest for $o_1$ and $o_3$ whereas $o_5$ is the best guest of $o_2$ and $o_4$.

*2) Finding next best guest of any object $o_u$:* Let $o_v$ be the current best guest of the object $o_u$. The next best guest of $o_u$ can be determined in O(1). Below, we describe how to find the next best guests for the right increasing functions and the procedure is similar for the right decreasing functions.

For the non-chromatic queries and the homochromatic queries, the next best guest $o_{v'}$ for an object $o_u$ is the right adjacent object of $o_v$. In the example of Fig. 3(c), let $o_3$ be the current guest of $o_1$. The next best guest of $o_1$ is $o_6$ which is the right adjacent object of $o_3$.

For the heterochromatic queries, the next best guest of $o_u$ is $o_{v+1}$ if $o_{v+1}$ has a color different than $o_u$. Otherwise, the right adjacent object of $o_{v+1}$ is guaranteed to have a different color and hence is the next best guest of $o_u$. Consider the example of Fig. 3(b) and assume that the current best guest of the object $o_2$ is $o_3$. When $(o_2, o_3)$ is reported, the algorithm checks $o_4$ to see if it is the next best guest of $o_2$. Since $o_2$ and $o_4$ have the same color, the next best guest of $o_2$ is $o_6$ which is the right adjacent object of $o_4$.

*3) Finding the adjacent objects:* Now we illustrate how to add pointers to the adjacent objects in O(N) time. For the non-chromatic queries, the procedure is trivial. So, we first discuss the procedure for determining the right adjacent objects for the heterochromatic queries. The procedure starts with setting the right adjacent object of $o_N$ to NULL. Then, it starts scanning the sorted list of the objects from right to left. For each object $o_u$, if $o_{u+1}$ has a different color than $o_u$ then $o_{u+1}$ is set as the right adjacent object of $o_u$. Otherwise, the right adjacent object of $o_{u+1}$ is set as the right adjacent object of $o_u$.

Consider the example of Fig. 3(b). The right adjacent object of $o_6$ is set to NULL. The right adjacent object of $o_5$ is $o_6$ because they have different colors. The right adjacent object of $o_4$ is not $o_5$ because they have same color. So, the right adjacent object of $o_5$ (which is $o_6$) is set as the right adjacent object of $o_4$. The algorithm continues in this way. The left adjacent objects can be set similarly by scanning the list from left to right.

For the homochromatic queries, we assign the right adjacent objects as follows. While we scan the list from right to left, we maintain the last seen object of each color. For any object $o_u$, its right adjacent object is the last seen object of the same color (NULL if no object has been seen of this color). The left adjacent objects are set similarly by scanning the list from left to right.

*4) Complexity:* The first pair is returned in $O(N\ Log\ N)$ (the objects are sorted and $O(N)$ pairs are inserted in the heap). We remark that this meets the lower bound of returning the closest pair in one dimension [14]. Since our general framework covers the closest pairs, the lower bound of the algorithm is $O(N\ Log\ N)$ hence our algorithm is optimal.

As illustrated earlier, the next best guest of any object $o_u$ can be determined in $O(1)$. For each host $o_u$, the heap contains at most one pair $(o_u, o_v)$. Hence, the maximum size of the heap is $O(N)$ which implies that each heap operation takes $O(Log\ N)$. In other words, a source incrementally returns the next best pair in $O(Log\ N)$.

### *B. External Memory Source*

The basic idea of the external memory algorithm is the same as the internal memory algorithm. However, there are following two main challenges: 1) the heap cannot be stored in the internal memory and 2) finding the next best guest of an object requires accessing the sorted list of the objects which is stored in the external memory (this means that the algorithm would need to access the external memory every time the next best guest is to be determined).

We address the first challenge by using the external memory priority queue proposed by Arge [15]. The basic idea of the external priority queue (or heap) is to retrieve and insert the elements in a batch which reduces the amortized I/O cost. Arge shows that the external priority queue can do an insert or delete operation in $O(\frac{1}{B} Log_{\frac{M}{B}} \frac{N}{B})$ amortized I/O where $B$ is the number of elements that can be stored in one disk page, $M \geq 2B$ is the number of elements that can be stored in the internal memory and $N$ is the number of elements in the priority queue. For details, please see [15].

We introduce the notion of dummy pairs to address the second challenge (i.e., to find the next best guest of an object without accessing the external memory). A dummy pair with a host $o_u$ and a guest $o_v$ is denoted as $(\overline{o_u, o_v})$. The pairs $(o_u, o_v)$ we introduced earlier are called the regular pairs hereafter. Recall that when a regular pair $(o_u, o_v)$ is retrieved from the heap, a pair $(o_u, o_{v'})$ is created and inserted in the heap where $o_{v'}$ is the next best guest of $o_u$. In contrast, when a dummy pair $(\overline{o_u, o_v})$ is retrieved from the heap, a dummy pair $(\overline{o_{u'}, o_v})$ is created and inserted in the heap where $o_{u'}$ is the next best host of $o_v$. The best host $o_u$ is defined in a similar way as

the best guest. More specifically, we say that an object $o_u$ is a better host of $o_v$ than $o_{u'}$ if $s(o_u, o_v) < s(o_{u'}, o_v)$. Finding the next best host is similar to finding the next best guest as described in previous section.

In Fig. 4, for each object, we show a regular pair with its best guest (curved arrows pointing right) and a dummy pair with its best host (connector style arrows pointing left). The scoring function is the sum of the attribute values and the score of each pair is shown on its edge.

Recall that when a pair $(o_u, o_v)$ is retrieved from the heap, the next best guest $o_{v'}$ is determined by using the adjacent object information of $o_v$. For our external memory algorithm, we propose to store the adjacent object information with both the regular pairs and the dummy pairs. More specifically, with a regular pair $(o_u, o_v)$, we attach the information of adjacent objects of the host $o_u$. In contrast, for a dummy pair $(\overline{o_u, o_v})$ we attach the information of adjacent objects of the guest $o_v$. The object that stores the adjacent object information in a pair is marked with a star. For example, $(\star o_u, o_v)$ denotes that the adjacent object information of $o_u$ is attached with the pair $(o_u, o_v)$.

**Algorithm 2  Creating and maintaining external memory source**

**InitializeSource()**

1: sort the objects in ascending order of their values
2: **for** each object $o_i$ **do**
3: attach adjacent object's information with $o_i$
4: $o_j \leftarrow$ the best guest of $o_i$
5: $o_k \leftarrow$ the best host of $o_i$
6: insert the pair $(\star o_i, o_j)$ into heap with score $s(o_i, o_j)$
7: insert the dummy pair $(\overline{o_k, \star o_i})$ into heap with score $s(o_k, o_i)$

**getNextBestPair()**

1: get the top pair $(\star o_u, o_v)$ from the heap
2: get the next top pair (which is dummy pair $(\overline{o_u, \star o_v})$)/* Lemma 2 */
3: **if** next best guest of $o_u$ exists **then**
4: $o_{v'} \leftarrow$ the next best guest of $o_u$
5: insert the pair $(\star o_u, o_{v'})$ in heap with score $s(o_u, o_{v'})$
6: **if** next best host of $o_v$ exists **then**
7: $o_{u'} \leftarrow$ the next best host of $o_v$
8: insert the dummy pair $(\overline{o_{u'}, \star o_v})$ into heap with score $s(o_{u'}, o_v)$
9: **return** $(\star o_u, o_v)$

Algorithm 2 presents the details of creating and maintaining an external memory source.. The main idea behind the algorithm is that the heap is modified such that whenever a pair $(\star o_u, o_v)$ is retrieved from the heap, its dummy pair $(\overline{o_u, \star o_v})$ is the next best pair in the heap. These two pairs are retrieved and are used as follows; The next best guest of $o_u$ is determined by using the adjacent object information of $o_v$ which is stored in the dummy pair $(\overline{o_u, \star o_v})$. Similarly, the next best host of $o_v$ can be determined by using the adjacent object information of $o_u$ which is stored in the regular pair $(\star o_u, o_v)$. It is easy to see that the next best pairs can be formed without accessing the external memory.

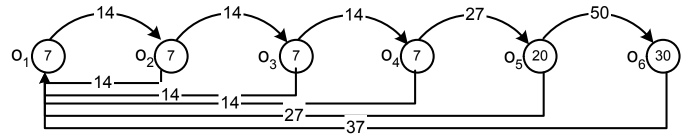



Fig. 4. Illustration of dummy pairs

**Modifying the heap priority function.** It is important to clarify that if there exists more than one pair with the same score then the next best pair may not be the dummy pair of $(o_u, o_v)$. Consider the example of Fig. 4, where several pairs have score 14. If the heap accesses the pair $(o_2, o_3)$, the next best pair cannot be its dummy pair $(\overline{o_2, o_3})$ because it has not been inserted in the heap yet. To guarantee that the next best pair is always the dummy pair of the retrieved pair, we modify the heap priority function as follows.

If two pairs have the same score, the heap gives priority based on the IDs of their guest objects. More specifically, if the scoring function is a right increasing function then the pair with the smaller ID of the guest object is given preference. If the scoring function is a right decreasing function then the pair with the larger ID of the guest object is given preference. The ID of each object in a source is its position in the list sorted in ascending order of attribute values. For instance, the ID of an object $o_u$ is $u$.

If two pairs have the same score and the same guest object then the heap gives priority based on the IDs of their host objects. More specifically, if the scoring function is a left increasing function then the heap prefers the pair with the larger ID of the host object. If the function is a left decreasing function then the heap prefers the pair with the smaller ID of the host object.

If two pairs have the same score, the same guest object and the same host object then one of them is a regular pair and the other is its dummy pair. In this case, the heap gives priority to the regular pair.

Lemma 2 guarantees the correctness of the algorithm. Due to the space limits, we omit the proof. The interested readers can see the proof (and a detailed example) in our technical report [16].

LEMMA 2 : Assume that the heap uses the priority function as described above. If a pair $(o_u, o_v)$ is the top pair of the heap then its dummy pair $(\overline{o_u, o_v})$ is the second top pair.

Please note that once the source is created, it does not require to access the external memory to create new pairs. The only external memory I/Os are due to insertion and deletion from the external memory heap. The cost of returning the first pair is sorting the objects and inserting $O(N)$ pairs in the external heap. Hence, the cost is $O(\frac{N}{B}Log_{\frac{M}{B}}\frac{N}{B})$ which is I/O equivalent to $O(N\ Log\ N)$ internal memory algorithm and hence is optimal [17]. The amortized I/O cost for retrieving next best pair is $O(Log_{\frac{M}{B}}\frac{N}{B})$ which is I/O equivalent to the cost of internal memory source.

## V. Query Processing Algorithm

In this section, we present our query processing algorithms and provide the complexity analysis. More specifically, we present the techniques to answer score-based top-$k$ pairs queries, skyline pairs queries and rank-based top-$k$ pairs queries in Section V-A, V-B and V-C, respectively.

### A. Score-based Top-k Pairs Queries

*1) Technique:* We apply the threshold algorithm (TA) [11], [9], [12] to combine the scores of a pair from different sources and return the top-$k$ pairs. However, please note that TA assumes that the sources support the random accesses (see Section II-B). In other words, when a pair is returned from a source $S_i$, TA needs to obtain its score on every other attribute. We enable TA to access the scores of a pair on the other attributes as follows.

For internal memory algorithms, we assume that the objects are stored in the main memory (this consumes $O(dN)$ memory space). When a pair $(o_u, o_v)$ is returned from one of the sources, we use the object table and retrieve the attribute values of $o_u$ and $o_v$ and compute the score of $(o_u, o_v)$ on every other attribute.

For the external memory algorithm, doing the random access requires accessing the object table (which exists in the external memory). This would be quite expensive because we need to look up the attribute values of two objects for each seen pair and this may require two I/Os. One solution is to apply NRA algorithm [11] because it does not require random accesses. However, NRA algorithm requires an unbounded buffer (see Section II-B) and the main memory consumption may be prohibitively large (it may be $O(V)$ where $V$ is the total number of valid pairs).

To enable random accesses for TA, we modify each source $S_i$ such that each pair stores $d$ attribute values of both of the objects. This increases the amortized I/O cost of creating the external memory source by a factor $d$ because the number of entries that can be stored in one disk block is reduced. However, doing this allows us to compute the score of each pair on every attribute without any additional I/O. Although this approach may increase the disk usage, the external memory sources are required only during the query processing and the data can be deleted after the query has been answered.

*2) Complexity Analysis:* The number of elements accessed by TA is always less than or equal to the number of elements accessed by Fagin's Algorithm (FA) [11] (see Section II-B). FA algorithm stops the sorted accesses when exactly $k$ elements are returned from all $d$ sources. Let $V$ be the number of elements in each source. The expected number of sorted accesses by FA is $T = O(V^{(d-1)/d}k^{1/d})$ under the assumption that the score of an element in one source is independent of its score in other sources [18].

As the cost of TA is always less than or equal to FA, the number of pairs our algorithm is expected to access from each source is $O(T)$ assuming that the score of a pair in one source is independent of its score in the other sources. The total number of accesses from all $d$ sources is $O(dT)$. As shown earlier, the cost of accessing a pair from a source is $O(Log\ N)$, hence the total expected cost [4] for the internal memory algorithm is given by Eq. (1).

$$O(dT\ Log\ N) = O(d\ V^{\frac{d-1}{d}} k^{\frac{1}{d}} Log\ N) \qquad (1)$$

For the non-chromatic queries, the total number of valid pairs $O(V)$ is $O(N^2)$. Hence the expected cost of our algorithm to answer a two dimensional closest pair query is $O(N\ Log\ N)$ which is optimal in algebraic decision tree model [14].

The cost of our external memory algorithm can be obtained similarly. The amortized I/O cost of accessing $dT$ ($T$ pairs from each source) is $O(\frac{dT}{B}(Log_{\frac{M}{B}} \frac{N}{B}))$ where $B$ is the number of pairs that can be stored in one block and $M \geq 2B$ is the number of pairs that can be stored in the main memory reserved for an external priority queue. For a two dimensional non-chromatic closest pair queries, the expected amortized I/O cost is $O(\frac{N}{B}(Log_{\frac{M}{B}} \frac{N}{B}))$ which is I/O equivalent to $O(N\ Log\ N)$ internal memory algorithm hence is optimal [17].

The space usage of the internal memory algorithm is $O(dN)$ because the main algorithm stores a table containing $N$ objects with $d$ attributes for each object and each source stores a table of $N$ objects with one attribute value for each object. The main memory usage of the external memory algorithm is $O(k+dM)$ where $M$ is the memory used for each source. The minimum memory an external source requires is $2B$, hence the minimum main memory requirement is $O(k + 2dB)$.

### B. Skyline Pairs Query

A pair $(x, y)$ is said to *dominate* another pair $(a, b)$ if for every attribute $i$, $s_i(x, y) \leq s_i(a, b)$ and for at least one attribute $j$, $s_j(x, y) < s_j(a, b)$. A skyline pairs query returns every pair that is not dominated by any other pair.

It can be shown that for any monotonic global scoring function, the best pair is always one of the skyline pairs. In other words, the skyline pairs query gives shortlisted candidate pairs such that for every candidate pair there exists a global scoring function for which it is the best pair. Hence if the users cannot define a suitable scoring function, they can select a pair from the skyline pairs that best meets their requirement.

Consider the example of a person who is interested in buying a broadband internet connection and a home phone connection. He might want to retrieve the pairs (broadband and phone) that have low total monthly cost, low total setup fee and shorter average contract length. Suppose that a database stores the information of broadband and home phones provided by different companies. While the score-based top-$k$ pairs queries can be used to retrieve the top-$k$ pairs, the user may instead prefer to retrieve all the pairs that are not dominated by any other pair (i.e., return every pair such that no other pair has

[4]Note that the cost analysis includes the cost of creating the sources. The cost of creating $d$ sources is $O(d(N\ Log\ N))$ which is dominated by Eq. (1). Same holds for the cost analysis of the external memory algorithm. Our experiment results also include the cost of creating the sources.

lower total monthly cost, lower total setup fee and shorter average contract length).

*1) Technique:* For ease of the presentation, we assume that all the pairs in a source have unique scores. Later, we will present the approach to handle the case when more than one pair can have same score. Our algorithm is similar to Fagin's Algorithm (FA) (see Section II-B). However, unlike FA algorithm, we address the problem of unbounded buffer. Our algorithm works as follows.

**1 .** Do the sorted accesses on each source $S_i$. For each newly seen pair $p$, determine its score on all other attributes. Compare $p$ with existing skyline pairs and include it in the set of skyline pairs if it is not dominated by any existing skyline pair. Otherwise, discard it.

**2.** Terminate when at least one object has been seen under the sorted accesses from all the sources. Report the skyline pairs.

The correctness of the algorithm follows from the fact that a pair $p$ cannot be dominated by any pair $p'$ that is accessed after it. This is because the score of $p'$ is larger than $p$ in at least one source. The termination condition is also correct because if a pair $p$ is seen in every source then every pair $p'$ that has not been seen in *any* source is dominated by $p$.

If more than one pair the have same score in a source $S_i$ then a pair $p$ can be dominated by a pair $p'$ that is accessed after it. This is because $p'$ may have a score equal to $p$ in the source $S_i$ and may have smaller scores in all other sources. We address this issue as follows. Let $x_i$ be the score of a pair $p$ that has been accessed from a source $S_i$. We discard the pair $p$ if it is dominated by any of the existing skyline pairs. Otherwise, we insert it in a list $C$ which contains the candidate skyline pairs. When a pair $p'$ is accessed from $S_i$, if its score is equal to $x_i$ it is compared with every pair in $C$ and the pairs that are dominated by $p'$ are deleted. Whenever the score of $p'$ is larger than $x_i$, all the pairs in $C$ are confirmed as the skyline pairs and are inserted in the set of skyline pairs.

Let $score_i$ be the score of a pair $p$ in a source $S_i$ such that $p$ has been seen under sorted accesses on all sources. The algorithm terminates if the score $x_i$ of the last pair seen in a source $S_i$ is larger than $score_i$. This is because every unseen pair has a score on $S_i$ larger than that of $p$ and cannot have score less than the score of $p$ on every other source. The proof of correctness is straight forward and is omitted.

A $k$-skyband [19] query returns every element that is dominated by at most $(k-1)$ other elements. A $k$-dominant skyline [20] query returns every element that is not dominated by any other element in $k$ or more dimensions. We remark that the extension of the algorithm to answer $k$-skyband pairs query and $k$-dominate skyline pairs query is straight forward. Due to space limits, we omit the details.

*2) Analysis:* We assume that the pairs have unique scores in each source. The number of accesses from each source is equal to the accesses by FA (because the algorithm stops when at least one object has been returned from all sources). So, the expected number of accesses from each source is $T = O(V^{(d-1)/d})$ (the value of $k$ is one). The expected number of total accesses on all the sources is $O(dT)$.

For each retrieved pair, we compare it with all the existing skyline pairs. The average number of skyline pairs is estimated to be $O(Log^{d-1}V)$ [21]. Since $V$ is at most $O(N^2)$, the expected number of skyline pairs is $O(Log^{d-1}N)$. Hence the expected cost of the internal memory skyline pairs algorithm is $O(dT\ Log^{d-1}N)$. The expected amortized I/O cost is the same as the cost of score-based top-$k$ ($k = 1$) pairs query obtained because the cost was obtained using the number of accesses by FA.

The lower bound on the cost of the skyline pairs queries is $O(N\ Log\ N)$ which can be obtained by reducing it to the closest pair query. It is easy to see that the expected cost of our algorithms meets the lower bound of the skyline pairs queries if two or less attributes are involved. The expected main memory usage of the internal memory algorithm is $O(dN + Log^{d-1}N)$ because in addition to the object table, it also stores the existing skyline pairs. The expected main memory requirement of the external memory algorithm is $O(k + 2dB + Log^{d-1}N)$.

### C. Rank-based Top-k Pairs Queries

In order to define a suitable scoring function, the users must have sufficient domain knowledge. Moreover, it is difficult to define a global scoring function on the attributes that are incompatible (e.g., dollars and inches) [22]. In such cases, the users can issue a rank-based top-$k$ pairs query which is defined below.

First, we define the rank of a pair $(a, b)$ on an attribute $i$ denoted by $rank_i(a, b)$. Let $s_i$ be the loose monotonic scoring function for the attribute $i$. $rank_i(a, b)$ is the number of pairs $(x, y)$ for which $s_i(x, y) < s_i(a, b)$. In other words, if the pairs are sorted in ascending order of their scores on $i^{th}$ attribute, $rank_i(a, b)$ is the rank of the pair $(a, b)$ in the sorted order.

Given a global scoring function $f$, the final rank-based score $R_SCORE$ of a pair $(a, b)$ is;

$$R_SCORE(a, b) = f(rank_1(a, b), \cdots, rank_d(a, b)) \quad (2)$$

Given a set of objects $O$, a rank-based top-$k$ pairs query returns a set of pairs $P \subseteq O \times O$ that contains $k$ pairs such that for any pair $(a, b) \in P$ and any pair $(a', b') \notin P$, $R_SCORE(a, b) \leq R_SCORE(a', b')$.

*1) Technique:* When a pair $p$ is seen on a source $S_i$, although its score on the other sources can be determined, it might not be possible to determine its rank on the other sources. In other words, the random access on a source cannot determine the rank of a pair in this source. However, if a pair $p$ is seen under the sorted access then its rank is the number of pairs that have been returned by this source and have smaller scores. This can be easily done by maintaining a counter for each source. The problem of rank-based top-$k$ pairs can be solved by using NRA [11] because the sorted accesses are possible but the random accesses are not possible.

As mentioned in Section II-B, there are two major weaknesses of NRA. First is that whenever a new element is seen

under the sorted access, the best possible scores of all the previously seen pairs are to be updated. This problem has been addressed by LARA algorithm [13] which we briefly described in Section II-B. The second problem is that NRA uses an unbounded buffer. We reduce its memory usage by the following observation. A pair $p$ that is dominated by $k$ other pairs cannot be the top-$k$ pair. Hence, we only need to maintain the $(k+1)$-skyband pairs. Other pairs can be safely pruned.

*2) Analysis:* In the worst case, the growing phase of LARA (see Section II-B) completes when there are at least $k$ elements that are seen on all the sources. Hence, the expected number of pairs accessed from each source is at most equal to the number of pairs accessed by FA. So, the expected number of pairs accessed from each source during the growing phase is $T = O(V^{(d-1)/d} \cdot k^{1/d})$. In the growing phase, when a pair $p$ is retrieved, it is compared against all $(k+1)$-skyband pairs to see if it can be pruned. The expected size of $(k+1)$-skyband is $O(k\ Log^{d-1}N)$ [23]. So, the expected cost of the growing phase is $O(dkTLog^{d-1}N)$ because in total $dT$ pairs are accessed and each pair is compared with every pair in the $(k+1)$-skyband.

Now, we estimate the number of elements accessed by the shrinking phase of LARA (which cannot be more than the number of elements accessed by NRA). We assume that the global function is sum of the local scores. Moreover, we assume that the scores in every source are unique. As stated earlier, when $O(T)$ elements are accessed from each source, the algorithm is expected to see $k$ elements that have been returned by all the sources. The worst possible score of these $k$ elements is $W_k = dT$ (the rank of the pair is $T$ in each source). If $dT$ elements are accessed from each source, then the algorithm can stop. This is because the final score of every object that is not seen in at least one source cannot be smaller than $W_k = dT$. Hence, the number of accesses by NRA on each source is at most $dT$ where $T = O(V^{(d-1)/d} \cdot k^{1/d})$. The total number of accesses on all $d$ sources is $O(d^2T)$. The cost of each access in the shrinking phase is $O(Logk + 2^d)$ [13]. Hence the expected total cost of the shrinking phase is $O(d^2T(Log\ N + Log\ k + 2^d))$.

The total cost of the internal memory rank-based top-$k$ pairs query is the sum of the cost of the growing phase and the cost of the shrinking phase as computed above. The expected amortized I/O cost of the external memory algorithm is $O(\frac{d^2T}{B}Log_{\frac{M}{B}}\frac{N}{B})$ because $d^2T$ pairs are expected to be accessed from the sources.

The expected main memory requirement for the internal memory algorithm is $O(dN + k\ Log^{d-1}\ N)$ because the pairs in $(k+1)$-skyband are also kept in the memory. The expected main memory requirement of the external memory algorithm is $O(2dB + k\ Log^{d-1}\ N)$.

## VI. EXPERIMENTS

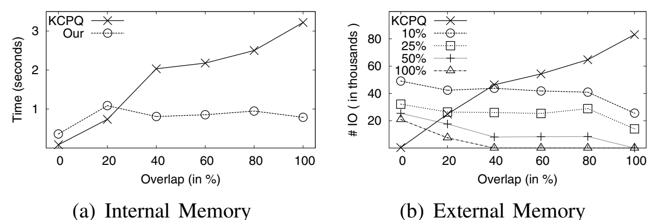
We conducted extensive experiments on both real and synthetic datasets. Due to the space limitation, we present only the most representative results. There does not exist any previous work for the skyline pairs queries and the rank-based top-$k$ pairs queries. However, several algorithms exist for the $k$ closest pairs query which is a special case of the score-based top-$k$ pairs queries. Moreover, naïve algorithms for the skyline pairs queries and the rank-based top-$k$ pairs queries perform extremely bad (in many cases they either ran out of memory or did not finish within two days). For these reasons, we focus on evaluating our score-based top-$k$ pairs queries algorithm. At the end of this section, we show the performance of the other two algorithms.

We show that our algorithm for the score-based top-$k$ pairs queries outperforms the existing best known algorithm (KCPQ) [3] for the $k$-closest pairs queries. For the queries that use more generic functions, we compare our algorithm with a naïve algorithm because there does not exist any other work to handle such queries.

### *A. k-Closest Pairs Queries*

We compare our algorithm with the best known $k$-closest pairs algorithm called KCPQ [3]. In accordance with [3], the page size for both of the algorithms is set to $1K$. The $k$ closest pairs query joins two data sets each containing $100,000$ objects and returns the $k$ closest pairs. $k$ is set to 10 in all experiments unless mentioned otherwise.

It has been noted that the overlap between the datasets is one of the main factors [3] that affect the performance of the existing algorithms. Fig. 5 shows the effect of the overlap on KCPQ and our algorithm. In Fig. 5(a), we run both of the algorithms in the internal memory and observe that our algorithm is 2 to 3 times faster when the overlap is at least $40\%$. For the smaller overlaps, the performance of KCPQ is better because most of the intermediate nodes of the R-trees are quickly pruned. However, its performance is still not significantly better than our algorithm. Note that our algorithm is not sensitive to the data overlap.

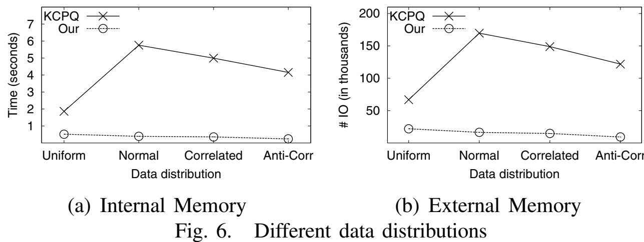



(a) Internal Memory (b) External Memory

Fig. 5. Effect of overlapping

Fig. 5(b) shows the performance of both of the algorithms in the external memory. The heap of KCPQ algorithm contains the intermediate nodes of the R-trees. Consequently, it uses larger amount of main memory. The buffer size for our algorithm is set according to the main memory usage of KCPQ. More specifically, we run our algorithm with the buffer size set to $100\%, 50\%, 25\%$ and $10\%$ of the memory used by KCPQ. Fig. 5(b) demonstrates that when the overlap is $40\%$ or more, our algorithm performs better even when the memory used by our algorithm is $10\%$ of the memory used by KCPQ.

We also conducted several experiments on different data distributions. More specifically, we generated the datasets following uniform, normal, correlated and anti-correlated dis-




(a) Internal Memory (b) External Memory

Fig. 6. Different data distributions

tributions. For each distribution, we generated two datasets with $50\%$ overlap between them. Fig. 6 demonstrates that our algorithm is not affected by the data distribution and performs significantly better than KCPQ.

We compared the two algorithms for several other parameters and datasets and observed that although our algorithm supports more general scoring functions and does not require pre-built indexes, it outperforms KCPQ for all settings except when the overlap is too small.

### B. Generic Scoring Functions

For the general scoring functions, we compare our algorithms with a naïve algorithm. The naïve algorithm uses nested loop to join a dataset with itself (block nested loop for external memory processing). The disk page size is set to $4K$. The buffer size for each of our external memory source is set to 2 pages (this is the minimum required by the external priority queue [15]).

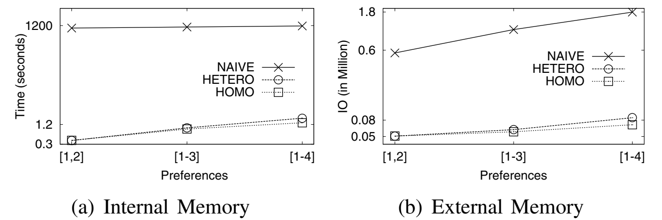
*1) Real Data:* The real dataset[5] consists of location data consisting of $304,895$ location points belonging to 87 zip codes of USA. The zip codes roughly map to different towns (or suburbs). Each point in the dataset corresponds to a residential block. We extracted the coordinates of the streets and the number of addresses along each street. We treat the center of each street as a residential block and the number of addresses along the street as the population of the block. For each block, we randomly generate a value which denotes the average rent of the houses in the residential block. All of the attributes are normalized to a unit space. The global scoring function we used is the sum of the local scores.

| Preference | Heterochromatic | Homochromatic |
|---|---|---|
| 1&2: Distance | close | far |
| 3: Population | high | high |
| 4: Rent | low | low |

TABLE I

THE QUERIES USED ON REAL DATA

We use several heterochromatic and homochromatic queries each involving two to four attributes. Table I shows some of the queries we use on the real data. First two preferences involve two attributes (i.e., the two location coordinates of each block). A heterochromatic query on these two attributes retrieve the closest pairs of blocks such that each block is located in a different suburb. For a query involving $d$ preferences, we use the first $d$ preferences for that query listed in the table. For example a homochromatic query on three attributes retrieves the pairs of blocks (located in the same suburb) that are far from each other and have high total population. $k$ is set to 10 for all queries.




(a) Internal Memory (b) External Memory

Fig. 7. Real data

Fig. 7 shows that the naïve algorithm is three orders of magnitude slower than our internal memory algorithm and uses an order of magnitude more IOs. The query time for our algorithm is low which demonstrates the applicability of our approach in the real world applications. Similar results were observed when the queries were run for other parameters.

*2) Synthetic Data:* The default synthetic dataset contains the points following a uniform distribution. Each object is randomly assigned a color. The number of colors vary from 50 to 250. The local scoring functions used by the algorithms are the sum and the absolute difference. The global scoring function is a weighted aggregate (we allow negative weights). For each dimension, a local scoring function is randomly chosen (sum or absolute difference) and is assigned a random weight.

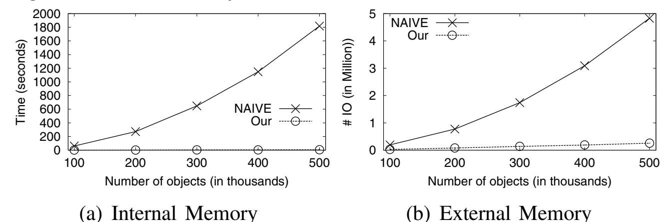
| Parameter | Range |
|---|---|
| Number of objects ($\times 1000$) | 100, 200, **300**, 400, 500 |
| Number of colors | 50, **100**, 150, 200, 250 |
| Number of attributes | 2, 3, **4**, 5, 6 |
| k | 1, **10**, 25, 50, 100 |

TABLE II

EXPERIMENT PARAMETERS

We present the results for the homochromatic top-$k$ queries. The results for the non-chromatic and the heterochromatic queries follow similar trends. Table II shows the default parameters in bold.

Fig. 8 and Fig. 9 study the effect of increasing the number of objects and the number of attributes, respectively. While the performance of both of the algorithms[6] degrades, our algorithm scales very well.



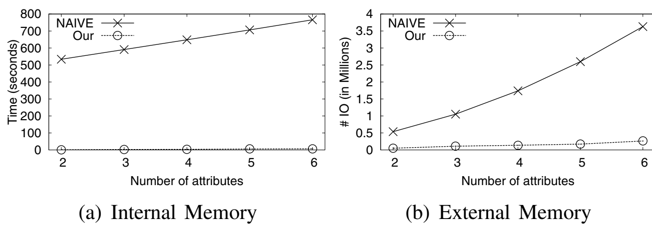

(a) Internal Memory (b) External Memory

Fig. 8. Effect of number of objects

Fig. 10 studies the effect of $k$. The performance of our algorithm is better for smaller $k$. The naïve algorithm is not

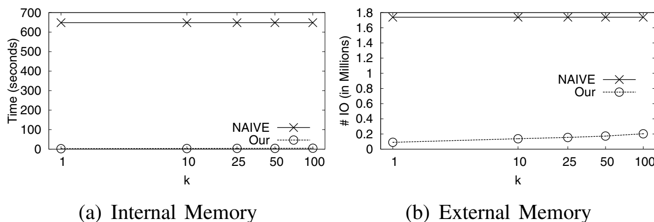
[5]http://www.census.gov/geo/www/tiger/

[6]To compare the scalability of the two algorithms, we show the results in linear scale. The readers interested in seeing the same results in logscale can see our technical report [16].




(a) Internal Memory (b) External Memory

Fig. 9. Effect of number of attributes

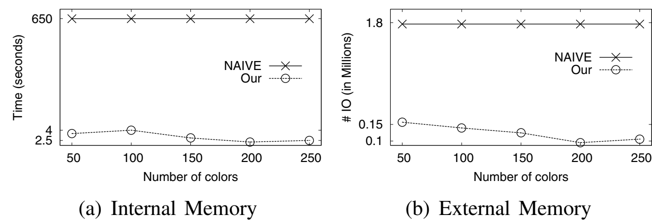



(a) Internal Memory (b) External Memory

Fig. 10. Effect of k

affected by $k$ because it considers all the pairs regardless of the value of $k$.

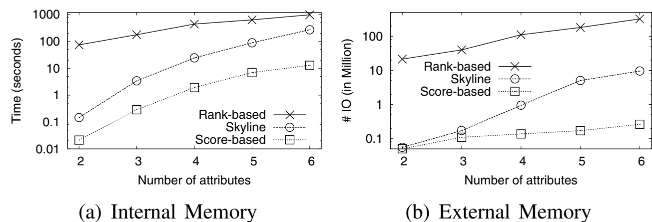
Fig. 11 studies the effect of number of colors. Our algorithm performs slightly better when the number of colors is large. This is mainly because the number of valid pairs decreases when the number of colors is large. However, the effect is not very significant because the number of pairs that are accessed from each source is not significantly affected.




(a) Internal Memory (b) External Memory

Fig. 11. Effect of number of colors

Finally, we present the results for the skyline pairs queries and the rank-based top-$k$ pairs queries. As stated earlier, the naïve algorithms perform extremely bad. Therefore, we compare the performance of our proposed algorithms (for the score-based queries, the rank-based queries and the skyline queries) to give the readers an insight about the cost of each type of query.




(a) Internal Memory (b) External Memory

Fig. 12. Comparison of different top pairs queries

Fig. 12 shows the cost of the three algorithms when different number of attributes are involved in the query. The score-based top-$k$ queries are the easiest to solve among the three and the rank-based top-$k$ pairs queries are the hardest. The cost of each of the algorithms increases with the number of attributes used in the query.

## VII. Conclusion

We present a unified approach to answer a broad class of top-$k$ pairs query including the $k$ closest pairs queries, the $k$ furthest pairs queries and their variants. The expected performance of the proposed algorithms is optimal when the queries involve two or less attributes. Extensive experiments demonstrate the efficiency of our proposed algorithms.

**Acknowledgments:** The second author was supported by the ARC Discovery Grants (DP110102937, DP0987557, DP0881035), Google Research Award and NICTA. The fourth author's research was supported by NSFC (Project 61073005).

## References

[1] M. Smid, "Closest-point problems in computational geometry," in *Handbook on Computational Geometry, published by Elsevier Science*, 1997.
[2] G. R. Hjaltason and H. Samet, "Incremental distance join algorithms for spatial databases," in *SIGMOD*, 1998.
[3] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest pair queries in spatial databases," in *SIGMOD*, 2000.
[4] C. Yang and K.-I. Lin, "An index structure for improving nearest closest pairs and related join queries in spatial databases," in *IDEAS*, 2002.
[5] J. Shan, D. Zhang, and B. Salzberg, "On spatial-range closest-pair query," in *SSTD*, 2003, pp. 252–269.
[6] K. C.-C. Chang and S. won Hwang, "Minimal probing: supporting expensive predicates for top-k queries," in *SIGMOD Conference*, 2002, pp. 346–357.
[7] K. Mouratidis, S. Bakiras, and D. Papadias, "Continuous monitoring of top-k queries over sliding windows," in *SIGMOD Conference*, 2006.
[8] R. Fagin, "Combining fuzzy information from multiple systems," in *PODS*, 1996, pp. 216–226.
[9] S. Nepal and M. V. Ramakrishna, "Query processing issues in image (multimedia) databases," in *ICDE*, 1999.
[10] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top- query processing techniques in relational database systems," *ACM Comput. Surv.*, vol. 40, no. 4, 2008.
[11] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 614–656, 2003.
[12] U. Güntzer, W.-T. Balke, and W. Kießling, "Optimizing multi-feature queries for image databases," in *VLDB*, 2000.
[13] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung, "Efficient top- aggregation of ranked inputs," *ACM Trans. Database Syst.*, vol. 32, no. 3, p. 19, 2007.
[14] M. Ben-Or, "Lower bounds for algebraic computation trees (preliminary report)," in *STOC*, 1983.
[15] L. Arge, "The buffer tree: A technique for designing batched external data structures," *Algorithmica*, 2003.
[16] M. A. Cheema, X. Lin, H. Wang, J. Wang, and W. Zhang, "A unified framework for computing best pairs queries," in *UNSW Technical Report, 2010. ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/1005.pdf*.
[17] J. S. Vitter, "External memory algorithms and data structures: Dealing with massive data," *ACM Computing Surveys*, vol. 33, p. 2001, 2001.
[18] R. Fagin, "Combining fuzzy information from multiple systems," *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 83–99, 1999.
[19] 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.
[20] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang, "Finding k-dominant skylines in high dimensional space," in *SIGMOD*, 2006.
[21] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson, "On the average number of maxima in a set of vectors and applications," *J. ACM*, 1978.
[22] R. Fagin, R. Kumar, and D. Sivakumar, "Efficient similarity search and classification via rank aggregation," in *SIGMOD*, 2003.
[23] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu, "Probabilistic skyline operator over sliding windows," in *ICDE*, 2009, pp. 1060–1071.