A Unified Framework for Computing Best Pairs Queries

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

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

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

[⋆] *Tsinghua University, China*
jimwang@tsinghua.edu.cn

Technical Report
UNSW-CSE-TR-1005
Feb 2010

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Top-k pairs queries have many real applications. k closest pairs queries, k furthest pairs queries and their bichromatic variants are few 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 ranking 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 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 a pair by combining its scores on different attributes. The proposed framework also supports the skyline pairs queries; that is, return the pairs that are not dominated by any other pair. 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 and is parallelizable.

1 Introduction

Given a set of objects $\{o_1, \cdots, o_N\}$ and a ranking function to compute the score of a pair of objects (o_u, o_v). A top-k pairs query returns k pairs with the best scores among all possible pairs. An important and well studied special case of the top-k pairs query is the k closest pairs query which returns k pairs with smallest distances. The k closest pairs queries have been extensively studied in the context of computational geometry (see [22] and references therein).

Due to the popularity of location based services, the spatial database community has also conducted significant research on the k closest pairs queries, k furthest pairs queries and their variants [13, 7, 25, 21, 1]. However, there does not exist a unified approach that efficiently answers the k closest pairs queries and their variants under different L_p distances. To the best of our knowledge, we are first to provide a unified framework that efficiently supports a broad class of top-k pairs queries including the above mentioned queries.

The top-k pairs queries have also many interesting applications in traditional databases. For example, the k most similar (or dissimilar) pairs queries might be used to find the pairs of objects that are the most similar (or dissimilar) to each other based on any user defined similarity measure. Another interesting variation is to find the pairs of objects that are similar to each other in one subspace and dissimilar in another subspace.

To the best of our knowledge, we are first to study the problem of top-k pairs queries based on generic scoring functions. We present a fresh approach to answer the top-k pairs queries which is efficient and supports a broad class of queries. Below, we present some of the queries supported by our proposed framework. Formal definitions of the queries are given in Section 2.1.

Score-based top-k pairs queries. Previous work supports limited distance functions (e.g., closest pair in Euclidean space). However, a user may want to retrieve top-k pairs based on a more general scoring function. 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 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 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

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

a *global* scoring function that computes the final score of a pair by combining its scores on all attributes computed by the local scoring functions.

Our framework supports any global scoring function that is *monotonic* and any local scoring function that is *loose monotonic*. Monotonic functions cover a wide range of functions and are used in many real applications. Although we define monotonic and loose monotonic scoring functions in Section 2.1, we remark here that the loose monotonic functions are more general than the monotonic functions.

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

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 scoring functions on the attributes that are incompatible (e.g., dollars and inches) [10]. In such cases, the users can issue rank-based top-k pairs queries where each pair is ranked according to each attribute involved in the query. The final score of each pair is computed by combining the ranks of the pair on each attribute.

Skyline pairs queries. A skyline pairs query chooses every pair which is not *dominated* by any other pair; that is, no other pair has better score on each of the involved attributes. 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 and rank-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 lower total monthly cost, lower total setup fee and shorter average contract length). Our approach can also be extended to answer k-skyband [18] pairs and k dominant skyline [5] pairs. To the best of our knowledge, we are first to study the rank-based top-k pairs and skyline pairs queries.

Chromatic and non-chromatic queries. We classify top-k pairs queries 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 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 (e.g., 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. He may issue a heterochromatic top-k pairs query by adding a condition in `where` clause of the query. Please 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 [13, 7] solve bichromatic queries and the extension to heterochromatic queries is either non-trivial or inefficient.

Below, we summarise our contributions.

- We 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 and can parallelize the query processing.
- We theoretically analyse the performance of the proposed algorithms. The expected performance of the top-k pairs queries and skyline pairs queries is optimal when the number of attributes involved is two or less. Moreover, our external memory algorithm uses less memory compared to the existing work on k closest pairs queries [13, 7].
- Our extensive experiments demonstrate significant improvement over the existing best known solution for the k closest pairs query. For the more general top-k pairs queries, we compare our algorithm with the naïve algorithm and observe more than an order of magnitude improvement.

Rest of the paper is organized as follows. In Section 2, we formally define the problem and present an overview of the related work. We present our framework and its advantages in Section 3. In Section 4, we present our techniques to create and maintain internal memory and external memory sources. We present our query processing algorithms in Section 5. Experiment results are given in Section 6. Section 7 concludes the paper.

2 Preliminaries

2.1 Problem Definition

Monotonic and loose monotonic functions

A function f is 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 loose monotonic functions which are more general than the monotonic functions.

Given a set of values $x_1 \leq x_2 \leq \cdots \leq x_n$. A scoring function $s(.,.)$ defined on a pair of values is a loose monotonic function if for every x_i both of the following are true: i) for every $j > i$ for the fixed i, $s(x_i, x_j)$ either monotonically increases or monotonically decreases as j increases, and ii) for every $k < i$ for the fixed i, $s(x_i, x_k)$ either monotonically increases or monotonically decreases as 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 each loose monotonic function into different categories. A loose monotonic function is called right increasing (resp. decreasing) function if for every $j > i$ for the fixed i, $s(x_i, x_j)$ monotonically increases (resp. decreases) as 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 $k < i$ for the fixed 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.

Queries

Let d be the number of attributes specified by the user for the top-k pairs query. For each attribute i, the user specifies a scoring function s_i that computes the score of a pair on the attribute i. Such scoring function is called local scoring function and the score $s_i(a, b)$ of a pair (a, b) returned by the local scoring function is called its local score. The user defines a global scoring function f that takes as parameter d local scores and returns the final score of a pair.

$$SCORE(a, b) = f(s_1(a, b), \cdots, s_d(a, b)) \tag{2.1}$$

The global scoring function f may be any monotonic scoring function and a local scoring function s_i may be any *loose monotonic* scoring function. The users are allowed to define a different local scoring function for each attribute.

Now, we define 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 $s_i(x, y) < s_i(a, b)$.

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

$$R\_SCORE(a, b) = f(rank_1(a, b), \cdots, rank_d(a, b)) \tag{2.2}$$

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 $(a', b') \notin P$, $SCORE(a, b) \leq SCORE(a', b')$.

Rank-based top-k pairs query. Given a set of objects O, a non-chromatic rank-based 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 $(a', b') \notin P$, $R\_SCORE(a, b) \leq R\_SCORE(a', b')$.

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 non-chromatic skyline pairs query returns every pair that is not dominated by any other pair.

Chormatic queries. Given a set of objects O such that each object is assigned a color. A chromatic top-k pairs query is similar to a non-chromatic query except an additional constraint that 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 same color. On the other hand, a heterochromatic top-k pair query considers only the pairs that contain two objects having different colors.

2.2 Related Work

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 [22] for a nice survey). Below, we give an overview of the previous work in the context of spatial databases.

Hjaltason et al. [13] are 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. [7] propose several algorithms for k closest pairs queries. Similar to the previous algorithm [13], they also index the datasets by R-trees. They use bounds based on the minimum and maximum distances 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 memory used by their algorithm is small compared to the algorithm proposed in [13], there is no guarantee on the amount of main memory usage (e.g., the size of the heap can be $O(V)$ where V is the total number of possible pairs).

Yang et al. [25] proposed a data structure to further improve the k closest pairs algorithm. Their algorithm works for the case when all the pairs have unique distances [21]. Several variants of k closest pairs queries have also been studied in [23, 1, 21, 19].

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, 16, 8, 17]. Ilyas et al. [14] 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. Fig. 3.1 shows three sources where the objects are sorted on their scores. Let x_i be the score of an object in a source S_i. The final score of the object is computed using a monotonic function $f(x_1, \cdots, x_d)$ where d is the number of sources. The algorithms report k pairs with 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 sorted order according to its score. For example, first sorted access on the source S_1 returns p_1 and the second sorted access returns p_2. A *random* access returns the score of a given object from a source. For example, a top-k algorithm may request source S_2 to return the score of the object p_1. The source finds the object p_1

and returns its score 12. It is important to note that not all sources can support both types of accesses (e.g., a search engine provides sorted accesses but does not support random accesses).

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 source $S_i, \cdots, S_d$. 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 random accesses on 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, 17, 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 p returned from a source S_i, do random accesses on every other source to obtain its scores in the other sources. Compute the final score of p 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 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) NRA [11] assumes that the sources do not support random accesses. The algorithm works as follows.

1. Do sorted accesses in parallel on each of the d sources. For each seen object p, compute its best possible score $B(p)$ and worst possible score $W(p)$ 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 $W(p)$.

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

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

Mamoulis et al. [15] present some interesting observations and propose an algorithm LARA that significantly improves the performance of NRA. LARA consists of two phases. In *growing phase*, the objects that are seen under sorted accesses form a candidate set. They prove that during the growing phase no candidate object can be pruned. Hence, update of the best possible scores is not required. Let $\underline{x}_i$ be the last score seen on a source S_i and W_k be the k^{th} smallest

worst score of seen objects. The growing phase completes when $t \geq W_k$ where $t = f(\underline{x}_1, \cdots, \underline{x}_d)$ and denotes the best possible score of any unseen object.

In *shrinking phase*, the candidates are divided in 2^d categories based on the sources on which they have been seen. For each group, the candidate with smallest worst score is called the leader. They prove that as new objects are accessed only the best possible scores of the leaders are to be updated. If the leader of a group can be pruned, the whole group is pruned. The algorithm stops when there is no leader with best possible score smaller than W_k .

3 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 top-k queries that combine the scores from different ranked sources (please see Fig. 3.1). More specifically, we maintain d sources 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 algorithm (e.g., FA, TA and NRA) views these sources as ranked inputs and can be used to retrieve the top-k pairs by combining the ranked inputs.

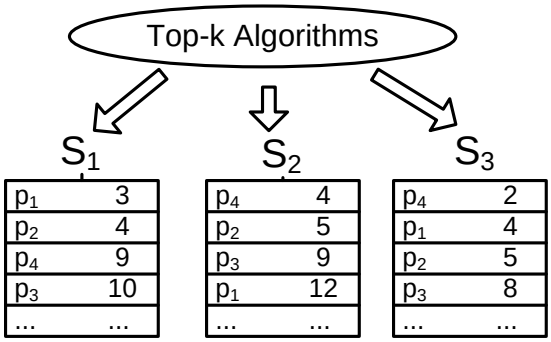


Figure 3.1: Our framework

As mentioned in previous section, combining the scores from ranked sources has received significant research attention. Most of the existing work can be applied to our framework to solve the problem of top-k pairs queries. These algorithms assume that the sources can report the elements in sorted order. Hence, it is important to develop efficient techniques to create and maintain the sources such that each source can return the pairs in sorted order.

Below, we highlight some of the advantages of using our framework.

1. No pre-built indexes required. The existing solutions [13, 7] to k closest pairs queries use R-trees to index the objects and then use a join to retrieve the k closest pairs. R-tree based solutions have the following weaknesses: i) creating R-trees might not be helpful when the scoring function is not a distance metric, ii) R-trees may not perform well if only a subset of dimensions is used by the scoring functions, and iii) many applications require retrieving the top-k pairs among the objects that meet some selection conditions. In such cases, if the R-trees is pre-built on all objects then its performance may be poor because it may contain many invalid objects.

Our framework does not require any pre-built indexes and efficiently solves the top-k pairs queries avoiding the above mentioned problems.

2. Known memory requirement. Existing techniques use heap to store the intermediate nodes of the R-trees. The size of heap may become large and the system may run out of memory. On the other hand, our proposed algorithm for the score-based top-k queries has a fixed memory requirement (it requires $O(k)$ space in addition to $2d$ buffer pages). For skyline pairs queries, the space usage is linear to the number of skyline pairs (i.e., answer size). The memory usage of rank-based top-k queries is same as the skyline pairs queries.

3. Efficient. Although our proposed approach supports more general top-k pairs queries and does not require any pre-built index, our experiment 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. Parallelizable. Our proposed approach can be easily parallelized. More specifically, each source can be processed by a different processor. The main algorithm requests the pairs from difference sources and computes the top-k pairs. The main algorithm does not need to wait for a particular source and can continue computation based on the pairs received from the other sources.

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

In next section, we present optimal algorithms to create and maintain the sources. In Section 5, we present the query processing algorithms.

4 Maintaining The Sources

In this section, we present algorithms to create and maintain the sources. More specifically, we present an optimal internal memory in Section 4.1 and an optimal external memory algorithm in Section 4.2.

4.1 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 ease of presentation, we assume[1] that $s(o_u, o_v) = s(o_v, o_u)$. 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 right side of o_u in the sorted list $o_1 \leq o_2 \leq \cdots \leq o_N$. For chromatic

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

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 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 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)

Algorithm 1 presents the details of creating and maintaining a source. Initially, all the objects are sorted in ascending order of their attribute values such that $o_1 \leq o_2 \leq \cdots \leq o_N$ (ties are broken arbitrarily). Then, for 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 the 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 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.

Example 1: Consider the example of Fig. 4.1 which shows six objects o_1 to o_6 sorted on 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. 4.1(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. 4.1(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. 4.1(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.

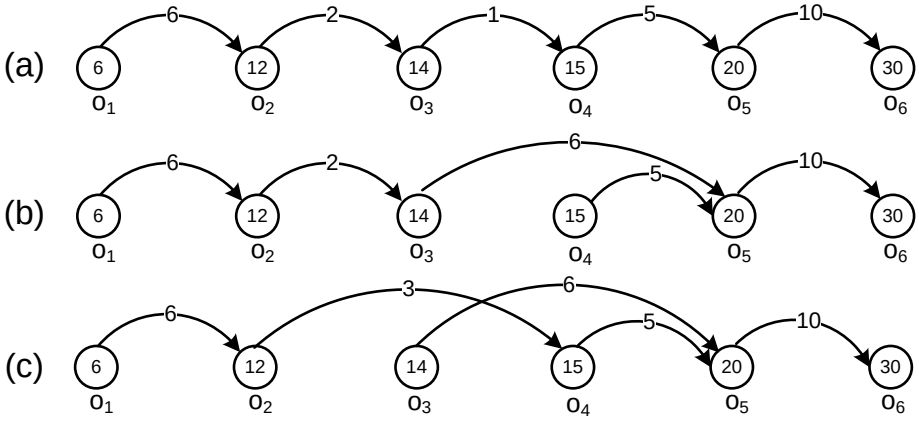


Figure 4.1: Illustration of Algorithm 1

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 guest for each of the 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. 4.2 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. 4.2(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.

In Section 4.1, we show that left and right adjacent objects of all objects can be determined in O(N). For each object, we store pointers to its adjacent objects which enable the algorithm to access the adjacent objects of any object in O(1).

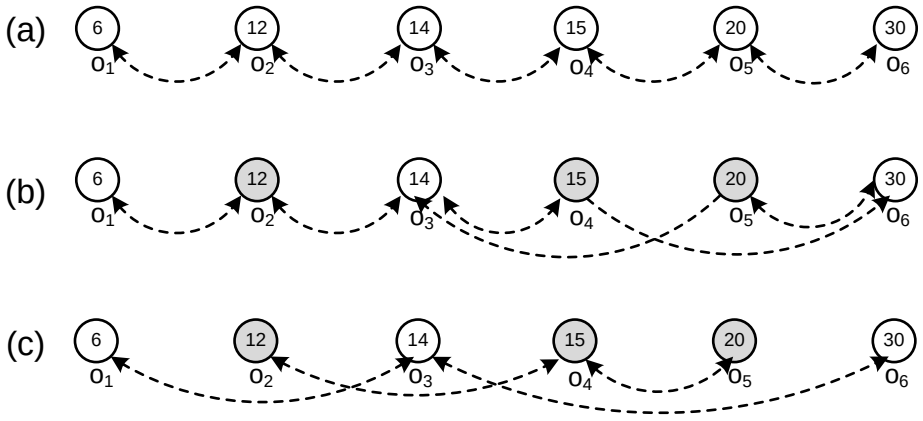


Figure 4.2: Adjacent objects (a) Non-chromatic (b) Heterochromatic (c) Homochromatic

Finding the best guest of o_u

For right increasing functions. Recall that if the scoring function is right increasing[2] then the score $s(o_u, o_v) \leq s(o_u, o_{v'})$ if $v < v'$ (i.e., $o_{v'}$ is on right side of o_v in the sorted list $o_1 \leq o_2 \leq \cdots \leq o_N$). Hence, for any object o_u, its best guest is its right adjacent object. For example, in Fig. 4.2(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. 4.2(a) the best guest of every object is o_6 if the scoring function is right decreasing function (e.g., $s(o_u, o_v) = -(o_u + o_v)$).

For 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. 4.2(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 homochromatic queries, we scan the sorted list $o_1 \leq \cdots \leq o_N$ from left to right and maintain the last seen object of each color. For each object o_u, its best guest is the last seen object of the same color. In the example of Fig. 4.2(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.

The cost of finding the best guests for all N objects is $O(N)$ for both the chromatic and non-chromatic queries.

Finding next best guest of o_u

Let o_v be the current best guest of the object o_u. The next best guest of o_u is determined in O(1) as follows.

For right increasing functions. For non-chromatic queries and the homochromatic queries, the next best guest $o_{v'}$ for an object o_u is the right adja-

[2]The algorithm can determine if the function is right increasing or right decreasing by using any three values $x < y < z$. If $s(x, y) < s(x, z)$, the scoring function is right increasing.

cent object of o_v. In the example of Fig. 4.2(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 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. 4.2(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.

For right decreasing functions. The basic idea is similar as for the right increasing functions. More specifically, for non-chromatic queries and homochromatic queries, the next best guest of o_u is the left adjacent object of o_v. Consider the example of Fig. 4.2(c) and assume that the current best guest of o_1 is o_6. The next best guest of o_1 is o_3 which is the left adjacent object of o_6.

For heterochromatic queries, the next best guest of o_u is o_{v-1} if it has a color different than o_u. Otherwise the left adjacent object of o_{v-1} is the next best guest of o_u. In Fig. 4.2(b), assume that the current best guest of o_3 is o_5. The next best guest of o_3 is o_4 because it has a color different than o_3.

Finding the adjacent objects

Now we illustrate how to add pointers to the adjacent objects for all objects in O(N) time. For non-chromatic queries, the procedure is straight forward. 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. 4.2(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 and sets right adjacent objects of all the objects. The left adjacent objects can be set similarly by scanning the list from left to right.

For homochromatic queries, we assign the right adjacent objects as follows. While we scan the list, we maintain the last seen object of each color (an array of size m can be used where m is the number of unique colors). The algorithm starts scanning the list of the objects from right to left. 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.

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 [3]. Since our general framework covers the closest pairs, the lower bound of the algorithm is $O(N\ Log\ N)$ hence

our algorithm is optimal. Now, we analyse the cost of incrementally returning next best pairs.

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 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 $O(Log\ N)$.

The lower bound of an algorithm that returns T closest pairs is $O(N\ Log\ N + T)$ [20] where the closest pairs are not reported in sorted order. Reporting them in sorted order would require a total run time of $O(N\ Log\ N + T\ Log\ T)$. On average, the cost of returning each pair would be $O(Log\ T)$. Our source reports each pair in $O(Log\ N)$. Next, we present a simple strategy that improves the complexity of our algorithm to return next best pair from $O(Log\ N)$ to $O(Log\ T)$ for $T < N$ where T is the number of pairs retrieved from the source. The cost becomes $O(Log\ N)$ when $T > N$.

The improvement in the complexity is achieved by making the following simple change in Algorithm 1. The source is created using a list L instead of the heap. More specifically, at line 4 of the `InitializeSource()` function, we push the pairs in a list L instead of inserting the pairs in heap. When all N pairs have been inserted in the list L, we sort the list of pairs in ascending order of their scores.

Whenever a request arrives to retrieve the next best pair from this source, the best pair (o_u, o_v) is determined at line 1 of the `getNextBestPair()` as follows. The next best pair is either the top pair of the heap (which is initially empty) or the top pair of the list. The best pair (o_u, o_v) is reported and the next best guest $o_{v'}$ of o_u is determined and the pair $(o_u, o_{v'})$ is inserted in the heap. Note that the size of heap is $O(T)$ when $T < N$, hence the cost of returning the best pair is $O(Log\ T)$ for $T < N$. As before, the heap cannot have more than N pairs even when $T > N$, so the cost remains $O(Log\ N)$ in this case.

4.2 External Memory Source

The basic idea of the external memory algorithm is same as the internal memory algorithm. However, the heap cannot be stored in the internal memory. For this reason, we use external memory priority queue proposed by Arge [2]. 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 [2].

The main challenge in creating and maintaining an external memory source is to determine the next best guest of an object without accessing the external memory. Recall that when a pair (o_u, o_v) is retrieved from the heap, the next best guest $o_{v'}$ of o_u is determined and the pair $(o_u, o_{v'})$ is inserted in the heap. The object table and the sorted list of objects cannot be stored in the internal memory. Hence a straight forward approach would access the sorted list from the external memory to create the new pair $(o_u, o_{v'})$. Clearly, the overall I/O cost is prohibitive in this case.

To better illustrate the challenge, we demonstrate a failed attempt to solve this problem. Consider the example of Fig. 4.3 where the scoring function is the sum of the attribute values. Fig. 4.3(a) shows the initial state of the source where for each object, a pair with its best guest has been created and inserted in the heap. The best pair (o_1, o_2) is retrieved from the heap and is reported. In order to determine the next best guest of o_1, we need to know the adjacent object of o_2. This would require an I/O. One attempt to address this problem is to store the adjacent object information of o_u and o_v with every pair (o_u, o_v) during the creation of the source. Below, we show that this does not solve the problem.

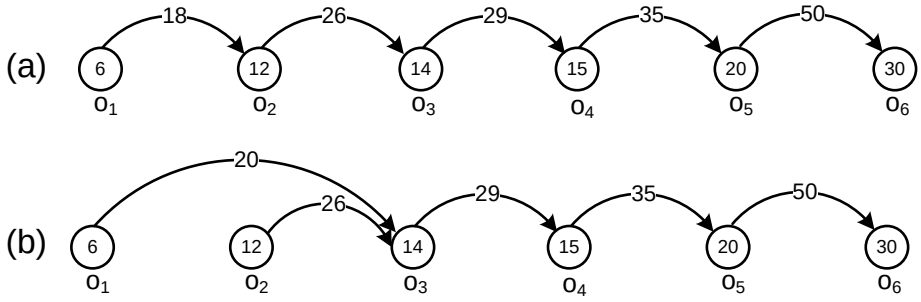


Figure 4.3: The challenge in maintaining a source

Consider that during the creation of the source, when a pair (o_u, o_v) is inserted in the heap we also attach the information (IDs and attribute values) of adjacent objects of o_u and o_v with the pair. Consider again the example of Fig. 4.3. The best pair (o_1, o_2) is retrieved and reported. The next best guest of o_1 is the adjacent object of o_2. Since the adjacent object information of o_1 and o_2 was attached with the pair, the algorithm determines that the next best guest of o_1 is the adjacent object of o_2. Hence, a new pair (o_1, o_3) is created and inserted in the heap (see Fig. 4.3 (b)). When the main algorithm sends a request to retrieve the next best pair, the pair (o_1, o_3) is found at top of the heap and is reported. Note that now the next best guest of o_1 cannot be determined without accessing the external memory because during the creation of the pair (o_1, o_3), we did not attach adjacent object information with o_3 because this information was not available.

Before we present a solution which does not need to access external memory to find the best guest of an object, we introduce the notion of dummy pair. A dummy pair with host o_u and guest o_v is denoted by $(\overline{o_u, o_v})$. The pairs (o_u, o_v) we introduced earlier are called regular pairs hereafter. There are following two differences in how we treat the regular pairs and the dummy pairs.

1. 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 best hosts and next best hosts is similar as described in previous section.

2. 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 3`
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 shows the details of creating and maintaining the source. During the creation of source, for each object o_i a pair $(\star o_i, o_j)$ is inserted in the heap where o_j is its best guest. A dummy pair $(\overline{o_k, \star o_i})$ is also inserted in the heap where o_k is the best host of o_i. Fig. 4.4(a) shows the source where for each object o_i, a pair with its best guest and a dummy pair with its best host is created. The scoring function is the sum of attribute values. The regular pairs are shown with curved arrows pointing right and the dummy pairs are shown with connector style arrows pointing left.

When the main algorithm sends a request to return the next best pair, the source retrieves the top two pairs from the heap. We can prove that if the top pair is $(\star o_u, o_v)$ then the next top pair is its dummy pair $(\overline{o_u, \star o_v})$ (Lemma 3). The pair $(\star o_u, o_v)$ is reported and a new pair $(\star o_u, o_{v'})$ is inserted in the heap where $o_{v'}$ is the next best guest of o_u. This can be determined without accessing the external memory because the dummy pair $(\overline{o_u, \star o_v})$ stores the adjacent object information for o_v. Similarly, the next best host $o_{u'}$ of the object o_v is determined and a dummy pair $(\overline{o_{u'}, \star o_v})$ is inserted in the heap. The next best host can also be determined without accessing the external memory because the regular pair $(\star o_u, o_v)$ stores the adjacent object information of o_u.

The key observation is that when a pair $(\star o_u, o_v)$ is accessed from the heap its dummy pair $(\overline{o_u, \star o_v})$ is present in the heap and is the next best pair in the heap. We need to modify the heap preference function in order to guarantee that the algorithm is correct (Lemma 2 and 3) when there are more than one pair in the heap with same score.

Modifying the heap priority function. We modify the heap such that if two pairs have same score, the heap gives priority to the pairs based on their guest objects. More specifically, if the scoring function is right increasing function then the pair with smaller ID of the guest object is given preference. If the scoring function is right decreasing then the pair with 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 same score and same guest object then the heap gives priority based on their host objects. More specifically, if the scoring function is left increasing function then the heap prefers the pair with larger ID of the host object. If the function is left decreasing function then the heap prefers the pair with smaller ID of the host object.

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

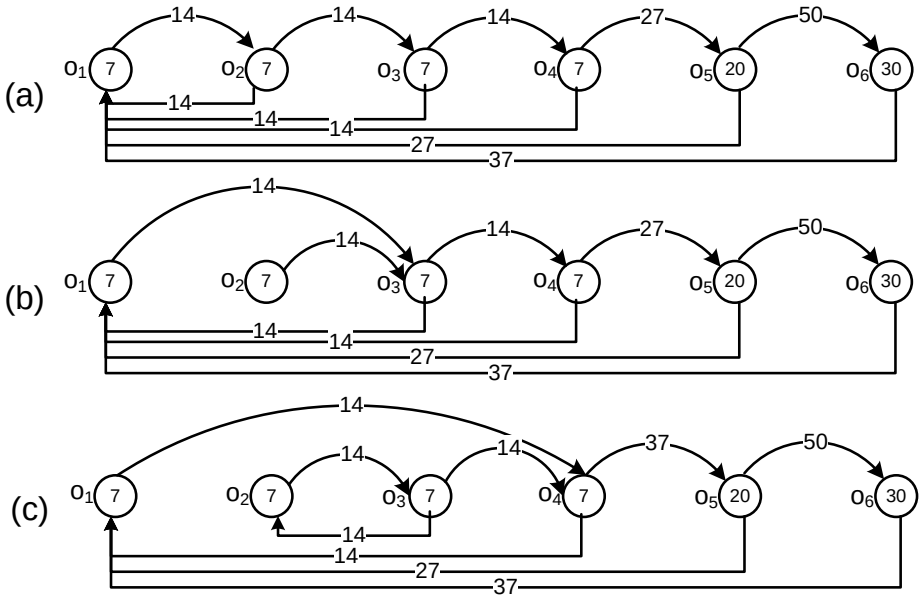


Figure 4.4: Illustration of Algorithm 2

Example 2: Consider the example of Fig. 4.4 where the regular pairs are shown with curved arrows (from left to right) and the dummy pairs are shown with connector style arrowed lines (from right to left). The scoring function is the sum of attribute values which is right increasing and left decreasing function. So, the heap prefers the pair with smaller guest IDs and smaller host IDs.

Fig. 4.4(a) shows the initial state after the source is created (e.g., for each object, a pair is created with its best guest and a dummy pair is created with its best host). The smallest score is 14 and the pairs with the smallest score are $(\star o_1, o_2), (\star o_2, o_3), (\star o_3, o_4), (\overline{o_1, \star o_2}), (\overline{o_1, \star o_3})$ and $(\overline{o_1, \star o_4})$.

Recall that the heap gives priority to the pairs with smaller IDs of guests if more than one pair have same score. Hence, the top pair is $(\star o_1, o_2)$ and the second top pair is $(\overline{o_1, \star o_2})$. The next best guest of o_1 is o_3 which is determined by the adjacent object information of o_2. The pair $(\star o_1, o_3)$ is inserted in the heap (see Fig. 4.4(b)). There does not exist a next best host for the guest o_2 so we do not insert a new dummy pair.

The heap still contains few pairs with the same score. As the heap gives priority to the pairs with smaller guest IDs, the pairs $(\star o_2, o_3), (\star o_1, o_3)$ and $(\overline{o_1, \star o_3})$ are preferred over others. To break the tie between these pairs, the heap gives priority to the pairs with smaller host IDs. Hence $(\star o_1, o_3)$ and its dummy pair $(\overline{o_1, \star o_3})$ are the top two pairs.

The pair $(\star o_1, o_3)$ is reported and the next best guest of o_1 is determined to be o_4. The pair $(\star o_1, o_4)$ is inserted in the heap. The next best host of o_3 is o_2, so a dummy pair $(\overline{o_2, \star o_3})$ is created and is inserted in the heap (Fig. 4.4 (c)). The next top two elements in the heap are $(\star o_2, o_3)$ and its dummy pair $(\overline{o_2, \star o_3})$. The algorithm continues in this way. ■

It is important to note that the algorithm may not work if the heap priority function is not changed in the way we proposed. Consider the example of Fig. 4.4(a) where several pairs have same score. If the heap priority is set such that the pair (o_2, o_3) is the top pair, the next top pair cannot be its dummy pair because it does not exist in the heap. Below, we prove the correctness of our algorithm if the heap is modified as we proposed.

The correctness of the algorithm follows from Lemma 2 and 3. We prove the lemmas for a loose monotonic function that is right increasing and left decreasing function (e.g., the sum function used in Example 2). The proofs for other functions (e.g., right increasing and left increasing, right decreasing and left increasing, right decreasing and left decreasing) are similar.

Recall that for right increasing and left decreasing functions the heap gives priority to a pair with smaller ID of the guest if two pairs have same score. If the pairs have the same score and same guest object, the heap gives priority to the pair with smaller ID of the host. To avoid complex notations, we do not show the star with the pairs.

LEMMA 2 : If a dummy pair $(\overline{o_u, o_v})$ is the top pair of the heap then its regular pair (o_u, o_v) has already been retrieved from the heap.

PROOF.

Assume that $(\overline{o_u, o_v})$ is the top pair. If the object o_u does not have any pair $(o_u, o_{v'})$ in the heap it implies that it has hosted o_v (i.e., (o_u, o_v) has been retrieved from the heap). If there exists a pair $(o_u, o_{v'})$ in the heap and $v' < v$, then $s(o_u, o_{v'}) \leq s(\overline{o_u, o_v})$ because the function is right increasing. This contradicts that $(\overline{o_u, o_v})$ is the top pair because the heap would prefer $(o_u, o_{v'})$ (even if the score $s(o_u, o_v) = s(o_u, o_{v'})$, the heap would prefer the pair $(o_u, o_{v'})$ because it has a guest with smaller ID).

If $v' = v$, this means that the regular pair (o_u, o_v) exists in the heap and a dummy pair cannot be the top pair in presence of its regular pair. If $v' > v$, this implies that the pair (o_u, o_v) has already been retrieved because, for a right increasing function, the pair of o_u with its guests that have smaller IDs are considered first (e.g., in Example 2, (o_1, o_2) is considered before (o_1, o_3)).

LEMMA 3 : If a pair (o_u, o_v) is the top pair of the heap then its dummy pair $(\overline{o_u, o_v})$ has already been created and is present in the heap (it implies that the dummy pair is the second top pair).

PROOF. We prove this by contradiction. Assume that $(\overline{o_u, o_v})$ is not present in the heap. At any stage, each host o_v contains a dummy pair $(\overline{o_{u'}, o_v})$ in the

heap. If no such pair exists or $u' > u$ then this implies that the dummy pair $(\overline{o_u, o_v})$ has already been retrieved which violates Lemma 2. If $u' < u$ then $(\overline{o_{u'}, o_v})$ would be the top pair instead of (o_u, o_v). This is because $s(o_{u'}, o_v) \leq s(o_u, o_v)$ and the dummy pair $(\overline{o_{u'}, o_v})$ would be preferred even when $s(o'_u, o_v) = s(o_u, o_v)$ because the heap prefers a pair with smaller host ID.

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 [24].

5 Query Processing Algorithms

In the previous section, we presented efficient techniques to create and maintain internal memory and external memory sources. As stated in Section 3, the existing top-k algorithms (e.g., FA, TA and NRA) can be applied to combine the scores of pairs on different sources to obtain the top-k pairs. In this section, we present the techniques to combine the scores to answer the top pairs queries. We also conduct the complexity analysis and show that the expected performance of the algorithms is optimal when two or less attributes are involved in the query.

5.1 Score-based Top-k Pairs Queries

We apply the threshold algorithm (TA) to combine the scores from different sources and return the top-k pairs. However, as stated in Section 2.2, TA assumes that the sources support random accesses. 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 other attributes as follows.

For internal memory algorithms, we assume that the objects are stored in main memory (this consumes $O(dN)$ memory space). When a pair (o_u, o_v) is seen in 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. This is equivalent to doing a random access on the sources.

For the external memory algorithm, doing random access requires accessing the object table (which exists in the external memory). This would be quite expensive because we need to look up attribute values of two objects for each seen pair which may require two I/Os. One solution is to apply NRA algorithm because it does not require random accesses. However, as discussed in Section 2.2, FA and NRA algorithms require unbounded buffers and the main memory consumption may be prohibitively large (it may be $O(V)$ where V is the total number of valid pairs).

To address this issue, we modify each source S_i such that each pair stores d attribute values of both of the objects in it. This increases the amortized I/O cost of the external priority queue 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.

Analysis

The number of elements accessed by TA is always less than or equal to the number of elements accessed by FA [11]. FA algorithm stops the sorted accesses when exactly k elements are returned from all d sources under the sorted accesses. 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 [9].

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 other sources. Total number of accesses from all d sources is $O(dT)$. As showed earlier, the cost of accessing a pair from a source is $O(Log\ N)$, hence the total expected cost for the internal memory algorithm is;

$$O(dT\ Log\ N) = O(d\ V^{\frac{d-1}{d}}k^{\frac{1}{d}}Log\ N) \tag{5.1}$$

For non-chromatic queries the total number of valid pairs is $O(N^2)$. Hence the expected cost of our algorithm to answer two dimensional closest pair query is $O(N\ Log\ N)$ which is optimal in algebraic decision tree model [3].

The cost of 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 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 [24].

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 $O(k + 2dB)$.

5.2 Skyline Pairs Query

For ease of presentation, we assume that all 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 FA. However, we address the problem of unbounded buffer. Our algorithm works as follows.

1 . Do 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[1]. Otherwise, discard it.

[1]In our implementation, we use main memory R-tree to index the existing skyline pairs. The newly seen pairs are compared to existing skyline pairs using the R-tree. We observed that

2. Terminate when at least one object has been seen under sorted accesses from all 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 an object is seen in all sources, every pair that has not been seen in any source is dominated by it.

If more than one pair 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 source S_i and may have smaller scores in all other sources. We address this as follows. Let x_i be the score of a pair p that has been accessed from a source S_i. If it is dominated by the existing skyline pairs we discard it. Otherwise, we insert it in a list C which contains 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 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 all newly seen pairs have score on S_i larger than 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 [18] query returns every element that is dominated by at most $(k-1)$ other elements. A k-dominant skyline [5] 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.

Analysis

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

For each retrieved pair, we compare it with all existing skyline pairs. The average number of skyline pairs is estimated to be $O(Log^{d-1}V)$ [4]. 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 same as the cost of score-based top-k ($k = 1$) pairs query because the cost of score-based top-k queries was obtained using the number of accesses by FA.

A lower bound on the cost of skyline pairs queries can be obtained by reducing the closest pair query to it. If the size of skyline is larger than $O(N\ Log\ N)$ then the skyline query cannot be answered in $O(N\ Log\ N)$. Otherwise if the size of skyline is smaller and the skyline pairs query can be answered in less

this approach performs significantly better than comparing the new pair with every existing skyline pair.

than $O(N\ Log\ N)$, the closest pair query can be answered by scanning the set of skyline pairs once. This would mean that the closest pair can be answered in less than $O(N\ Log\ N)$ which contradicts the lower bound of closest pair queries. Therefore, $O(N\ Log\ N)$ is a lower bound on the skyline pairs queries.

It is easy to see that the expected cost of our algorithms meets the lower bound for the queries that involve two attributes. 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 $O(k + 2dB + Log^{d-1}N)$.

5.3 Rank-based Top-k Pairs Queries

When a pair p is seen on a source S_i, although its score on other sources can be determined, it might not be possible to determine its rank on the other sources. However, if a pair p is seen under 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 using NRA because the sorted accesses are possible but the random accesses are not possible.

As mentioned earlier, 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 previously seen pairs are to be updated. This problem has been addressed by LARA algorithm [15] which we briefly described in Section 2.2. The second problem is that NRA uses 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.

Analysis

In the worst case, the growing phase of LARA (see Section 2.2) completes when there are at least k elements that are seen on all 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 $O(k\ Log^{d-1}N)$ [26]. 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, the algorithm is expected to see k elements that have been returned by all sources when T elements have been accessed from each source. 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 sources is $O(d^2T)$. The cost of each access in shrinking phase is $O(Logk + 2^d)$ [15]. Hence the expected total cost of the shrinking phase is $O(d^2T(Log\ N + Log\ k + 2^d))$.

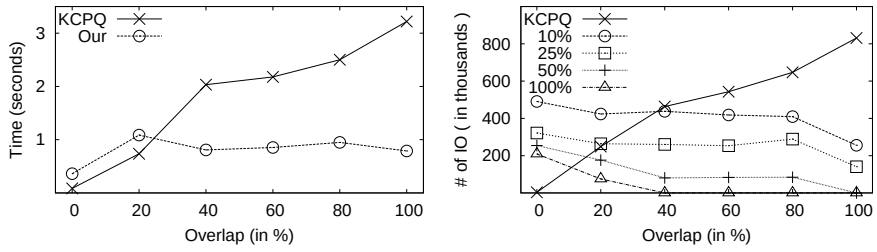
The total cost of the internal memory rank-based top-k pairs query is the sum of the cost of growing phase and the cost of shrinking phase computed above. The expected amortized I/O cost of the external memory algorithm is $O(\frac{d^2T}{B} Log_{\frac{M}{B}} \frac{N}{B})$ because d^2T pairs are expected 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 $O(2dB + k\ Log^{d-1}\ N)$.

6 Experiments

In this section, we evaluate the performance of our algorithms. We conducted extensive experiments on both real and synthetic datasets. Due to space limitation, we present only the most representative results. There does not exist any previous work for the rank-based top-k pairs queries and the skyline pairs queries. However, there exists several algorithms for k closest pairs queries which is a special case of the score-based top-k pairs queries. Moreover, naïve algorithms for the rank-based and skyline 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 the score-based top-k algorithm. At the end of this section, we show the performance of the other two algorithms.

The score-based top-k algorithm is compared with state of the art k closest pairs query algorithm (KCPQ) [7]. We use both real and synthetic datasets and compare our algorithm with KCPQ. In accordance with [7], the page size for both algorithms is set to $1K$. We generated several synthetic datasets following different data distributions. The default datasets follow uniform distribution. The k closest pair 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.



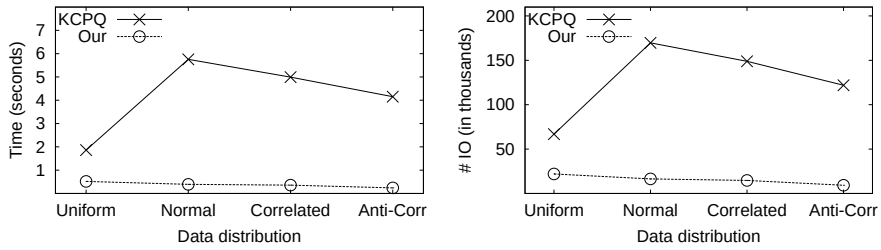
(a) Internal Memory (b) External Memory

Figure 6.1: Effect of overlapping

It has been noted that the overlap between the datasets is one of the main factors [7] affecting the performance of the existing algorithms. Fig. 6.1 shows the effect of overlap on the algorithms. In Fig. 6.1(a), we run both algorithms

in internal memory and observe that our algorithm is 2 to 3 times faster when the overlap is more than 40%. For smaller overlaps, the performance of KCPQ is better because most of the intermediate nodes of R-trees are quickly pruned. However, its performance is still not significantly better than our algorithm. Moreover, our algorithm is not sensitive to the data overlap.

Fig. 6.1(b) shows the performance of both algorithms in 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 buffer size set to $100\%, 50\%, 25\%$ and 10% of the memory used by KCPQ. Fig. 6.1(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.



(a) Internal Memory (b) External Memory

Figure 6.2: Different data distributions

We also conducted experiments on different data distributions. More specifically, we generated datasets following uniform, normal, correlated and anti-correlated distributions. For each distribution, we generated two datasets with 50% overlap between them. Fig. 6.2 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 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 small.

For the general scoring functions, we compare our algorithms with a naïve algorithm. The naïve score-based top-k 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 the external memory source is set to 2 pages (this is the minimum required by the external priority queue [2]).

The real dataset[1] 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 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 average rent of the houses in the residential block. All attributes are normalized to a unit space. The global

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

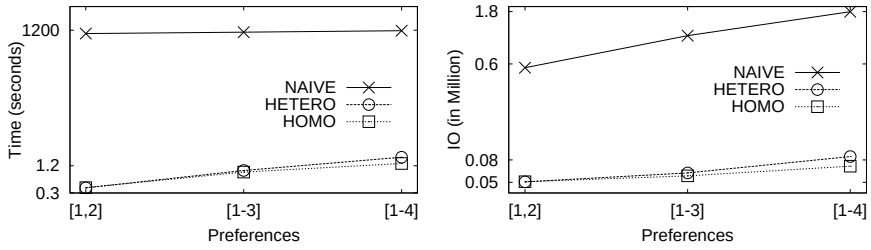
scoring function we used is sum of the local scores.

We use several heterochromatic and homochromatic queries involving two to four attributes. Table 6.1 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 same suburb) that are far from each other and have high total population. k is set to 10 for all queries.

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

Table 6.1: The queries used on real data

Fig. 6.3 compares the performance of our algorithm for heterochromatic and homochromatic top-k queries involving two to four attributes. Naïve algorithm is three order of magnitude slower than our internal memory algorithm and uses an order of magnitude more IO than our external memory algorithm. The query time for our algorithm is low which demonstrates the applicability of our approach in real world applications. Similar results were observed when queries were run under different parameters (e.g., different k).



(a) Internal Memory (b) External Memory

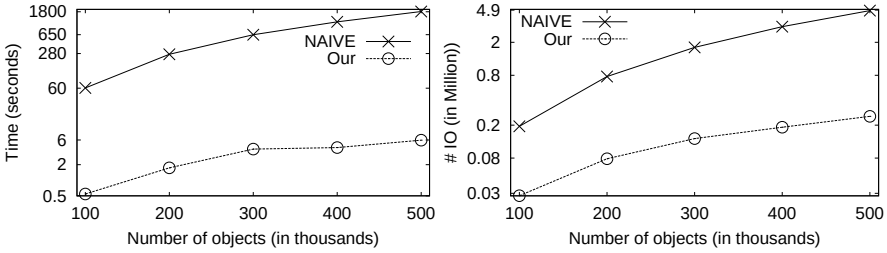
Figure 6.3: Real data

To further evaluate the performance, we study the effect of different parameters using the synthetic data sets. The default dataset contains the points following 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 algorithm 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 assigned a random weight. We present the results for homochromatic top-k queries. Non-chromatic and heterochromatic queries follow similar trends. Table 6.2 shows the default parameters.

Fig. 6.4 studies the effect of number of objects. The performance of each algorithm degrades with the increase in number of objects. This is because the number of possible pairs increases with the dataset size.

Parameter	Range
Number of objects (×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

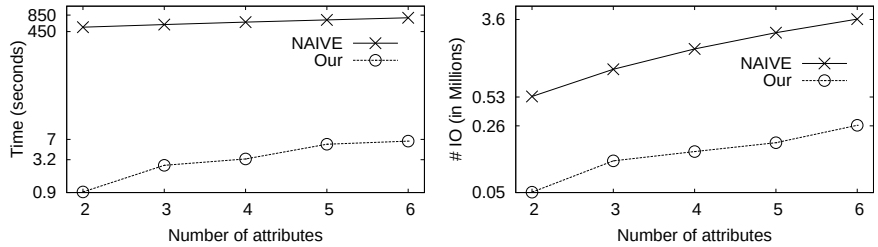
Table 6.2: Experiment Parameters



(a) Internal Memory (b) External Memory

Figure 6.4: Effect of number of objects

Fig. 6.5 studies the effect of number of attributes involved in the query. As expected, the performance of our algorithm degrades when more attributes are used in the top-k pairs queries. The I/O cost of naïve algorithm also increases significantly. This is because the number of disk pages that store the objects increases when each object has more attributes.



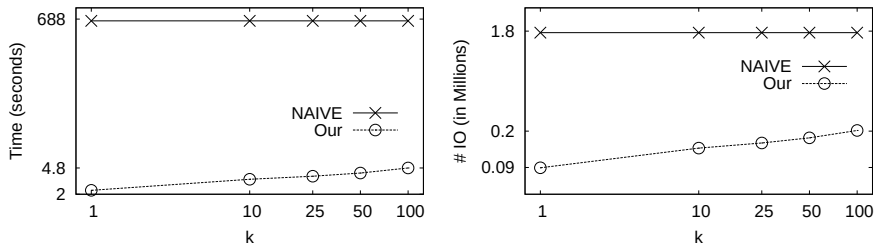
(a) Internal Memory (b) External Memory

Figure 6.5: Effect of number of attributes

Fig. 6.6 studies the effect of k. The performance of our algorithm is better for smaller k. The naïve algorithm is not affected by k because it considers all pairs regardless the value of k.

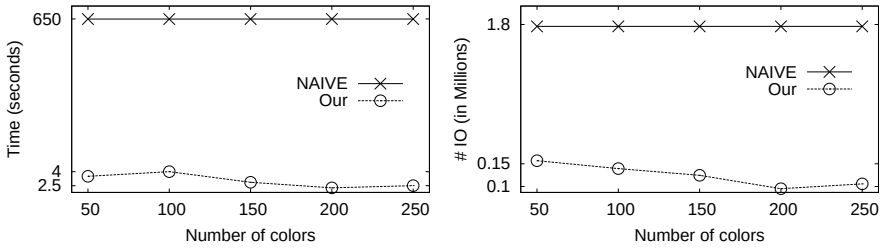
Fig. 6.7 studies the effect of number of colors. The performance of our algorithm is slightly better when the number of colors is large. This is mainly because the number of valid pairs decreases when the number of colors is large. However, the effect is not very significant because the number of pairs that are accessed from each source is not significantly affected.

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



(a) Internal Memory (b) External Memory

Figure 6.6: Effect of k



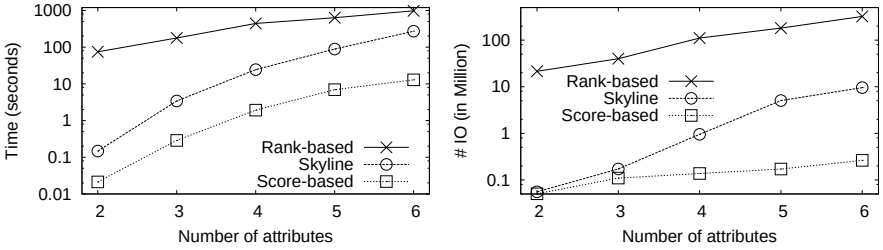
(a) Internal Memory (b) External Memory

Figure 6.7: Effect of number of colors

Fig. 6.8 shows cost of the three queries when different number of attributes are involved in the query. Score-based top-k queries are easiest to solve among the three and the rank-based top-k pairs queries are the hardest. The cost of all the queries increase with the number of attributes used in the query.

7 Conclusion

In this paper, we present a unified approach to answer a broad class of top-k pairs query including k closest pairs queries, k furthest pairs queries and their variants. We are also first to study the problem of rank-based top-k pairs queries and the skyline pairs queries. The expected performance of the proposed algorithms is optimal when the queries involve two or less attributes. Our approach does not require any pre-built indexes and is parallelizable.



(a) Internal Memory (b) External Memory

Figure 6.8: Comparison of different top pairs queries

Bibliography

[1] F. Angiulli and C. Pizzuti. An approximate algorithm for top-k closest pairs join query in large high dimensional data. *Data Knowl. Eng.*, 53(3):263–281, 2005.

[2] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[3] M. Ben-Or. Lower bounds for algebraic computation trees (preliminary report). In *STOC*, pages 80–86, 1983.

[4] 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*, 25(4):536–543, 1978.

[5] 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 Conference*, pages 503–514, 2006.

[6] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.

[7] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD Conference*, pages 189–200, 2000.

[8] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.

[9] R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.

[10] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD Conference*, pages 301–312, 2003.

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

[12] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, 2000.

[13] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, 1998.

[14] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top- query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[15] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung. Efficient top- aggregation of ranked inputs. *ACM Trans. Database Syst.*, 32(3):19, 2007.

[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] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, 1999.

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

[19] S. Qiao, C. Tang, H. Jin, S. Dai, and X. Chen. Constrained k-closest pairs query processing based on growing window in crime databases. In *ISI*, pages 58–63, 2008.

[20] J. S. Salowe. Enumerating interdistances in space. *Int. J. Comput. Geometry Appl.*, 2(1):49–59, 1992.

[21] J. Shan, D. Zhang, and B. Salzberg. On spatial-range closest-pair query. In *SSTD*, pages 252–269, 2003.

[22] M. Smid. Closest-point problems in computational geometry. In *Handbook on Computational Geometry (Editors: J. Sack and J. Urrutia), published by Elsevier Science*, 1997.

[23] L. H. U, N. Mamoulis, and M. L. Yiu. Continuous monitoring of exclusive closest pairs. In *SSTD*, 2007.

[24] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33:2001, 2001.

[25] C. Yang and K.-I. Lin. An index structure for improving nearest closest pairs and related join queries in spatial databases. In *IDEAS*, pages 140–149, 2002.

[26] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu. Probabilistic skyline operator over sliding windows. In *ICDE*, pages 1060–1071, 2009.