Efficiently Monitoring Top-k Pairs over Sliding Windows

Zhitao Shen[†], Muhammad Aamir Cheema[†], Xuemin Lin[*‡†], Wenjie Zhang[†], Haixun Wang[§]

[†]*The University of New South Wales, Australia*
{shenz,macheema,lxue,zhangw}@cse.unsw.edu.au
[‡]*East China Normal University, China*
[§]*Microsoft Research Asia*
haixunw@microsoft.com

***Abstract*—Top-k pairs queries have received significant attention by the research community. k-closest pairs queries, k-furthest pairs queries and their variants are among the most well studied special cases of the top-k pairs queries. In this paper, we present the first approach to answer a broad class of top-k pairs queries over sliding windows. Our framework handles multiple top-k pairs queries and each query is allowed to use a different scoring function, a different value of k and a different size of the sliding window. Although the number of possible pairs in the sliding window is quadratic to the number of objects N in the sliding window, we efficiently answer the top-k pairs query by maintaining a small subset of pairs called K-skyband which is expected to consist of $O(K\ log(N/K))$ pairs. For all the queries that use the same scoring function, we need to maintain only one K-skyband. We present efficient techniques for the K-skyband maintenance and query answering. We conduct a detailed complexity analysis and show that the expected cost of our approach is reasonably close to the lower bound cost. We experimentally verify this by comparing our approach with a specially designed *supreme* algorithm that assumes the existence of an oracle and meets the lower bound cost.**

I. Introduction

Given a scoring function $s(o_i, o_j)$ that computes the score of a pair of objects (o_i, o_j), 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. Due to the importance of the top-k pairs queries, numerous algorithms have been proposed to answer several variants of the top-k pairs queries [1], [2], [3], [4], [5], [6]. However, to the best of our knowledge, we are first to propose a framework to efficiently answer a broad class of top-k pairs queries over sliding windows.

Top-k pairs queries over sliding windows have many interesting applications in different areas such as wireless sensor network, stock market, traffic monitoring and internet applications etc. For instance, top-k pairs queries can be used for *pair-trading* [7]. 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 underperforming 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. Pair-trading can be profitable only if the trader is the first one to capitalize on the opportunity [7]. Hence, the trader may want to continuously monitor the top-k pairs from the most recent data (e.g., a sliding window containing most recent n items).

Consider another example of an online auction website. A user may be interested in finding the pairs of products that have similar specifications but are sold at very different prices (i.e., different final bids). Such pairs may be used to understand the users behavior and market trends, e.g., suitable bidding time for buyers and suitable bidding closing time for sellers etc. An analyst or a user may issue the following query to obtain top-k pairs of such products sold during last 7 days.

```
Select a.id, b.id from auction a, auction b
where a.id < b.id
order by dist(a.spec,b.spec) - |a.bid - b.bid|
limit k
window [7 days]
```

Here $dist(a.spec, b.spec)$ computes the distance (or difference) between their specifications and $|a.bid - b.bid|$ denotes the absolute difference between the final bids they receive. Note that the query prefers the pairs of products that have small difference between their specifications but have large difference between their selling prices. The condition $a.id < b.id$ ensures that a pair (a, b) is not repeated as (b, a).

While the above example shows a simple scoring function, in real-world applications, the users may specify a more sophisticated scoring function. Our framework allows the users to define arbitrarily complex scoring functions. A query that retrieves top-k pairs among the most recent n data items (i.e., sliding window of size n) and uses the scoring function s is denoted as $Q_{(k,n,s)}$.

A. Contributions

Our framework has following features.

Unified framework. To the best of our knowledge, we are the first to study top-k pairs queries over sliding windows. We present a unified framework that efficiently solves the

[*]Corresponding Author.

top-k pairs queries involving *any* arbitrarily complex scoring function. In our framework, the server maintains N most recent objects where N indicates the size of the largest sliding window any query is allowed to use. Each object has D attributes and the users may define any scoring function that uses $d \leq D$ of these attributes to compute the scores. Our framework handles multiple top-k pairs queries where each query is allowed to use a different scoring function, a different size of sliding window $n \leq N$ and a different value of k.

Intuitively, it may be possible to improve the performance if the scoring functions satisfy certain properties. We propose optimizations to significantly enhance the performance for a broad class of scoring functions called *global scoring functions* [1]. We remark that k-closest pairs queries, k-furthest pairs queries and their variants are among many of the popular queries that use the global scoring functions.

Low storage requirement. Our system uses $O(ND)$ space to maintain the most recent N objects. The system may receive different queries (issued by a single user or different users) and several queries having different values of k and n may share the same scoring function. For each unique scoring function, our system maintains a small subset of candidate pairs called *K-skyband* (to be formally introduced in Section III). All the queries that use this scoring function are answered using only the pairs in the K-skyband. We show that the expected size of the K-skyband is $O(K\ log(N/K))$ where K is the maximum value of k of the queries that use this scoring function and N is the size of the largest sliding window any query is allowed to use. Hence, in addition to $O(ND)$ memory space, our system uses $O(K\ log(N/K))$ memory for each unique scoring function. Note that the total number of possible pairs is $O(N^2)$ and $O(K\ log(N/K))$ is much smaller. Later, we show that $O(ND)$ is the lower bound storage requirement (see Theorem 4).

Efficient skyband maintenance. As the new objects arrive and the old objects expire, the skyband is needed to be maintained. Based on a novel concept of K-staircase, we present efficient techniques to maintain the K-skyband. We show that $O(N)$ is a lower bound cost for maintaining the K-skyband for arbitrarily complex scoring functions or when the system is unaware of the properties of the scoring functions. For this case, the expected cost of our algorithm is $O(N \cdot (log(log\ N) + log\ K))$ which is reasonably close to the lower bound cost. Note that, in practice, K is usually small (e.g., less than 1000) and $log(log\ N)$ is less than 2 even for a very large value of N (e.g., $N = 10^{99}$).

Efficient query answering. We propose efficient techniques to answer the top-k pairs queries using the K-skyband. Given a K-skyband, the complexity of our technique to answer a top-k pairs query is $O(log|SKB| + k)$ in the worst case where $|SKB|$ is the size of the K-skyband. The expected cost of our technique is $O(log(log\ n) + log\ K + k)$ where n is the size of the sliding window used by the query and K is the largest value of k any query may use. Note that the lower bound cost for query answering is $O(k)$ and the expected cost of our algorithm is reasonably close.

Extensive evaluation and analysis. As discussed above, we conduct a detailed complexity analysis to evaluate our algorithms and demonstrate that the cost of our proposed approach is reasonably close to the lower bound cost. To experimentally verify this, we design an algorithm called *supreme* algorithm that assumes the existence of an oracle that can conduct certain calculations without requiring any computation time. The usage of oracle allows the supreme algorithm to meet the lower bound. Our extensive experiments on real and synthetic data demonstrate that our algorithm performs reasonably well as compared to the supreme algorithm and is more than three orders of magnitude faster than a naïve algorithm.

II. Background Information

A. Related Work

1) Top-k Query Processing: Given a set of objects and a user defined scoring function, a top-k query retrieve the k objects with the smallest scores. The problem has been extensively studied [8], [9], [10]. Fagin's algorithm (FA) [9], threshold algorithm (TA) (independently proposed in [9], [10], [11]) and no-random access (NRA) [9] propose some of the top-k processing algorithms that combine multiple ranked lists and return the top-k objects.

2) Data Stream Processing: Processing the top-k queries and k nearest neighbor queries [8], [12], [13] on the data stream has been extensively studied. Mouratidis et al. [8] propose an efficient technique to compute top-k queries over sliding windows. They make an interesting observation that a top-k query can be answered from a small subset of the objects called k-skyband [14]. Our algorithm is similar in the sense that we also maintain the K-skyband to answer the top-k pairs queries. However, we use a single K-skyband to answer multiple queries having different values of $k \leq K$ and different sizes of the sliding windows. Also, the previous techniques [8], [12] to maintain K-skyband are not applicable to our problem because the techniques rely on the fact that the newly arrived objects cannot be dominated by any of the existing objects. Hence, these techniques unconditionally include the newly arrived objects in the K-skyband. On the other hand, in our problem, the newly formed pairs may or may not be dominated by the existing pairs, which make the request of online maintenance technically more challenging.

3) Top-k Pairs Queries Processing: The database community has devoted significant research attention to the processing of k-closest pairs queries[2], [3], [5] and their variants [15], [16], [6]. All of the above mentioned techniques are applicable only to the k-closest pairs queries or their variants. Cheema et al. [1] propose a unified framework to efficiently answer a broad class of the top-k pairs queries including the queries mentioned above. k-closest pairs queries on moving objects are studied in [17], [15]. However, the extension of these techniques to answer k-closest (or top-k) pairs queries over sliding windows is either non-trivial or inefficient.

B. Preliminaries

Sliding windows. Consider a stream of objects. For a fixed number N, a *count-based* sliding window contains the most recent N objects of the data stream. Similarly, for a fixed value T, a *time-based* sliding window contains the objects that arrive within last T time units. For the ease of presentation, in the rest of the paper, we consider only the count-based windows. However, our techniques can also be applied to answer the top-k pairs queries over the time-based sliding windows.

Age of a pair of objects. Let o be the i^{th} most recent object. We say that the age of the object o is i and we denote the age of an object as $o.age$. Note that a sliding window of size N consists of every object o for which $o.age \leq N$. We say that an object o has been expired if $o.age > N$.

A pair of objects (o_i, o_j) expires if at least one of the objects o_i and o_j expire. Note that the age of a pair (o_i, o_j) is $max(o_i.age, o_j.age)$. For the simplicity of the notations, we denote the age of a pair p as $p.age$. A sliding window of size N contains every pair p for which $p.age \leq N$.

Score of a pair. Given a scoring function $s(\cdot,\cdot)$, the score of a pair (o_i, o_j) is $s(o_i, o_j)$. For the simplicity of notations, the score of a pair p is denoted as $p.score$.

Top-k pairs query. A top-k pairs query $Q_{(k,n,s)}$ takes three parameters k, n and s and considers a set of pairs P that consists of every pair x for which $x.age \leq n$. The query $Q_{(k,n,s)}$ returns an answer set from P that consists of k pairs such that for every pair p in the answer set and for any other pair $p' \in P$, $p.score \leq p'.score$ (the scores are computed using the scoring function s).

Snapshot vs continuous queries. Note that the set of objects in the sliding window changes dynamically as the new objects arrive and the old objects expire from the sliding window. Hence, some users may be interested in continuous update of the results. In contrast, some users may only be interested in retrieving the top-k pairs from the current sliding window. The queries that require continuous updates of the results are called continuous queries and the queries that compute the results only once are called snapshot queries.

III. Solution Overview

Before we present our framework, we revisit the concept of K-skyband [14]. Then, we prove that K-skyband is the minimal set of pairs required to be maintained in order to answer top-k pairs queries.

K-Skyband. Let x and y be two points in d dimensional space. For any point x, $x[i]$ denotes the value of x in i^{th} dimension. A point x dominates a point y if for every dimension i, $x[i] \leq y[i]$ and for at least one dimension j, $x[j] < y[j]$. Given a set of points P, a K-skyband consists of every point $x \in P$ that is dominated by at most $(K-1)$ other points of P.

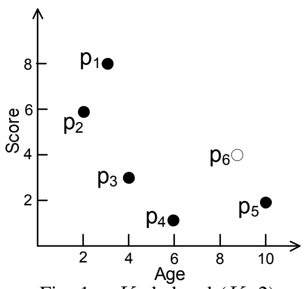


Fig. 1. K-skyband (K=2)

Consider the example of Fig. 1 that shows six points p_1 to p_6 in a two-dimensional space. The point p_6 is dominated by two points p_3 and p_4. Hence, the K-skyband (K=2) does not contain the point p_6. The 2-skyband consists of the points p_1, p_2, p_3, p_4 and p_5 because each of these points is dominated by at most one other point.

Given a pair of objects $p = (o_i, o_j)$ and a scoring function s, the pair can be mapped to a two dimensional age-score space where score is $p.score = s(o_i, o_j)$ and age is $p.age = max(o_i.age, o_j.age)$. Fig. 1 shows six pairs of objects shown in the age-score space.

Theorem 1: Let P be the set of all possible pairs of most recent N objects and each pair be mapped to the age-score space using a scoring function s_1. Let $SKB_{(K,s_1)}$ be the K-skyband of P in the age-score space. Every top-k pairs query $Q_{(k,n,s_1)}$ can be answered using the pairs in $SKB_{(K,s_1)}$ if $k \leq K$ and $n \leq N$.

Proof: It is sufficient to show that a pair $p' \notin SKB_{(K,s_1)}$ cannot be among the top-k pairs of any query $Q_{(k,n,s_1)}$. Since p' is not in the K-skyband, it implies that there are at least K other pairs such that for each such pair p, $p.score \leq p'.score$ and $p.age \leq p'.age$. Hence, for any sliding window of size $n \leq N$ that contains p', there exist at least K other pairs that are in the sliding window and have scores at most equal to the score of p'. Hence, such top-k pairs query can be answered without considering p'. ■

Consider the example of Fig. 1. Any top-k pairs query $Q_{(k,n,s)}$ can be answered by considering only the pairs p_1 to p_5 where $k \leq (K = 2)$, $n \leq (N = 10)$ and s is the scoring function used to map the pairs to the age-score space. The next theorem shows that the K-skyband is a minimal set of pairs required to be maintained in order to guarantee the correctness.

Theorem 2: Let $SKB_{(K,s_1)}$ be the K-skyband as defined in Theorem 1. For any algorithm that does not maintain a pair $p \in SKB_{(K,s_1)}$, there exists a query $Q_{(k,n,s_1)}$ that cannot be answered correctly.

Proof: Consider a query $Q_{(K,p.age,s_1)}$ (i.e., $k = K$ and $n = p.age$). Since p is a pair in the K-skyband, there exist at most $(K-1)$ other pairs with age at most equal to $p.age$ and scores smaller than $p.score$. In other words, there are at most $(K-1)$ pairs in the sliding window of size $n = p.age$ that have scores smaller than $p.score$. Hence, p must be among the top-K pairs of the query[1]. ■

A. Expected size of K-skyband

Existing analysis to estimate the expected size of K-skyband (e.g., [18]) assumes that i) the values of objects in one dimension are independent of their values in the other dimensions and ii) the values of the objects on each dimension are unique. Unfortunately, the existing analysis cannot be directly applied to our problem because the second assumption does not hold in our problem settings. This is because many pairs have the same value on the age dimension (i.e., have the same age). Nevertheless, we conduct an analysis and show

[1]Note that the proof assumes that there does not exist any other pair p' for which $p'.age = p.age$ and $p'.score = p.score$. We remark that even if such pairs exist, we can easily handle this case by assuming that the score of one pair is slightly larger (larger by an infinitely small value) than the other based on some criteria such as the IDs of the objects in the pairs.

that the expected size of the K-skyband we need to maintain is $O(K\ log(N/K))$.

We assume that the scores of pairs are independent of their ages. This is a reasonable assumption for the scoring functions that do not use ages of the objects to determine the scores of pairs.

Lemma 1: Let p be a pair with age x. Assuming that the scores of pairs are independent of their ages, the probability that p is in K-skyband is $min(K/x^2, 1)$.

Proof: Consider an object o_i and assume that $o_i.age = x$. Every pair (o_i, o_j) for which $o_j.age < o_i.age$ has age equal to $o_i.age$. Hence, the number of pairs with age equal to x is $(x-1)$. Also, for any pair p with $p.age = x$, the number of pairs that have age less than x is $1+2+\cdots+(x-2) = O(x^2)$. Let p' be one of these $O(x^2)$ pairs. Note that the pair p is dominated by p' iff $p'.score \leq p.score$. Hence, the probability that a pair with age x is not dominated by any other pair in the sliding window is $1/x^2$ assuming that every pair is equally probable to have the smallest score. Similarly, the probability that a pair with age x is dominated by at most K other pairs is $min(K/x^2, 1)$. ■

Theorem 3: Assuming that the scores of pairs are independent to the ages of the pairs, the expected size of the K-skyband is $O(K\ log(N/K))$.

Proof: From Lemma 1, the probability that a pair p with age x is dominated by at most K other pairs is $min(K/x^2, 1)$. As stated in the proof of Lemma 1, the number of pairs with age equal to x is $(x-1)$. Hence, the expected number of pairs that have age equal to x and are in K-skyband is $(x-1) \times min(K/x^2, 1)$. The expected total number of pairs that are in K-skyband is approximately $\sum_{x=2}^{N} x \cdot min(K/x^2, 1)$. Let $y = \lfloor \sqrt{K} \rfloor$. This expression can be simplified as follows.

$$\begin{aligned}\sum_{x=2}^{N} min(\frac{K}{x}, x) &\approx \sum_{x=2}^{y} x + \sum_{x=y+1}^{N} \frac{K}{x} \\ &\approx K + K \sum_{x=y+1}^{N} \frac{1}{x} \\ &\approx K + K(H_N - H_y)\end{aligned}$$

where $H_N = \sum_{x=1}^{N} 1/x$ and is called N^{th} harmonic number. For the case when $y = 1$ (i.e., $K < 4$) , the term $\sum_{x=2}^{y} x$ is considered zero and note that this does not affect our complexity analysis.

It is well known that H_N grows almost as fast as natural log of N. More precisely, H_N is known to be (e.g., see [19]) approximately equal to $ln(N) + \gamma$ where $\gamma \approx 0.577$ is *Euler's constant*. Hence, H_N and H_y can be approximated to $ln(N)$ and $ln(y)$, respectively . So, the expected number of pairs in K-skyband is $O(K \cdot (ln(N) - ln(\sqrt{K}))$ or $O(K\ log(N/K))$. ■

B. Framework

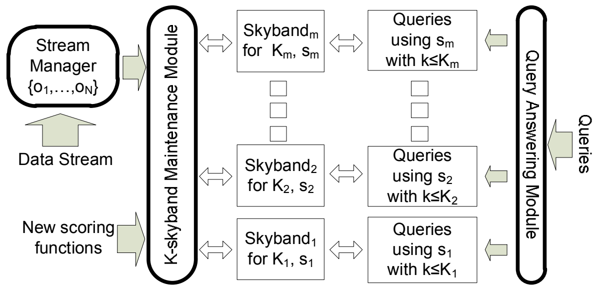
In real world scenarios, different users have different requirements. Therefore, different users may choose different scoring functions each involving a different set of attributes. Similarly, different users (or even a single user) may issue the top-k pairs queries with different values of k and n. We present a framework that aims to handle all these different queries efficiently. Our framework consists of the following three modules:

1. Stream Manager. Assume that each object has D attributes and every query issued on the system can use $d \leq D$ of these attributes in its scoring function. Moreover, suppose that N is the maximum size of the sliding window any query is allowed to use. The stream manager maintains $(D+1)$ lists each consisting of N elements. For every $0 < i \leq D$, the i-th list stores the objects sorted in ascending order of i-th attribute values of the objects. The $(D+1)$-th list is sorted in ascending order of the ages of the objects. Clearly, the storage requirement is $O(ND)$. The theorem below shows that this is the minimum amount of storage required to answer the top-k pairs queries.

Theorem 4: To answer a top-k pairs query over the sliding window of size N, the lower bound on storage requirement is $O(ND)$ where D is the number of attributes involved in the scoring function.

Proof: Assume that an object o is deleted such that $o.age \leq N$. Since the values of the newly arrived objects are unknown, a new object o' may arrive in the stream such that $s(o, o')$ is minimum (i.e., the pair (o, o') is one of the top-k pairs). If the object o is deleted from the stream, this pair will not be considered and the system will miss the correct answer. Hence, the object o must not be deleted. Moreover, the system must store all D attribute values of each object because the scoring function s may involve $d \leq D$ attributes. Hence, the lower bound on the storage requirement is $O(ND)$. ■

2. Skyband Maintenance Module. Let $S = \{s_1, \cdots, s_m\}$ be the set of unique scoring functions used by different queries. For each scoring function s_i, the skyband maintenance module maintains a set of skyband pairs $SKB_{(K_i, s_i)}$ where K_i is the maximum value of k for any query that uses the scoring function s_i (see Fig. 2).


Stream Manager {o1,…,oN}
Data Stream
New scoring functions
K-skyband Maintenance Module
Skyband$_m$ for K$_m$, s$_m$
Skyband$_2$ for K$_2$, s$_2$
Skyband$_1$ for K$_1$, s$_1$
Queries using s$_m$ with k≤K$_m$
Queries using s$_2$ with k≤K$_2$
Queries using s$_1$ with k≤K$_1$
Query Answering Module
Queries
Fig. 2. Framework

If a user issues a query $Q_{(k,n,s_i)}$ that uses a scoring function s_i not being used by any of the existing queries in the system, the skyband maintenance module creates a new skyband $SKB_{(K_i, s_i)}$ for this new scoring function. Upon receiving the object updates and new queries, the skyband maintenance module updates all the skybands in the system.

3. Query Answering Module. The query answering module is responsible for answering the snapshot or continuous top-k pairs queries. A query $Q_{(k,n,s_i)}$ is answered using the skyband $SKB_{(K_i, s_i)}$.

In Section IV, we present the details of the query answering module. The details of the skyband maintenance module is presented in Section V. The techniques for stream manager are simple and are omitted due to the space limitations.

IV. Query Answering Module

In this section, we present our query answering technique. As discussed earlier, to answer a query $Q_{(k,n,s_i)}$, the query answering module uses the skyband $SKB_{(K_i,s_i)}$. For the ease of presentation, we denote K_i as K and $SKB_{(K_i,s_i)}$ as skyband in this section.

A. Snapshot Top-k Pairs Queries

A straight forward approach to answer a top-k query is to scan the list of skyband pairs in increasing order of their scores. Any pair p for which $p.age > n$ is ignored. The algorithm stops when k pairs with age at most equal to n are retrieved. These k pairs are reported. Note that the cost of this algorithm is $O(|SKB|)$ in the worst case where $|SKB|$ is the size of the K-skyband. Next, we present an approach that answers the top-k pairs query in $O(log(|SKB|) + k)$ in the worst case.

To enable efficient computation of the queries, the skyband maintenance module indexes all the K-skyband pairs in a priority search tree (PST) [20]. Algorithm 1 shows the PST construction algorithm and Fig. 4 shows a PST constructed using the pairs in 2-skyband of Fig. 3. The pairs are labeled such that the age of a pair p_i is i. The number inside each node corresponds to its score. For each node, PST also stores the median value used to split the left and right subtrees (see line 3 of Algorithm 1). For example, the age of root node p_1 is 1, its score is 6 and the left and right subtrees are decided based on the median score 4 (shown under the dotted line).

Algorithm 1 PrioritySearchTree(P)

```
1: if $P$ is empty then return NULL
2: Choose an element $p$ with smallest age among $P$
3: $median$ = median of score values of elements in $P$
4: $P_R$ = {elements in $P$ with score greater than $median$}
5: $P_L$ = $P - P_R - \{p\}$
6: p.right-subtree = PrioritySearchTree($P_R$)
7: p.left-subtree = PrioritySearchTree($P_L$)
8: return $p$
```

Before we describe the properties of PST, we define a few terms. Ancestor of a node is its parent or (recursively) the parent of its ancestor. For example, in Fig. 4, the nodes p_1 and p_2 are the ancestors of the node p_3. Two nodes are called cousins to each other if they have a common ancestor and they do not have a child-ancestor relationship with each other. For example, the nodes p_4 and p_6 are cousins to each other because they have a common ancestor p_1. A node x is called a left cousin of a node y if they share a common ancestor e and x is in the left subtree of e and y is in the right subtree of e. Right cousins are defined similarly. In Fig. 4, the node p_6 is a left cousin of the node p_4 and the node p_4 is a right cousin of the node p_6.

The priority search tree has the following properties: 1) the age of a node cannot be smaller than the age of its ancestor (e.g., the age of p_3 is larger than the ages of its ancestors

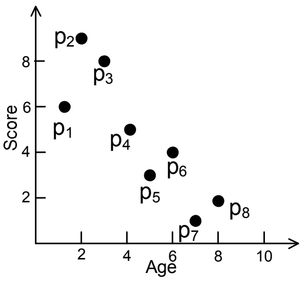


Fig. 3. 2-skyband

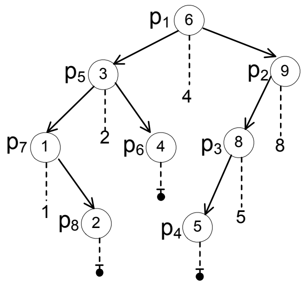


Fig. 4. Priority Search Tree

p_1 and p_2), 2) the score of a node is always greater than the scores of its left cousins and is always smaller than the scores of its right cousins (e.g., the score of p_6 is greater than the scores of its left cousins (p_7 and p_8) and is smaller than the scores of its right cousins (p_2, p_3 and p_4). Note that the score of a child may be smaller or larger than (or even equal to) the score of its ancestor.

We utilize the above mentioned properties to efficiently answer a top-k pairs query $Q_{(k,n,s)}$. Algorithm 2 shows our query processing algorithm that traverses the PST in an order very similar to the *post-order* traversal. In a post-order traversal, for any node e, its left subtree is visited before its right subtree and the node e is visited in the end. Our algorithm traverses the PST in the post-order except the following two differences: i) it only considers the nodes that lie in the sliding window (see lines 9 and 10) and ii) the algorithm terminates when k objects are visited in the post-order (line 3). It can be proved that the top-k pairs are among the pairs that are either visited or are among the marked nodes in the stack S (line 11). Finally, the set of candidates is scanned and k pairs with the smallest scores are obtained (line 12).

Algorithm 2 TopPairs(PST,k, n)

```
1: visitedSet = $\phi$
2: if root.age $\leq n$ then insert root in a stack $S$
3: while visitedSet.size $< k$ AND $S$ is not empty do
4:    $e$ = top element of $S$
5:    if $e$ is a leaf OR is marked then
6:       insert $e$ in visitedSet and remove from $S$
7:    else
8:       mark $e$
9:       if e.rightChild.age$\leq n$ then push e.rightChild in $S$
10:      if e.leftChild.age $\leq n$ then push e.leftChild in $S$
11: $candidates = visitedSet \cup$ marked nodes in stack $S$
12: visit candidates to obtain $k$ pairs with smallest scores
```

Example 1: Consider the 2-skyband shown in Fig. 3 and the PST shown in Fig. 4. Consider a query that wants to retrieve top-2 pairs in the sliding window of size 7. The post-order traversal returns two nodes p_7 and p_6 and the stack contains the nodes p_1, p_5 and p_2. The nodes p_1 and p_5 are the marked nodes and p_2 is not a marked node. The top-2 pairs are p_7 and p_5 which are selected from the candidates (p_7, p_6, p_1 and p_5). Note that our algorithm does not consider the node p_8 because it does not lie in the sliding window.

Proof of correctness. The algorithm returns k nodes in post-order traversal. Let x be the node with the largest score among these k nodes. Any other node y that has score smaller than

$x.score$ must satisfy one of the followings: 1) y is one of the left cousins of x; 2) y is a child of x or 3) y is an ancestor of x. Since our algorithm visits the nodes in post-order, any node that satisfies the condition 1 or 2 is either visited by our algorithm or is not visited because it does not lie in the sliding window (its age is greater than n). Hence, any node that lies in the sliding window and may possibly have score smaller than the score of x is one of its ancestors. Note that the stack contains the unvisited ancestors of all the visited nodes. Moreover, every ancestor of a visited node is a marked node in the stack (see line 8) and our algorithm considers all the marked nodes of the stack (see line 11 of Algorithm 2). Hence, our algorithm correctly determines the top-k pairs.

Complexity analysis. Priority search tree is always a balanced tree [20] because the left subtree and right subtree of a node are determined based on the median score. Therefore, the height of the tree in the worst case is $O(log|SKB|)$ where $|SKB|$ is the number of pairs stored in PST. Hence, the number of candidates at line 11 of Algorithm 2 is $O(log|SKB| + k)$. This is because the number of elements in stack at any time is bounded by the height of the tree. To obtain the top-k pairs, we use the the median of medians selection algorithm [21] to obtain the k pairs in time linear to the number of candidates. Hence the complexity of the algorithm is $O(log|SKB| + k)$ in the worst case.

As shown earlier, the expected size of K-skyband for a sliding window of size N is $O(K \cdot log(N/K))$ (Theorem 3). Note that our algorithm does not access a node e and its children if e does not lie in the sliding window of size n. This means that we essentially consider only the pairs in K-skyband that lie in the sliding window of size n. Hence, the expected cost is $O(log|SKB_n| + k)$ where $|SKB_n|$ is the size of K-skyband for the sliding window of size n. Hence, the expected cost is $O(log(K \cdot log\ (n/K)) + k) = O(log(log\ n) + log\ K + k)$. We remark that in the worst case the expected cost is $O(log(log\ N) + log\ K + k)$ because the maximum size of the stack in the worst case may still be $O(log|SKB|)$ even though we ignore the nodes with age greater than n. This is because the PST is a balanced tree with respect to the overall data set and may not necessarily be balanced for a subset of the data.

B. Continuous Top-k Pairs Queries

The initial results of a continuous top-k pairs query are computed using the algorithm presented earlier for computing the snapshot queries. The results of a query $Q_{(k,n,s)}$ may change if one of the top-k pairs expires or if a new pair has a score smaller than the score of one of the existing top-k pairs. We first handle the expired pairs and then handle the new pairs.

Handling pairs expired from K-skyband. For each query $Q_{(k,n,s)}$, we maintain two lists of top-k pairs one sorted on their ages and the other sorted on their scores. We use the list of top-k pairs that is sorted on the ages to determine when a pair expires. Let p be an expired pair. We delete p from both of the sorted lists.

Handling new pairs in K-skyband. The skyband maintenance module provides a list of new pairs added to the K-skyband. The list is provided sorted in ascending order of the scores of the new pairs. We scan the list in ascending order and every pair p is added to the answer of the query if $p.score < score_k$ where $score_k$ is the largest score among the scores of the top-k pairs. Whenever such a pair p is added to the answer, the pair with the largest score in the top-k pairs is deleted and the $score_k$ is updated accordingly. The algorithm stops scanning the sorted list when $p.score \geq score_k$. This is because all the remaining pairs are guaranteed to have scores greater than $score_k$ and are not needed to be considered.

Note that after handling the expired pairs and the newly arrived pairs, the answer set of a query may contain less than k pairs (e.g., when the number of deleted pairs is greater than the number of pairs added in the answer set). In such cases, we call Algorithm 2 to compute the top-k pairs from scratch in $O(log|SKB| + k)$.

Complexity analysis. In the worst case, the complexity of updating the results is $O(log|SKB| + k)$ because we call Algorithm 2 when the number of deleted pairs is greater than the number of inserted pairs. This worst case may happen only when one or more pairs are deleted from the top-k pairs. We analyse the probability of this case to happen.

For any object o_i, the number of pairs containing o_i in the sliding window of size n is $O(n)$. The total number of possible pairs in sliding window is $O(n^2)$. The probability that any of the pairs related to an object o_i has the smallest score among all possible pairs is $n/n^2 = 1/n$. The probability that any of the pairs related to the object o_i is one of the top-k pairs is k/n. Hence, the probability that any of the expired pairs is among the top-k pairs is k/n. Note that this probability is low since k is usually much smaller than n. Therefore, the probability of the worst case to happen is k/n and the expected amortized complexity of updating the results is $O(k/n(log|SKB| + k))$ per update.

V. Skyband Maintenance Module

A. Handling arbitrarily complex scoring functions

In this section, we present the details of skyband maintenance module (SMM) for arbitrarily complex scoring functions. The K-skyband needs to be updated when an object expires or when a new object arrives. Below, we describe how to handle both of the cases.

Handling when an object expires. Handling an expired object is easy because we only need to delete the relevant pairs from the K-skyband. Note that the age of an expired object o_i is the largest among all the objects in the sliding window. Moreover, every pair that is to be deleted has age equal to $o_i.age$. We keep a list of K-skyband pairs sorted on their ages and for each pair in the list we store a pointer to the relevant node in the PST. We use this list to delete every pair p for which $p.age = o_i.age$.

Handling when an object arrives. When a new object o_i arrives, we may need to update the K-skyband. For arbitrarily complex scoring functions, we need to consider all valid pairs of o_i with the existing objects in the sliding window. The number of pairs to be considered in this case is $O(N)$. Note that $O(N)$ is the lower bound cost for handling a new object because, for arbitrarily complex scoring functions, if we do not consider a pair (o_i, o_j) then we may miss the correct result because (o_i, o_j) may be one of the top-k pairs.

Algorithm 3 Handling new object (o)

```
1: Let $S$ be the pairs in $K$-skyband sorted on scores
2: for each new pair $p$ of the object $o$ do
3:     compute the score and age of $p$
4:     if $p$ is not dominated by $K$-skyband then
5:         insert $p$ in $S$ in sorted order
6: UpdateSkybandAndStaircase($S$)/* Algorithm 4 */
```

Algorithm 3 shows the details of handling a newly arrived object o. We say that a pair p is dominated by a K-skyband if there are at least K pairs in the K-skyband that dominate p. For each new pair p, we first need to check whether it is dominated by the existing K-skyband or not (line 4). The pairs that are not dominated by the K-skyband are added to the existing K-skyband which is kept sorted in ascending order of the scores of pairs (line 5). After all the pairs are considered, the algorithm updates the K-skyband (line 6).

As mentioned earlier, for each new pair p, we need to check whether it is dominated by the existing K-skyband or not (line 4). A naïve approach to do so is to consider all the pairs in the existing K-skyband and count the number of pairs that dominate p. If the number of dominating pairs is less than K then the pair p is not dominated by the K-skyband. Note that the complexity of this approach is linear to the size of the K-skyband, i.e., $O(|SKB|)$. Next, we present an approach that checks whether a pair p is dominated by the K-skyband or not in $O(log(|SKB|)$. First we introduce the concept of *K-staircase*.

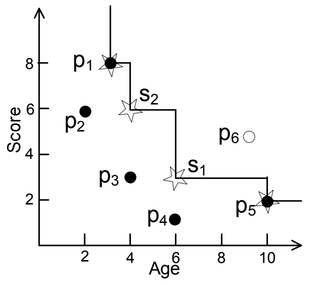


Fig. 5. 2-staircase

1) K-staircase: Given a set of points P, the K-staircase is a set of points $SCase$ such that if a point p is dominated by *any* point $x \in SCase$ then there are *at least* K points in P that dominate p. Moreover, for any point p', if there does not exist *any* point $x \in SCase$ that dominates p' then there are *at most* $K-1$ points in P that dominate p. Note that the points in the K-staircase can be used to check whether a point is dominated by the K-skyband or not. More specifically, a point p is dominated by the K-skyband if and only if it is dominated by at least one point of the K-staircase.

Fig. 5 shows a set of points $P = \{p_1, \cdots, p_6\}$. The K-staircase ($K = 2$) is also shown which consists of the points p_1, p_5, s_1 and s_2 (shown as stars). Note that the points in the staircase are not necessarily the points in the set P (see s_1 and s_2). Before we show our algorithm to compute the K-staircase, we present the intuition.

Consider a point p_3 that is in K-skyband ($K = 2$) as shown in Fig. 5. Among the points that have scores at most equal to $p_3.score$, we identify a point that has K^{th} smallest age. In Fig. 5, the points that have scores at most equal to $p_3.score$ are p_3, p_4 and p_5 and the point p_4 has the K^{th} ($K = 2$) smallest age among these points. Based on p_3 and p_4, we determine a K-staircase point s_1 such that $s_1.score = p_3.score$ and $s_1.age = p_4.age$. Please note that such a point s_1 is dominated by at least K points of P. Hence, any point that is dominated by s_1 is dominated by at least K points of P. Moreover, any point that dominates s_1 is dominated by at most $K-1$ points of P. To construct K-staircase, we repeat the above procedure for every point of the K-skyband and determine a relevant K-staircase point. Below, we present the details.

2) Updating K-skyband and K-staircase: Recall that in Algorithm 3, we need to update the K-skyband and K-staircase after all the new pairs are added to the existing K-skyband (see line 6). In this section, we present our technique to efficiently update the K-skyband and K-staircase. In [22], the authors presented an algorithm to construct the K-skyband from a set of two-dimensional points P. Since our algorithm to construct the K-staircase has a similar structure, we embed the two algorithms to construct both the K-skyband and K-staircase in parallel. If the points in the dataset P are sorted in the ascending order of their scores, the algorithm constructs the K-skyband and K-staircase in $O(|P| \cdot log\ K)$ where $|P|$ is the number of points in P.

Algorithm 4 UpdateSkybandAndStaircase(P, K)

```
1: Initialize a max-heap $H$ with key set to age of elements
2: Let $P$ be sorted in ascending order of scores
3: for each pair $p$ in $P$ do
4:     if $|H| < K$ then
5:         add $p$ to $SKB_K$
6:         insert $p$ in $H$
7:         if $|H| = K$ then
8:             insert $(p.score, H.top().age)$ into $K$-staircase
9:     else
10:        if $p.age \geq H.top.age$ then
11:            discard $p$
12:        else
13:            add $p$ to $SKB_K$
14:            insert $p$ in $H$
15:            H.pop()/* delete top element of H */
16:            insert $(p.score, H.top().age)$ into $K$-staircase
17: output $SKB_K$ and $K$-staircase.
```

Algorithm 4 presents the details. The points in P are accessed in ascending order of their scores (if two points have the same score, the point with the smaller age is accessed first). An accessed point p cannot be the K-skyband point if the algorithm has accessed at least K other points with age at most equal to $p.age$ (line 10). This is because all of these K points have scores at most equal to $p.score$ (recall that the points are being accessed in ascending order of scores).

If a point p is in K-skyband then we identify a K-staircase point x such that $x.score = p.score$ and $x.age = H.top().age$ where $H.top().age$ is the maximum age of a pair in the heap (line 16). Note that the heap stores K smallest ages and $H.top().age$ corresponds to the K^{th} smallest age among the points that have been accessed (i.e., have scores smaller than $p.score$).

Checking dominance using K-staircase. We say that a point p is dominated by the K-staircase $SCase$ if and only if there exists a point $x \in SCase$ that dominates the point p. As stated earlier, a point p is dominated by K-skyband if and only if p is dominated by the K-staircase. Next, we show that checking whether a point p is dominated by the K-staircase can be done in $O(log|SKB|)$.

Note that the points of the K-staircase returned by Algorithm 4 are sorted on their scores. To check whether a point p is dominated by the K-staircase or not, we do a binary search on the points in the K-staircase and retrieve a point x that has score smaller than $p.score$ and the point next to x in the K-staircase has score greater than $p.score$. It can be proved that if p is not dominated by x then the point is not dominated by any point in the K-staircase. This is because all the points of the K-staircase that have scores smaller than x have age greater than $x.age$ (see the K-staircase of Fig. 5). Since the size of K-staircase is bounded by the size of K-skyband, checking whether a point is dominated by K-staircase takes $O(log|SKB|)$.

3) Complexity analysis: The following lemma is important in analysing the complexity.

Lemma 2: When a new object arrives, the expected number of new pairs that are not dominated by the existing K-skyband is $O(K)$.

Proof: For a newly arrived object o_{new}, there are $O(N)$ new pairs in the sliding window. Let p_x be a new pair with age equal to x. The set of new pairs is $\{p_2, p_3, \cdots, p_N\}$. From Lemma 1, a pair with age x has probability $min(K/x^2, 1)$ not to be dominated by K-skyband. Hence, $\sum_{x=2}^{N} min(K/x^2, 1)$ gives the number of new pairs that are not dominated by the K-skyband. The summation can be approximated to $\sqrt{K} + K \cdot \sum_{x=\sqrt{K}+1}^{N} 1/x^2$. This is reduced to $\sqrt{K} + K \cdot C$ where C is a constant smaller than $\pi^2/6$ (see Basel's problem[2]). Hence, the number of such pairs is $O(K)$. ■

Cost of handling a new object. We analyse the complexity of Algorithm 3.

lines 2 to 4: For a newly arrived object, Algorithm 3 considers $O(N)$ new pairs (line 2). For each of these pairs, the algorithm checks whether it is dominated by the K-staircase or not. Hence, the total cost of these lines is $O(N \cdot log|SKB|)$.

line 5: According to Lemma 2, the number of pairs that are not dominated by the K-skyband is $O(K)$. These $O(K)$ pairs are inserted in the K-skyband set S. The cost of each such operation is logarithmic to the size of S. Hence, the cost of line 5 is $O(K \cdot log(|SKB| + K))$ where $O(|SKB| + K)$ is the expected size of S after all K pairs are added.

line 6: At line 6, Algorithm 4 is called. The cost of Algorithm 4 to compute the K-skyband and the K-staircase for a sorted dataset of size $|S|$ is $O(|S| \cdot logK)$ [22]. Since the size of S is $O(|SKB| + K)$, the cost of computing the K-skyband and the K-staircase (line 6 of Algorithm 3) is $O((|SKB| + K) \cdot logK)$. After the K-skyband is updated, the new pairs inserted in the K-skyband are inserted in the priority search tree (PST) and the pairs that are not among the K-skyband pairs anymore are deleted from the PST. Since the size of the K-skyband is expected to remain the same before and after the update, the number of new pairs is equal to the number of pairs deleted from the PST, i.e., $O(K)$ according to Lemma 2. The cost of inserting and deleting these pairs from the PST is $O(K \cdot log|SKB|)$.

Overall cost of Algorithm 3: The above analysis demonstrates that the overall complexity of Algorithm 3 is $O(N \cdot log|SKB| + K \cdot log(|SKB| + K) + (|SKB| + K) \cdot logK)$. Since $|SKB|$ is larger than K and N is larger than $|SKB|$ if $K \ll N$ (which is usually the case), the overall complexity of Algorithm 3 is $O(N \cdot log(|SKB|))$.

Cost of handling an expired object. When an object o_i expires, the number of pairs that are to be deleted from the K-skyband is at most K. This is because the K-skyband contains at most K pairs that have equal age (the K pairs with the smallest scores). Recall that each deletion and insertion on PST takes $O(log|SKB|)$. In the worst case, K pairs are to be deleted and the worst case cost is $O(K \cdot log|SKB|)$.

Overall cost. Note that the cost of handling a new object dominates the cost of handling an expired object. Hence, the overall cost is $O(N \cdot log(|SKB|)$. Since the expected size of $|SKB|$ is $O(K \cdot log\ (N/K))$, the overall expected complexity is $O(N \cdot (log(log\ N) + log\ K))$.

B. Optimization for certain scoring functions

In the previous subsection, we showed that the skyband can be maintained by considering $O(N)$ new pairs when a new object arrives in the data stream. In this section, we show that for a broad class of scoring functions we can reduce the number of considered pairs. We call these scoring functions the *global scoring functions*. The global scoring functions are based on monotonic and loose monotonic functions as defined in [1]. To make the paper self contained, we give formal definitions of these functions.

Monotonic function. 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 $1 \leq i \leq n$.

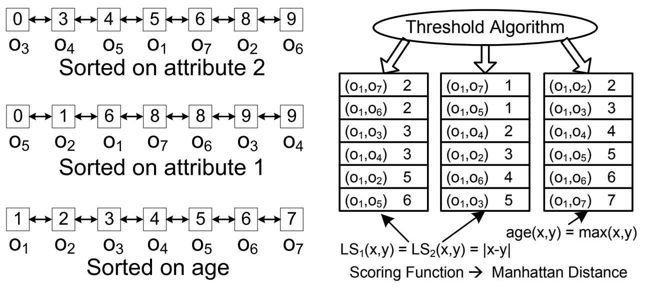
Loose monotonic scoring function. Let $ls(.,.)$ be a scoring function that takes two values as parameter and returns a score. A function $ls(.,.)$ is a loose monotonic function if for every value x_i both of the following are true: i) for a fixed x_i and every $x_j > x_i$, $ls(x_i, x_j)$ either monotonically 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.

Note that the loose monotonic scoring functions are more general than the monotonic scoring functions, i.e., every monotonic function is a loose monotonic function but the converse may not be true. The absolute difference of two values (e.g., $|x_i - x_j|$) is a loose monotonic function but not a monotonic function. The average of two values is a loose monotonic function as well as a monotonic function.

Global scoring function. Let d be the number of attributes used by the scoring function. For each attribute i, the user

[2]http://en.wikipedia.org/wiki/Basel_problem

specifies a loose monotonic scoring function $ls_i(.,.)$ that computes the score of a pair on the attribute i. Such scoring function is called a local scoring function and the score $ls_i(a,b)$ 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 global scoring function gsf that takes d local scores as parameter and returns the final score of a pair (a,b) as $gsf(ls_1(a,b),\cdots,ls_d(a,b))$. We require that such global scoring function must be a monotonic function. Note that the global scoring functions are more general than the monotonic scoring functions used by many real world applications [1]. For instance, k-closest pairs queries, k-furthest pairs queries and their variants can be answered by using global scoring functions (see [1] for details).



(a) Sorted Lists (b) Applying TA

Fig. 6. Optimization for global scoring functions

1) Technique: Let D be the total number of attributes of the objects. As described in Section II and shown in Fig. 6(a), the stream manager maintains $(D+1)$ sorted lists (D lists each sorted on one of the attributes and one list sorted on the ages). The global score (i.e., final score) of a pair is computed by combining $d \le D$ local scores where the i-th local score corresponds to the score of a pair on the i-th attribute.

For a newly arrived object o and for an attribute i, we can incrementally retrieve the pairs of objects related to the object o in ascending order of their i-th local scores (see [1] for details). Fig. 6(b) shows an example where, for a newly arrived object o_1, the lists can be used to incrementally retrieve the pairs of o_1 in sorted order of the scores. We iteratively retrieve these pairs in ascending order of scores for each attribute i and then apply an algorithm similar to the threshold algorithm (TA) [9] to terminate the algorithm before visiting all $O(N)$ new pairs of the newly arrived object.

Algorithm 5 presents the details. The algorithm accesses the pairs in round-robin fashion from the $d+1$ attributes where the $(d+1)^{th}$ attribute corresponds to the age of a pair (line 4). Each accessed pair p is mapped to age-score space and is inserted in S if it is not dominated by the K-staircase (line 6).

Let $\underline{ls_i}$ be the local score of the last retrieved pair for the i^{th} attribute and $\underline{age}$ be the age of the last pair retrieved for the age attribute. Note that $\underline{ls_i}$ corresponds to the smallest possible local score of any unseen pair for the i^{th} attribute. Hence, $gsf(\underline{ls_1},\cdots,\underline{ls_d})$ is the smallest possible final score of any unseen pair where $gsf()$ denotes the global scoring function. Similarly, $\underline{age}$ is the smallest possible age of any unseen pair. Hence, we map a dummy point (see line 10) to the age-score space with the smallest possible age and the

Algorithm 5 handling new object o)

```
1: $S$ = points in $K$-skyband sorted on scores
2: dummy point = $(0,0)$
3: **while** dummy point not dominated by $K$-staircase **do**
4:     **for** $i = 1$ to $i = d+1$ **do**
5:         access next best pair $p$ of $o$ in ascending order of $i^{th}$
           local score
6:         **if** $p$ is not dominated by $K$-staircase **then**
7:             insert $p$ in $S$ in sorted order of scores
8:     Let $\underline{ls_i}$ be the score of last pair seen for $i^{th}$ attribute
9:     Let $\underline{age}$ be the age of last pair seen from the age list
10:    dummy point = $(\underline{age}, gsf(\underline{ls_1},\cdots,\underline{ls_d}))$
11: UpdateSkybandAndStaircase($S$)
```

smallest possible score. If this dummy point is dominated by the K-staircase then any unseen pair will also be dominated by the K-staircase. For this reason, we do not need to consider remaining unseen pairs (see line 3) if the dummy pair is dominated by the K-staircase.

2) Complexity analysis: Note that the main difference between Algorithm 3 and Algorithm 5 is that Algorithm 3 considers $O(N)$ new pairs when a new object arrives whereas Algorithm 5 considers fewer pairs by using the threshold algorithm (TA). Let M be the number of the pairs considered by Algorithm 5. We estimate the value of M and obtaining the overall complexity is similar to that of the Algorithm 3.

We access the pairs in round robin fashion for the $d+1$ attributes. Note that the algorithm may terminate if at least K pairs have been seen for each of these $d+1$ attributes. This is because for any unseen pair there would be at least K pairs that have both the score and age less than it. Fagin showed that the number of elements accessed from the $d+1$ lists in such case is $M = (d+1)\cdot N^{d/(d+1)}\cdot K^{1/(d+1)}$ [9].

VI. Experiments

A. Experimental settings

Real data. We use a publicly available data set[3] collected from 54 sensor nodes deployed in the Intel research lab in Berkeley between February 28th and April 5th, 2004. Each node measures environment readings such as temperature, humidity and light. The data set consists of 2.3 million records collected from these sensors. We use the following scoring function. $s(o_x,o_y) = \frac{|o_x.time-o_y.time|}{|o_x.temp-o_y.temp||o_x.humidity-o_y.humidity|}$

The scoring function prefers the pairs of sensor readings that are taken within small duration of time and report quite different temperature and humidity. We remark that we tried several other inherently different scoring functions and the experimental results demonstrated similar trends.

Synthetic data. We generate synthetic data following uniform, correlated and anti-correlated [23] distributions and each data set consists of 2 million objects. Let $o[i]$ be the value of the object o in i^{th} dimension. For a scoring function that uses d dimensions, we use the following four different scoring functions.

$s_1(o_x,o_y) = \sum_{i=1}^{d} |o_x[i] - o_y[i]|$

$s_2(o_x,o_y) = -\sum_{i=1}^{d} |o_x[i] - o_y[i]|$

[3]http://db.csail.mit.edu/labdata/labdata.html

$s_3(o_x, o_y) = \prod_{i=1}^{d} |o_x[i] - o_y[i]|$
$s_4(o_x, o_y) = -\prod_{i=1}^{d} |o_x[i] - o_y[i]|$

Note that the scoring function s_1 retrieves the k-closest pairs and s_2 retrieves the k-furthest pairs according to the Manhattan distance between the pairs. Analogously, s_3 and s_4 retrieve top-k similar pairs and top-k dissimilar pairs, respectively, according to the product of the differences of the attributes. We conducted experiments for several other scoring functions and obtained results similar to the ones reported in this paper.

Parameter	Range
Data distribution	real, **uniform**, correlated, anticorrelated
# of attributes (d)	2, **3**, 4, 5, 6
N (in thousands)	**10**, 50 100, 500, 1000
K	1, 5, 10, **20**, 50, 100

TABLE I
EXPERIMENT PARAMETERS

Unless mentioned otherwise, for a fixed value of k and n, we issue four queries $Q_{(k,n,s_i)}$, one for each of the four scoring functions, and report the average query cost per object update. The table I shows the different parameters used in our experiments and the bold values are the default values used in the experiments unless mentioned otherwise.

B. Evaluating Overall Cost

To the best of our knowledge, we are the first to study the problem of top-k pairs over data stream. This problem is inherently different from other related problems such as k-closest pairs queries on moving objects [17], [15], static top-k pairs queries [1] and incremental distance join [2] etc. Although at first it may seem easy to extend previous techniques, a careful analysis demonstrates that the extension of these techniques to answer k-closest (or top-k) pairs queries over sliding windows is either non-trivial or inefficient.

We evaluate our algorithm (Algorithm 3) that answers the queries involving arbitrarily complex scoring function. Since it uses a K-staircase to maintain the K-skyband, our algorithm is called *SCase*. For an extensive evaluation of our algorithm, we carefully design two competitors called *Naïve* and *Supreme*. Below, we present the details.

Naïve Algorithm. A naïve approach to answer continuous top-k pairs query is to maintain all $O(N^2)$ pairs in sorted order of their scores. However, this approach appeared to be too slow. Another serious drawback is that the space complexity is quadratic and is prohibitive for large sliding windows. Therefore, we devised a better naïve approach that uses $O(KN)$ space. For each newly arrived object, K pairs related to it with the smallest scores are computed. All $O(KN)$ pairs are kept sorted on their scores. When an object o_i expires, all the pairs related to it are deleted. Note that the object o_i may be among the top-K pairs of an unexpired object o_j. After we delete the pairs related to o_i, we need to update the top-k pairs of every such object o_j.

Supreme algorithm. We assume that there exists an oracle that answers questions without requiring any computation time. We use this oracle such that the supreme algorithm meets the lower bound cost[4]. More specifically, for query answering, we assume that the supreme algorithm requests oracle to return, in sorted order of scores, only the pairs of K-skyband that lie in the sliding window. The supreme algorithm returns first k pairs and requests oracle to stop. Clearly, the query answering cost of the supreme algorithm is $O(k)$ that meets the lower bound.

As implied by Theorem 2, every algorithm must maintain the pairs in K-skyband for exact answering of top-k pairs queries. To maintain K-skyband, the supreme algorithm uses Algorithm 3 and computes only line 2 and line 3. The remaining steps are answered by the oracle in no time. Note that the skyband maintenance of the supreme algorithm meets the lower bound of $O(N)$.

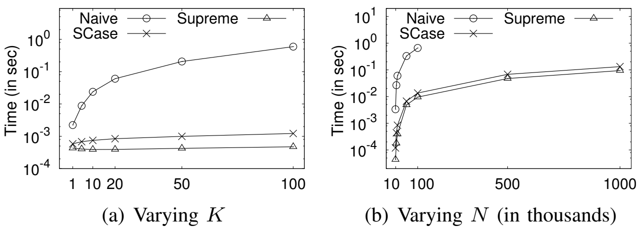


Fig. 7. Overall cost evaluation on the real data

In Fig. 7, we compare our algorithm with other algorithms using the real sensor data set. We issue 100 top-k pairs queries $Q_{(k,n,s)}$ where $k \leq K$ and $n \leq N$ are randomly chosen for each query. Our algorithm demonstrates two to three orders of magnitude improvement over the naïve algorithm and performs reasonably well as compared to the supreme algorithm. For $N \geq 500,000$, the naïve algorithm did not complete its execution in 7 days and the estimated completion time was around 2 months. Therefore, we do not show results for the naïve algorithm for the larger values of N.

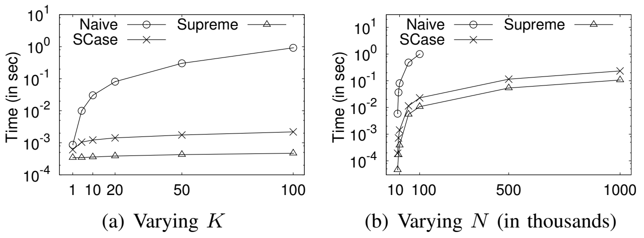
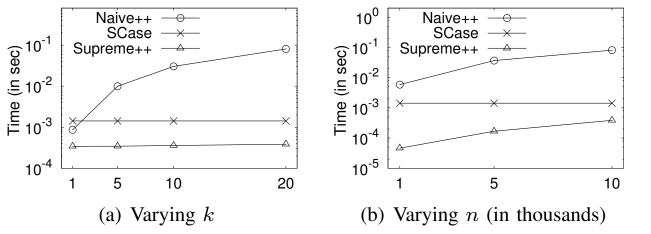


Fig. 8. Effect of K and N on synthetic data

In Fig. 8 and Fig. 9, we perform experiments on synthetic data sets to conduct a more detailed evaluation. Since we also want to observe the performance of the algorithms for varying n and varying k, we decide not to randomly generate n and k. Instead, in each experiment, we run four queries each using a fixed value of n and k and using one of the four scoring functions (s_1, s_2, s_3 and s_4) presented in Section VI-A. In Fig. 8(a) and Fig. 8(b), we study the effect of K and N on both algorithms. For each query, we set $n = N$ (the largest sliding window) and $k = K$ (the largest possible value of k).

[4]Note that the performance of an algorithm also depends on the way it is implemented. However, we remark that the supreme algorithm is a reasonable benchmark to evaluate the scalability of our approach. Having said this, for a fair evaluation, the supreme algorithm is implemented by using the code that is a subset of the code used by our algorithm.

The results are similar to the results obtained using the real data set.



(a) Varying k (b) Varying n (in thousands)

Fig. 9. Effect of k and n on synthetic data

In Fig. 9, we study the effect of k and n. As stated earlier, our algorithm does not know the values of n and k in advance hence maintains a K-skyband for most recent N objects. In contrast, for a more strict evaluation of our algorithm, we assume that both the naïve and the supreme algorithms know the values of n and k in advance. In effect, the supreme algorithm maintains k-skyband (note that $k \leq K$) for most recent n objects only. The naïve algorithm uses only $O(kn)$ memory instead of $O(KN)$ memory. We call these variations of the supreme and naïve algorithms as **supreme++** and **naïve++**, respectively.

The results are reported in Fig. 9(a) and Fig. 9(b). In Fig. 9(a), the naïve++ algorithm performs better for $k = 1$ because it needs to maintain only $O(n)$ pairs in total whereas we need to maintain 20-skyband ($K = 20$) for most recent $N = 10,000$ objects.

Fig. 9(b) shows that our algorithm outperforms naïve++ algorithm even for $n = 1000$ although it incurs maintenance cost to maintain a K-skyband for a window size N of $10,000$. Note that the complexity of supreme++ is $O(n)$ and the complexity of our algorithm is $O(N \cdot (log(log\ N) + log\ K)$. Hence, the cost of supreme++ increases with increase in n whereas the cost of our algorithm remains unaffected.

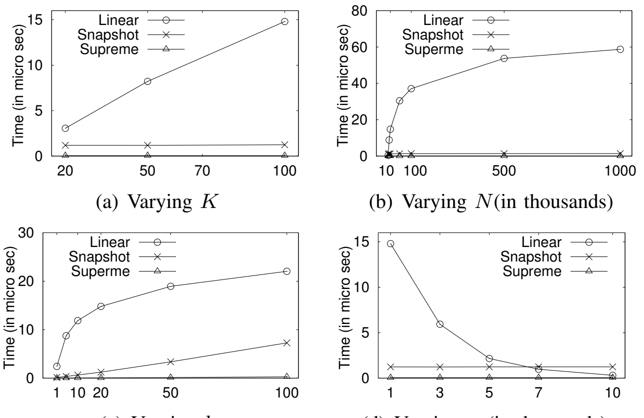
C. Evaluating Query Answering Module

In this section, we evaluate the performance of our query answering module.

1) Snapshot Query Answering: We compare our query answering algorithm with the supreme query answering algorithm as well as another algorithm called *linear* algorithm. The linear algorithm is the approach we discussed in the first paragraph of Section IV-A and it takes time linear to the size of K-skyband in the worst case. Our query answering algorithm (Algorithm 2) is called *snapshot*. We study the effect of each of the parameters K, N, k and n, separately.

In Fig. 10(a) and Fig. 10(b), we study the effect of varying K and N, respectively. The default value of n is 1000 and the default value of k is 20. As expected, the cost of supreme algorithm is negligible. This is because, in all the experiments, the supreme algorithm needs to iterate over a link list of size k. The snapshot algorithm outperforms the linear algorithm and scales better with the increase in the values of K or N. The cost of linear algorithm increases because the size of K-skyband increases with the increase in K or N.

In Fig. 10(c) and Fig. 10(d), we fix the values of K and N and study the effect of k and n on both of the algorithms. The default value of K is chosen to be 100 so that we can



(a) Varying K (b) Varying N(in thousands)

(c) Varying k (d) Varying n (in thousands)

Fig. 10. Linear vs Snapshot Algorithm

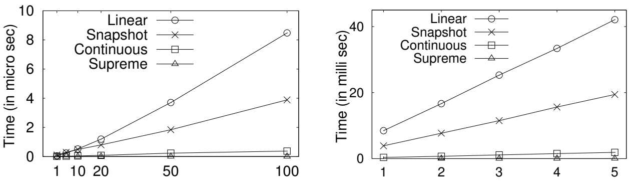
answer the queries with any $k \leq 100$. The snapshot algorithm performs better than the linear algorithm for varying k.

Fig. 10(d) shows that the linear algorithm performs slightly better than the snapshot algorithm when the value of n is close to N. This is because the linear algorithm accesses the pairs in K-skyband in ascending order of scores and terminates when k pairs are found with age at most equal to n. The algorithm is expected to terminate earlier when n is large. Note that when $n = N$ the cost of linear algorithm is $O(k)$ which is impossible to be outperformed.

Recall that our complexity analysis shows that the cost of snapshot algorithm is $O(log(log\ n) + log\ K + k)$. As anticipated by our complexity analysis, the cost of our snapshot algorithm increases with increase in k (see Fig. 10(c)) but is not significantly affected by a moderate increase in K or n (see Fig. 10(a) and Fig. 10(d)).

2) Continuous Query Answering: Next, we evaluate the performance of our continuous query algorithm which is denoted as *continuous* in the figures. The supreme algorithm for continuous query answering assumes that the oracle notifies it whenever a pair is deleted or added to the existing answer and the supreme algorithm updates the results accordingly. We also choose the linear algorithm and the snapshot algorithm as competitors such that these algorithms compute the results from scratch whenever the results are to be updated.

In Fig. 11(a), we show the effect of K on the continuous query algorithm for 1000 queries that randomly choose the values of n and k. Fig. 11(a) shows the average cost per query per object update. Clearly, our continuous query algorithm outperforms the linear and snapshot algorithms and scales better.



(a) Varying K (b) Varying $|Q|$(in thousands)

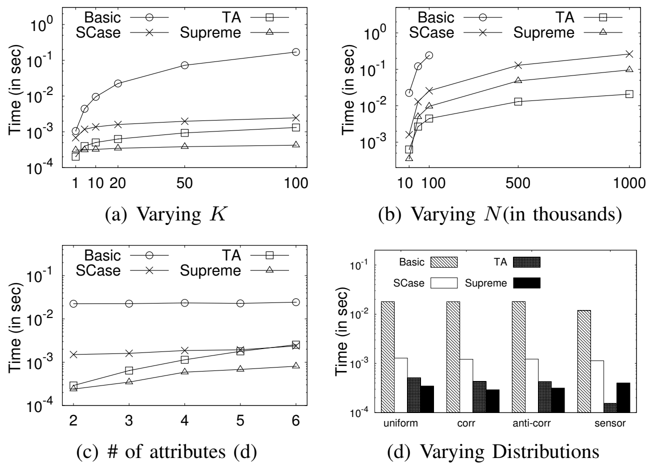
Fig. 11. Evaluation of continuous queries algorithm

Fig. 11(b) shows the performance of the algorithms for the increasing number of queries. Each query $Q_{(k,n,s)}$ uses a randomly chosen value of k and n. Fig. 11(b) shows the total cost for all the queries per object update. Our continuous query algorithm outperforms the linear and snapshot approaches.

D. Evaluating Skyband Maintenance Module

In this section, we evaluate our skyband maintenance module. We compare four algorithms. The *SCase* algorithm is the Algorithm 3 which uses K-staircase and can be applied on any arbitrarily complex scoring function. The *basic* algorithm is the same as Algorithm 3 but does not use K-staircase. As stated in Section II-A2, previous algorithms to maintain K-skyband [8], [12] cannot be directly applied. Nevertheless, we embedded all applicable optimizations (e.g., dominance counter) of their techniques in the *basic* algorithm. The *TA* algorithm is Algorithm 5 which is applicable only on the queries using global scoring functions. The supreme algorithm maintains the skyband as discussed in Section VI-B. Note that TA has an advantage over all other algorithms (including the supreme algorithm) that it knows that the scoring function is a global scoring function and uses its properties.

In Fig. 12(a) and Fig. 12(b), we study the affect of K and N, respectively. As expected, the TA algorithm always outperforms the basic and SCase algorithms. This shows the effectiveness of using optimizations for global scoring functions. Also, note that SCase algorithm outperforms the basic algorithm which shows the effectiveness of using the K-staircase. TA outperforms even the supreme algorithm when window size N is large. This is because TA utilizes the properties of the global scoring function and does not compute the score of all $O(N)$ objects when a new object arrives.



(a) Varying K (b) Varying N(in thousands)

(c) # of attributes (d) (d) Varying Distributions

Fig. 12. Skyband Maintenance Techniques

In Fig. 12(c), we vary the number of attributes d used by the scoring functions and study the effect on the algorithms. The performance of TA degrades as the number of attributes increases. This verifies our complexity analysis given in Section V-B. The cost of supreme algorithm increases mainly because the cost of computing the score of a pair increases as the number of attributes increases. The basic and SCase algorithms are not affected by the number of attributes because the main cost in these two algorithms is not the cost of computing the scores of the pairs.

In Fig. 12(d), we show the effect of data distribution on the algorithms. TA consistently performs better than SCase and the basic algorithm on each different data set. Also, SCase algorithm performs significantly better than the basic algorithm.

VII. Conclusion

We present efficient techniques to answer a broad class of top-k pairs query over sliding windows. We provide a detailed complexity analysis and show that the storage requirement and the performance of our algorithms is reasonably close to the lower bound. We verify this by an extensive experimental evaluation and demonstrate the efficiency of our approach.

Acknowledgments: The work was partially done when the corresponding author was taking a Chinese academic program at East China Normal University; and the first, second and forth authors were visiting East China Normal University, and supported by NSFC61021004. The work was also supported by ARC DP120104168, DP110102937 and DP0987557.

References

[1] 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.
[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 Conference*, 2000.
[4] M. Smid, "Closest-point problems in computational geometry," in *Handbook on Computational Geometry*, 1997.
[5] C. Yang and K.-I. Lin, "An index structure for improving nearest closest pairs and related join queries in spatial databases," in *IDEAS*, 2002.
[6] J. Shan, D. Zhang, and B. Salzberg, "On spatial-range closest-pair query," in *SSTD*, 2003, pp. 252–269.
[7] G. Vidyamurthy, *Pairs Trading: quantitative methods and analysis*. John Wiley & Sons, Inc., 2004.
[8] K. Mouratidis, S. Bakiras, and D. Papadias, "Continuous monitoring of top-k queries over sliding windows," in *SIGMOD Conference*, 2006.
[9] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *JCSS*, 2003.
[10] S. Nepal and M. V. Ramakrishna, "Query processing issues in image (multimedia) databases," in *ICDE*, 1999.
[11] U. Güntzer, W.-T. Balke, and W. Kießling, "Optimizing multi-feature queries for image databases," in *VLDB*, 2000.
[12] C. Böhm, B. C. Ooi, C. Plant, and Y. Yan, "Efficiently processing continuous k-NN queries on data streams," in *ICDE*, 2007.
[13] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas, "Ad-hoc top-k query answering for data streams," in *VLDB*, 2007.
[14] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *TODS*, 2005.
[15] L. H. U, N. Mamoulis, and M. L. Yiu, "Continuous monitoring of exclusive closest pairs," in *SSTD*, 2007.
[16] F. Angiulli and C. Pizzuti, "An approximate algorithm for top-k closest pairs join query in large high dimensional data," *Data Knowl. Eng. 2005*.
[17] P. Zhou, D. Zhang, B. Salzberg, G. Cooperman, and G. Kollios, "Close pair queries in moving object databases," in *GIS*, 2005, pp. 2–11.
[18] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu, "Probabilistic skyline operator over sliding windows," in *ICDE*, 2009.
[19] D. E. Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.
[20] E. M. McCreight, "Priority search trees," *SIAM Journal on Computing*, vol. 14, no. 2, pp. 257–276, May 1985.
[21] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *JCSS*, 1973.
[22] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava, "Ranked join indices," in *ICDE*, 2003.
[23] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.