Stabbing the Sky: Efficient Skyline Computation over Sliding Windows

Xuemin Lin   Yidong Yuan   Wei Wang
NICTA & University of New South Wales
Sydney, Australia
{lxue, yyidong, weiw}@cse.unsw.edu.au

Hongjun Lu
Hong Kong University of Sci. & Tech.
Hong Kong, China
luhj@cs.ust.hk

Abstract

We consider the problem of efficiently computing the skyline against the most recent N elements in a data stream seen so far. Specifically, we study the n-of-N skyline queries; that is, computing the skyline for the most recent n ($\forall n \leq N$) elements. Firstly, we developed an effective pruning technique to minimize the number of elements to be kept. It can be shown that on average storing only $O(\log^d N)$ elements from the most recent N elements is sufficient to support the precise computation of all n-of-N skyline queries in a d-dimension space if the data distribution on each dimension is independent. Then, a novel encoding scheme is proposed, together with efficient update techniques, for the stored elements, so that computing an n-of-N skyline query in a d-dimension space takes $O(\log N + s)$ time that is reduced to $O(d \log \log N + s)$ if the data distribution is independent, where s is the number of skyline points. Thirdly, a novel trigger based technique is provided to process continuous n-of-N skyline queries with $O(\delta)$ time to update the current result per new data element and $O(\log s)$ time to update the trigger list per result change, where δ is the number of element changes from the current result to the new result. Finally, we extend our techniques to computing the skyline against an arbitrary window in the most recent N elements. Besides theoretical performance guarantees, our extensive experiments demonstrated that the new techniques can support on-line skyline query computation over very rapid data streams.

1 Introduction

For two points $x = (x_1, x_2, ..., x_d)$ and $y = (y_1, y_2, ..., y_d)$ in the d-dimensional space, x *dominates* y if $x_i \leq y_i$ for $1 \leq i \leq d$. Given a set P of points, the *skyline* comprises the points in P, which are not dominated by another point in P (see Figure 1 for example). Skyline computation roots in many applications [26] that involve a multi-criteria decision making. For instance, in the stock market buyers may want to know the top deals so far, as one of many kinds of statistic information, before making trade decisions; consequently, a query "what are the top buy deals (transactions) of a given stock" may be issued. Here, each deal (transaction) is recorded by the price (per share) and the volume (number of shares). This is a typical example of ranking the data items (deals) by more than one criterion; that is, price and volume in this case. Obviously, in such an application a deal a is better than another deal b if a involves a higher volume and is cheaper (per share) than those of b, respectively. The top deals, thus, are the set of deals which are not worse than another deal; they form the skyline of all the deals (Figure 1 shows such an example).

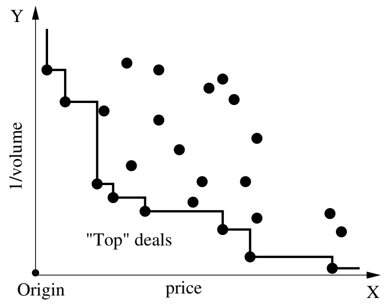


Figure 1. Skyline of Buy Deals of a Stock

Skyline query processing and its variants [3, 17, 20, 26] have been extensively studied, and a number of algorithms have been developed. The main memory algorithms may be found in [3, 17, 14], while the techniques related to database applications may be found in [4, 6, 15, 23, 27]. Without pre-processing, the best known bounds of time complexity among the existing algorithms for computing the skyline of n points in a d-dimensional space are $O(n \log^{d-2} n)$ for $d \geq 4$ and $O(n \log n)$ for $d = 2, 3$ [17]. The skyline computation problem is also related to several other well-known problems, such as *convex hulls*, *top-K* queries, and *nearest neighbour* search. The techniques can be found in the literature [12, 13, 22, 24].

None of the algorithms referenced above was originally designed to support on-line computation in the presence of rapid updates of data elements. On the other hand, in many applications data updates may rapidly happen. For instance, in the *sliding window* computation model (i.e., the most recent N elements) against data streams a deletion (expiring the oldest element) and an insertion (inserting a new element) are always associated with the new arriving element.

The existing techniques are not able to efficiently support on-line computation against sliding windows over a rapid data stream.

Sliding window computation is very important in data stream applications [1] with the aim to provide the most recent on-line information. Regarding the above example of identifying the skyline of buy deals, some clients may want to know the top deals among the most recent N deals; that is, the skyline of the most recent N data elements. Moreover, different users may have different favourite thresholds of N. Therefore, it is important for an information provider (system) to organise the most recent N elements in an effective way, so that any "n-of-N skyline" queries (the computation of the skyline of the most recent n ($\forall n \leq N$) elements) can be processed efficiently. In this paper, we focus on the problem of on-line processing n-of-N skyline queries over a data stream seen so far.

Very recently, approximation techniques for computing convex hulls and nearest neighbours have been developed against data streams [11, 16]. However, these techniques are not applicable to the skyline computation over data streams due to the inherent difference between those problems and the skyline problem. Moreover, we focus on a precise computation in this paper. The algorithm in [14] can efficiently support an on-line skyline computation regarding deletion/insertion of data elements. Since the algorithm focuses on the skyline computation for a whole data set, the data structures maintained are only applicable to the computation over the most recent N elements, while an n-of-N (for $n < N$) skyline query still has to be processed as if there were no pre-processing. Further, this technique only supports element deletions in the 2-dimensional space.

These are the motivations of our research in the paper. To the best of our knowledge, there is no similar work existing in the literature in the context of skyline computation over data streams. Our contribution can be summarized as follows.

- A novel pruning technique has been developed to minimize the number $\mathcal{N}$ of elements to be kept in the most recent N elements for processing all n-of-N queries. Note that $\mathcal{N} \leq \mathcal{N}$ as $\mathcal{N}$ is the number of elements in a subset of the most recent N elements. We also showed that in a d-dimensional space $\mathcal{N} = O(\log^d N)$ if the data distribution on each dimension is independent.
- A novel encoding scheme with linear size $O(\mathcal{N})$ on the stored elements is developed, together with the efficient update algorithms based R-tree and interval tree techniques. This encoding scheme effectively reduces the time complexity for processing an n-of-N skyline query ($\forall n \leq N$) to $O(\log \mathcal{N} + s)$ from $O(n \log n)$ for $d = 2, 3$ and $O(n \log^{d-2} n)$ for $d \geq 4$, where s is the number of skyline points. In fact, the time complexity of our query algorithm may be further reduced to $O(\min\{\log \mathcal{N}, d \log \log N\} + s)$ when the data distribution is independent.
- A new trigger based technique for continuously processing an n-of-N skyline query is developed. Upon the arrival of a new data element, it guarantees $O(\log \delta)$ time to update the current query result where δ is the number of element changes from the current result to the new result. It takes $O(\log s)$ time to update the triggers list per result change.
- We extend our techniques to the applications where an arbitrary window query in the most recent N elements is involved.

Besides theoretical guarantees, our extensive experiments indicated that the new techniques can accommodate on-line computation against very rapid data streams. As shown later, the techniques can be immediately applied to the applications where the most recent information is specified on a time period.

The rest of the paper is organized as follows. In section 2, we present background information in skyline and stream computation. Sections 3 and 4 provide our techniques for processing n-of-N and arbitrary window skyline queries, respectively. Results of comprehensive performance studies are discussed in section 5. Section 6 concludes the paper.

2 Preliminaries

In this section, we briefly introduce the related work in the skyline computation, followed by different stream computation models. Then, we present our framework for online computing the skyline of the most recent n ($\forall n \leq N$) data elements in a data stream.

Table 1 summarises the notation used throughout the paper.

Table 1. Notation

Notation	Definition
N, n	number of the most recent elements
M	number of the elements seen so far
P_N	the most recent N elements
R_N	non-redundant elements in P_N
S_N	skyline points in P_N
$\mathcal{N}$	$\lvert R_N \rvert$
e	an element in a data stream
$\kappa(e)$	position (integer) of e in a data stream
s	number of skyline points
d	space dimension

2.1 Skyline Query Processing

A *skyline query* is to compute the skyline of a set P of n points; every point (data element) in the skyline is called *skyline point*. An efficient computation of skyline query was first investigated by Kung *et al.* in [17] where an $O(n(\log n)^{d-2})$ time algorithm for $d \geq 4$ and an $O(n \log n)$ time algorithm for $d = 2, 3$ were developed. Bentley *et al.* [3] provided an efficient algorithm with an expected linear running time if the data distribution on each dimension is independent.

Kapoor [14] studied the problem of dynamically maintaining an effective data structure for an incremental skyline computation in the 2-dimensional space. The data structure adopts the *red-black* tree [7] to organise the n elements in P in order of increasing values in one dimension. Then, the skyline of the elements in each subtree is implicitly maintained. The data structure takes $O(n)$ space with the $O(\log n)$ update time for an insertion/deletion. The data structure also takes $O(\gamma \log n + s)$ time to compute the skyline of P where s is the number of skyline points and γ is the number of changes of the data structure since last computation. The technique has been extended to the d-dimensional space (for any $d \geq 3$), where only insertions can be involved, with $O(\log^{d-1} n)$ updated time per insertion and $O(\gamma \log^{d-1} n + s)$ time to compute the skyline.

Börzsönyi *et al.* [4] studied the skyline computation problem in the context of databases and proposed an SQL syntax for the skyline query. They also developed the skyline computation techniques based on *block-nested-loop* [25] and *divide-conquer* [7] paradigms, respectively. Chomicki *et al.* [6] proposed another block-nested-loop based computation technique to take the advantages of a pre-sorting. Tan *et al.* [27] proposed the first *progressive* technique that can output skyline points without having to scan the whole data set. Two auxiliary data structures are proposed, *bitmap* and *search tree*. Kossmann *et al.* [15] presented another *progressive* technique based on the nearest neighbour search technique on R-tree [22, 12], which adopts a divide-and-conquer paradigm on the dataset indexed by R-tree. Papadias *et al.* [23] proposed a branch and bound search technique to progressively output skyline points on datasets indexed by R-tree. One of the most important properties of the technique in [23] is that it guarantees the minimum I/O costs.

2.2 Skyline and Data Streams

In many applications, a data stream may be *append-only* [1, 9]; that is, there is no deletion of data element involved. In this paper, we studied the skyline computation problem restricted to the append-only data stream model. In a data stream, elements are positioned according to their relative arrival ordering and labelled by integers. Note that the position/label $\kappa(e)$ means that the element e arrives $\kappa(e)$th in the data stream.

Consider that precisely computing skyline in an on-line fashion naturally requires main memory based processing algorithms. On the other hand, it may not be physically feasible to store a whole data stream in main memory since the volume of a whole stream is theoretically unbounded. In this paper, we studied the problem of skyline computation restricted to the most recent N elements, seen so far, that can fit in the main memory. Specifically, we investigate the on-line computation of skyline restricted to the stream computation models below.

1. n**-of-**N **model** [18]. We will investigate the problem of effectively organising the most recent N elements in a data stream seen so far, so that the computation of skyline against any most recent n ($n \leq N$) elements can be processed efficiently. Note that a *sliding window* model [1] is a special case of the n-of-N model where $n = N$. □

2. (n_1, n_2)**-of-**N **model.** This is a generalization of the n-of-N model. Here, we want to compute the skyline of the elements between the most n_2th recent element and the most n_1th recent element (for any $n_1 \leq n_2 \leq N$). □

Note that the n-of-N model gives the skyline based on the most recent information, while the (n_1, n_2)-of-N model provides recent "historic" information. Combining the results from the two models may indicate a trend change from the most recent n_2 elements to the most recent n_1 elements.

The skyline queries against the n-of-N model and (n_1, n_2)-of-N model are, thereafter, abbreviated to "n-of-N query" and "(n_1, n_2)-of-N query", respectively.

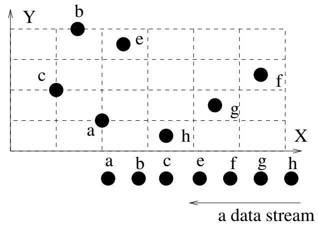


Figure 2. A Data Stream

2.3 On-line Computation

Consider the example in Figure 2 where elements arrive according to the alphabetic ordering. Suppose that the data stream so far consists of 6 elements a, b, c, e, f, and g. The skyline S_6 against these most recent 6 elements consists of a and c, while the skyline S_4 against the most recent 4 points (i.e., c, e, f, and g) consists of c and g. Further, when the new element h arrives, S_6 and S_4 are updated to $\{c, h\}$ and $\{e, h\}$, respectively.

This example shows that the skyline S_n of the most recent n elements P_n is not a subset of the skyline S_N of the most recent N elements P_N when $n < N$, though P_n is a subset of P_N. Therefore, the problem of efficiently processing any n-of-N query ($\forall n \leq N$) is more challenging than the computation of skyline on the most recent N elements only for a given N. Moreover, S_n (S_N) may be changed along the update of the most recent n (N) elements upon the arrival of a new element. Consequently, in an on-line environment efficient techniques are required to compute the up to date skyline. Furthermore, although in this paper the data streams interested are append-only, our techniques have to deal with the deletion of data elements; this is because the deletion of the oldest element due to its expiration from the most recent N elements happens every time when a new element arrives. Below we outline our on-line techniques for processing n-of-N queries.

- We effectively characterize the "critical" dominance relationships among the data elements in the most recent N elements P_N and model them by a graph that is

a *forest*. This leads to the minimization of the number of data elements to be kept from P_N.
- We encode such a graph by a set of intervals in the 1-dimensional space, and map an n-of-N query into a "stabbing" query on the intervals. Then, an n-of-N query is processed by computing the corresponding stabbing query.
- The graph and its encoding scheme are dynamically maintained to reflect an on-line change of the most recent N elements in a data stream.
- A novel trigger based algorithm is developed to process a continuous n-of-N query.

The techniques are also extended to cover (n_1, n_2)-of-N queries. Below, we briefly introduce the "stabbing" queries and the processing time complexity. The stabbing query processing techniques will be used as a black-box in our algorithms.

Stabbing queries

Give a set of m intervals and a *stabbing* point p in the 1-dimensional space, the *stabbing query* is to find all intervals which contain p. By the interval tree techniques in [21, 24], a stabbing query can be processed in $O(\log m + l)$ where l is the number of intervals in the result. By storing an interval only in the tree node that is the lowest common ancestor (LCA) of the two end points of the interval, the space complexity of the interval tree is $O(m)$. It has been also shown that the time complexity of an update (insertion or deletion) to an interval tree is amortised to $O(\log m)$ per deletion or insertion. Note that the intervals here can be closed, half closed, or open at both ends.

3 Processing n-of-N Queries

In this section, we present our techniques for efficiently processing n-of-N queries. We first minimize the number of data elements to be kept for processing all n-of-N queries. Then, we present an effective encoding scheme on the stored elements to support n-of-N query processing. Thirdly, we present our techniques to efficiently update the data structures involved. This is followed by our novel techniques for processing continuous queries.

3.1 Minimizing the Number of Elements

Suppose that in the most recent N data elements (points) P_N, there are 2 elements e and e' such that e' dominates e and e' arrives later than e. It is immediate that in any most recent n ($n \leq N$) elements containing e', e is not a skyline point. A data element e is *redundant* with respect to the most recent N elements if e is expired (i.e. outside the most recent N elements) or is dominated by a younger element e' (later issued than e).

Theorem 1. *Suppose that P_N is the set of the most recent N data items. Then,*

1. *a skyline point (data element) in the most recent n (for any $n \leq N$) elements must be a non-redundant element in the current P_N;*
2. *any non-redundant element in P_N is a skyline point in the most recent n (for some $n \leq N$) elements;*
3. *once an element e becomes redundant, e will be no longer qualified as a skyline point over any most recent n elements ($\forall n \leq N$).*

Proof. The theorem can be immediately verified from the definition of a redundant element. □

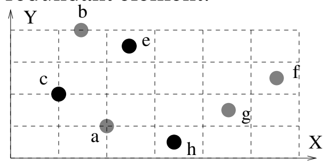


Figure 3. Redundant Points

Consider the data stream in Figure 2 with 7 elements and $N = 6$. The non-redundant elements are shown in Figure 3 as the black points. Note that the element a is not included since it is already expired.

Let R_N be the set of non-redundant elements in P_N. According to Theorem 1, we need only to keep R_N instead of P_N to exactly answer all n-of-N queries; that is, the redundant elements should be removed. Moreover, Theorem 1 also implies that $|R_N|$ gives the minimum number of elements we should keep to support the precise computation of all n-of-N queries. We can show the following theorem.

Theorem 2. *In a data stream on a d-dimensional space, suppose that the data distribution on each dimension, including the arrival order, is independent. Restricted to any dimension, data values are always distinct. Then, the average value of $\mathcal{N}$ is $O(\log^d N)$ where $\mathcal{N} = |R_N|$.*

Proof. Assume that we currently have M elements. We map each element $e = (x_1, x_2, ..., x_d)$ in P_N into a point $p_e = (x_1, x_2, ..., x_d, M - \kappa(e))$ in the $(d+1)$-dimensional space.

According to the definition of R_N, R_N corresponds to the set of skyline points, by the above mapping, of $\{p_e : e \in P_N\}$ in the $(d+1)$-dimensional space. From Theorem 2 in [3], this theorem immediately follows. □

Note that when d is small, $\mathcal{N}$ is much smaller than N if data follow the distribution in Theorem 2. Moreover, our initial experiment results also demonstrated that for low dimensionality $|R_N|$ is small even data do not follow the distribution assumptionsi in Theorem 2. The table in Figure 4 reports our experiment results for evaluating the size of R_N regarding different data distributions, N values ($N = 10^5$ or 10^6), and space dimensions (dim). In our experiment, we make the distribution of elements' arrival order independent to the data distribution, while the data distribution are either *independent*, or *correlated*, or *anti-correlated* [4].

3.2 Encoding R_N for n-of-N Queries

An element e in R_N may be dominated by many other elements in R_N that arrive earlier than e. Clearly, the number of the dominance relations among the elements in R_N

Dim	Inde-pendent		Cor-related		AntiCor-related	
	$N = 10^5$	10^6	10^5	10^6	10^5	10^6
2	65	120	16	38	179	287
3	226	411	40	55	1.3K	2.6K
4	894	1.9K	64	120	5.5K	14K
5	2.6K	5.6K	126	209	14K	47K

Figure 4. $|R_N|$

may be $O(\mathcal{N}^2)$ even with transitive reduced. One can imagine such an example where the dominance relations induce a full bipartite graph. Therefore, storing all the dominance relations is not only expensive in storage space but also too expensive to be maintained.

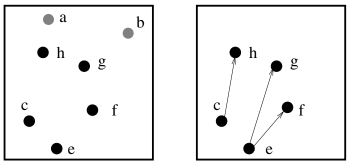
The example below shows that we do not have to keep all the dominance relations.

Example 1. *Suppose that the* 3 *data elements* a, b, *and* c *arrive according to their alphabetic ordering, and* c *is dominated by both* a *and* b.

Clearly, in this example if the dominance relation $b \to c$ is released due to the expiration of b then the dominance relations $a \to c$ has already been released, since a is expired earlier than b. Therefore, we only need to keep $b \to c$ to hold a "lock" on c.

In R_N, a dominance relation $e' \to e$ is *critical* if and only if e' is the youngest one (but older than e) in R_N, which dominates e; that is, $\kappa(e')$ is maximized among all the elements (other than e), in R_N, dominating e. Note that $\kappa(e') < \kappa(e)$ since R_N does not contain any redundant elements in P_N. In the example 1, $b \to c$ is critical. A graph on R_N is a *dominance* graph if the edge set consists of the critical dominance relations; it is denoted by G_{R_N}. We use $e' \xrightarrow{c} e$ to denote "e' critically dominates e".

It should be clear that the dominance graph is a *forest* as each element (as a vertex) has at most one incoming arc.



(a) Stream S (b) Dominance Graph

Figure 5. Dominance Graph

Example 2. *Assume that a data stream consists of data elements* a, b, c, e, f, g, *and* h *as depicted in Figure 5(a), which arrive according to their alphabetic order. Figure 5(b) illustrates the corresponding dominance graph after removing the redundant elements. Note that* $N = 7$ *in this example.*

The theorem below is fundamental to encoding the dominance graph to process n-of-N queries. It can be immediately verified according the definition of a dominance graph.

Theorem 3. *For a given* n *(*$n \leq N$*), an element* $e \in P_n$ *is skyline point for the* n*-of-*N *query if and only if either*

- *e is a root of the current dominance graph G_{R_N}, or*
- *there is an edge $e' \xrightarrow{c} e$ in G_{R_N} such that e' arrives earlier than the nth most recent element; that is, $\kappa(e') < M - n + 1 \leq \kappa(e)$.*

Here, M *is the number of the total elements seen so far.*

Our encoding scheme on G_{R_N} is quite straightforward: G_{R_N} is represented by its edges. That is, 1) every edge $e' \to e$ in G_{R_N} is represented by the interval $(\kappa(e'), \kappa(e)]$, and 2) each root e in G_{R_N} is represented by the interval $(0, \kappa(e)]$.

According to Theorem 3 an element e in G_{R_N} is in the answer of an n-of-N query ($n \leq N$) if and only if $\kappa(e)$ is the right end of an interval $(a, \kappa(e)]$ that contains $M - n + 1$. Based on the encoding scheme above, the problem of computing an n-of-N query is converted to the *stabbing query* problem with the stabbing point $M - n + 1$ that is discussed in section 2.3.

Let I_{R_N} denote the interval tree on the intervals obtained by the encoding scheme on G_{R_N}. We can process an n-of-N query as follows.

> Stab the intervals in I_{R_N} by $M - n + 1$, and then return the data elements e such that each $\kappa(e)$ is the right end of a stabbed interval.

Example 3. *Regarding the example in Figure 5,* a, b, ..., *and* h *arrived at time* 1, 2, 3, ... 7, *respectively. The dominant graph can be encoded by the following intervals:* $(0, 3]$, $(0, 4]$, $(3, 7]$, $(4, 5]$, *and* $(4, 6]$. *When* $n = 6$, $M - n + 1 = 2$ *as* $M = 7$. *Clearly, the intervals* $(0, 3]$ *and* $(0, 4]$ *are the results of stabbing query; consequently,* c *and* e *are the skyline points for the most recent* 6 *elements among the* 7 *already arrived elements.*

Since G_{R_N} is a forest, it is immediate that the number of intervals in I_{R_N} is $O(|R_N|)$. Consequently, our query processing algorithm runs in $O(\log \mathcal{N} + s)$ where $\mathcal{N} = |R_N|$. Clearly, the query processing time becomes $O(\min\{\log \mathcal{N}, d \log \log N\} + s)$, when the data distribution on each dimension is independent, based on Theorem 2.

3.3 Maintaining R_N & the Encoding Scheme

Upon the arrival of a new element e_{new} in the data stream, e_{new} is added to R_N and the oldest element e_{old} in R_N should be removed if it is expired. Therefore, R_N may have to be updated, as well as the interval tree I_{R_N}. Our algorithm is described in Algorithm 1.

The lines 3-8 in Algorithm 1 describe the updates if e_{old} in R_N is expired. Although the oldest element in P_N is always expired once a new element arrives, the oldest element e_{old} in R_N is not necessarily always expired. For example, regarding the stream in Figure 5 the oldest element c in R_N ($N = 6$) should not be expired if the next new element arrives. Note that e_{old} is always a root in G_{R_N}.

The lines 9-16 describe the updates of R_N and the interval trees by inserting e_{new}. Updating the interval trees

Algorithm 1 : Maintaining R_N & its Encoding Scheme

Description:

```
while new element e_new do
    if the oldest e_old in R_N is expired then
        R_N := R_N − {e_old};
        remove (0, κ(e_old)] from I_{R_N};
        for ∀e_old →^c e do
            update (κ(e_old), κ(e)] in I_{R_N} to (0, κ(e)]
        end for
    end if
    find D_{e_new} ⊆ R_N dominated by e_new;
    for ∀ e ∈ D_{e_new} do
        remove the intervals in I_{R_N} with κ(e) as an end
    end for
    R_N := R_N − D_{e_new} + {e_new}
    determine the critical relation e →^c e_new;
    add (κ(e), κ(e_new)] (or (0, κ(e_new)]) to I_{R_N}
end while
```

can be done in $O(\log \mathcal{N})$ time per update, as discussed in section 2.3. The critical issues are to compute$D_{e_{new}}$ and determine the critical dominance relation for e_{new} if it exists.

The problem of computing $D_{e_{new}}$ is the well-known *dominance reporting* [5, 19] problem. The research in this area has been focused on queries. The most efficient query algorithms are presented in [5, 19]; however no update techniques of their supporting data structures are reported. Therefore, these data structures are not quite applicable to our problem where updates are invoked by new data element arrivals.

As pointed out by [8], the most in-memory data structures for points are difficult to be balanced when data are updated. We propose to use the in-memory R-tree [2, 10, 8, 28] to organize R_N to support the two kinds of computation above. The data structures adopted in our algorithms are depicted in Figure 6.

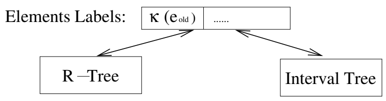


Figure 6. Data Structures Maintained

The label set $\{\kappa(e)\}$ for the elements in R_N are stored according to an increasing ordering. There is a 1-1 mapping to link the elements stored in the R-tree and the label set; the links between right (left) ends of intervals and the label set are also maintained. The label set and these links are maintained for constant time computing the relation between an interval end and the corresponding element, as well for determining if e_{old} is expired.

We propose to use **depth-first search** paradigm [22] on R-tree in Algorithm 1 to computing $D_{e_{new}}$. A node in R-tree is further expanded if e_{new} falls to the "candidate region" of its bounding box. For instance, in the 2-dimensional space, the shade region of Figure 7 (a) is the candidate region of the bounding box. If e_{new} falls in the shade region then the node corresponding to the bounding box will be further investigated; otherwise the subtree rooted at bounding box will be discarded in our search. Moreover, the whole subtree can also be discarded in our search if e_{new} falls in the l-corner as depicted; in this case all the elements in the subtree will be included in $D_{e_{new}}$. Similar situations can be identified in a d-dimensional space (for $d > 2$).

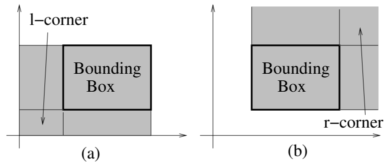


Figure 7. Dominance Testing

In our depth-first search, we immediately remove a new discovered element in $D_{e_{new}}$ from the R-tree but do not immediately balance the R-tree for every deletion. The R-tree is balanced after deleting every element in $D_{e_{new}}$. To effectively balancing of the R-tree after deletion of $D_{e_{new}}$, we adopt the $B+$-tree update strategy (bottom-up) combining with the technique in [2]. Moreover, to make the depth-first search effective we modify the bounding box of a node when the depth-first search backwardly returns to the node. For example, regarding the example in Figure 8 we modify the bounding box $B1$ to the bounding box $B4$ after deleting the subtrees rooted at $B5$ and $B6$, and then returning to $B1$ in the depth first search. This will prevent us from unnecessarily investigating any remaining children of $B1$.

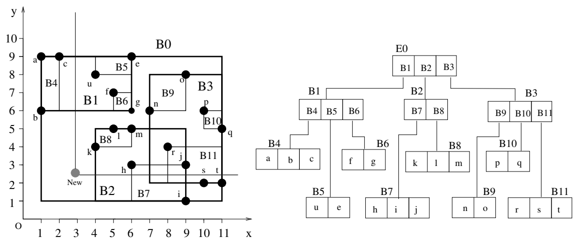


Figure 8. Example

We propose to use the **best first search** paradigm [12] on R-tree in Algorithm 1 to determining the critical dominance relation on e_{new}. At each node v in the R-tree, we propose to maintain the maximal value m_v of $\kappa(w)$ among all the data elements w in the subtree rooted at v. To efficiently do the best first search, the max-heap [7] on m_v among the nodes to be expanded is maintained, and then the heap top node is chosen to be expanded. The criteria of expanding the heap top node v are similar to those in the dominance reporting. As depicted in Figure 7(b), we expand v if and only if e_{new} falls in the shade area surrounding the bounding box; if e_{new} falls in the r-corner as depicted then the algorithm terminates and outputs the element e' with $\kappa(e') = m_v$ in the subtree rooted at v. We terminate the algorithm if the heap is empty or the current node v under investigation is an element e' that dominates e_{new}. Consequently, $(\kappa(e'), \kappa(e_{new})]$ is added to I_{R_N} if e'

exists; otherwise $(0, \kappa(e_{new})]$ is added.

We use the standard R-tree insertion technique [2] to insert e_{new} to the R-tree. As discussed in [28], the R-tree cost models are generally quite sophisticated. We are unable to analyse the time complexity of our search on a R-tree. However, our experiment results indicated that our R-tree based update technique is practically quite efficient when d is small; it is sufficient enough to support the real time updates against high speed stream. We have done the experiments up to $d = 5$.

3.4 Continuous n-of-N Queries

Continuous queries are issued once and run continuously to generate results along with the updates of the underlying datasets. With the arrival of a new element, the result S_n of an n-of-N query might be changed. A simple way is to re-run our query processing algorithm (stabbing query) in section 3.2 per arrival of a new data element; this requires $O(\log \mathcal{N} + s)$ per new element. In this subsection, we present a novel trigger based incremental algorithm with $O(\delta)$ time to update the current result and $O(\log s)$ time to update the trigger list per result change. Here, δ is the number of element changes from the current result to the new result. The correctness of our algorithm is based on the following proposition.

Proposition 1. *Once a new element e_{new} arrives, the current result S_n of an n-of-N query may have the following changes:*

- **Deletion:** *a data element $e \in S_n$ is removed if e_{new} dominates e or e is expired.*
- **Insertion:** *a data element $e \in R_N$ is added to S_n if in the updated G_{R_N} after applying Algorithm 1 for inserting e_{new}, either 1) e is e_{new} and $\nexists e'$ such that $\kappa(e') \geq M - n + 1$ and $e' \xrightarrow{c} e_{new}$, or 2) e is critically dominated by the just expired element e'' in S_n and e'' is not dominated by e_{new}.*

Here, M is the total number of elements.

Below is our algorithm - Algorithm 2.

In our algorithm we maintain a *min-heap* [7] on $\{\kappa(e) : e \in S_n\}$ for processing efficiency; that is, the heap top always has the smallest value. Here, e_{top} in the algorithm is the element in S_n corresponding to the heap top of the min-heap; it is expired from the most recent n elements if $\kappa(e_{top}) < M - n + 1$. Thus, every time we need only to check the heap top in the current solution to see if the trigger should be fired. If the trigger fires (i.e. $\kappa(e_{top}) < M - n + 1$), then the element on the heap top should be processed by Algorithm 2.

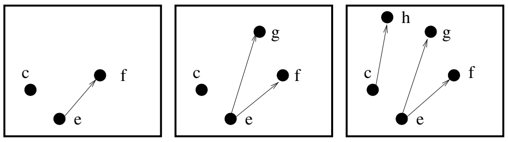
Note that $D_{e_{new}}$ is the set of new redundant elements dominated by e_{new} which are discovered by Algorithm 1. We assume the deletion of elements in $D_{e_{new}}$ in Algorithm 1 will notify the processing of a continuous query q. This can be efficiently done by linking an element e to the continuous queries which are using e as part of the result.

Algorithm 2 : Processing Continuous n-of-N Queries

Description:

1: **while** new element e_{new} **do**
2: $M := M + 1$;
3: **for** $e \in S_n \cap D_{new}$ **do**
4: Removal (e, S_n)
5: **end for**
6: **if** $\nexists e' \xrightarrow{c} e_{new}$ with $\kappa(e') \geq M - n + 1$ **then**
7: Add (e_{new}, S_n)
8: **end if**
9: **while** $\kappa(e_{top}) < M - n + 1$ **do**
10: Removal (e_{top}, S_n)
11: **for** $\forall e$ with $e_{top} \xrightarrow{c} e$ in G_{R_N} **do**
12: Add (e, S_n)
13: **end for**
14: **end while**
15: **end while**

In Algorithm 2, Removal (e, S_n) is to remove e from S_n and also remove $\kappa(e)$ from the min-heap - the trigger list; Add (e, S_n) is to add e to S_n and insert $\kappa(e)$ into the min-heap.



(a) a b c e f (b) b c e f g (c) c e f g h

Figure 9. An Example

Example 4. *Regarding the stream in Example 2 as depicted in Figure 5 (a), suppose that $N = 5$ and $n = 4$. The stream initially has 5 elements $\{a, b, c, e, f\}$. As new elements arrive, the n-of-N query results are updated as follows according to Algorithm 2. The result comprises $\{c, e\}$ for the n-of-N query and remains unchanged when the most recent N elements change from $\{a, b, c, e, f\}$ to $\{b, c, e, f, g\}$; Figures 9(a) and 9(b) showed the corresponding dominance graphs, respectively. Once the element h arrives, the new (updated) dominant graph is showed in Figure 9(c). Since $\kappa(c) = 3 < 7 - 4 + 1$, c is expired and thus h is added to the solution. Consequently, e and h form the answer once h arrives.*

The computation of the elements critically dominated by e_{top} (line 11) in Algorithm 2 can be done in $O(l)$ time by just following the links between R-tree to label set and the label set to interval trees, respectively. Here, l is the number of children of e_{top}. Consequently, it takes $O(\delta)$ time to update the current result and takes $O(\log s)$ [7] to update the min-heap per element change in S_n.

4 (n_1, n_2)-of-N Queries

In this section, we investigate the problem of processing (n_1, n_2)-of-N queries; that is, compute the skyline of the elements arriving between the most n_2th recent element and the most n_1th recent element in a data stream, where $n_1 \leq n_2 \leq N$. Unlike processing n-of-N queries, all elements in P_N need to be kept for processing all (n_1, n_2)-of-N queries; the reason is straightforward: n_1 could equal n_2. However, similarly to processing n-of-N queries the data structures to be maintained are the R-tree on R_N (the non-redundant elements in P_N), and interval trees. Below, we first characterize the property for a data element to be a skyline point for an (n_1, n_2)-of-N query.

An element e in the most recent N elements P_N may be dominated by many other elements in P_N. We use a_e to denote the youngest element e' that dominates e and arrives before e; that is,

$$\kappa(a_e) = \max\{\kappa(e') : e' \text{ dominates } e \text{ \& } \kappa(e') < \kappa(e)\}. \quad (1)$$

Similarly, we use b_e to denote the oldest element e' that dominates e and arrives after e; that is,

$$\kappa(b_e) = \min\{\kappa(e') : e' \text{ dominates } e \text{ \& } \kappa(e') > \kappa(e)\}. \quad (2)$$

In case that a_e (b_e) does not exist, a dummy data element e_0 (e_∞) is used to represent a_e (b_e) with $\kappa(e_0) = 0$ ($\kappa(e_\infty) = \infty$). Note that $a_e \rightarrow e$ has been defined as the critical dominance relation in the last section. We call $b_e \rightarrow e$ *backward critical dominance relation*, a_e the *critical ancestor* of e, and b_e the *backward critical ancestor* of e. It can be immediately shown that e is a skyline point of an (n_1, n_2)-of-N query if and only if its critical ancestor arrives earlier than the most n_2th recent element and backward ancestor arrives later than the most n_1th recent element.

Theorem 4. *An element e in P_N is a skyline point for an (n_1, n_2)-of-N query if and only if*

$$\kappa(a_e) < M - n_2 + 1 \leq \kappa(e) \leq M - n_1 + 1 < \kappa(b_e), \quad (3)$$

where M is the number of elements seen so far.

A graph, with the vertex set $P_N \cup \{e_0, e_\infty\}$ and with edge set consisting of the critical and backward critical relations, is called the *CBC dominance* graph of P_N; it is denoted by CG_{P_N}. Clearly, the number of edges is $O(|P_N|)$ since every data element (vertex) has only two incoming arcs.

Regarding the data stream in Figure 2, Figure 10 illustrates the CBC dominance graph if $N = 7$ after omitting the dummy vertices e_0 and e_∞ and the edges attached to them.

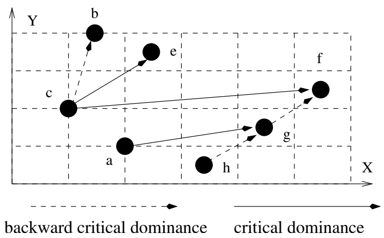


Figure 10. A CBC Dominance Graph

Theorem 4 is fundamental to our algorithms for processing an (n_1, n_2)-of-N query. As with section 3.2, CG_{P_N} can be represented by the edges that are encoded by interval trees.

A CG_{P_N} is encoded as follows. For each element e in P_N, use $((\kappa(a_e), \kappa(e)], \kappa(b_e))$ to represent its two incoming arcs $a_e \rightarrow e$ and $b_e \rightarrow e$. Then, we build the interval tree on the intervals $\{(\kappa(a_e), \kappa(e)] : e \in P_N\}$. To process (n_1, n_2)-of-N query, we apply the stabbing query algorithm by using $M - n_2 + 1$ to stab the intervals while checking the condition $\kappa(e) \leq M - n_1 + 1 < \kappa(b_e)$. Below is a description of our algorithm.

Algorithm 3 : Processing (n_1, n_2)-of-N Query

Description:

1: Stab the intervals by $M - n_2 + 1$;
2: **for** each element e in stabbing result **do**
3:     **if** $\kappa(e) \leq M - n_1 + 1 < \kappa(b_e)$ **then**
4:         return e;
5:     **end if**
6: **end for**

Note that in line 2 of the algorithm, "element e in the stabbing result" means the interval $(\kappa(a_e), \kappa(e)]$ is stabbed. It is immediate that Algorithm 3 runs in $O(\log N + l)$ where l is the number of intervals stabbed. Consider that an n-of-N query is a special case of a (n_1, n_2)-of-N query. To retain the same time complexity of processing n-of-N queries, we split the interval tree on CG_{P_N} into two interval trees as follows.

I_{R_N}: Intervals with the right end corresponding to an element in R_N.

R_{R_N-}: Intervals with the right end corresponding to an element in $P_N - R_N$.

It can be immediately shown that I_{R_N} consists of the intervals encoded from the dominant graph defined in section 3.2; thus, we use the same notation. The data structures used are depicted in Figure 11

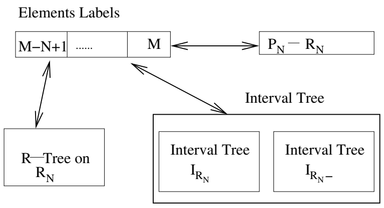


Figure 11. Data Structure Maintained

It is immediate that the backward critical dominance relations and the critical dominance relations may be determined iteratively against the current non-redundant elements R_N as follows. Once a new element e arrives, if an element e' in the current R_N is dominated by e then e is the backward critical ancestor of e'. If an element e'' in R_N critically dominates e then e'' is the critical ancestor of e. Consequently, the maintenance algorithm of CG_{P_N} via I_{R_N} and I_{R_N-} are similar to Algorithm 1. Below, in Algorithm 4 we present the maintenance algorithm by describing its similarity and difference with Algorithm 1.

It is immediate that the update costs of the R-tree on R_N are exactly the same as those in Algorithm 1. While update costs of the interval trees per new element is amortised to $O(\log N)$ since every data element moves at most once

Algorithm 4 : Data Structures Maintenance

- It always expires the oldest element in P_N; the subsequent deletion in R-trees and update of interval trees are the same as that in Algorithm 1. Inserting the new element e_{new} to R_N is also the same as Algorithm 1.
- The search paradigms on the R-tree of R_N are the same as those in Algorithm 1. That is, the depth-first search is used to determine the new redundant elements of which e_{new} is the backward critical ancestor, while the best-first search is used to determine the critical ancestor of e_{new}.
- The interval corresponding to the critical dominance will be added to I_{R_N} in the same way as in Algorithm 1. For a new redundant element e' caused by e, remove e' from the R-tree and place it in $P_N - R_N$. Then, remove $((\kappa(a_{e'}), \kappa(e')], \infty)$ from I_{R_N} and insert $((\kappa(a_{e'}), \kappa(e')], \kappa(e))$ to I_{R_N-}.

from I_{R_N} to I_{R_N-}.

Note that unlike an n-of-N query, to process a continuous (n_1, n_2)-of-N query it is necessary to keep some data elements that are not the current skyline elements. A space-efficient algorithm has been developed by us to minimize the number of candidate result elements to be kept. Due to the space limit, we are unable to present it in this paper.

5 Performance Evaluation

As mentioned earlier, there is no existing technique designed to support efficient computation of n-of-N and (n_1, n_2)-of-N queries with an effective on-line preprocessing. In our performance study, we implement the most efficient main-memory algorithm [17] for computing the skyline of a set of points and use it as a benchmark algorithm to evaluate our techniques. Below are the algorithms that have been implemented and evaluated.

KLP: The skyline computation algorithm [17].

nN: Our query processing algorithm for n-of-N queries; that is, the stabbing query processing algorithm.

mnN: Our algorithm (Algorithm 1) for continuously maintaining the data structures for supporting n-of-N queries.

cnN: Our algorithm (Algorithm 2) for processing continuous n-of-N queries.

n12N: Our query processing algorithm (Algorithm 3) for (n_1, n_2)-of-N queries.

mn12N: Our algorithm (Algorithm 4) for continuously maintaining the data structures for (n_1, n_2)-of-N queries.

All the experiments have been carried out on a Pentium 4 PC with a 2.8GHz processor and 1GB of main memory. As we do not have real data, we evaluate our techniques against the 3 most popular synthetic benchmark data, *correlated*, *independent*, and *anti-correlated* [4].

In our experiments, we evaluate the efficiencies of our algorithms, as well as the sensitivity and scalability against the data distributions, dimensionality, and window sizes. Our performance evaluation is conducted against the space dimensions from 2 to 5. This is because that in this paper our techniques focused on a lower dimensional space.

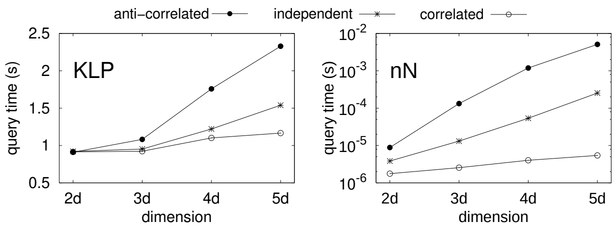
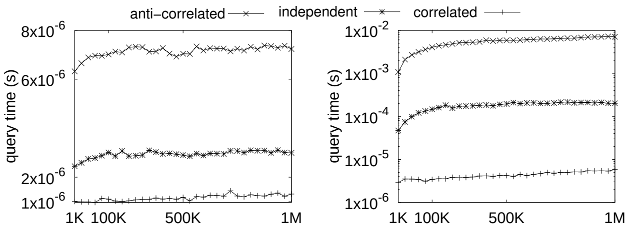


Figure 12. KLP vs nN



(a) 2d (b) 5d

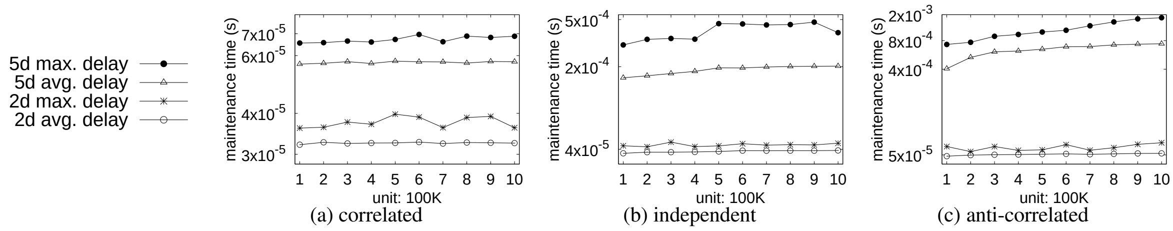
Figure 13. Performance against Different n

5.1 Evaluating n-of-N query algorithm

Note that the time complexity of nN is $O(\log \mathcal{N} + s)$ where $\mathcal{N} \leq N$. In most sliding window applications, we may expect that $N \leq 10^6$. A variation of N from 1K to 1M will not change the time complexities dramatically; the time complexity is mainly decided by s - the number of skyline points. Clearly, s should be determined by n ($n \leq N$) in an n-of-N query, data distributions, and the space dimensionality. In this subsection, we evaluate the efficiencies of nN against the space dimensions from 2 to 5, different data distributions, and different n values. In the experiments conducted in this subsection, we fix $N = 10^6$.

In this set of experiments, we randomly choose 1000 different n values varying from 1000 to 10^6. Each n is thus mapped to an n-of-N query with $N = 10^6$ to evaluate nN. The application of KLP to processing an n-of-N query is very straightforward - applying KLP to computing the skyline of the most recent n elements.

For each d ($2 \leq d \leq 5$), we generate 3 synthetic data sets with the distributions, correlated, independent, and anti-correlated, respectively. Each data set has 2×10^6 data elements. To simulate a data stream by a data set, we assign the relative arriving ordering of data elements according to their generation ordering in the synthetic data; by doing this we may expect that the data distribution of any most recent N elements can approximately retain the original data distribution of the whole data set. For each data stream, we randomly take 1000 snapshots of the most recent N elements P_N. For each P_N, the interval tree I_{R_N} is generated and then nN is executed against I_{R_N}. Meanwhile, in each P_N we take the most recent n elements and then apply KLP on the most recent n elements to compute the skyline. The experiment results are reported in Figure 12, where we calculate the average query processing costs of these 1K queries for each pair of data set and space dimension.



(a) correlated (b) independent (c) anti-correlated

Figure 14. mnN Performance

The results showed that the average processing time by KLP is more than 1 second. This indicates that KLP will cause a significant processing delay even against a data stream with a very low arrival speed - 1 element per second. Thus, it is not efficient enough to support on-line computation of n-of-N queries. Therefore, we no longer evaluate the performance of KLP in our performance study.

In the experiment, we also report the impact of n. We fix the space dimensionality by 2 and 5, respectively, then we report the query processing time against different n. Since the time of each execution of nN is too short to be recorded, we divided these 1K queries into 33 disjoint sets queries such that each set consists of about 33 queries with the consecutive values of n; then we record the average processing time for each set as one query processing time. The results are reported in Figure 13. The results showed that our query processing techniques are not very sensitive to the changes of n. However the dimensionality and data distribution have a great impact on the efficiency of our techniques; this is because they have a great impact on the value of s.

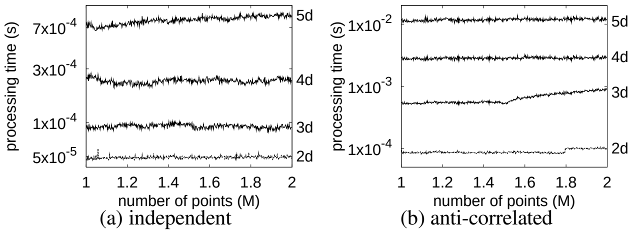
5.2 Efficiency of Maintenance Techniques: mnN

In this subsection, we study the performance of mnN - the algorithm for continuously maintaining the data structures for processing n-of-N queries. In this set of experiments, we choose two space dimensions $d = 2, 5$. For each of these two space dimensions, we generate 3 data streams in the same way as those in the last subsection, correlated, independent, and anti-correlated. Then we record the average cost and maximum cost, respectively, of processing one data item against different N values. Ten different N values are chosen; that is, $N = i \times 10^5$ for $1 \leq i \leq 10$. The experiment results are reported in Figure 14.

As illustrated, the correlated data has the best performance and the anti-corrected data has the worst performance. This is because correlated data leads to the smallest size of R_N on average, while anti-correlated data generates the largest size of R_N on average. The experiment results demonstrated that our maintenance techniques can support on-line update of data structures against a very rapid data stream. Even for $5d$ anti-correlated data streams, mnN can handle the element arrival rate about 500 elements per second in the worst case in real time. The experiment results also suggest that the update costs (average and maximum) per data item follow a logarithmic function regarding N.

5.3 System Scalability

In this subsection, we evaluate the scalability for system to handle a number of n-of-N queries against rapid streams. Since our techniques are based on sliding windows (i.e. one element in and one element out), it is not necessarily to evaluate a data stream with a very massive volume as long as there are enough sliding times. Moreover, in most sliding window applications it is quite rare to have $N > 10^6$. Furthermore, the performance study in the last subsection indicated that the data structure maintenance costs are more expensive when N gets larger, and anti-correlated data and independent data lead to more expensive maintenance costs than the correlated data.



(a) independent (b) anti-correlated

Figure 15. Overall Performance

By the above observations, in this set of experiments we choose $N = 10^6$, limit the data set size to 2×10^6, and do the experiment against two kinds of data streams, independent and anti-correlated. For each d ($1 \leq d \leq 5$), we generate two data streams (independent and anti-correlated) with 2×10^6 data elements in the same way as those in section 5.1. We also randomly generate 2×10^6 n-of-N queries and randomly assign them among the most recent 1M elements. Then, we run the algorithms, mnN to continuously maintain the data structures and run nN for processing n-of-N queries. We record the processing time between two consecutive data elements a and b, including the time of processing the queries between the two data elements and the time to maintain the data structures due to the former element a. Since such time is too short to be recorded, we use average time for processing 1000 elements as the processing time of one element; that is, the total processing time of 1000 elements divided by 1000.

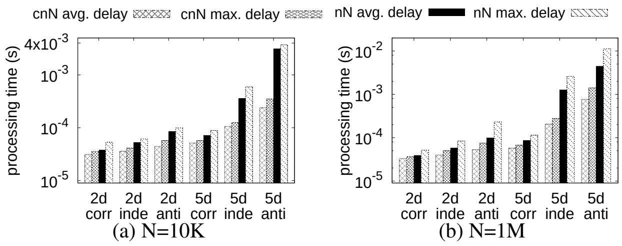
The experiment results are reported in Figure 15. Note that we report only the performance from the $10^6 + 1$th elements. This is because the window starts sliding for data structures maintenance when the window is full - having

10^6 elements. In fact, the initial costs are lower. In order to avoid a misleading, we cut the initial costs part.

The experiment results showed that for $d = 2, 3$, the system can support such a load on-line against a very rapid data stream with the arrival speed higher than 1K elements per second. The performance degenerates for anti-correlated data when $d = 4, 5$. However, for anti-correlated data, our techniques can still handle a rapid data stream on-line with the element arrival speed about 300 elements per second for $d = 4$ but can handle only a data stream with a medium arrival speed for $d = 5$ - about 80 elements per second.

5.4 Continuous n-of-N queries

Now we evaluate the performance of our continuous query processing techniques - cnN. To make a comparison, we also run our nN algorithm once per new data item arrival to continuously process an n-of-N query.

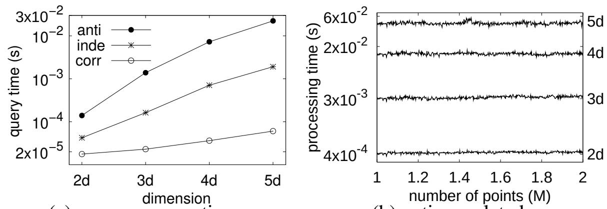


(a) N=10K (b) N=1M

Figure 16. Performance Evaluation of cnN

We use $2d$ and $5d$ data to do the evaluation. The 6 data streams are generated in the same way as those in section 5.2. We choose N = 10K and 1M. In the system, 20 n-of-N queries are generated such that 10 for N = 1M and 10 for N = 10K. For N = 10K (N = 1M), these 10 queries are with $n = i \times \frac{N}{10}$ (for $1 \leq i \leq 10$), respectively. We record the average delay (processing time) and maximum delay of an element, respectively. Note that a delay of an element e means the processing time involving processing e before processing next element; this includes the data structure maintenance costs and query processing costs. Again to record precisely such a delay per element, we use the average delay per 1000 elements instead.

Figure 16 reports the experiment results. It is interesting to note that running nN per new data element also has a very reasonable performance especially in a lower dimensional space, while our cnN technique can support such a system work-load against very rapid stream with an arrival speed higher than 1000 elements per second.



(a) average query time (b) anti-correlated

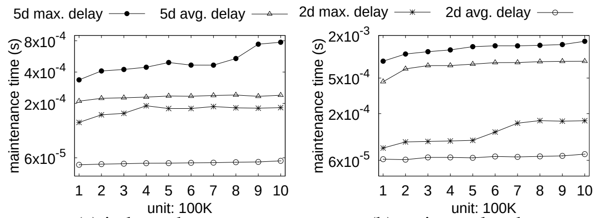
Figure 17. Performance of n12N

5.5 (n_1, n_2)-of-N Processing Techniques

In this subsection, we conduct experiments to evaluate the performance of our techniques, n12N, mn12N, and cn12N for processing (n_1, n_2)-of-N queries.

The settings of the first experiment are the same as those in section 5.1 except that we randomly generate 1K (n_1, n_2)-of-queries with the constraint that $n_2 - n_1 \geq 500$. We record the average query processing time of these 1K queries for a d (space dimension) and a data stream (via its snapshots). The results are reported in Figure 17(a). The performance of n12N follows a very similar pattern to that of nN; however it is slightly slower due to the fact that n12N has to stab the elements more than required by a query.

In the second set of experiments, we evaluate the system scalability by applying n12N and m12N. The settings and the experiments are the same as those in section 5.3 except the randomly generated 2×10^6 queries are now (n_1, n_2)-of-N queries. Figure 17(b) reports the experiment results against anti-correlated data for $d = 2, 3, 4, 5$, respectively. The experiment results indicated that our techniques can support the system workload of on-line processing 2M queries (though not always simultaneously) against very rapid stream with arrival speed higher than 1K elements/second for $d = 2, 3$. The performance significantly drops when d is increasing. For $d = 4$, we may still be able to handle a stream with a medium arrival speed - about 70 elements/second, while for $d = 5$ we can only handle a slow data stream with the arrival speed about 22 elements per second.



(a) independent (b) anti-correlated

Figure 18. Performance Evaluation of mn12N

We repeat the experiments in section 5.2, except the algorithm mn12N is employed this time, for the performance evaluation of our continuous maintenance techniques of the data structures to support (n_1, n_2)-of-N queries. The results are reported in Figure 18 for independent data and anti-correlated data. These experiment results confirmed our theoretical analysis that mn12N and mnN should have about the same efficiency.

5.6 Summary of the Performance Study

As a short summary, our experiment results clearly demonstrated that our on-line skyline computation algorithms are very efficient in practice besides the theoretical complexity guarantees. They also showed that the techniques for continuously maintaining the proposed data structure can on-line support very rapid data streams. Our processing techniques for ad-hoc queries combining with our data structure maintenance algorithms can process a massive number of queries on-line against very rapid data streams in a lower-dimensional space ($d = 2, 3, 4$), and but are only able to handle streams with medium or low arrival speed for $d = 5$.

Note that our continuous query processing techniques in the paper may not be able to support the processing of massive continuous queries by a single CPU resource against data streams, though they perform greatly for a small number of continuous queries. This is partially due to the inherent difficulty of continuously processing skyline queries - there could be many skyline points involved; thus it could be expensive to materialize many query results.

6 Conclusions

In this paper, we presented novel techniques for on-line skyline computation over the most recent n elements (for any $n \leq N$) in a rapid data stream. While many researchers are working on the optimal off-line processing of skyline queries, this work is among the first attempts to develop efficient incremental techniques to on-line support skyline computation. Our algorithms are not only efficient and scalable in practice but also have theoretically guaranteed performance. Moreover, we also extend the techniques to cover an arbitrary window query in the most recent N elements. Our experiment results demonstrated that the techniques can be used to process rapid data streams in lower dimensional spaces with the space dimension not greater than 5.

Note that if we replace the element position labels by element arriving time then our techniques can be immediately applied to the most recent elements specified by a time period. As a possible future work, we will investigate if the maintenance of our data structures may be carried out by algorithms with theoretical guarantees. We will also investigate the problem of approximate skyline computation over data streams.

Acknowledgement. The research of the first and the second authors was partially supported by the ARC Discovery grant - DP0346004.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *POS*, 2002.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *SIGMOD*, 1990.

[3] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors. *JACM*, 1978.

[4] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. *ICDE*, 2001.

[5] B. Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *Discrete & Computational Geometry*, 1987.

[6] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. *ICDE*, 2003.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[8] V. Gaede and O. Gunther. Multidimentional access methods. *Computing Surveys*, 30(2):170–231, 1998.

[9] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *STOC*, pages 471–475, 2001.

[10] A. Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD*, 1984.

[11] J. Hershberger and S. Suri. Convex hulls and related problems in data streams. *MPDS*, 2003.

[12] G. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[13] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. *SIGMOD*, 2001.

[14] S. Kapoor. Dynamic maintenance of maxima of 2-d point sets. *SIAM J. Comput.*, 2000.

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

[16] N. Koudas, B. C. Ooi, K. L. Tan, and R. Zhang. Approximate nn queries on streams with guaranteed error/performance bounds. *VLDB*, 2004.

[17] H. Kung, F. Luccio, and F. Preparata. On finding the maximum of a set of vectors. *JACM*, 22(4):469–476, 1975.

[18] X. Lin, H. Lu, J. Xu, and J. X. Yu. Continuously maintaining quantile summaries of the most recent n elements over a data stream. In *ICDE*, 2004.

[19] C. Makris and A. Tsakalids. Algorithms for three-dimensional dominance searching in linear space. *IPL*, 1998.

[20] D. Mclain. Drawing contours from arbitrary data points. *Computer Journal*, 1974.

[21] K. Mehlhorn. *Data Structures and Algorithms: 3. Multidimensional Searching and Computational Geometry*. Springer, Berlin, 1984.

[22] F. V. N. Roussopoulos, S. Kelly. Nearest neighbor queries. *SIGMOD*, 1995.

[23] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal progressive alogrithm for skyline queries. *SIGMOD*, 2003.

[24] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[25] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.

[26] R. Steuer. *Multiple Criteria Optimization*. Wiley New York, 1986.

[27] K. Tan, P. Eng, and B. Ooi. Efficient progressive skyline computation. *VLDB*, 2001.

[28] Y. Theodoridis, E. Stefanakis, and T. Sellis. Efficient cost models for spatial queries using r-trees. *IEEE TKDE*, 2000.