Efficiently Processing Continuous k-NN Queries on Data Streams

Christian Böhm[1], Beng Chin Ooi[2], Claudia Plant[3], Ying Yan[4]
[1]: *University of Munich, Germany, boehm@ifi.lmu.de*
[2]: *National University of Singapore, ooibc@comp.nus.edu.sg*
[3]: *UMIT, Austria, claudia.plant@umit.at*
[4]: *Fudan University, China, yingyan@fudan.edu.cn*

Abstract

Efficiently processing continuous k-nearest neighbor queries on data streams is important in many application domains, e. g. for network intrusion detection. Usually not all valid data objects from the stream can be kept in main memory. Therefore, most existing solutions are approximative. In this paper, we propose an efficient method for exact k-NN monitoring. Our method is based on three ideas, (1) selecting exactly those objects from the stream which are able to become the nearest neighbor of one or more continuous queries and storing them in a skyline data structure, (2) delaying to process those objects which are not immediately nearest neighbors of any query, and (3) indexing the queries rather than the streaming objects. In an extensive experimental evaluation we demonstrate that our method is applicable on high throughput data streams requiring only very limited storage.

1. Introduction

Query processing on data streams has become very popular in recent years, e.g. [3, 4, 5, 12] to mention a few. There is is a vast number of solutions for various types of data, e.g. relational, semi-structured and time series, but most existing approaches are approximate because of the special conditions in a streaming environment. In contrast to conventional query processing, where queries are immediately answered from a database with a large but finite and previously known number of objects, in stream based query processing the queries are *subscribed* to a data stream. This means that, upon every arrival of a new object from the stream, the set of registered queries has to be checked. If the new object qualifies for one or more queries, it is reported as a result of the queries. There are two main challenges: The streaming objects arrive with a very high frequency and usually not all of them can be stored in the system. As an application scenario, consider network monitoring. The system administrators may specify various network packages which are suspicious due to different reasons (intrusion, rule violation, misuse etc.) Then, the stream of network packages is constantly surveyed for packages which are similar to the suspicious objects. As a second example, take advertisements of an arbitrary market segment. Each user may specify the properties (price, color, size, weight etc.) of a product he/she is interested in. The system permanently informs the user about those advertisements which fit best his/her requirements.

We distinguish between two different types of similarity queries: range queries and nearest neighbor queries. For both types, the user selects an object, the *query object* which is the starting point of the search. For a range search, the user must additionally specify the query radius, i.e. a threshold for the maximally allowed distance from the query object. Since similarity measures are often not very intuitive, it may be difficult to specify such a query radius. Therefore, in practice, the k-nearest neighbor query (k-NN) is more important, because the user only has to specify k, the number of objects that he wants to retrieve, and the system automatically retrieves the k *most* similar results. In this paper, we focus on k-NN queries but our technique can be extended to range queries in a straightforward way.

We define for each object a life span in which the object is valid. In many applications such as advertisements, the objects themselves may be associated with a time of expiry. If this is not the case, the time of validity can either be specified globally for the whole system or may be individually specified for each query. By specifying a global or query-specific lifespan, the user is also enabled to control how frequently he/she wants to get a query result. E.g. if the life span is set to one hour, the user will get at least every hour a new result (but maybe also some additional results if good hits arrive before the current nearest neighbor expires).

In addition to global, query-specific, and object-specific expiry, we can also distinguish between time-based and number-based expiry. In time-based expiry, the life span

1-4244-0803-2/07/$20.00 ©2007 IEEE.

of an object is defined in terms of e.g. seconds. In contrast, for number-based expiry, the user defines a maximum number of objects n which is at all times simultaneously valid. After the object number $n+1$ has arrived from the stream, object number 1 automatically expires. Number based expiry is also practically useful because n gives a intuitive quality measure for the results: each result is *the best out of n objects.* Number based expiry is useful for global and query-specific expiry, but not for object-specific.

We will also distinguish between monotonic and non-monotonic expiry. Monotonic expiry means that objects expire in the same order as they have arrived from the stream. Non-monotonic expiry is only possible in combination with object-specific expiry. In this context, the queries have also a life span which starts at the time stamp of subscription $subscr(q)$ and ends at the time stamp of unsubscription $unsubscr(q)$.

Problem Specification. We consider a data stream S as a sequence of objects $(o_1, o_2, ..)$. Besides its coordinates, each object has a time stamp of its appearance at the stream, denoted by $app(o)$ and a time stamp of expiry, denoted by $exp(o)$. We call the time span between an objects appearance and its expiry the life span $l(o)$ of an object o. In this time the object is *valid*.

Definition 1 (Valid objects.) *For a given timestamp t the set of valid objects V_t is defined as*
$V(t) = \{o \in S | app(o) \leq t \leq exp(o)\}$.

Definition 2 (Valid objects w.r.t q.) *For a given time stamp t the set of objects which are valid for query q, denoted $V_q(t)$ corresponds to*
$V_q(t) = \{o \in V(t) | subscr(q) \leq app(o) \leq unsubscr(q)\}$.

Definition 3 (k-NN.) *Let $dist$ be a metric distance function. The k-nearest neighbors of a query q at a given time stamp t, $NN_t^k(q)$ is the minimal subset of valid objects $V_q(t)$ containing at least k elements with*
$\forall o \in NN_t^k(q) \wedge \forall p \in V_q(t) \setminus NN_t^k(q) : dist(o,q) < dist(p,q)$.
We denote by $NN_t(q)$ the nearest neighbor of a query q, i.e. $NN_t(q) = NN_t^1(q)$. Wherever non-ambiguous we write NN.

Conceptually at every time stamp all subscribed queries are evaluated. If there are changes among the k-NN of a query, they are reported to the user who owns the subscription. The result set of the system at a time stamp contains all objects that need to be reported.

Definition 4 (Result set.) *For a given time stamp t and a set of queries Q the result set $R(t)$ consists of the following objects o:*
$R(t) = \{o | \exists q \in Q : o \in NN_t^k(q) \wedge o \notin NN_{t-1}^k(q)\}$.

Contributions. The key contributions of our approach can be summarized as follows:

1. We propose a framework for exact k-nearest neighbor query processing on data streams.
2. We demonstrate how the basic idea of a skyline can be exploited for continuous k-nearest neighbor queries to assure exact answer guarantee at very low memory consumption.
3. We further reduce memory consumption by object delaying without giving up the exactness property.
4. An efficient index structure for continuous queries is provided. This index allows highly dynamic updates and is small enough to fit into main memory.

Our paper is organized as follows: The next section is dedicated to related work. In Section 3 we introduce our idea for exact NN processing using a skyline data structure. In Section 4 we show how we can further reduce memory consumption by object delaying. Section 5 gives a detailed description of our index structure used for the queries. Section 6 illustrates how the result reporting is implemented and Section 7 extends our method to k-NN queries. In Section 8 we provide an extensive experimental evaluation. Section 9 concludes the paper.

2. Related work

k-NN Queries on Data Streams. k-NN queries on static databases is a well studied problem for which many index structures have been proposed, e.g. [15, 16, 1]. Most of them are not suitable in our context because of the high throughput. In the special case of streaming time series research activities mainly focus on discovering patterns to predict the coordinates of new objects. This information is used to discard objects that are probably irrelevant to the query and improve response time of the system, e.g. [4]. Prediction-based methods are not applicable if the objects attribute values are independent from their time of appearance and the objects arrived before. For this case approximate approaches have been proposed, such as [7]. The data space is partitioned into cells and a $B+$ tree together with a Z-curve is used to index the space. For each grid cell some objects are retained to guarantee an absolute error bound for k-NN queries. However, in some applications exact answers are essential, e.g. in health monitoring. In any case exact answers are of highest compliance to the user and give a sound base for data analysis and interpretation.

Skylines and Query Monitoring. In this paper, we use a skyline based object buffer associated with each

query, which we call the *query skyline*. This allows us to discard most of the objects from the stream immediately without giving up the exact answer guarantee (cf. Section 3). In general, the skyline of a data set contains the data points which are maximal or minimal in two or more of the attributes. The problem was first proposed by [2]. It attracted much attention both on static data sets, e.g. [13, 10] and streaming data [8]. Papadias et al. introduced the concept of *k-skyband* [10] which is also related to our work. The k-skyband of a data set contains all objects which are dominated by at most $k-1$ other objects. In [9] an interesting algorithm for monitoring top-k-queries is proposed. The k-skyband has geometric implications on the data space, leading to areas that are excluded to contain relevant objects. In this approach all objects are indexed in a grid and the order of processing of the cells is determined by the k-skyband. However, this approach relies on the assumption that all valid objects can be kept in memory.

Query Indexing. In a data stream context it is usually impossible to index all the data objects because of memory limitations. Typically the number of queries is much smaller than the number of valid objects and naturally the system needs to have enough memory to store them. The idea of indexing the queries instead of the data objects has been successfully applied in the context of moving objects [17]. Organizing queries in a grid index outperforms object indexing for a large number of moving objects, because the objects are highly dynamic causing a lot of index updates. In our context indexing the queries instead of the objects means that we change the problem of k-nearest neighbor search (of the objects) into a bi-chromatic reverse k-nearest nearest neighbor search (of the queries), for more details see Section 5. The problem of RNN has been extensively studied for static databases. A lot of sophisticated tree-based index structures have been prosed, e. g. [6, 11, 14]. For indexing the queries these structures are not flexible enough to support frequent updates and lead to a memory overhead which is not necessary. We propose a grid-style index which is described in detail in Section 5.

3. Skyline Based Object Maintenance

Usually only a very small fraction of the objects arriving from a data stream is relevant to a query. For high throughput streams also only a small fraction of the objects can be stored in the system. These general conditions require a decision strategy to discard irrelevant objects. Most of existing proposals focus on sophisticated filtering and load shedding strategies carefully selecting objects which are considered as unimportant and can be discarded. Various decision strategies with error bounds have been proposed but exact answers can not be guaranteed (cf. Section 2). We develop a criterion to decide upon arrival of a new object from the stream if it may become the nearest neighbor in future, or can definitely not. Among the stored potential relevant objects pruning is performed. The basic idea is that a new object can often exclude many other objects which can not become nearest neighbors until the new object expires. In this section, we restrict ourselves to the case where the queries are continuous 1-nearest neighbor queries, continuous k-nearest neighbor queries for $k > 1$ will be considered in Section 7.

To decide whether or not an object o may become the nearest neighbor of a query q we consider the 2-dimensional space of the query distance and the time of expiry. Note that this space is always 2-d, regardless of the dimensionality of the feature space of the data and query objects. Our method is applicable to objects of arbitrary dimensionality, and can even be extended to non-vector metric objects. An object o may become the nearest neighbor, unless there exists another object p which at the same time (1) is *closer* to the query than o and which (2) *expires later* than o. If both conditions hold, then o cannot be the nearest neighbor now (because at least one closer object p is known). Moreover, it cannot become the nearest neighbor later, because the object p lives longer. A set of objects which are maximal (minimal) with respect to two (or more) different conditions (such as, in this case, distance and expiry) is called a *skyline* [8]. In this context, we are considering a two dimensional distance-expiry skyline, which we call the *query skyline*, formally:

Definition 5 (Query Skyline.) *Given a time stamp t, a query q and two points $o, p \in V_q(t)$. We say o* dominates *p if $exp(o) > exp(p)$ and $dist(o,q) < dist(p,q)$. The query skyline $s(q,t)$ of query q and the set of valid points $V_q(t)$ consists of all points in $V_q(t)$ that are not dominated by other points in $V_q(t)$.*
$s(q,t) = \{o \in V_q(t) |$
$\nexists o' \in V_q(t) : exp(o') > exp(o) \wedge dist(o',q) < dist(o,q)\}$.

In the following, we will formally prove that the objects of the skyline in the distance-expiry space (and only these objects) need to be stored as potential nearest neighbors. The current nearest neighbor is also in the skyline at all times.

Lemma 1 (Correctness.) *At each time stamp t the query skyline $s(q,t)$ of each query contains $NN_t(q)$ the nearest neighbor of q at time stamp t.*

Proof 1 *$NN_t(q)$ is by definition of the nearest neighbor not dominated by any other object $o \in V_q(t)$, i. e. $\nexists o \in V_q(t) : dist(o,q) > dist(NN(q),q)$.*

```
algorithm skylineMaintenance (Object o, Query q)
Skyline s := q.associatedSkyline;
SkylineElement c := s.last; // current element
SkylineElement c';
if nonMonotonicExpiry
  // search for suitable position:
  while c ≠ null and exp(c) > exp(o)
    c' := c;
    c := c.pred;
  if c ≠ s.last and dist(c', q) < dist(o, q)
    return; // o is dominated, leave s unchanged
c := skylinePruning(o, q, c);
if c = null
  s.insertAsFirst(o);
  reportNeighbor(o, user(q));
else
  s.insertAfter(o, c);

function skylinePruning (Object o, Query q,
                         SkylineElement c): SkylineElement
SkylineElement c';
while c ≠ null and dist(c, q) > dist(o, q)
  c' := c;
  c := c.pred;
  s.delete(c');
return c;
```

Figure 1. Skyline Maintenance.

Lemma 2 (Completeness.) *A valid object o which is not in the query skyline $s(q,t)$ at time stamp t, is the nearest neighbor at no time $t' \geq t$.*

Proof 2 *Since o is not in the skyline, it is dominated by another object p, i.e. $exp(p) > exp(o)$ and $dist(p,q) < dist(o,q)$. Therefore, o is not the element having minimal distance to q for any time stamp $t' \in [t..exp(p)]$. Since $exp(o) < exp(p)$, o is not valid after that time interval, and thus, o is additionally not the nearest neighbor for any $t' \in [exp(p)..\infty[$.*

We can represent the skylines in main memory as double-linked lists, ordered by time of expiry (and at the same time, automatically ordered by distance, because the monotonicity of the objects in the skyline can be easily proven). When a new object o arrives from the data stream, we first have to locate the potential position of this object in the skyline. We start our search at the upper end, because in most applications the objects have at least approximately uniform life spans. If the potential position is not at the end, we have to check if the new object is excluded by the object immediately following the potential position. It is also a consequence of the monotonicity that, if the immediate successor does not exclude the new object, then none of the (transitive) successors can exclude the object. If o is not excluded, it needs to be appended and we proceed as follows: We test whether o excludes its direct predecessor. Due to monotonicity, if o does not exclude the direct predecessor, it

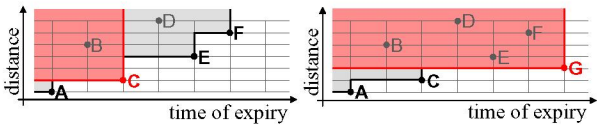

Figure 2. Skyline Example.

cannot exclude any of its transitive predecessors, and we're done. Otherwise, the dominated object is removed from the linked list, and we do the same test with the new predecessor (if any). The pseudocode of the algorithm is depicted in Figure 1. An example is visualized in Figure 2, where we have a set of 6 objects, A-F. Two objects, B and D are dominated, i.e. the actual skyline is stored in the linked list A-C-E-F. As an example, object C is highlighted, along with the space which is excluded by C (containing the object B). When a new object G arrives it is simply appended, since the expiry is assumed monotonic here. We have to consider the two predecessors F, and then E, and discard them. Since C is not dominated by G none of the predecessors (A) must be checked.

An interesting question is how many objects do we have to expect to be in the skyline. It has been shown in [8] that, under reasonable assumptions, the size of the skyline is in $O(\log_2 n)$. In our first experiments we were able to confirm a typical size of 10 elements for $n = 1,000,000$ valid objects. Although this is very moderate, we will present in Section 4 an idea to further reduce the typical skyline size to 2-3 objects.

Mindful readers may have noticed that in case of monotonic time stamps, the newest object must always be appended to the skyline of every query. Therefore, one might wonder if it is really necessary to access all stored skylines for every new object (which would make query indexing, described in Section 5 obsolete). The answer is yes, but we will propose a simple trick in the next section (4) which greatly reduces this effort without affecting the correctness and completeness of the result of our algorithm.

4. Object Delaying

New objects cannot be dominated by other objects in the query skylines because they have the latest expiry. These unnecessarily appended objects are discarded soon by new objects which are closer to the query. Therefore the size of the query skylines stays logarithmic despite of this problem. But for efficient online stream processing it is essential to avoid any overhead in space and time. The skyline size can be further reduced to a few objects by a simple but effective trick. We delay every new object o from the stream for a certain time span before trying to insert it into the ac-

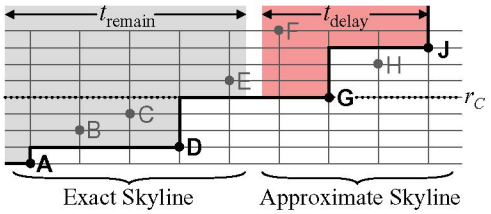

Figure 3. Approximate and Exact Skyline.

tual query skylines. By doing so we avoid inserting o into the skyline of a query q which is far away from o (where o qualifies *only* because it is new). It is likely that while o is delayed, objects from the stream arrive that are closer to the query. Naturally these new objects dominate o (closer to the query and also later expiry) so that o can be safely discarded. Very few objects need to be immediately inserted because they are really relevant, e.g. because they replace the current nearest neighbor or have the potential to become nearest neighbor during the delay time. In the following we explain in detail how to determine the objects requiring immediate insertion. We introduce a simple ring buffer to store the new objects temporarily.

Definition 6 (Buffer.) *The buffer B with size $|B|$ stores the $|B|$ newest objects from the stream, it holds: $\forall o \in B$, $\forall p \notin B$: $exp(o) > exp(p)$. We additionally denote by t_b the earliest expiry among all elements in the buffer. The set of objects in the buffer at time stamp t is denoted by delayed(t).*

In this buffer objects are processed in first-in-first-out order if the objects have monotonic expiries (for non-monotonic expiry, we either use a priority queue or we use a fifo-buffer as well, but in the latter case we only insert such objects the life span of which is at least two times the delay time, cf. Lemma 3). In addition we divide the skyline of a query into two parts. We call the first part of the skyline the *exact skyline*. Objects are inserted into this skyline when they come out of the buffer and qualify to be inserted then.

Definition 7 (Exact Skyline.) *The exact skyline of a query q at time stamp t contains all objects $o \in V_q(t) \setminus$ delayed(t) which are not dominated by other elements $p \in V_q(t) \setminus$ delayed(t) or by elements of the approximate skyline at time stamps $t_b..t$.*

The other part of the skyline is called *approximate skyline*. It consists of new objects requiring immediate insertion. When a new object o arrives it is appended to all approximate skylines for which it could be possible that object o could become the nearest neighbor immediately or during the time when the object is delayed in the ring buffer. Additionally, it is appended to some of the approximate skylines for which o is also a "good" object, according to an arbitrary strategy. For maintenance and good performance of our index structure described in the next section it is beneficial to have some "good" objects in the approximate skylines. In our system, the queries are stored in a grid based index (cf. Section 5). Our strategy is, therefore, to insert the new object o to all those queries which are stored in the grid cells which encompass o. Formally, the approximative skyline consists of all non-dominated objects from a subset of the delayed objects:

Definition 8 (Approximate Skyline.) *Let S be an arbitrary subset of delayed(t) which contains all objects with a distance below a cut radius r_C to q. The approximate skyline of q is the set of elements from S which are not dominated by other elements from S.*

Note that the objects in the exact skyline need to be physically stored there. In contrast, for the approximate skyline, it is sufficient to store pointers to the objects only, because they are physically stored in the delay buffer. Whenever the delay buffer is full (usually upon every new insertion of an object) we remove the oldest object p from it (remove it also from all approximate skylines) and check, to which of the exact skylines it must be appended. It must be appended to those exact skylines for which no object is known which dominates p. However, in the meantime (during the delay of p), we have seen many new objects with later expiry times. Our hope is that for each query at least one of these new objects is very close, and, therefore, p is indeed excluded now. Provided we have appended any good object to the approximate skyline, we can use the top element of the approximate skyline for discarding p. All elements in the approximate skyline have later expiries than p. Therefore, p is discarded if the top element of the approximate skyline is closer to the query than p. Therefore, the top element of the approximate skyline exactly defines the cut radius r_C which is used for query indexing (cf. Section 5).

In the last paragraph, we have seen that an object can be safely discarded if it is dominated by the top element of the approximate skyline. In the extreme, the top element of the approximate skyline can dominate all elements of the exact skyline. The top element of the approximate skyline will become the nearest neighbor of the associated query. In this case, the question is justified whether or not this top element is really the nearest neighbor, and under what conditions this can be guaranteed. We will show in the following that this is guaranteed if (1) every object is inserted into the approximate query skyline which is closer to the query than the current top element, and (2) the remaining time of validity after being inserted into the exact skyline t_{remain} is at least as high as the delay time t_{delay}.

Lemma 3 *Assume that the above conditions hold, and that an object o is not appended to the approximate skyline of*

query q. Then, it is guaranteed that o cannot become the nearest neighbor of query q until o is removed from the delay buffer (and inserted into the exact skyline).

Proof 3 *According to the definition of the approximate skyline, o has a distance to q which is larger than r_C, the distance of the top element o_t of the approximate skyline. Therefore, it cannot become the nearest neighbor of q until o_t expires. The top element o_t is still delayed ($o_t \in delayed(t)$) but may be extracted from the delay buffer in an arbitrarily short time. But then, it is still valid for t_{remain}. The new object o remains in the buffer for t_{delay}. If $t_{delay} \leq t_{remain}$ then o cannot become the nearest neighbor of q during the delay time.*

Note that, after the delay time, o is considered to be inserted into the exact query skyline. To perform this test, o is compared to the *new* top element of the approximate skyline. If o is dominated by the new top element, it is guaranteed that o will become never the nearest neighbor of q.

An example of a pair of exact and approximate query skylines is shown in Figure 3. The exact skyline consists of the objects A and D and the approximate skyline of the objects G and J. B and C are dominated by D. E would actually belong to the exact skyline but is dominated by the top element of the approximate skyline G which also defines the cut radius r_C. H should actually belong to the approximate skyline. But according to the definition of the approximate skyline, it is possible to discard H as it is above the cut radius. According to Lemma 3, H cannot become the nearest neighbor before G expires which takes at least t_{remain} time. Before the expiry of G the object H is taken out of the buffer and is tested for insertion into the exact skyline. The same holds for J, but it is inserted anyhow because it is in the same grid cell as the query.

5. Query Indexing

Due to object delaying (cf. Section 4) the number of objects of the stream that need to be stored in the query skyline is very small. In addition, the streaming objects are highly dynamic, so it would not be beneficial to use an index structure for the objects. In this section we propose to index the queries in a grid-style index, which is highly flexible w.r.t. updates and can be held in main memory.

According to Definition 3, at each time stamp t the answer of a continuous nearest neighbor query q is that object which has the smallest distance to q among all objects $o_j \in V_t(q)$ which are valid at time t. We are now interested in the question, given a new (and by definition, valid) object o at timestamp t, to which queries $q \in Q$ is o the nearest neighbor among all valid objects $o \in V_t(q)$? We can write this formally in the following way: $q \in Q : NN_t(q) = \{o\}$.

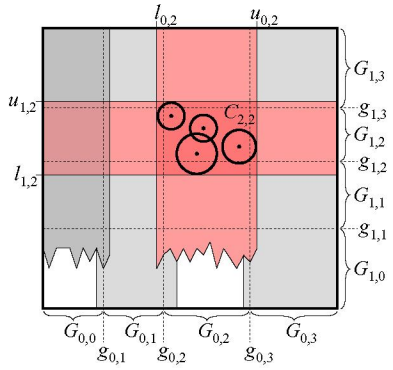

Figure 4. Query Indexing.

One might recognize that this resembles to the definition of the reverse nearest neighbor problem. More exactly, the object o searches for the *bichromatic* reverse nearest neighbor among the queries, as defined, e.g. in [11].

For the efficient evaluation of a bichromatic reverse nearest neighbor query, usually the following consideration is applied: o is the nearest neighbor of those $q \in Q$ to which it is closer than the current nearest neighbor of q. This can be easily decided if the radius $r(q)$ of the current nearest neighbor is stored with every query $q \in Q$. For continuous NN-queries which are supported by a delay buffer we need a cut radius r_C which is guaranteed to be an upper bound of the nearest neighbor radius for t_{remain} time. As discussed in Section 4, r_C corresponds to the distance between q and the top element of the approximate skyline. Instead of considering the queries as points of a metric or vector space, we consider them as spheres with a radius $r_C(q)$.

For indexing, we merely need a structure which allows us to efficiently determine all spheres in which the point o is contained. We apply a very simple, grid based method for vector objects which is described in the following. Note that this is used for storing the queries only. Therefore, it might fit into main memory for a moderately high number of queries and regardless of the overall size of the data stream and regardless of the number of valid objects. Our data structure is visualized in Figure 4. Each of the d dimensions is independently partitioned into a number of ρ quantiles according to a sample of the set of queries. If initially no sample of the queries is known, then the quantiles can be uniformly distributed in the data space. These quantiles are depicted in Figure 4 in dashed lines and marked with $g_{i,j}$ where $i \in 0, ..., d-1$ is the dimension of the grid line and $j \in 0, ..., \rho$ is a sequential number which is strictly monotonic (i.e. $g_{i,j_1} < g_{i,j_2} \Leftrightarrow j_1 < j_2$). The set of all quantiles constitutes a d-dimensional $\rho \times ... \times \rho$ grid. The rows and columns are marked with $G_{i,j}$ where i

is again the dimension and $j \in 0, ..., \rho - 1$ is the sequence number. Initially the lower limit of $G_{i,j}$ is $g_{i,j}$ and the upper limit is $g_{i,j+1}$. The cells of the grid are marked with $C_{a_0,...a_{d-1}}$ where a_i corresponds to some grid row or column G_{i,a_i}. The queries are associated to the grid cells according to the coordinates $q_i, i = 0, ..., d-1$ of the center, i.e. $q \in C_{a_0,...,a_{d-1}} \Leftrightarrow q_i \in [g_{i,a_i}, g_{i,a_i+1}]$. Note that, until now, this structure resembles the well-known VA-file [15].

We now extend this structure to index spheres rather than points. Additionally to the original quantile lines $g_{i,j}$ we determine for each row and column $G_{i,j}$ the lower and upper boundaries $l_{i,j}, u_{i,j}$ of all query spheres (the spheres centered by q having the current cut radius $r(q)$) which are stored in any of the cells:

$$l_{i,j} = \min_{q \in Q | q_i \in [g_{i,j}, g_{i,j+1}]} \{q_i - r_C(q)\}$$

$$u_{i,j} = \max_{q \in Q | q_i \in [g_{i,j}, g_{i,j+1}]} \{q_i + r_C(q)\}$$

Here q_i denotes attribute number i in the query vector q. As depicted in Figure 4, these limits usually extend each line and column a little bit to the left and to the right. Therefore, these modified grid cells are now allowed to overlap. But if the radii of the stored spheres are small, it is also possible that some grid cells do not grow but shrink instead, and the neighboring grid cells become disjoint. During query processing, the queries may frequently change their associated cut radius $r_C(q)$: As a consequence of the arrival of a new object, the cut radius of a query may decrease. Likewise, the cut radius may increase if the current nearest neighbor of a query expires. We handle the increase of a radius by immediately checking the lower and upper boundaries of the cell to which q is associated, and updating them if necessary. From time to time, the boundaries are also checked if they are able to be contracted. By this simple method it is guaranteed that each grid cell is always a conservative approximation of the contained query sphere. Thus, it is possible to safely decide according to the cell boundaries that a cell cannot contain any queries to which a given object could be the nearest neighbor. Figure 5 summarizes the required steps for processing a new object in pseudocode.

```
algorithm processNewObject (Object o)
if buffer.isFull()
  Object o' :=buffer.getAndRemoveMinimumExpiry();
  process(o', false);
buffer.insert(o);
process(o, true);

algorithm process (Object o, boolean isApproximate)
for all g ∈ gridCells
  if o ∈ g.occupiedSpace
    for all q ∈ g.associatedQueries
      maintenance(o, q, isApproximate);

algorithm maintenance (Object o, Query q, boolean isApproximate)
// monotonic expiry only due to space limitations
Skyline e := q.associatedExactSkyline;
Skyline a := q.associatedApproximateSkyline;
SkylineElement c, c';
if isApproximate
  c := skylinePruning (o, q, a.last); // cf. Figure 1
if not isApproximate or a.isEmpty()
  c := skylinePruning (o, q, e.last);
  if e.isEmpty()
    reportNeighbor(o, user(q)));
if isApproximate
  a.append(o);
else
  e.append(o);
```

Figure 5. Processing of a New Object o.

6. Result Reporting

Our system has to report every change of the nearest neighbor of any subscribed query $q \in Q$ immediately to the user who owns the subscription. There are two possible causes of a nearest neighbor change, (1) the arrival of a new object from the data stream which takes over immediately the top position of the query skyline, and (2) the expiry of the current nearest neighbor of q. Case (1) is immediately detected during processing of the new object. Case (2) is more complicated because, in principle, the query skyline of every query must be continuously monitored for the expiry of its first element.

To do this in a very efficient way, we store pointers to the queries in a priority queue (heap) which we call the *action-queue*. We need only one global action queue in the system with a size of $|Q|$, the number of subscribed queries. The elements of the action queue are ordered by the time stamp of the expiry of the top element of the query skyline. This allows an efficient access (in constant time) to the element which will expire next in any query. This access can be done periodically upon every arrival of a new object. As an alternative, the next relevant expiry (the top element of the action queue) can be triggered by a timer.

Whenever an element is dequeued from the action queue, and whenever a newly arrived object takes over the top position of a query q, the action queue must be updated which requires $O(\log_2 |Q|)$ time where $|Q|$ is the number of subscribed queries (note that we have to register every query exactly once in the action queue with the expiry of the top element of the query skyline). This reorganization upon the arrival of new objects can even be avoided in the case of monotonic object expiry: In this case, we know that the expiry t_{new} of the new top element of q is later than of the old one (t_{old}). We can leave the element in the action queue as it is, wait for t_{old} and then state that the top element of the action queue has already been dominated. Therefore the new expiry t_{new} can then be registered without notifying the user about any change.

7. k-Nearest Neighbor Queries

In this section, we shortly describe the generalization of our technique for k-nearest neighbor queries with $k > 1$. The three ideas of our technique, query indexing, skyline based object maintenance and delay buffering can also be applied in the general case. Particularly the concept of query-skylines needs some modifications. We start again with a simple variant without distinction between exact and approximate query skyline:

An object can now be safely discarded from memory if it is guaranteed that it will become at no future point-of-time one of the k nearest neighbors of any query $q \in Q$. Translating this into the language of skylines, an object can be discarded if it is dominated by at least k other objects, where the dominance is exactly the same as in Definition 5. We can then define a k-skyline as the set of objects which is dominated by less than k objects, which corresponds to the concept of the k-skyband [10]. In addition we store for each object a counter with the information by how many other objects it is dominated.

When we use a delay buffer, we have again to distinguish between approximate and exact skylines. Our notion of dominance can be naturally extended to this case (objects of the approximate skyline can also dominate objects from the exact skyline, and, thus, increase their counter). To associate a query with a cut radius we need to guarantee that no object exceeding the cut radius can become the nearest neighbor during the delay time or during its remaining life time. This requires that we have to store the objects of the approximate k-skyline in a way which allows efficient access to the element with distance-rank k. A linked list can be used for the management of the elements with rank 1 to k. A priority queue (heap) can be used for the remaining elements with rank $> k$.

8. Experiments

Data Sets and Methodology. For a systematic analysis of the properties of our method we generated synthetic data sets of various dimensionality. In particular, we used uniformly distributed and clustered gaussian data. The clustered gaussian data sets contain 10 clusters (standard deviation 0.1) which are randomly distributed on the data space. We also show the performance on two netflow data sets. We used a 2-d data set containing records extracted from netflow IP data logs, cf. [7]. We additionally used a 3-d netflow data set containing records (source port, destination port, packet size) extracted from the LBL-TCP 3 data set, available at the Internet Traffic Archive (http://ita.ee.lbl.gov/). The IDs of the source and destination hosts have been left out here. For all experiments we assume number based expiry. In this context for an object o a life span of e.g. $l(o) = 20,000$ means that o expires after 20,000 new objects from the stream have arrived. We used $N = 1,000,000$ 2-d streaming objects with a lifespan of $l(o) = 20,000$, $|Q| = 500$ queries, a buffer size of $|B| = 2000$, and a resolution of $\rho = 20$ when not otherwise specified. We further assume that at each time stamp a new object from the stream arrives. We randomly separated objects from the stream and used them as continuous queries. Experiments were run on a PC with a 2.4 GH pentium processor and 512 MB main memory under Java.

Query Indexing. Figure 6(a) shows the number of distance calculations with varying resolution of the query index on uniform and clustered gaussian data. For uniformly distributed data, $\rho = 20$ grid lines per dimension show the best trade-off between number of cells and queries per cell. For clustered gaussian data a higher resolution of 35 is better, since queries are densely populated in some regions of the data space. In Figure 6(b) the scalability w.r.t. the number of subscribed queries $|Q|$ for our algorithm and a variant without index is depicted. Upon arrival of a new object o, this variant simply checks all queries if o needs to be inserted. In this experiment we used uniformly distributed data. Hence our index structure is simple and efficient in maintenance, query indexing pays off already for $|Q| = 100$. For $|Q| = 1600$ the index leads to a speed up factor of 80. Figures 6(c) and 6(d) show the performance in the case of new dynamic queries arriving during the runtime of the system on netflow data. In Figure 6(c) in addition to 400 static queries 100 dynamic queries have been inserted. Despite of the inserted queries, the runtime scales linearly with N. Figure 6(d) shows the performance w.r.t. the number of dynamic queries. New queries can be very efficiently inserted in our index. This leads to a sublinear increase in runtime. During the runtime the grid cells are expanded as explained in Section 5. To maintain a high performance of the grid index it is beneficial to adjust the upper and lower bounds in fixed time intervals. For uniform and clustered gaussian data we achieve good performance if we do this upon arrival of every 1,000 new objects as Figure 6(e) shows.

Buffer Size. The size of the buffer can be chosen depending on available memory. As shown in Section 4 our method works correctly with an arbitrarily small buffer size. According to Lemma 3 an upper limit for the buffer size $|B|$ is then half of the live span of the streaming objects. Figure 6(f) shows the overall memory usage for varying buffer sizes $|B| = 20...4,000$ for two dimensional uniformly distributed data. Here we increased the live span of the objects to $l(o) = 500,000$. The memory usage for the buffer, for the query skylines and the overall memory

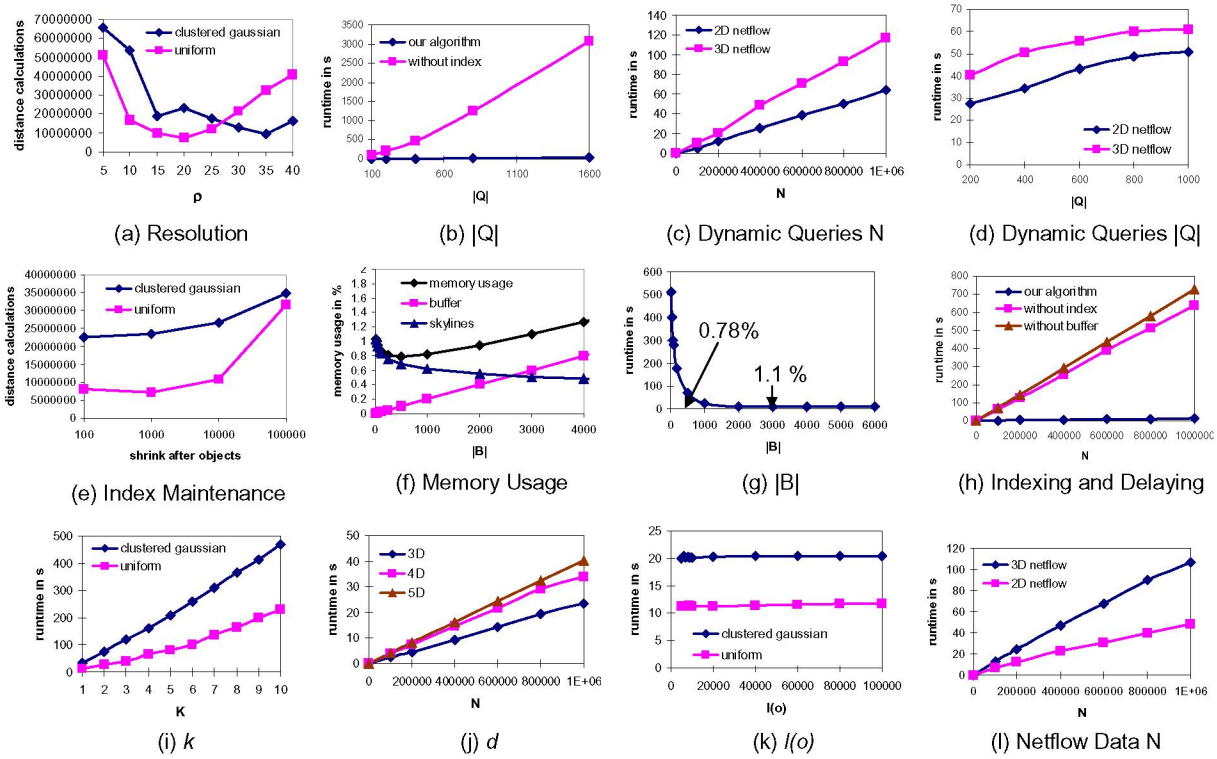

Figure 6. Experimental Evaluation.

usage is depicted. Even for this relatively long life span of the objects, memory usage is very low. For all examined buffer sizes we needed to store in total less than 2% of the valid objects. As a minimum 0.78% of the valid objects need to be stored for a buffer size of $|B| = 500$ objects. Figure 6(g) gives the runtimes in seconds for the previous experiment. Although the configuration minimizing memory usage allows a throughput of 14,085 objects per second, a much higher throughput of 93,110 objects per second can be achieved with a buffer size of $|B| = 3000$, which means storing 1.1% of the valid objects. Larger buffer sizes do not considerably further improve runtime.

Scalability. Figure 6(h) demonstrates the benefits of query indexing and object delaying. The runtime in seconds on uniformly distributed 2-d data is reported for our technique and variants without index and without buffer for various numbers of objects N. The version without buffer performs worst. This demonstrates the effectiveness of object delaying. For $N = 1,000,000$ this version attains a throughput of 1,372 objects per second. The variant without index is slightly better and attains a throughput of 1,570 objects per second. The combination of query indexing and object delaying leads to impressive performance gains over both aspects when used alone. One million stream objects are processed in 11.58 seconds, this corresponds to a throughput of 86,356 objects per second. The throughput is 55 times higher than for the variant without index and even 63 times higher than for the variant without buffer. Figure 6(i) shows the scalability our method for k-nearest neighbor processing as described in Section 7 on 2-d uniform and clustered gaussian data. Starting with $|B| = 2000$, we increased the buffer size linearly with k. It can be assumed that for a larger k also more memory is available, at least for storing the query results. With this configuration, for $k = 10$ we store 5% of the valid objects in the buffer. For both data sets, the runtime scales linear with k. Figure 6(j) demonstrates the scalability with the dimensionality on clustered gaussian data of dimensionality 3, 4, and 5. For this experiment, the resolution was set to $\rho = 10$. All curves scale linearly with the number of objects N. There is only a small increase in runtime from 4 to 5 dimensional data. Figure 6(k) displays runtime for various life spans $l(o) = 5,000..1,000,000$ on uniform and clustered gaussian data. The runtime remains almost constant with increasing $l(o)$, even if all

objects are valid at all time stamps. This demonstrates that our method is applicable high throughput data streams ranging from relatively short object live spans to scenarios without expiry. Figure 6(l) shows the runtime in seconds for varying N on netflow data. Here we set $\rho = 15$ for both, 2-d and 3-d data sets. For both data sets, the runtime scales linearly with the data size.

9. Conclusions

In this paper, we have proposed an efficient technique for processing a large number of continuous k-nearest neighbor queries on data streams of feature vectors which are important e.g. in multimedia and marketing applications. In contrast to previous methods for query processing on streams, the result of our method is not approximative but exact and complete. Our method is based on three major ideas:

1. The **query skyline** enables us to carefully select those objects from the stream which have the potential of becoming answers to any of the subscribed queries.
2. A **delay buffer** is used to retard processing of those objects which are not immediate answers to the queries and thus greatly improves the efficiency of processing.
3. A **query index** is used to organize the subscribed queries in a simple, grid based way.

Our extensive experimental evaluation demonstrates that our technique scales well with the size of the stream, the number of queries, the number k of answers per subscribed queries, the dimensionality, and the available buffer size for artificial and real-world stream data. In particular, we demonstrate that the combination of our 3 major ideas leads to a very low memory consumption: Only between 0.78% and 1.1% of the valid stream objects must be retained in memory to guarantee completeness and correctness of the result of our technique. In addition, we demonstrate that the combination of our 3 major ideas improves the throughput by a large factor of about 60.

Acknowledgement. Most of this work has been done during a research visit of Christian Böhm, Claudia Plant and Ying Yan to Beng Chin Ooi at National University of Singapore.

References

[1] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The x-tree : An index structure for high-dimensional data. In *VLDB*, pages 28–39, 1996.

[2] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[3] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, pages 40–51, 2003.

[4] L. Gao and X. S. Wang. Continually evaluating similarity-based pattern queries on a streaming time series. In *SIGMOD*, pages 370–381, 2002.

[5] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[6] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD*, pages 201–212. ACM Press, 2000.

[7] N. Koudas, B. C. Ooi, K.-L. Tan, and R. Z. 0003. Approximate nn queries on streams with guaranteed error/performance bounds. In *VLDB*, pages 804–815, 2004.

[8] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *ICDE*, pages 502–513, 2005.

[9] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD*, pages 635–646, 2006.

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

[11] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of influence sets in frequently updated databases. In *VLDB*, pages 99–108, 2001.

[12] H. Su, E. A. Rundensteiner, and M. Mani. Semantic query optimization for xquery over xml streams. In *VLDB*, pages 277–288, 2005.

[13] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310.

[14] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB*, pages 744–755, 2004.

[15] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

[16] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB*, pages 421–430, 2001.

[17] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005.