A Unified Framework for Answering k Closest Pairs Queries and Variants

Muhammad Aamir Cheema, Xuemin Lin, Haixun Wang, Jianmin Wang and Wenjie Zhang

Abstract—Given a scoring function that computes the score of a pair of objects, a top-k pairs query returns k pairs with the smallest scores. In this paper, we present a unified framework for answering generic top-k pairs queries including k-closest pairs queries, k-furthest pairs queries and their variants. Note that k-closest pairs query is a special case of top-k pairs queries where the scoring function is the distance between the two objects in a pair. We are the first to present a unified framework to efficiently answer a broad class of top-k queries including the queries mentioned above. We present efficient algorithms and provide a detailed theoretical analysis that demonstrates that the expected performance of our proposed algorithms is optimal for two dimensional data sets. Furthermore, our framework does not require pre-built indexes, uses limited main memory and is easy to implement. We also extend our techniques to support top-k pairs queries on multi-valued (or uncertain) objects. We also demonstrate that our framework can handle *exclusive top-k pairs queries*. Our extensive experimental study demonstrates effectiveness and efficiency of our proposed techniques.

Index Terms—Closest pairs queries, furthest pairs queries, top-k queries, multi-valued objects, uncertain objects

1 INTRODUCTION

Given a scoring function $s(o_u, o_v)$ that computes the score of a pair of objects (o_u, o_v), a top-k pairs query returns k pairs with the smallest scores among all possible pairs of objects. k closest pairs queries, k furthest pairs queries and their variants are some well studied examples of top-k pairs queries that rank the pairs on distance functions.

The problems of k closest (or most similar) pairs queries, k furthest pairs queries and their variants have received significant research attention from the computational geometry community (see [1] for a nice survey) and the database community [2], [3], [4], [5]. However, all of the existing techniques focus on a specific problem and there does not exist a unified approach that can answer different variants of top-k pairs queries (e.g., different L_p distances, generic scoring functions etc.). We are the first to provide a unified framework that supports a broad class of top-k pairs queries including the queries mentioned above.

An interesting variation of top-k pairs queries 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. Such top-k pairs queries can be used for *pair-trading* [6]. Pair-trading is a market neutral strategy according to which two correlated stocks that follow same day-to-day price movement (e.g., Coca-Cola and Pepsi) may be used to earn profit when the correlation between them weakens, i.e., one stock goes up and the other goes down. The profit can be earned by buying the under-performing stock and selling it when the divergence between the two stocks returns to normal. A top-k pairs query can be issued to obtain the pairs of stocks that are correlated (e.g., they belong to the same business sector and have similar fundamentals such as market caps, dividends etc.) and display different trends.

Consider another example of a sales company. The manager might want to retrieve two salespersons who make similar sale (i.e., the total cost of their sold items is similar) but receive very different salaries. Suppose that the relevant information is stored in a table named `worker`. The manager may issue the following query to retrieve the top-k pairs of such salespersons.

```
Q1: select a.id, b.id from worker a, worker b
where a.id <> b.id
order by
|a.sale - b.sale| - |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 salespersons with larger difference in their salaries and smaller difference in their sales.

While the example shows a simple ranking criterion, in the real applications, the users may define more sophisticated scoring functions. Our framework supports a generic ranking function that is based on *local* and *global* functions. Specifically, the function that computes the score of a pair of objects on a single attribute is called a local scoring function and the function that computes the final score of a pair (by combining the local scores) is called the global scoring function. Our framework supports any global scoring function that is *monotonic* and any local scoring function that is *loose monotonic*. 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. In the above example, the local scoring functions are $|a.sale - b.sale|$ and $-|a.salary - b.salary|$. The global scoring function is the sum of the local scores.

Our proposed framework also supports more generalized top-k pairs queries such as *chromatic* and *non-chromatic* top-

- *Muhammad Aamir Cheema is with Clayton School of Information Technology, Monash University, Australia. E-mail: aamir.cheema@monash.edu*
- *Xuemin Lin and Wenjie Zhang are with the School of Computer Science and Engineering, University of New South Wales, Australia. E-mails: {lxue, zhangw}@cse.unsw.edu.au*
- *Haixun Wang is with Google Research. E-mail: haixun@google.com*
- *Jianmin Wang is with Tsinghua University China and Tsinghua National Laboratory for Information Science and Technology, China E-mail: jimwang@tsinghua.edu.cn*

k pairs queries. Suppose that each object in the database has been assigned a color. A chromatic top-k pairs queries considers only the pairs of objects that meet certain color requirement. In contrast, a non-chromatic top-k pairs query does not consider the colors of the objects (i.e., all pairs are considered). The *chromatic* queries are further classified into *homochromatic* and *heterochromatic* top-k pairs queries. 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.

In the example of salespersons, assume that the user wants to consider only the pairs of salespersons 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 Q1. We remark that a *bichromatic* query is a special case of heterochromatic queries where the number of colors is restricted to two. Although there exist techniques to solve bichromatic k closest pairs queries [2], [3], their extension to heterochromatic queries is either non-trivial or inefficient.

Below, we summarize our contributions in this paper.

1. We are the first to propose a unified framework for a broad class of top-k pairs queries including k-closest pairs queries, k-furthest pairs queries and their chromatic variants. Some features of our proposed framework include low memory consumption, no requirement of pre-built data structure and easy implementation.

2. We conduct extensive theoretical analysis to evaluate the performance of the proposed algorithms and show that the expected performance is optimal when the number of attributes involved is two or less.

3. Our extensive experimental study demonstrates a significant improvement over the existing best known solution for k closest pairs query. For generic top-k pairs queries, a comparison with a naïve algorithm demonstrates up to three orders of magnitude improvement.

This paper is an extended version of [7] and we make the following additional contributions in this version.

4. We formally define top-k pairs queries on multi-valued (or uncertain) objects and present efficient techniques based on non-trivial lower bounds. Extensive experimental study demonstrates the efficiency of our proposed techniques and effectiveness of the lower bounds.

5. We extend our techniques to efficiently answer *exclusive top-k pairs queries* (defined in Section 4.2) and provide experimental results to demonstrate the effectiveness of our proposed optimizations.

The rest of the paper is organized as follows. We formally define the problem and give an overview of the most relevant work in Section 2. Section 3 describes our framework and presents techniques to answer top-k pairs queries on regular (single-valued) objects. In Section 4, we present our techniques to answer top-k pairs queries on multi-valued objects and exclusive top-k pairs queries. Our extensive experimental study is given in Section 5 followed by conclusion in Section 6.

2 PRELIMINARIES

2.1 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 $f(s_1(a, b), \cdots, s_d(a, b))$.

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.

2.2 Related Work

The problem of k closest pairs queries has received significant research attention by the computational geometry community (see [1] for a nice survey). Hjaltason et al. [2] are the first to study the problem of closest pairs in the context of spatial databases. While the proposed solution has a nice feature that it returns the pairs incrementally, its priority queue size may be prohibitively large. Shin et al. [8] also propose algorithms for incremental distance join and demonstrate that their algorithms outperform the techniques in [2].

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).

Shen et al. [9] study the top-k pairs queries over sliding windows. We remark that their focus is on efficiently updating the results of top-k queries for the data streams. In contrast, we focus on efficiently computing the initial results of the top-k queries. Zhang et al. [10] study the similarity joins on multi-valued objects. Similar to this work, they use ϕ-quantile score to define the queries. In contrast to this work, they only consider Euclidean distance whereas this work allows a broad class of scoring functions.

Top-k queries retrieve the top-k objects based on a user defined scoring function. The problem has been extensively studied [11], [12], [13], [14], [15], [16]. Ilyas et al. [17] 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 threshold algorithm (TA) which is a well known algorithm and is used in our techniques.

Threshold Algorithm (TA). TA (independently proposed in [18], [15], [19]) 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. Let W_k be the largest of the scores of the objects maintained in the heap.

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 $t \geq W_k$. Report the objects in the heap as top-k objects.

3 Techniques

3.1 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., TA) view these sources as the ranked inputs and can be used to retrieve the top-k pairs by combining these ranked inputs.

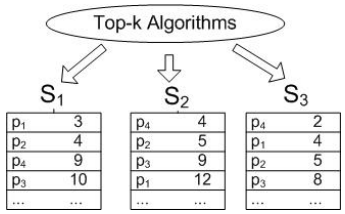


Fig. 1. Our framework

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 straightforward solution to create a source S_i is to sort all 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 straightforward approach may be prohibitive.

In Section 3.2, we present an internal memory optimal algorithm to create and maintain such sources. The internal memory algorithm uses $O(N)$ space and is optimal in time complexity. An external memory I/O optimal algorithm can also be developed using the ideas similar to internal memory algorithm. However, due to the space limitations, we omit the details and refer the readers to the conference version of this paper [7].

Our proposed techniques have various advantages such as known main memory requirement, no need for pre-built indexes, optimality for two-dimensional data sets, and ease of implementation and extension (see [7] for details).

3.2 Maintaining The Sources

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[1] $s(o_u, o_v) = s(o_v, o_u)$. 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 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 (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 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.

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)).

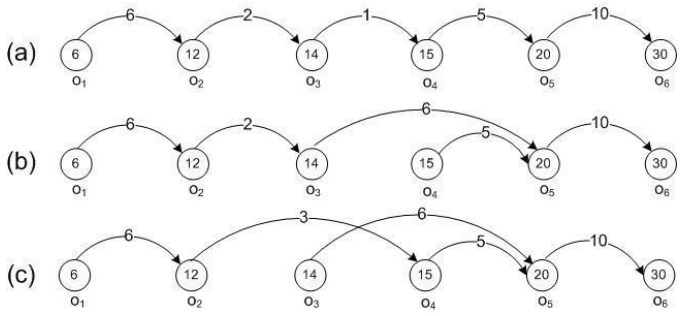


Fig. 2. Illustration of Algorithm 1

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. For a formal proof, see Lemma 1 in [7].

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 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).

3.2.1 *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 2.1 for the definitions).

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 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 $u < v$.

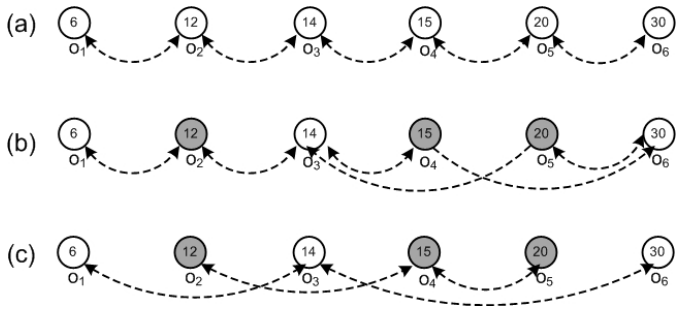


Fig. 3. Adjacent objects (a) Non-chromatic (b) Heterochromatic (c) Homochromatic

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.

3.2.2 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.

3.2.3 Finding the adjacent objects

Now we illustrate how to add pointers to the adjacent objects in $O(N)$. 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.

3.2.4 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 [20]. 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 $O(\log N)$.

3.3 Query Processing Algorithm

3.3.1 Technique

We apply the threshold algorithm (TA) [18], [15], [19] to combine the scores of a pair from different sources and return the top-k pairs. Recall that TA assumes that the sources support the random accesses (see Section 2.2). In other words, when a pair is returned from a source S_i, TA needs to obtain its score on every other attribute. 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. We use our external memory algorithm (see [7]) when the available memory is less than $O(dN)$.

Incrementally returning top-k pairs. Incremental algorithms return the results one-by-one, i.e., the partial results are incrementally reported to the users without waiting for all the

results to be computed. Next, we show that we can modify TA such that it reports top-k pairs incrementally.

Recall that TA maintains W_k which corresponds to the largest of the scores of the objects maintained in the heap (see Section 2.2). The algorithm terminates when the threshold t becomes at least equal to W_k. Let W_i be the i-th largest score of the objects maintained in the heap. We modify TA such that it starts by setting $i = 1$ and reports the best pair as soon as $t \geq W_1$. The algorithm continues by iteratively incrementing i by one and reporting the i-th best pair as soon as $t \geq W_i$. It can be easily shown that the complexity of this variation of TA is the same as that of the original TA.

3.3.2 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) [18]. FA 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 [21].

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[2] is given by Eq. (1).

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

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 [20].

4 Extensions

4.1 Top-k Pairs Queries on Multi-Valued Objects

An object having multiple instances is called a multi-valued object [22], [10]. Multi-valued objects exist in many real world applications. For instance, a real estate development company may evaluate different towns by modelling each town as a multi-valued object such that each residential property in the town is its instance that has several attributes such as its price, household income, and number of people living in the property etc. Similarly, the performance of a basketball player in a game may be measured by his statistics (scores, assists, rebounds, steals, blocks etc.) and may be treated as an instance of the player; consequently, each player has a set of instances [22] where each instance corresponds to his performance in a particular game. In this section, we formally define top-k pairs queries on multi-valued objects[3] and propose efficient query processing techniques.

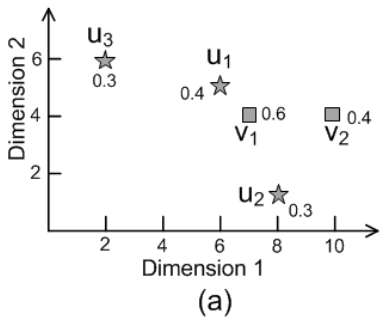
4.1.1 Problem Definition

Similar to existing research on multi-valued objects [22], [10], we define top-k pairs based on ϕ-quantile scores. Let $U = \{u_1, \cdots, u_m\}$ denote a multi-valued object with m instances where each instance u_i is a multi-dimensional object and $w(u_i)$ denotes its weight s.t. $\sum_{i=1}^{m} w(u_i) = 1$. Given two multivalued objects $U = \{u_1, \cdots, u_m\}$ and $V = \{v_1, \cdots, v_n\}$, the weight of a pair of instances $p = (u_i, v_j)$ s.t. $u_i \in U$ and $v_j \in V$ is $w(p) = w(u_i) \times w(v_j)$ and the score of the pair is denoted as $p.score$ which is computed using a given scoring function (e.g., by applying the global scoring function as stated in Section 2).

Aggregated weight of a pair of instances. Let $L = \{p_1, \cdots, p_{m \times n}\}$ be the list of all possible pairs of instances of U and V sorted in ascending order of their scores, i.e., $p_i.score \leq p_j.score$ for $i < j$. The aggregated weight of p_j (denoted as $p_j.aggW$) is $p_j.aggW = \sum_{i=1}^{j} w(p_i)$.

ϕ-quantile score. Given a value ϕ ($0 < \phi \leq 1$), the ϕ-quantile score of the pair of multivalued objects (U, V) is the score of the first pair p_j in L in the sorted order such that $\sum_{i=1}^{j} w(p_i) > \phi$. In other words, ϕ-quantile score of (U, V) is the score of the pair p_j such that $p_j.aggW > \phi$ and $p_j.aggW - w(p_j) \leq \phi$. Such a pair p_j is called pivot pair of (U, V).

Note that 0.5-quantile score represents the median[4] of the scores of all possible pairs between the instances of U and V. When the score of each pair of instances (u, v) is computing using the global scoring function $SCORE(u, v)$ as described in Section 2, its ϕ-quantile score is denoted as $SCORE_\phi(U, V)$.



Pair	Score	weight
(u_1, v_1)	2.0	0.24
(u_2, v_1)	4.0	0.18
(u_2, v_2)	5.0	0.12
(u_1, v_2)	5.0	0.16
(u_3, v_1)	7.0	0.18
(u_3, v_2)	10.0	0.12

(b)

Fig. 4. Illustration of ϕ-quantile score

EXAMPLE 2 : Consider the example of Fig. 4(a) where two multi-valued objects $U = \{u_1, u_2, u_3\}$ and $V = \{v_1, v_2\}$ are shown in a two-dimensional space (along with the weight of each of their instances). Assume that the score of a pair of instances (u_i, v_j) is the Manhattan distance between them (i.e., local scoring function is the absolute difference and the global scoring function is the summation). Fig. 4(b) shows the list of

2. 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). Our experimental results also include the cost of creating the sources.

3. We remark that uncertain objects can also be modelled as multi-valued objects where the weight of each instance corresponds to its occurrence probability. Hence, our techniques can also be applied to answer top-k pairs queries on uncertain objects.

4. http://wikipedia.org/wiki/Median#Medians_of_probability_distributions

TABLE 1
Notations

Notation	Definition
U	a multi-valued object
u_i	instance of a multi-valued object U
$w(u_i)$	weight of an instance
p	a pair of instances (u, v)
$w(p)$ or $w(u, v)$	weight of a pair of instances (u, v)
$p.score$	global score of a pair of instances
$p.score[i]$	ith local score of a pair of instances
(U, V)	a pair of multi-valued objects
$SCORE_\phi(U, V)$	ϕ-quantile global score of (U, V)
$S_\phi[i](U, V)$	ϕ-quantile local score of (U, V) in ith dimension
$LB\_SCORE_\phi(U, V)$	a lower bound on $SCORE_\phi(U, V)$
$LB\_S_\phi[i](U, V)$	a lower bound on $S_\phi[i](U, V)$

all possible pairs sorted on their scores along with the weights of the pairs. The aggregated weight of the pair (u_2, v_2) is 0.54. The 0.5-quantile score of (U, V) is $SCORE_{0.5}(U, V) = 5$ which is the score of the pair (u_2, v_2). Note that the pair (u_2, v_2) is the pivot of (U, V) when $\phi = 0.5$. The 0.8-quantile score is $SCORE_{0.8}(U, V) = 7$ and the pivot corresponds to (u_3, v_1).

Top-k Pairs of Multi-Valued Objects. Given a set of multi-valued objects O and a quantile value ϕ $(0 < \phi \leq 1)$, a 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 $(A', B') \notin P$, $SCORE_\phi(A, B) \leq SCORE_\phi(A', B')$.

4.1.2 Solution Overview

In this section, we provide an overview of our solution for the top-k pairs query over multi-valued objects. For simplicity of presentation, we focus on non-chromatic queries and assume that the local scoring function is the absolute difference of the values (i.e., $|a - b|$). However, by using similar ideas, our techniques can be immediately applied to chromatic queries and any other local scoring function that is loose monotonic. We choose absolute difference as the local scoring function because it is a loose monotonic function but not a monotonic function and is more challenging to handle for this reason. First, we define a few terms and notations.

The local score of a pair of instances $p = (u, v)$ in i^{th} dimension is denoted as $p.score[i]$, i.e. $p.score[i] = |u[i] - v[i]|$. The ith local ϕ-quantile score of a pair of multi-valued object (U, V) is computed using $p.score[i]$ of the pairs of instances (i.e., L is sorted on $p.score[i]$) and is denoted as $S_\phi[i](U, V)$.

Pair	S[1]	weight
(u_1, v_1)	1.0	0.24
(u_2, v_1)	1.0	0.18
(u_2, v_2)	2.0	0.12
(u_1, v_2)	4.0	0.16
(u_3, v_1)	5.0	0.18
(u_3, v_2)	8.0	0.12

(a)

Pair	S[2]	weight
(u_1, v_1)	1.0	0.24
(u_1, v_2)	1.0	0.16
(u_3, v_1)	2.0	0.18
(u_3, v_2)	2.0	0.12
(u_2, v_1)	3.0	0.18
(u_2, v_2)	3.0	0.12

(b)

Fig. 5. Local ϕ-quantile scores

EXAMPLE 3 : Continuing the example of Fig. 4(a), Fig. 5(a) and Fig. 5(b) show the pairs of instances of U and V in sorted order of their local scores in first and second dimensions, respectively. Recall that the ith local score of a pair of instances (u, v) is $|u[i] - v[i]|$. 0.8-quantile local scores are $S_{0.8}[1](U, V) = 5$ (pivot pair is (u_3, v_1)) and $S_{0.8}[2](U, V) = 3$ (pivot pair is (u_2, v_1)).

Similar to our framework presented in Section 3.1, we divide the problem into d one-dimensional problems and then apply a variation of TA algorithm to compute top-k pairs of multi-valued objects. To achieve this, we present Lemma 1 that ensures that the global ϕ-quantile score $SCORE_\phi(U, V)$ of a pair (U, V) can be lower bounded by using its $\frac{\phi}{d}$-quantile local scores $S_{\phi/d}[i](U, V)$ in each of d dimensions. We use $LB\_SCORE_\phi(U, V)$ to denote a lower bound on $SCORE_\phi(U, V)$ and $LB\_S_{\phi/d}[i](U, V)$ to denote a lower bound on $S_{\phi/d}[i](U, V)$. Table 1 shows the notations used throughout this paper.

LEMMA 1 : Let $LB = f(LB\_S_{\frac{\phi}{d}}[1](U, V), \cdots, LB\_S_{\frac{\phi}{d}}[d](U, V))$ where f is the global scoring function and $LB\_S_{\frac{\phi}{d}}[i]$ is the ith $\frac{\phi}{d}$-quantile local score of (U, V). LB is a lower-bound of $SCORE_\phi(U, V)$.

Proof: We prove this by contradiction. Assume that $SCORE_\phi(U, V) < LB$. This implies that the total weight of the pairs of instances that have $p.score$ smaller than LB is larger than ϕ. Note that a pair p can have a global score smaller than LB if and only if its local score $p.score[i]$ in at least one dimension i is smaller than $LB\_S_{\phi/d}[i](U, V)$. By definition of $S_{\phi/d}[i](U, V)$, in a given dimension i, the total weight of such pairs is at most ϕ/d. This implies that the total weight of all such pairs in d dimensions is at most ϕ. Hence, the total weight of the pairs of instances that have $p.score$ smaller than LB is at most ϕ which contradicts the assumption. ■

Remark. Replacing $LB\_S_{\phi/d}[i]$ with $LB\_S_\phi[i]$ in Lemma 1 does not give a correct lower bound. Consider the example of Fig. 4 and 5 where $SCORE_{0.8}(U, V) = 7$, $S_{0.8}[1](U, V) = 5$ and $S_{0.8}[2](U, V) = 3$. $LB = 5 + 3 = 8$ which is not less than $SCORE_{0.8}(U, V)$. Note that our algorithm computes $LB\_S_{\phi/d}[i]$ on-the-fly and does not require pre-computation (for details, see Section 4.1.3).

Framework. We create and maintain d sources such that 1) each source S_i *incrementally* returns the pair (U, V) with the smallest $LB\_S_{\frac{\phi}{d}}[i]$ (i.e., a sorted access) in $O(\log N)$ where N is the total number of multi-valued objects; and 2) for a given pair of multi-valued objects (U, V), $LB\_S_{\phi/d}[i](U, V)$ can be computed in $O(1)$ (i.e., a random access). The details on how to construct the sources are presented in Section 4.1.3. A variation of TA algorithm is used to compute the top-k pairs of multi-valued objects as shown in Algorithm 2.

The algorithm accesses each source in a round-robin fashion. A pair (U, V) returned by a source S_i is possibly among the top-k pairs. Therefore, we need to compute its ϕ-quantile global score $SCORE_\phi(U, V)$ and update the list of top-k pairs accordingly (lines 7 and 8). $SCORE_\phi(U, V)$ is computed using Algorithm 3 (to be described in Section 4.1.4).

Algorithm 2 Top-k Algorithm

1: **for** each source S_i in round-robin fashion **do**
2: access next pair $(U,V) \in S_i$ with smallest $LB\_S_{\phi/d}[i]$
3: $L[i] = LB\_S_{\phi/d}[i](U,V)$
4: **if** p was never seen in any other source **then**
5: compute $LB\_SCORE_\phi(U,V)$ /* `using Lemma 1` */
6: **if** $LB\_SCORE_\phi(U,V) <$ score of k-th best pair **then**
7: compute $SCORE_\phi(U,V)$ /* `Algorithm 3` */
8: update top-k pairs and k-th best pair if required
9: $t = f(L[1], \cdots, L[d])$
10: **if** $t \geq$score of k-th best pair **then**
11: **return** top-k pairs

Since computing $SCORE_\phi(U,V)$ is expensive, our algorithm tries to avoid this computation. Specifically, if (U,V) has already been seen in any other source then we do not need to compute $SCORE_\phi(U,V)$ because it has already been considered (line 4). If the pair has not been seen in any other source, we compute the lower bound $LB\_SCORE_\phi(U,V)$ by doing random access on each other source and applying Lemma 1. If $LB\_SCORE_\phi(U,V)$ is larger than the score of k-th best pair seen so far then (U,V) cannot be among the top-k pairs. Hence, the computation of $SCORE_\phi(U,V)$ is not required (line 6).

Similar to the standard TA, the algorithm stops when the best possible score of any unseen pair (i.e., threshold) cannot be smaller than the score of k-th best pair. Specifically, let $L[i]$ denote the lower bound local score of the last pair accessed from the source S_i (line 3). Clearly, the best possible ϕ-quantile global score of any unseen pair is at least $t = f(L[1], \cdots, L[d])$. Therefore, if t is not smaller than the score of k-th best pair, the algorithm terminates by reporting top-k pairs (lines 9 to 11).

Space complexity. Let N be the total number of objects and m be the average number of instances in a multi-valued object. In the next section, we show that creating and maintaining a source requires sorting *critical* and *terminal* instances of all N objects along with an interval tree and a heap that contain at most N elements. Hence, $O(N)$ storage is required to create and maintain each source. The memory required for all d sources is $O(dN)$. A naïve algorithm to compute the exact score of a pair of objects at line 7 of Algorithm 2 requires storing at most $O(m^2)$ pairs of instances. Note that these $O(m^2)$ pairs can be deleted as soon as the exact score is computed. Although (in practice) our algorithm to compute the exact score (Algorithm 3) requires significantly smaller space, in the worst case, its space complexity is also $O(m^2)$. Furthermore, the main algorithm maintains k best pairs of multi-valued objects. Hence, the worst case space complexity of Algorithm 2 is $O(dN + m^2 + k)$.

4.1.3 Creating and Maintaining Sources

In this section, we describe how to create and maintain a source S_i. Similar to our solution in Section 3.2, we need to sort the instances of all multi-valued objects to create the sources. After this sorting operation that takes $O(Nm \log m + N \log N)$ (as explained later in this section), every sorted access (i.e., retrieve next best pair according to $LB\_S_{\phi/d}[i]$) takes $O(\log N)$ and every random access (i.e., compute $LB\_S_{\phi/d}[i]$ of a given pair of objects) takes $O(1)$ where N is the total number of multi-valued objects and m is the average number of instances in each multi-valued object. First, we describe how to compute a lower bound $LB\_S_{\phi/d}[i](U,V)$ for a given pair (U,V).

LEMMA 2 : Let L denote the list of every pair (u_i, v_j) where $u_i \in U$ and $v_j \in V$. Let $L' \subseteq L$ be the set of pairs such that the total weight of the pairs of instances in L' is at least $(1-\phi/d)$. Let $p \in L'$ be a pair with the smallest ith local score in L', i.e., there is no pair $q \in L'$ s.t. $q.score[i] < p.score[i]$. $p.score[i]$ is a lower bound on $S_{\phi/d}[i](U,V)$.

Proof: Note that the total weight of the pairs in $L - L'$ is at most ϕ/d. Hence, for every pair p' for which $p'.aggW > \phi/d$, there must exist at least one pair $p \in L'$ for which $p.score[i] < p'.score[i]$ (otherwise $p'.aggW \leq \phi/d$). This implies that the pivot pair cannot have a score smaller than $p.score[i]$, i.e., $S_{\phi/d}[i](U,V)$ cannot be smaller than $p.score[i]$. ■

EXAMPLE 4 : Consider the example of Fig. 5(a) and assume that $\phi = \phi/d = 0.8$ (assuming $d = 1$ for simplicity in this particular example). Note that $S_{0.8}[1](U,V) = 5$. A lower bound on $S_{0.8}[1](U,V)$ is obtained as follows. Without loss of generality, assume that L' contains (u_1, v_2) and (u_3, v_1). The total weight of these pairs is $0.34 > (1 - \phi/d)$. The score of pair (u_1, v_2) is 4 which is smaller than the score of (u_3, v_1). Therefore, $LB\_S_{0.8}[1](U,V) = 4$. We remark that L' can contain any combination of pairs as long as the total weight is at least $(1 - \phi/d)$.

Before we present Lemma 3 that provides an efficient way to construct L' and $LB\_S_{\phi/d}[i](U,V)$, we introduce a few terms.

Terminal instance. For a multivalued object U, its *terminal* instance in i^{th} dimension is the instance with the largest value in ith dimension and is denoted as u_t.

Critical instance. Let $\{u_1, \cdots, u_m\}$ denote the instances of a multi-valued object U sorted in ascending order of their values in ith dimension. The critical instance of U is an instance u_c such that $\sum_{j=c}^{m} w(u_j) \geq 1 - \phi/d$ and $\sum_{j=c+1}^{m} w(u_j) < 1 - \phi/d$.

EXAMPLE 5 : Fig. 6 shows two multi-valued objects U and V with 4 and 5 instances, respectively. The weight of each instance $u_i \in U$ is 0.25 and the weight of each $v_i \in V$ is 0.2. The terminal instances of U and V are u_4 and v_5, respectively, and are shown with circles having thick boundaries. Assuming $(1 - \phi/d) = 0.6$, the critical instances of U and V are u_2 and v_3, respectively (shown with shaded circle).

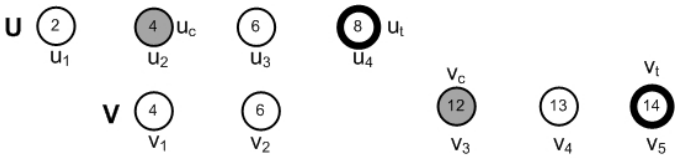


Fig. 6. Critical and terminal instances

LEMMA 3 : Let U and V be two multi-valued objects such that $u_t \leq v_t$ where u_t and v_t are the terminal instances of U and V, respectively. $LB\_S_{\phi/d}[i](U,V) = max(0, v_c - u_t)$ where v_c is the critical instance of v.

Proof: When $v_c \leq u_t$, the lower bound score is 0 which does not require a proof because the score (i.e., the absolute difference) cannot be less than 0. We prove the lemma for the case when $v_c > u_t$. Let L' be the set that contains every pair (u_i, v_j) where u_i is *any* instance in U (i.e., $u_i \in U$) and v_j is either a critical instance of V or the instance on the right side of critical instance (i.e., $v_j \in V$ for $j \geq c$). In Fig. 6, L' contains every pair between $\{u_1, \cdots, u_4\}$ and $\{v_3, v_4, v_5\}$. Clearly, the total weight of the pairs in L' is at least $1 - \phi/d$. This is because the sum of weights $v_j \in V$ for $j \geq c$ is at least $1 - \phi/d$. According to Lemma 2, the pair with the minimum score in L' is a lower bound. It is easy to verify that the pair (u_t, v_c) has the minimum score (i.e., $|u_t - v_c|$) in L'. ■

Now, we are ready to present the details of how to create and maintain the sources. The algorithm is the same as Algorithm 1 except the details of sorting the objects and computing best guest and next best guest of an object U. Next, we briefly describe these details using the example of Fig. 7 where five multi-valued objects U, V, W, X and Y are shown. The terminal instances of the objects are shown above the critical instances of the objects (and are sorted according to their values).

Sorting. The instances of each object are sorted in ascending order of their values in ith dimension. The terminal and critical instances of each object are also identified (this can be done using a linear scan on the sorted instances of each object). If the total number of objects is N and each object has m instances, this step takes $O(Nm \log m)$. Then, the following two sorted lists are created as shown in Fig. 7: 1) *terminal list* contains the terminal instances of the objects sorted according to their values; 2) *critical list* contains the critical instances sorted on their values. The elements in critical instances are connected according to the sorted order (see the arrows shown in broken lines). The cost of creating these lists is $O(N \log N)$. We also construct an interval tree [23] that indexes the interval $[u_c, u_t]$ for each multi-valued object U. The construction cost of the interval tree is $O(N \log N)$.

Finding best guest of U. An object U serves as a host to an object V only if $u_t \leq v_t$. An object V is called eligible guest of an object U if $u_t \leq v_t$. In Fig. 7, U will be the host of the objects V, X and Y. The best guest V of an object U is the object with minimum $LB\_S_{\phi/d}[i](U, V) = max(0, v_c - u_t)$. Hence, the best guest of U is an object V which has minimum v_c among all eligible guests of U. We consider the following two cases.

1. $v_c \leq u_t$**.** Note that every eligible guest V for which $v_c \leq u_t$ has $LB\_S_{\phi/d}[i](U, V) = 0$. Since for every such object V, $v_c \leq u_t$ and $v_t > u_t$, the interval $[v_c, v_t]$ overlaps with u_t. To efficiently identify such objects, we issue a *stabbing query* [23] on the interval tree that indexes $[v_c, v_t]$ for each multi-valued object V in our data set. A stabbing query returns every interval that overlaps a given value.

In Fig. 7, the best guest of object W is the object U (and $LB\_S_{\phi/d}[i](W, U) = 0$) which is returned by the stabbing query (the interval $[u_c, u_t] = [4, 8]$ overlaps $w_t = 6$). The cost of a stabbing query is $O(x + \log N)$ where x is the number of results. If $x > 1$, we suspend the stabbing query as soon as we find one object satisfying the query (i.e., the cost is at most $O(\log N)$). The stabbing query is resumed when the next best guest of U is to be found (as we describe later in this section). The suspension and resumption of the stabbing query requires just a single pointer pointing to the relevant node in the interval tree. Hence, the storage complexity does not increase.

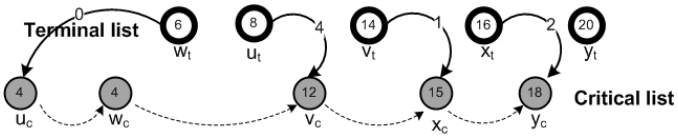


Fig. 7. Computing best guest and next best guest

2. $v_c > u_t$**.** If the stabbing query does not return any object then it means there is no eligible guest with $v_c \leq u_t$. In this case, the best guest is identified as follows. Every object V for which $v_c > u_t$ is an eligible guest because $v_t > u_t$. Hence, a binary search on critical list can be conducted to find the object with minimum v_c among the objects that have $v_c > u_t$. This takes $O(\log N)$. In Fig. 7, the best guest of object U is V because 1) the stabbing query does not return any object; and 2) the object with minimum v_c among the objects for which $v_c > u_t$ is the object V. Hence, (U, V) with $LB\_S_{\phi/d}[i](U, V) = 4$ is inserted.

Fig. 7 shows the best guest object for each object by drawing an arrow from the host object to the guest object (the number on an arrow denotes $LB\_S_{\phi/d}[i]$). These four pairs are inserted in the heap when the source is initialized (see line 4 of Algorithm 1).

Finding next best guest of U. Let V be the current best guest of U. The next best guest of U can be determined as follows. If V was obtained using a stabbing query (case 1 above), the stabbing query is resumed to obtain another object V' satisfying the query. If such an object is found it is selected as the next best guest. If the stabbing query does not return any object, a binary search on critical list is conducted (as in case 2 above) to find the next best guest of U. In the case when V was obtained using a binary search, then an object V' is the next guest where V' is the object that has critical instance v'_c adjacent to v_c in the sorted list. Clearly, the cost of this operation is at most $O(\log N)$.

In Fig. 7, assume that the current pair (W, U) was retrieved and we want to find the next best guest of W. The stabbing query does not return any other object. Hence, a binary search is conducted as in case 2 and the object V is determined as the next best guest with $LB\_S_{\phi/d}[i](W, V) = |v_c - w_t| = 6$. If the current pair (U, V) is retrieved, the next best guest of U is X because x_c is adjacent to v_c in the critical list and $LB\_S_{\phi/d}[i](U, X) = |x_c - u_t| = 7$.

Remark. Note that the cost of sorted access does not depend on the number of instances m of the multi-valued objects. At each sorted access, the top pair from the heap is retrieved and the next best pair is determined in $O(\log N)$ and inserted in the heap. The size of the heap is $O(N)$. Hence, the cost of a sorted access is $O(\log N)$.

4.1.4 Computing $SCORE_\phi(U, V)$

In this section, we describe how to compute the score $SCORE_\phi(U, V)$ of a pair of objects (U, V) requested at line 7

of Algorithm 2. The algorithm applies a variation of TA on d sources that maintain the pairs of instances according to their local scores. Note that these sources are not the sources that we described in Section 4.1.3. The sources in Section 4.1.3 allow sorted and random accesses on pairs of *multi-valued objects* whereas Algorithm 3 requires sorted and random accesses on pairs of *instances* of a given (U, V). These sources are created using the techniques we presented in Section 3.2.

Algorithm 3 Compute $SCORE_\phi(U, V)$

Input: (U, V) and the value of ϕ
Output: $SCORE_\phi(U, V)$
1: initialize an empty list L
2: insert a dummy pair p in L with $p.aggW = 0$, $w(p)$=0 and $p.score = \infty$ and call it pivot
3: **for** each source S_i in round-robin fashion **do**
4: retrieve a pair p from S_i
5: $s[i] = p.score[i]$
6: **if** p was never seen in any other source **then**
7: Do random access on all sources to compute $p.score$
8: **if** $p.score \leq pivot.score$ **then**
9: $pivot.aggW = pivot.aggW + w(p)$
10: insert p in L in sorted order
11: **while** $pivot.aggW - w(pivot) > \phi$ **do**
12: $pivot.aggW = pivot.aggW - w(pivot)$
13: $pivot \leftarrow$ the pair on the left of pivot in sorted L
14: $t = f(s[1], \cdots, s[d])$
15: **if** $t \geq pivot.score$ or $t \geq$ score of k-th best pair **then**
16: **return** $pivot.score$

The algorithm initializes an empty list L that will maintain the accessed pairs in ascending order of their global scores (i.e., $p.score$). The algorithm will maintain a $pivot$ that always points to a pair p in L for which $pivot.aggW > \phi$ and $pivot.aggW - w(pivot) \leq \phi$. If at any stage, $pivot$ does not satisfy this condition, we say that $pivot$ has become invalid in which case we update $pivot$ to point to a pair that satisfies the condition. Note that if all pairs are considered, $pivot.score$ is $SCORE_\phi(U, V)$.

Initially, a dummy pair p is inserted with $p.aggW = 0$, $p.score = \infty$ and $w(p) = 0$. This dummy pair is selected as pivot (line 2). The algorithm proceeds with doing sorted accesses on each source S_i in a round robin fashion. For each accessed pair p from a source S_i, if p was never accessed before, its score $p.score$ is computed by doing random accesses on each other source (line 7). If $p.score > pivot.score$ then p is ignored. Otherwise, $p.score \leq pivot.score$ which indicates that the weight of p contributes to $pivot.aggW$ in which case $pivot.aggW$ is incremented by $w(p)$ (line 9). The pair p is inserted in L in sorted order (in case of tie between p and $pivot$, p is inserted before $pivot$) (line 10). Since $pivot.aggW$ has changed, we need to check whether $pivot$ is valid or invalid. If the $pivot$ has become invalid we compute a new $pivot$ by sequentially traversing the list L from $pivot$ towards the leftmost pair in L until we find a pair p that satisfies the pivot condition. This pair is set as $pivot$ and the algorithm continues (lines 11 to 13).

At the end of each round, we compute a threshold t as in TA algorithm that serves as a lower bound score for any unseen pair. At the end of a round, if $t > pivot.score$, it indicates that none of the unseen pair can contribute to $pivot.aggW$. Hence, the algorithm terminates after reporting $pivot.score$ which corresponds to $SCORE_\phi(U, V)$.

Note that exact computation of $SCORE_\phi(U, V)$ is not required if (U, V) is guaranteed not to be among the top-k pairs of multi-valued objects. Therefore, the algorithm can safely terminate if $pivot.score$ is not smaller than the score of k-th best pair of *multi-valued objects* seen so far (lines 14 to 16).

Remark. The cost of Algorithm 3 degrades as ϕ increases because the cost depends on the number of pairs that have $p.score < SCORE_\phi(U, V)$ and $SCORE_\phi(U, V)$ increases with ϕ. We improve the performance of Algorithm 3 for larger ϕ as follows. Note that the ϕ-quantile score can be converted to an identical $(1 - \phi)$-quantile score which is computed using the list L sorted in descending order of $p.score$ (instead of ascending order). Hence, when $\phi > 0.5$, we compute $(1-\phi)$-quantile score by applying ideas similar to Algorithm 3 and maintaining the list L in descending order of $p.score$.

4.2 Exclusive Top-k Pairs Queries

Consider a set S containing top-k pairs. We say that an object o_u satisfies the exclusiveness constraint if o_u appears *at most* once in the set of pairs S, i.e., there exists at most one pair related to o_u in S.

An exclusive top-k pairs (ETP) query retrieves k pairs with the smallest scores such that every object o_u satisfies the exclusiveness constraint. In such queries, if the best pair returned by the query is (o_u, o_v) then the algorithm continues retrieving the remaining top-$(k-1)$ pairs by ignoring the objects o_u and o_v and the pairs involving these two objects. The traditional top-k pairs queries studied earlier in this paper are called inclusive top-k pairs queries throughout this section.

An ETP query has many interesting applications. For example, an exclusive k closest pairs query can be issued to solve car-parking assignment problem [24] where each parking slot is to be reserved for *at most* one car (the car that is closest to it). Consider another example of a recruitment agent who has a list of applicants and a list of jobs. On a given day, he may want to arrange k interviews (one interview corresponds to one job-applicant pair). He may issue an exclusive top-k pairs query to retrieve k job-applicant pairs such that the scores (suitability) of the reported job-applicant pairs are better than the remaining pairs. Due to time constraints, it may not be possible for an applicant to appear in more than one interviews on a given day. Hence, the agent may issue the query such that every applicant satisfies the exclusiveness constraint.

In the above example, the agent may have no problem arranging more than one interviews for a single job in a single day. Hence, he may not require that every job also satisfies the exclusiveness constraint. Although, for the sake of simplicity, we focus to present the techniques for the case where every object is required to satisfy the exclusiveness constraint, we remark that our techniques can be easily extended to answer the queries where only a certain type of objects (e.g., applicants) are required to satisfy the exclusiveness constraints.

4.2.1 Technique

As described in Section 3.3.1, TA can be modified to return top-k pairs incrementally. We adopt this version of TA to solve ETP. As soon as ith best pair (o_u, o_v) is returned by TA, we

conduct the following three operations: 1) delete every pair that violates exclusiveness constraint, i.e., delete the pairs that involve o_u or o_v; 2) adjust the left and right adjacent objects such that no object has o_u and o_v as its left or right adjacent object; and 3) for each deleted pair (o_x, o_v), insert (o_x, o_w) in the heap where o_w is the next best guest of o_x. This step guarantees that the heap contains, for each host o_x, a pair formed with its best guest among the guests that it has not hosted earlier.

A straightforward approach to implement the first operation is to traverse through the heap to delete every pair related to o_u or o_v. The second and third operations are similar to the techniques presented in Section 3.2. Next, we present optimizations that significantly improve the performance as demonstrated by our experimental study.

Optimizations. Consider the example of Fig. 8(a). Recall from the description of Example 1 that all the pairs connected by the arrows are the pairs that have been inserted in the heap. More specifically, the pairs (o_1, o_2), (o_2, o_5), (o_3, o_5), (o_4, o_5) and (o_5, o_6) are inserted in the heap with scores 6, 8, 6, 5 and 10, respectively. Now assume that a pair containing o_5 has been reported as ith best pair by TA. The algorithm needs to delete the pairs (o_2, o_5), (o_3, o_5), (o_4, o_5) and (o_5, o_6) from the heap. Note that each deletion from the heap takes $O(\log N)$ where N is the number of pairs in the heap. Next, we present a strategy that guarantees that each object has at most two pairs related to it in the heap.

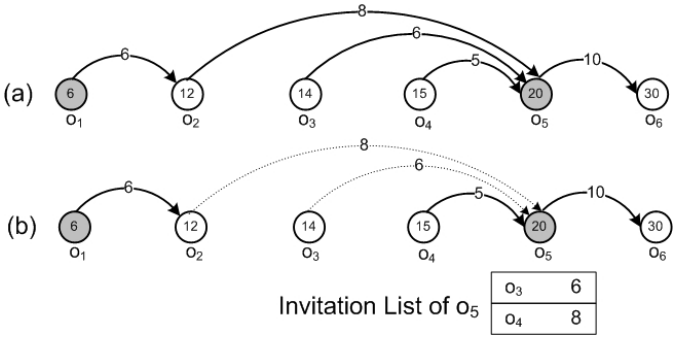


Fig. 8. Invitation List of o_5

Recall that Algorithm 1 ensures that each object o_v can have at most one pair in the heap such that o_v is the host object. However, there may be more than one pairs in the heap for which o_v is the guest object. For example, there are three pairs (o_2, o_5), (o_3, o_5), (o_4, o_5) for which o_5 is the guest object (see the incoming edges of o_5 in Fig. 8(a)). We modify Algorithm 1 such that each object o_v has at most one pair in the heap for which o_v is the guest object. In other words, even if there are more than one incoming edges of o_v, only at most one incoming edge is inserted in the heap (the one with the smallest score).

In the example of Fig. 8(b), the pair (o_4, o_5) is inserted in the heap. The other incoming edges of o_5 (shown in broken lines) are not inserted in the heap. Instead, they are inserted in a list called the *invitation list* of o_5 as shown in Fig. 8(b). The invitation list of an object o_5 records the objects that will host o_5 in future. For instance, when the pair (o_4, o_5) is reported to the main algorithm, the invitation list is used to insert a new pair (o_3, o_5) in the heap.

The invitation list of each object o_v is always kept sorted to ensure that the next pair that is to be inserted in the heap can be determined efficiently. Note that the cost of keeping the invitation list sorted is $O(\log m)$ per insertion where $m < N$ is the size of the invitation list. When an object o_v is deleted, for each object o_x in its invitation list, the next best guest o_w is determined and the object o_x is inserted in the invitation list of o_w.

4.2.2 Analysis

We provide the analysis for the simpler version of our algorithm and the cost of the optimized algorithm is always at most equal to the cost of the simpler version. Assume that the algorithm terminates after accessing Z elements from each source. Let M be the average number of valid pairs containing any object o_u. If a pair (o_u, o_v) is reported to the user, TA ignores at most $2(M-1)$ unseen pairs ($M-1$ pairs containing o_u and $M-1$ pairs containing o_v). If k pairs are reported, the number of ignored pairs is at most $2k(M-1)$. Assuming that the algorithm terminates after accessing Z pairs from each source, the number of pairs that the algorithm ignores is at most $\frac{Z \cdot 2k(M-1)}{V}$ where V is the total number of valid pairs. The algorithm stops when $T = O(V^{(d-1)/d} \cdot k^{1/d})$ have been seen from each source [21] excluding the pairs that have been ignored.

$$Z - \frac{Z \cdot 2k(M-1)}{V} = V^{(d-1)/d} \cdot k^{1/d} \tag{2}$$

The above equation can be solved to compute the value of Z. Note that Z is at most equal to V (the total number of valid pairs).

$$Z = min(V, \frac{V}{V - 2k(M-1)} \cdot V^{(d-1)/d} \cdot k^{1/d}) \tag{3}$$

For non-chromatic queries, V is at most N^2 and M is equal to $N-1$. Hence, the expected number of elements accessed from each source is $O(min(N^2, \frac{N}{N-2k} \cdot N^{2(d-1)/d} \cdot k^{1/d}))$. Note that this number is $O(N/(N-2k))$ times the number of pairs accessed for the score-based top-k pairs queries studied in Section 3.3 and $k << N$ in many real world applications.

5 Experiments

We present the results for top-k pairs queries and exclusive top-k pairs queries on single-valued objects in Section 5.1 and 5.3, respectively. Top-k pairs queries on multi-valued objects are evaluated in Section 5.2.

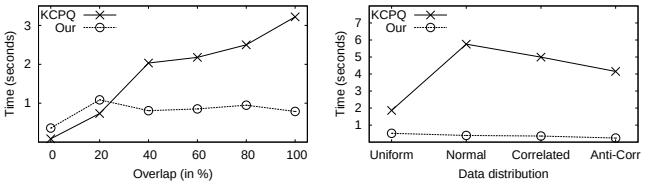
5.1 Top-k pairs queries on single-valued objects

5.1.1 k-Closest Pairs Queries

We compare our algorithm with the best known k-closest pairs algorithm called KCPQ [3]. 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 data sets is one of the main factors [3] that affect the performance

of the existing algorithms. Fig. 9(a) shows the effect of the overlap on KCPQ and our algorithm. It can be observed 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) Overlapping (b) Data distribution

Fig. 9. Effect of overlapping and data distribution

Fig. 9(b) studies the effect of different data distributions. More specifically, we generated the data sets following uniform, normal, correlated and anti-correlated distributions. For each distribution, we generated two data sets with 50% overlap between them. Fig. 9(b) 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 data sets 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.

5.1.2 Generic Scoring Functions

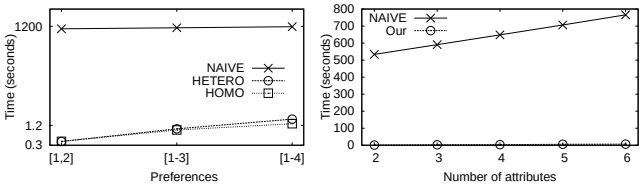
For the general scoring functions defined in Section 2, we compare our algorithms with a naïve algorithm. The naïve algorithm uses nested loop to join a data set with itself.

Real Data. The real data set consists of location data consisting of $304,895$ location points from 87 different towns in USA. Each location corresponds to the location of a residential block which has four attributes: two location coordinates, population and average rent of the properties in the block. The blocks that are in the same town are assigned the same color. We run several heterochromatic and homochromatic queries each involving two to four preferences (i.e., attributes). More details of the real data set and the scoring functions can be found in Section VI-B of [7].

TABLE 2
Parameters for regular queries

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

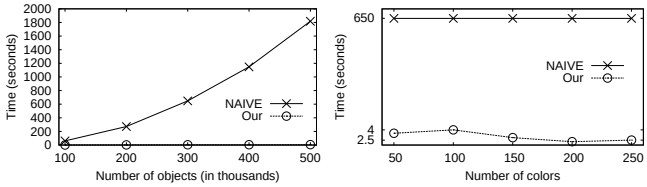
Synthetic Data. The default synthetic data set contains points following a uniform distribution. Each object is randomly assigned a color. The number of colors vary from 50 to 250. The local scoring functions used by the algorithms are the sum and the absolute difference. The global scoring function 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. For synthetic data set, we present the results for the homochromatic top-k queries. The results for the non-chromatic and the heterochromatic queries follow similar trends. Table 2 shows the default parameters in bold.



(a) Real Data (b) Synthetic Data

Fig. 10. Effect of number of attributes

Fig. 10 shows the effect of number of attributes on the queries issued on real and synthetic data sets. The naïve algorithm is up to three orders of magnitude slower than our algorithm. The query time for our algorithm is low which demonstrates the applicability of our approach in the real world applications.



(a) Data cardinality (b) Number of colors

Fig. 11. Effect of data cardinality and number of colors

Fig. 11(a) and Fig. 11(b) study the effect of increasing the number of objects and the number of colors, respectively. Our algorithm is more than an order of magnitude faster than the naïve algorithm and scales much better with the data cardinality. Fig. 11(b) demonstrates that 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.

5.2 Top-k pairs queries on multi-valued objects

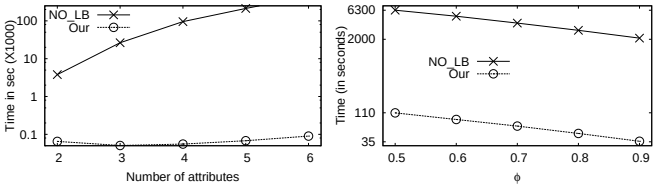
We conduct extensive experimental study on synthetic data sets. Table 3 shows the parameters where the default values are shown in bold. Specifically, we create N points following uniform distribution where each point corresponds to the center of a multi-valued object. A number m is randomly chosen between 1 and M where M demonstrates the maximum number of instances an object may have. For an object, its m instances are created following a uniform distribution such that the value of the instance in a dimension does not deviate more than $v\%$ of the whole space from the center of the object. The weights assigned to the instance follow normal distribution. We conducted experiments on various data distributions (for weights and locations of object centers and instances) and observed similar results. Note that our synthetic data sets have up to 5 Million instances in total. The local scoring function for each dimension is the absolute difference and the global scoring function is the summation of the local scores.

To the best of our knowledge, there does not exist any algorithm that answers top-k pairs queries involving generic

TABLE 3
Parameters for queries on multi-valued objects

Parameter	Range
Number of objects N	250, 1000, **2000**, 3000, 5000
Maximum # of instances M	100, 300 **500**, 700, 1000
Maximum variation v (in %)	0.25, 0.5, **1**, 2, 3
ϕ	0.5, 0.6 **0.7**, 0.8, 0.9
k	1, **10**, 25, 50, 100

scoring functions on multi-valued objects. Therefore, we considered various naïve algorithms as our competitors. However, due to the complexity of the problem, the algorithms either ran out of memory or did not return results even after a few days. Therefore, we compare our algorithm with a different version of our algorithm (denoted as *NO_LB*) that uses our framework but does not utilize the lower bound computation (i.e., it ignores lines 5 and 6 of Algorithm 2). Note that our comparison with this algorithm also demonstrates the effectiveness of our proposed lower bound.

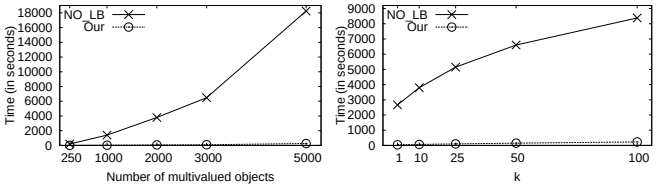


(a) Dimensionality (b) Effect of ϕ

Fig. 12. Effect of dimensionality and ϕ

Fig. 12(a) studies the effect of number of attributes (i.e., dimensionality). Our algorithm performs up to three orders of magnitude faster than *NO_LB*. Interestingly, the performance of our algorithm improves when the dimensionality is increased to 3 and it degrades as the number of attributes are further increased. This is due to the following reason. As the dimensionality increases, the probability that the multi-valued objects overlap in all dimensions is reduced. This results in improved lower bounds. On the other hand, larger dimensionality reduces the effectiveness of TA algorithm. *NO_LB* did not return results (for the data sets containing 6 attributes) even after several days and was terminated. In the rest of the experiments, the default number of attributes is 2.

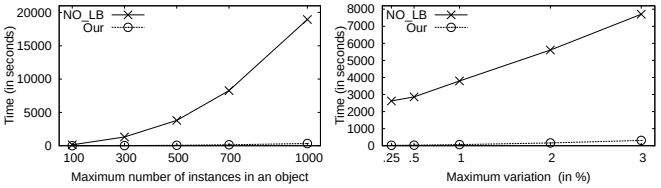
In Fig. 12(b), we vary the value of ϕ from 0.5 to 0.9 and study its affect on both algorithms. Note that a ϕ-quantile query can be converted to an identical (1-ϕ)-quantile query by changing the order in which the objects are sorted (as explained in Section 4.1.4). Hence, the results for $\phi < 0.5$ are not shown in experiments because the trend is a mirrored reflection of the reported graph. Fig. 12(b) shows that the performance improves for larger values of ϕ. This is because 1) the lower bound is more effective for larger values of ϕ; and 2) Algorithm 3 performs worse when ϕ is closer to 0.5.



(a) Number of objects (b) Effect of k

Fig. 13. Effect of data cardinality and k

Fig. 13(a) and 13(b) study the effect of number of multi-valued objects and the value of k, respectively. As expected, the running time of each algorithm increases as the number of objects or the value of k increases. However, our algorithm scales reasonably well.



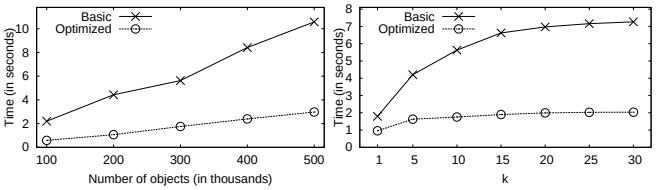
(a) Number of instances (b) Variation in values

Fig. 14. Effect of number of instances and maximum variation

Fig. 14(a) and Fig. 14(b) study the effect of number of instances and the maximum variation v. As the number of instances increases, the cost increases because the computation of global score of a pair becomes more expensive. Fig. 14(b) demonstrates that the performance degrades with the increase in the variation. This is because when v increases, the multi-valued objects have higher chance to overlap each other which negatively affects the lower bounds.

5.3 Exclusive top-k pairs queries

To the best of our knowledge, we are the first to study exclusive top-k pairs queries involving generic scoring functions. We compared our algorithm with a naïve algorithm that uses a nested loop to generate all pairs and then computes top-k pairs by applying exclusiveness constraint. We also compared our algorithm with a basic version of our algorithm denoted as *Basic* which does not utilizes the optimizations we presented in Section 4.2. Our algorithm is denoted as *Optimized*.



(a) Number of objects (b) Value of k

Fig. 15. Effect of data cardinality and k

Fig. 15(a) and Fig. 15(b) demonstrate the effect of number of objects and the value of k, respectively. As can be expected, the naïve algorithm was up to three orders of magnitude slower than our algorithm. Therefore, to clearly illustrate the effectiveness of our optimizations, we do not show the results for naïve algorithm. It can be observed that our optimizations improve the performance by up to four times.

6 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 at most two attributes. Extensive experiments demonstrate the efficiency of our proposed algorithms.

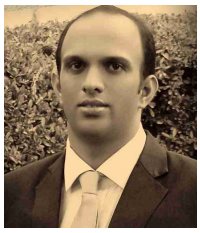
We also extend our framework to answer top-k pairs queries on multi-valued (or uncertain) objects where the score of a

pair of objects is based on the notion of ϕ-quantile score and exclusive top-k pairs queries. In future, we intend to study top-k pairs on uncertain data considering the possible worlds semantics. Another interesting direction for future work is to investigate the top-k groups queries where a group can contain two or more objects. Top-k groups queries can be useful in identifying groups of interesting objects, e.g., selecting a team of players with best overall statistics over specified attributes.

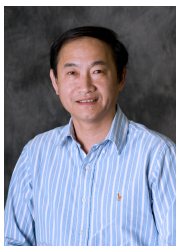
Acknowledgments. We are thankful to the anonymous reviewers for their helpful comments that have helped to significantly improve this paper. Muhammad Aamir Cheema is supported by ARC DE130101002 and DP130103405. Xuemin Lin is supported by NSFC61232006, NSFC61021004, and ARC DP120104168. The research work of Jianmin Wang is supported by NSFC61325008. Wenjie Zhang is supported by ARC DE120102144 and DP120104168.

REFERENCES

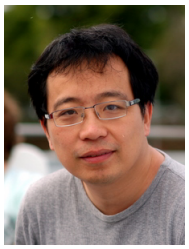
[1] M. Smid, "Closest-point problems in computational geometry," in *Handbook on Computational Geometry, published by Elsevier Science*, 1997.
[2] G. R. Hjaltason and H. Samet, "Incremental distance join algorithms for spatial databases," in *SIGMOD*, 1998.
[3] A. Corral, Y. Manolopoulos, Y. Theodoridis, 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] G. Vidyamurthy, *Pairs Trading: quantitative methods and analysis*. John Wiley & Sons, Inc., 2004.
[7] M. A. Cheema, X. Lin, H. Wang, J. Wang, and W. Zhang, "A unified approach for computing top-k pairs in multidimensional space," in *ICDE*, 2011, pp. 1031–1042.
[8] H. Shin, B. Moon, and S. Lee, "Adaptive multi-stage distance join processing," in *SIGMOD Conference*, 2000, pp. 343–354.
[9] Z. Shen, M. A. Cheema, X. Lin, W. Zhang, and H. Wang, "Efficiently monitoring top-k pairs over sliding windows," in *ICDE*, 2012.
[10] W. Zhang, L. Zhan, Y. Zhang, M. A. Cheema, and X. Lin, "Efficient top-k similarity join processing over multi-valued objects," *World Wide Web*, pp. 1–25, 2013.
[11] Z. Shen, M. A. Cheema, X. Lin, W. Zhang, and H. Wang, "A generic framework for top-k pairs and top-k objects queries over sliding windows," *IEEE Trans. Knowl. Data Eng.*, 2012.
[12] K. C.-C. Chang and S. won Hwang, "Minimal probing: supporting expensive predicates for top-k queries," in *SIGMOD Conference*, 2002.
[13] K. Mouratidis, S. Bakiras, and D. Papadias, "Continuous monitoring of top-k queries over sliding windows," in *SIGMOD Conference*, 2006.
[14] R. Fagin, "Combining fuzzy information from multiple systems," in *PODS*, 1996, pp. 216–226.
[15] S. Nepal and M. V. Ramakrishna, "Query processing issues in image (multimedia) databases," in *ICDE*, 1999.
[16] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung, "Efficient top- aggregation of ranked inputs," *ACM Trans. Database Syst.*, 2007.
[17] 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.
[18] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 614–656, 2003.
[19] U. Güntzer, W.-T. Balke, and W. Kießling, "Optimizing multi-feature queries for image databases," in *VLDB*, 2000.
[20] M. Ben-Or, "Lower bounds for algebraic computation trees (preliminary report)," in *STOC*, 1983.
[21] R. Fagin, "Combining fuzzy information from multiple systems," *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 83–99, 1999.
[22] W. Zhang, X. Lin, M. A. Cheema, Y. Zhang, and W. Wang, "Quantile-based knn over multi-valued objects," in *ICDE*, 2010, pp. 16–27.
[23] K. Mehlhorn, "Computational geometry," in *Data Structures and Algorithms 3*. Springer Berlin Heidelberg, 1984, vol. 3, pp. 79–268.
[24] L. H. U, N. Mamoulis, and M. L. Yiu, "Continuous monitoring of exclusive closest pairs," in *SSTD*, 2007.



Muhammad Aamir Cheema is a Lecturer at Clayton School of Information Technology, Monash University, Australia. He obtained his PhD from UNSW Australia in 2011. His research areas are spatio-temporal databases, mobile and pervasive computing and probabilistic databases. He is the recipient of 2012 Malcolm Chaikin Prize for Research Excellence in Engineering and 2013 Discovery Early Career Researcher Award. His PhD thesis was nominated for SIGMOD Jim Gray Doctoral Dissertation Award and ACM Doctoral Dissertation Competition. He has also won two CiSRA best research paper awards (in 2009 and 2010), two "one of the best papers of ICDE" (in 2010 and 2012), and two best paper awards at WISE 2013 and ADC 2010, respectively. He served as local organization co-chair for APWEB 2013 and workshop co-chair for MSTD 2013.

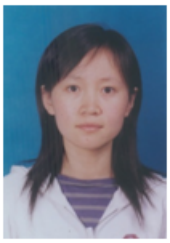


Xuemin Lin is a Professor in the School of Computer Science and Engineering, the University of New South Wales. He has been the head of database research group at UNSW since 2002. Before joining UNSW, Xuemin held various academic positions at the University of Queensland and the University of Western Australia. Dr. Lin got his PhD in Computer Science from the University of Queensland in 1992 and his BSc in Applied Math from Fudan University in 1984. During 1984-1988, he studied for PhD in Applied Math at Fudan University. He currently is an associate editor of ACM Transactions on Database Systems. His current research interests lie in data streams, approximate query processing, spatial data analysis, and graph visualization.



Haixun Wang is a staff research scientist at Google Research. Before joining Google, he was a senior researcher at Microsoft Research Asia, where he managed the database team. Haixun Wang has published more than 120 research papers in refereed international journals and conference proceedings. He won the IEEE ICDM 2013 Ten Year highest impact paper award, ER 2008 best paper award, etc. He is associate editor of Distributed and Parallel Databases (DAPD), IEEE Transactions of Knowledge and Data Engineering (TKDE), Knowledge and Information System (KAIS), Journal of Computer Science and Technology (JCST). He isPC co-Chair of CIKM 2012, ICMLA 2011, and WAIM 2011.

Jianmin Wang graduated from Peking University, China, in 1990, and got his M.E. and Ph.D. in Computer Software from Tsinghua University, China, in 1992 and 1995, respectively. He is now a professor at the School of Software, Tsinghua University. His research interests include unstructured data management, workflow & BPM technology, Enterprise Information System, benchmark for database system. He has published over 100 DBLP indexed papers in Journals, such as TKDE, TSC, DMKD, CII, DKE, FGCS, and IJIIS, and in conferences, such as VLDBSIGMOD, SIGIR, ICDE, AAAI, IJCAI, ICWS, and SAC. He has led to develop a product data/lifecycle management system (PDM/PLM), which has been implemented in hundreds enterprises in china. Nowadays, he leads to develop an unstructured data management system, LaUDMS



Wenjie Zhang is currently a lecturer in School of Computer Science and Engineering, the University of New South Wales, Australia. She received PhD in computer science and engineering in 2010 from the University of New South Wales. Since 2008, she has published more than 20 papers in SIGMOD, VLDB, ICDE, TODS, TKDE and VLDBJ. She is the recipient of Best (Student) Paper Award of National DataBase Conference of China 2006, APWebWAIM 2009, Australasian Database Conference 2010 and DASFAA 2012, and also co-authored one of the best papers in ICDE2010, ICDE 2012 and DASFAA 2012. In 2011, she received the Australian Research Council Discovery Early Career Researcher Award.