Approximate NN Queries on Streams with Guaranteed Error/performance Bounds

Nick Koudas
AT&T Labs-Research
koudas@research.att.com

Beng Chin Ooi   Kian-Lee Tan   Rui Zhang
National University of Singapore, Singapore
{ooibc, tankl, zhangru1}@comp.nus.edu.sg

Abstract

In data stream applications, data arrive continuously and can only be scanned once as the query processor has very limited memory (relative to the size of the stream) to work with. Hence, queries on data streams do not have access to the entire data set and query answers are typically approximate. While there have been many studies on the k Nearest Neighbors (kNN) problem in conventional multi-dimensional databases, the solutions cannot be directly applied to data streams for the above reasons. In this paper, we investigate the kNN problem over data streams. We first introduce the e-approximate kNN (ekNN) problem that finds the approximate kNN answers of a query point Q such that the absolute error of the k-th nearest neighbor distance is bounded by e. To support ekNN queries over streams, we propose a technique called DISC (aDaptive Indexing on Streams by space-filling Curves). DISC can adapt to different data distributions to either (a) optimize memory utilization to answer ekNN queries under certain accuracy requirements or (b) achieve the best accuracy under a given memory constraint. At the same time, DISC provide efficient updates and query processing which are important requirements in data stream applications. Extensive experiments were conducted using both synthetic and real data sets and the results confirm the effectiveness and efficiency of DISC.

1 Introduction

In many applications, including geographic information systems, content-based retrieval and data mining, finding the k Nearest Neighbors (kNN) to a query object is one of the most frequent operations. The database research community has in recent years provided several novel solutions to efficient kNN processing [22, 6, 21]. The kNN problem can be defined as follows: Given a set of points $S = \{P_0, P_1, ..., P_n\}$ in a d-dimensional space V, and a query point $Q \in V$, find a set kNN which contains k points in S such that, for any $P \in kNN$ and for any $P' \in S - kNN$, $dist(Q, P) \leq dist(Q, P')$.

To further improve performance, the $(1 + \epsilon)$-approximate nearest neighbors problem [1, 17] has been introduced which is defined as follows: Find a point $P \in V$ that is an $(1 + \epsilon)$-approximate nearest neighbor of the query point Q, so that for any point $P' \in S$, $dist(P, Q) \leq (1 + \epsilon)dist(P', Q)$. The k $(1 + \epsilon)$-approximate nearest neighbors problem can be similarly defined [2]. Here ϵ is in fact a bound for the relative error of the k-th nearest neighbor distance, which is specified by the users before the query.

KNN queries over multi-dimensional data streams is a pressing concern when mining streams for unknown patterns. For example, in computer aided manufacturing (CAM) systems, sensors are used to monitor the position, shape[1], size, surface characterization, material properties, etc, of parts passing through on a production line. The data are collected and sent to a control system. The control system analyzes the feedback information and then adjusts the parameters of the production line so as to control the quality of the parts. Often, we tend to identify parts with similar shape to a given part in order to discover patterns of other features. In highway traffic monitoring, sensors are embedded on highways to observe the passing vehicles. Estimates of vehicle speed and length can be obtained and utilized to provide useful traffic related information. Similarly in network traffic monitoring, network

---

[1]Even parts on a same production line have slightly different shapes and sizes due to manufacturing errors.

traffic streams (IP traffic) are usually logged using special programs, such as CISCO's netflow. The network management system will monitor the network packet header information to obtain information on traffic flow patterns [3], which involves finding packets similar to a given packet.

In addition, data stream applications typically operate in an environment where memory is limited (relative to the size of the stream) so that it is not feasible to work with the entire data set in memory. For this reason, one has to resort to approximate kNN answers in the case of continuously evolving data streams. All previous proposals for approximate kNN queries require the user to specify a relative error bound (ϵ) beforehand. However, in certain applications, absolute error bounds are more critical and preferable. In the CAM example, a query typically specifies absolute errors: "Identify 10 parts that are most similar in size to a given part A. The query specifies that as long as a part's resultant error (that is, the root-sum-square of the errors in width and length) to those of the 10 most similar parts is not more than 0.1mm the answer is acceptable." In the highway traffic monitoring example, it may also be more intuitive to specify errors by absolute bounds: "Find the 20 vehicles that are close to position A. An answer is acceptable as long as its distance to A is not larger than say 10 meters than that of the 20 closest vehicles." Similar examples can be drawn from the field of network monitoring and other engineering applications, in which users have good knowledge of the absolute errors acceptable.

Motivated by such applications, we introduce a new type of approximate nearest neighbors problem, called the *e*-**approximate kNN (ekNN) problem**, in which the answers are bounded by absolute value instead of relative one. Formally, we define it as following:

Definition 1 (ekNN) *Given a data set S and a query point Q, find a set ekNN which contains k points in S such that for any $P \in ekNN$ and for any $P' \in kNN$ (the actual kNN set of Q), $dist(Q, P) \leq dist(Q, P') + e$, where e is a bound for the absolute error of the k-th nearest neighbor distance.*

Subsequently, we define the *e*-**approximate kNN problem over Data Streams** as follows:

Definition 2 (ekNN over data streams) *Let X be a sequence of points $(P_0, P_1, P_2, ...)$ (in this paper, we view data records with multiple attributes as multi-dimensional points). X can be either finite or infinite. Each element $P_i(i = 0, 1, 2, ...)$ of X is a point in d-dimensional space and is allowed to be read for at most once in the order of the sequence. Let S_t be the set of points of X that have been read at time t. At any time t and for any query point Q, find the ekNN of Q from the elements of S_t.*

In particular, we identify and provide solutions to the following ekNN problems on data streams:

1. **memory optimization for a given error bound**: given an error bound *e*, use as little memory as possible to answer ekNN queries.

2. **error minimization for a given memory size**: given a fixed amount of memory, achieve the best accuracy for ekNN queries.

We propose a general scheme which aims to reduce the amount of information to be stored while guaranteeing a provable error bound. Specifically, we partition the underlying space into equal square-shaped cells, and then we prove that in each cell we only need to store at most K (for a user specified value K) points to guarantee some error bound. We will prove that the error bound is guaranteed for any ekNN query where $k \leq K$. Next, to facilitate efficient maintenance of K points in each cell, we propose a technique called DISC (aDaptive Indexing on Streams by space-filling Curves), in which points are stored in the leaf nodes of the B*-tree with the Z-values [19] of their cells as keys. DISC has two important properties: first, it only allocates memory for those points that are necessary to guarantee the error bound; second, by merging cells, DISC can adjust the structure to meet the memory constraint. These two properties make it adaptive to different data distributions. In addition, being a B*-tree based indexing structure, DISC provides fast access to a given cell. This facilitates efficient updates and query processing. Overall, DISC can achieve our goals of minimizing memory usage for a given error bound or obtaining best accuracy for a given memory constraint while retaining efficient updates and query processing. We present the ekNN search algorithm based on DISC and also show how to modify DISC to support sliding window ekNN queries. Extensive performance studies using synthetic and real data sets were conducted, and the results demonstrate that DISC is both query and memory efficient. Note that since DISC is essentially a B*-tree based technique, it can also be used as a disk-based structure.

The rest of the paper is organized as follows: Section 2 reviews related work. In Section 3, we propose a general scheme to reduce information while still answering the ekNN problem with some error bound. A brute-force method based on this framework is also presented in this section. Then we present DISC and the algorithms in Section 4. Section 5 reports the results of our experimental studies. Section 6 concludes the paper.

2 Related Work

Various multi-dimensional indexing structures [5, 14, 7, 23] and kNN query processing strategies have been proposed in the literature [12, 21, 22]. These methods assume that the data are disk-resident and can be scanned multiple times. As such, they are not suitable for processing data streams that typically require one-pass algorithms as the data are not stored on disk

and are too large to fit into memory. Moreover, it is unclear how these schemes can provide any guarantee on approximate answers to kNN queries.

A structure based on quadtrees for answering kNN queries approximately was proposed in [8]. The relative error is dependent on the dimensionality d so that the larger the value of d, the greater the relative error will be. Then the $(1 + \epsilon)$-approximate nearest neighbors problem was studied [1, 2, 17], in which the relative error ϵ is a constant specified by the user. An algorithm requiring exponential time in d and linear space was proposed in [1] and follow-up studies improved its time/space requirements [13, 17, 18]. These studies share the common feature of a relative error bound. The ND P-sphere tree [11] also accelerates kNN search by providing non-exact answers. The algorithm guarantees that for a user specified percentage of time, the returned answers are correct, but it cannot distinguish between the correct and incorrect answers. To our knowledge, there have been no studies on approximate kNN search specifying absolute error bounds. In addition, none of the above studies address the approximate kNN problem in the data stream model, where data can only be scanned once.

The management and processing of data streams has attracted lots of research interest recently. A survey can be found in [3]. In [10] the authors use the Fast Fourier Transform to solve the problem of pattern similarity search. The paper also studies the nearest neighbors problem over streams, but uses values from the incoming stream (time series) as queries to identify the nearest neighbors from an existing pattern database. In our setting, queries are specified by users on demand and we seek to locate nearest neighbors in the streaming data. [10] uses prediction to take advantage of batch processing. When the actual time series arrives, prediction error lower bounds and upper bounds are calculated and used together with the predicted distances to filter candidate patterns. In [9], hamming norms are used to measure the similarity between two streams, and in [20], a regression-based algorithm is proposed to mine frequent temporal patterns for data streams. Reverse nearest neighbor aggregate queries over streams have also been investigated in [16].

3   Analysis of the problem

In this section, we propose a scheme towards solving the ekNN problem with a guaranteed error bound. As we shall see, this scheme provides possibility to reduce the information to be stored, however, the scheme in itself does not guarantee achieving the goal of memory optimization or error minimization. The data structure used to implement it is also critical to achieve these two optimizations. Therefore we will first present the scheme, followed by analysis on adopting the most suitable structure to realize it.

Our overall approach consists of segmenting the underlying space into a number of cells and identifying dynamically a number of points to be stored in each cell (called the *footprints* of the data) as data stream by. We observe that, in order to guarantee the error bound e, which is the largest distance between two points in a cell, for kNN queries, we only need to maintain at most k points in each cell. In the case of data streams, the number of data is very large so that usually exceeds k in many cells. Therefore, by maintaining only k points, we can reduce the data to be stored. In the following, our scheme based on this observation is formally presented.

3.1   Capturing the Footprints

We consider the problem in a d-dimensional metric space V, which is a set of points with an associated distance function $dist$. The distance function $dist$ has the following properties:
1. $dist(P_1,P_2)=dist(P_2,P_1)$
2. $dist(P_1,P_2)>0$ $(P_1{\neq}P_2)$ and $dist(P_1,P_2)=0$ $(P_1{=}P_2)$
3. $dist(P_1,P_2)\leq dist(P_1,P_3)+dist(P_2,P_3)$

We divide the data space into a number of square-shaped cells and maintain at most K (K is a user specified constant) points in each cell. Specifically, as data stream by, each data point is placed in the cell it belongs to. If a cell already contains K points, there would be $K + 1$ points including the new one. Then, we discard a point according to some discarding policy. The discarding policy is clearly application dependent. For example, if the most recent information is of interest we will always delete the oldest point. When processing ekNN queries, we invoke an exact kNN query on the set of points maintained, that is, the *footprints* of the stream data. Contrasting the kNN answers obtained from the footprints of the data set and on the original data set, we prove that the difference of their k-th nearest neighbor distance is within e, which equals the largest distance between two points in a cell. So the kNN on the footprints is an approximate answer for the kNN query on the original data set with error bound e. We start by defining some functions necessary for the derivations that follow and formalize the scheme for capturing the footprints. Some commonly used symbols in this paper are summarized in Table 1.

We assume that the data space is normalized to a unit hypercube. Each of the d dimensions of X is divided equally into u segments (therefore X is divided into u^d cells). Let S be a set of points in X and c a cell in X. Define $S(c)$ as $\{P \in S | P \in c\}$, that is, the subset of S that is in the cell c.

Let T be a mapping on S which is defined as follows: for each cell c of X, if $|S(c)| > K$, image of $S(c)$ is the set of any K points in $S(c)$; if $|S(c)| \leq K$, image of $S(c)$ is $S(c)$.

Let S' be the image set of S under mapping T. For any query point $Q \in X$, kNN is the set of k nearest neighbors of Q in S and kNN' is the set of k nearest

Table 1: Symbols

Symbol	Meaning
c	A cell
d	Dimensionality
$dist(P_1, P_2)$	Function that returns the distance between the two points P_1 and P_2
e	The error bound of the k-th nearest neighbor distance
$far(S, P)$	Function that returns the farthest point in set S to point P
kNN	The set of the k nearest neighbors
$ekNN$	The set of the e-approximate k nearest neighbors
m	The order of the Z-curve
P	A data record, which is viewed as a multi-dimensional point
p_i	The i-th coordinate of point P
Q	A query point
S	A set of points
t	Current time
T	Some period of time
u	The number of segments a dimension is divided to
V	A metric data space
W	A query window
W_s	The smallest query window that contains $ekNN$

neighbors of Q in S'. Let $far(S, Q)$ be the function returning the point in S, which is of largest distance to Q among all the points of S.
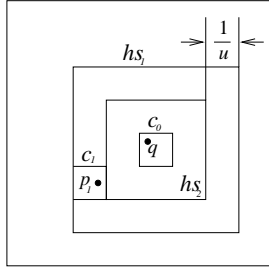

Figure 1: Diagram to explain Theorem 1

Theorem 1
For any positive integer $k \le K$,
$dist(far(kNN', Q), Q) \le dist(far(kNN, Q), Q) + d_M$,
where d_M is the maximum distance of two points within a cell.

Proof Suppose query point Q is in cell c_0 as Figure 1 shows. We have two cases to consider:
Case 1. $|S'(c_0)| \ge k$.
In this case, $far(kNN', Q)$ is within cell c_0. $|S(c_0)| \ge |S'(c_0)|$, so $|S(c_0)| \ge k$ and $far(kNN, Q)$ is also within cell c_0. The largest distance between two points within a cell is d_M. So

$$dist(far(kNN', Q), Q) \le d_M$$

and therefore,

$$dist(far(kNN', Q), Q) \le dist(far(kNN, Q), Q) + d_M.$$

Case 2. $|S'(c_0)| < k$.
Suppose hs_1 is the smallest hyper-square centered at c_0 and contains at least k points of S', hs_2 is the largest hyper-square centered at c_0 and contains at most $k - 1$ points of S'. Then side length of hs_1 minus side length of hs_2 must be $2/u$. Otherwise, either hs_1 can be smaller or hs_2 can be larger. Suppose there are k_1 points of S' in hs_2, there must be also k_1 points of S in hs_2 according to the definition of S'. Denote these k_1 points as $S(hs_2)$. Then $kNN - S(hs_2)$ and $kNN' - S(hs_2)$ are both in $hs_1 - hs_2$.

If all points in $kNN - S(hs_2)$ are in S', then $kNN' - S(hs_2) = kNN - S(hs_2)$, so $far(kNN', Q) = far(kNN, Q)$ and

$$dist(far(kNN', Q), Q) \le dist(far(kNN, Q), Q) + d_M$$

holds.

If any point in $kNN - S(hs_2)$ is not in S', say $P_1 \in kNN - S(hs_2)$ and $P_1 \notin S'$. Suppose $P_1 \in c_1$. $P_1 \notin S'$ means $|S(c_1)| > K$, and then S' must have K points in c_1. Let $P_2 = far(S'(c_1), Q)$. Then

$$dist(far(kNN', Q), Q) < dist(P_2, Q) \qquad (1)$$

According to the triangle inequality

$$dist(P_2, Q) < dist(P_1, Q) + dist(P_1, P_2) \qquad (2)$$

P_2 and P_1 are in the same cell, so

$$dist(P_1, P_2) < d_M \qquad (3)$$

$P_1 \in kNN - S(hs_2)$, so $P_1 \in kNN$, therefore

$$dist(P_1, Q) < dist(far(kNN, Q), Q) \qquad (4)$$

And from inequalities 1, 2, 3 and 4, we hence get
$dist(far(kNN', Q), Q) \le dist(far(kNN, Q), Q) + d_M$
$\square$

According to the theorem, if we divide the data space into u^d equal cells and use the above scheme to process the ekNN problem, $e = d_M$. In addition, if the maximum number of points maintained in a cell is K, for any ekNN query where $k \le K$, the above error bound is guaranteed. For example, if we maintain at most 5 points in a cell, then we can also search for 2NN with an error bounded by $e = d_M$. Note that d_M is determined by the distance function. Without loss of generality, we use the Euclidean distance function in the following discussions and our experimental studies. For the Euclidean metric, $d_M = \sqrt{d}/u$, and therefore the error bound is $e = \sqrt{d}/u$.

3.2 An Array-Based Method

A first method to implement this general scheme would be to organize the data in memory as a big d-dimensional array. Each element of the array represents a cell in the space. We may store at most K

points in each cell, so each array element is a structure consisting of K d-dimensional points. Stream data elements are placed in cells on demand as data stream by. If there are already K points, we discard one of them based on the discarding policy. Processing of ekNN queries using the array is straightforward. We just need to calculate the borders of the square which encloses the ekNN query sphere and check all the elements within the borders. In what follows, we refer to this method as the *array-based method.*

For the array-based method, we can calculate the memory size needed by the following equation:

$$Mem_{array} = u^d \cdot K \cdot d \cdot sizeof(attribute) \quad (5)$$

The array-based method is straightforward, and its processing is simple and fast in terms of memory accesses (reads/writes) and processor time, but the memory required is exponential to u. This static memory allocation strategy can cause excessive memory usage, especially for small error bounds, which implies a large value of u. Real data are often skewed and may be sparse; most cells contain much fewer than K points or even none at all, resulting in poor utilization of the statically allocated memory space. It is obvious that a structure capable of adapting to different data distributions is more desirable.

4   The DISC Method

To better utilize memory, cells that do not contain data points should not be explicitly maintained as opposed to the array-based method. Even within one cell, the number of points may be different, so space usage is different. This calls for a smart strategy to allocate space to each cell.

Besides the central objective of minimizing memory usage, the method should also provide fast updates and query processing. For the error minimization problem, the method may need some self-adjusting mechanism to achieve smallest error.

As discussed in the previous section, the array-based method needs too much memory despite its fast updates and query processing. Or we can organize the cells by a linked list and dynamically allocate only necessary space for each cell. The memory size problem is solved largely (we still have some extra cost due to the links), but the number of node accesses for update and query processing is linear to the number of points. Averagely, half the size of the linked list is accessed to locate a point. This is prohibiting for data stream applications.

A third way is to use a dynamic indexing structure such as an R-tree or a B-tree. On one hand, it dynamically allocates space in the unit of a leaf node so as to avoid excessive memory overheads as in the array-based method. On the other hand, the index provides fast access to the entries in the nodes. It is not as fast as the array-based method, but typically several node accesses are enough, which is much more efficient than linked lists in terms of updates and query processing. A dynamic index is in fact a compromise of the above two, and therefore it avoids the deficiency of either one.

A straightforward structure for multi-dimensional data is the R-tree or some of its variants. A point is stored as a leaf node entry. Since we need to differentiate between points from different cells, an identifier, id, is stored along with each point.

An alternative approach, which we adopt in this paper, is to employ a B*-tree[2] [15] together with a space-filling curve mechanism. Space-filling curves have been used to linearize multi-dimensional data spaces. Various types of space-filling curves exist in the literature; without loss of generality we adopt the Z-curve [19]. Efficient algorithms to compute Z-values can be found in [19]. Each cell corresponds to a Z-value. Footprints of the data stream are stored in the leaf nodes of a B*-tree using their corresponding cell Z-values as keys. Such an approach is expected to be more efficient than the R-tree scheme for the following reasons. Although a point is the unit of storage, a cell is the unit most of our operations deal with as we will see later in the algorithms. To locate a cell by the Z-value in a B*-tree, for each level of the tree, we only need to compare the search key with one value, since there is no overlap in the Z-values. In an R-tree, we need to compare the coordinates of the cell with $2d$ values (lower bound and upper bound for each dimension) for each level of the tree and there is overlap between the MBRs of the R-tree, which translates to more node accesses to update and search the R-tree. In addition, since the R-tree stores more information as keys, the fan-out of the R-tree nodes becomes lower and the height larger.

Another advantage of organizing the footprints in the Z-order is that cells can be arranged in a total order while maintaining cell proximity. The R-tree also keeps the points belonging to the same cell spatially close, but it still happens that they scatter in nearby MBRs. In DISC, points in the same cell are always consecutively stored in the leaf nodes. This property facilitates accesses on the cell level and make possible a very fast *merge-cells* operation, which is required for the error minimization problem and described in Section 4.2. We will also compare DISC to the R-tree in our experimental study. Since several points may belong to the same cell and have the same key in DISC, our B*-tree is designed to accommodate entries with equal keys. For the R-tree method, we have used the R*-tree [5] variant, which has a higher node utilization (about 73%). Moreover, we have also used the Z-values as the id's of cells for the R*-tree method.

Since we are utilizing space-filling curves, each dimension of the data space is partitioned into a number of intervals equal to an integral power of 2, the same for

---

[2]We employ the B*-tree for indexing (instead of B$^+$-tree) as its node utilization is about 85% or higher.

all dimensions. Let m denote the order of the Z-curve, then $u = 2^m$.

4.1 Index Creation

We begin by considering the first problem, namely the memory optimization problem for a given error bound e. To guarantee that this error bound is met by our query answers, we calculate the order of the Z-curve m_e according to Theorem 1 as follows.

$$\sqrt{d}/2^{m_e} \le e$$

Then

$$m_e \ge \log_2(\sqrt{d}/e) \qquad (6)$$

The larger the value of m_e, the more memory is required; we let m_e be the smallest integer that can satisfy inequality 6.

$$m_e = \lceil \log_2(\sqrt{d}/e) \rceil \qquad (7)$$

Algorithm **Build Index**, shown in Figure 3, describes how the index is constructed. In the algorithm, we initialize the value of m to m_e.

Before we discuss the algorithm, let us consider the second optimization problem, namely error minimization given a specific memory size constraint. The basic idea of the algorithm is to adjust the order of the Z-curve, m, to achieve the best accuracy while satisfying the constraint. Our aim is to minimize the error bound e in the ekNN search. Since the larger the value of m, the smaller the error bound e, and the data distribution is not known apriori, we start with a sufficiently large value for m; the exact value depending on the arithmetic precision we are working with. A value of 16 should suffice for most applications. As data arrive, it may turn out that m is too large and hence memory is exhausted; in this case, we merge small cells into a larger one, discard some points and still maintain at most K points in the larger cell. As a result some memory is freed, and processing of the stream continues. The Z-curve properties enable us to merge cells efficiently. In particular, a Z-value for a cell can be mapped efficiently (using simple bitwise operations) to Z-values corresponding to a curve of different order. For brevity, we omit the details which can be found in [19]. Related properties hold for other curves as well. Each time we need to perform cell merging, we will combine 2^d adjacent small cells into a larger cell as shown in Figure 2, in which cells c_0, c_1, c_2, c_3 are combined to form cell c_0'. The larger cell is still square-shaped. After merging the cells, the order of the Z-curve becomes $m - 1$. The index construction algorithm for this case is similar to that for the memory minimization problem; the difference lies in the merging phase. For brevity, we include this phase in the description of algorithm **Build Index**.

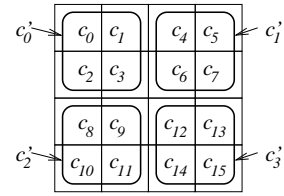We are now ready to look at algorithm **Build Index** (see Figure 3). In line 1, we let $m = m_e$ for the

Figure 2: Cell Merging

Algorithm **Build Index**
1. Initialize m
2. Read data from the stream, denote the point read in as P, calculate the Z-value of P, and we know which cell it belongs to, denote it as c
3. Search the B*-tree and obtain the number of points that also belong to cell c, denote the number as N_c
4. If $N_c < K$
   Insert P to the B*-tree
5. Else
   Among P and the K points in c, discard 1 and keep the other K points in the B*-tree
6. If memory runs out /*This only happens for the error minimization problem*/
   Merge cells and let $m = m - 1$.
   /* The merge cells algorithm is presented in the next subsection. */
7. Go to 2
End Build Index

Figure 3: Algorithm **Build Index**

memory optimization problem and let m be a large enough integer for the error minimization problem. In line 5, we should determine which point to discard according to the discarding policy. In our realization of the algorithm, we simply discard the new point P.

In the analysis of Section 3.1 we have assumed the data space is normalized to a unit hypercube. This may have difficulty when the maximum and minimum of the data are unknown. In DISC, we would set the maximum/minimum to safely large/small values. For example, we can use 10 times (suppose the data are positive) the observed maximum value in the history as the maximum value of the data space. This may result in most of the data gathered at the center of the data space. It will not cause a problem for DISC, because no memory would be wasted for the empty space. And this just shows the advantage of DISC's adaptation to the data distribution.

4.2 Algorithms to Merge Cells

For the error minimization problem, we adopted an adaptive approach that consists of merging 2^d adjacent cells to form a larger one in order to meet the memory constraint. Figure 2 shows a 2-dimensional example where the order of the Z-curve m equals 2 before merging. c_0 to c_{15} are the cells before merging. c_0' to c_3' are the cells after merging. The subscripts are the Z-values of the cells. Let us denote the larger cell as $M(c)$ if it contains c before merging, then

```
Algorithm General Merge-Cells (GMC)
1   For i from 0 to 2^{m-1} − 1
2       Search the index and obtain the number of points
        in the new cell c'_i, denote the number as N_{c'_i}
3       If N_{c'_i} > K
            Discard N_{c'_i} − K points according to the
            discarding policy
End General Merge-Cells
```

Figure 4: Algorithm **GMC**

$M(c_0) = M(c_1) = M(c_2) = M(c_3) = c'_0$. In general,

$$M(c_{zv}) = c'_{\lfloor zv/2^d \rfloor} \qquad (8)$$

where zv is the Z-value of the cell. Let S be a point set. We refer to the cells before merging as *old cells* and to the larger cells after merging as *new cells*. We present two algorithms to merge cells. The first cell merging algorithm applies to any index structure (including DISC and R-tree) that adopts our general scheme, that is, to maintain at most K points in each cell. The second cell merging algorithm is specially designed to exploit DISC's special property that the points are ordered according to the value of the Z-curve (versus the R-tree where points have no ordering). The latter scheme, referred to as the *bulk cell merging* scheme, scans all the leaf nodes once, and hence is expected to be more efficient than the former *general cell merging* algorithm.

In the first algorithm **General Merge-Cells** (GMC), we examine each new cell in the order of the Z-curve. For each new cell, we search the index and find all points belonging to this cell. If there are at most K points in the cell, we will leave them in the index; otherwise, we delete some of them according to the discarding policy and retain only K points. Algorithm **GMC** is presented in Figure 4. While the GMC algorithm is straightforward and applies to any structure, it is quite expensive since it searches the index 2^{m-1} times.

The second algorithm **Bulk Merge-Cells** (BMC), utilizes the property that the points in the leaf nodes of the B*-tree are ordered according to the Z-values. The 2^d adjacent points which will form a larger cell are adjacent in the leaf nodes, so we only need to scan all the leaf nodes once and merge the points in adjacent 2^d old cells into a new cell. In difference to an R-tree, the entries with close keys in the B*-tree are adjacent to each other, therefore in addition to deleting extra points in a new cell, we also need to move the remaining K points into the same cell. We use a *write cursor* pointing to the place where we would store the next points. Algorithm **BMC** is presented in Figure 5.

In line 16 of BMC (Figure 5), rebuilding internal nodes based on existing leaf nodes is very similar to bulk loading of a B+-tree. We do not discuss the details here for brevity.

Comparing the two merging algorithms, we note that BMC scans the leaf nodes only once, while GMC

```
Algorithm Bulk Merge-Cells (BMC)
1   Free all the internal nodes
2   Let ln be the first leaf node. Set write cursor at
    the beginning of ln. Let point set S be empty.
3   While (ln) //when ln is not NULL
4       For each point P in ln
5           If this is the first point in the first leaf node
6               c' = M(c), where c is the cell P belongs to
                S = S ∪ P
7           Else if P ∈ c'
                S = S ∪ P
8           Else if P ∉ c' //We entered the next cell
9               If |S| > K
                    Discard |S| − K points from S
10              Write the points in S to the position of
                write cursor and move the write cursor
                forward accordingly
11              Let S = ∅
12              S = S ∪ P
13              c' = M(c), where c is the cell P belongs to
14      ln = right neighbor of ln
15  Free all the leaf nodes after the write cursor
16  Rebuild internal nodes of the B*-tree based on the
    leaf nodes
End Bulk Merge-Cells
```

Figure 5: Algorithm **BMC**

entails many searches and updates for each new cell. So BMC is expected to be faster than GMC. We will compare them in the experiments.

We note that the merge-cells operation is expensive compared to other operations, especially when the memory is large. As it may take a while to reduce the order of the curve by 1, stream processing may be disrupted. Fortunately, it is not necessary to finish merging all cells at once. Cell merging can be performed incrementally. When the system load is heavy, say, there is a burst of incoming data or many queries, we stop the merge operation at the current new cell we are working on and record this stop position. If the update or the query accesses the points before that stop position, we process them assuming the order of the Z-curve to be $m-1$; if data belonging to cells after the stop position are required, we process them assuming the order of the Z-curve to be m. If the search involves more than one cell, some of which may be old and some are new, query processing is performed assuming the order of the Z-curve in the new cells , $m-1$. Old cells that are accessed in the search are temporarily combined to form larger new cells, but they are in fact merged later as cell merging resumes. The error bound returned with the query results in this case, is the one associated with the order $m-1$. Both GMC and BMC can be performed incrementally. However, it is important to complete the operation fast.

4.3   Query Processing

As analyzed in Section 3.1, an ekNN query in the original data set, is a kNN query in footprints of the data.

```
Algorithm KNN Search
1   S = ∅
2   For i from 1 to d
        wl_i = q'_i − 1/(2u); wh_i = q'_i + 1/(2u)
3   WindowQuery(W). From the points in W, get the k
    nearest points to Q and put them in S; if there are
    less than k points in W, put all of them in S.
4   if |S| < k or near(W, Q) < far(S, Q)
5       for i from 1 to d
            wl_i = wl_i − 1/u; wh_i = wh_i + 1/u
6       Go to 3
7   return S
End KNN Search
```

Figure 6: Algorithm **KNN Search**

Let Q be the query point and c_Q be the cell Q belongs to. Denote as Q' the center point of c_Q and as W a query window which is a d-dimensional interval $[wl_1, wh_1], [wl_2, wh_2], ..., [wl_d, wh_d]$. First, we initiate a square-shaped window query centered at Q' with an initial side length of $1/u$ and then increase it gradually. We maintain a k candidate answer set which always contains the nearest k points to Q within the current query window. The function $near(W, Q)$ returns the distance between Q and W's nearest side to Q. The algorithm terminates when $near(W, Q)$ is larger than or equal to the k-th farthest point in the candidate answer set. All the points outside the query window are farther from Q than $near(W, Q)$. So when the algorithm terminates, the farthest point in the candidate set is the k-th nearest point to Q among all the points inside and outside the query window. To avoid searching cells which are already visited in the previous iteration, we maintain a list of addresses of the B*-tree leaf nodes visited. WindowQuery(W) is a function to retrieve all the points in window query W. In DISC, each leaf nodes of the B*-tree corresponds to a continuous segment of the Z-curve. An efficient window query algorithm proposed in [4] accesses only those nodes with their corresponding Z-curve segments intersecting the query window. We use this algorithm for our WindowQuery() function. Figure 6 shows the algorithmic description of the **KNN search**.

For continuous ekNN queries, we maintain the $ekNN$ set as follows. Let W_s be the smallest window centered at Q' that contains all the points in $ekNN$. When a new data point P comes and $P \in W_s$, we may need to discard some points according to the discarding policy (for example, in the sliding window query discussed in the next subsection, points older than T_{sw} are discarded). If a point in $ekNN$ is discarded, the $ekNN$ set would have fewer than k points at the moment. After discarding, there are 3 cases to consider: 1) *There are still k points in ekNN.* If P is nearer to Q than the farthest point in $ekNN$, then P will replace the farthest point; otherwise $ekNN$ is kept unchanged. 2) *There are fewer than k points in ekNN and P is nearer to Q than the farthest point in ekNN*

before discarding. We add P to $ekNN$ and start kNN search as in the one-time search algorithm, but we set the initial search window as W_s. 3) *There are fewer than k points in ekNN and P is not nearer to Q than the farthest point in ekNN before discarding.* We just start kNN search as in the one-time search algorithm with the initial search window W_s. The proof of the above algorithm is straightforward and we omit it here due to the limitation of space.

4.4   Sliding Window ekNN Queries

In certain applications, recent stream data are of greater interest as opposed to data associated with the entire stream. This gives rise to the sliding window data stream model [3]. The ekNN problem can be expressed in this model as well. Formally, we wish to identify the $ekNN$ of a query point Q among all data stream elements arriving in the last T_{sw} time units.

DISC is capable of supporting such sliding window ekNN queries by simply employing a time-based discarding policy. Let t be the current time. Assume that each arriving stream element is tagged with a timestamp signifying its arrival time. Algorithm **Build Index** can be modified for the sliding window model as follows: When inserting a point P to a cell c, we first check the timestamp of existing points in c. We then delete the stale points, that is, the points that arrived earlier than $t - T_{sw}$. Finally, we insert P. For algorithm **KNN Search**, we only place points arriving later than $t - T_{sw}$ to the candidate answer set S. At any time, if we encounter stale points (during index building or kNN searching), we delete them immediately. Such modifications enable DISC to answer sliding window ekNN queries correctly. However, if there are data in the index that are older than $t - T_{sw}$, but no incoming stream data is added to the cells they belong to, such stale data will remain in the index, occupy space and affect space utilization. To avoid this, we need an operation to eliminate such stale data. This can be accomplished by scanning all the points and deleting stale data from the index. However, such an operation is expected to be time consuming. Again, like the cell merging process, this stale data elimination process can be done incrementally. There exists a tradeoff between memory utilization and processing capability. To achieve best accuracy when addressing the error minimization optimization problem in the sliding window model, we eliminate stale data before each call to the **Merge-Cell** operations. This way, some additional space becomes available and it may be possible to avoid cell merging.

We should take care when processing continuous ekNN queries over sliding windows. Even no new points come in W_s, there still could be stale data due to time. Therefore, in this case we need to check whether the set contains stale data in each time unit to guarantee the correctness of the $ekNN$ set. Or if the $ekNN$

answers are not requested all the time, we can check for stale data when we retrieve answers from the maintained $ekNN$ set. If there were stale data, we discard them and invoke the kNN search on the footprints with the initial search window W_s. This is still much faster than invoking the search from scratch.

5   Experiments

In this section, we present the results of an extensive experimental study using DISC. While we have implemented and worked with an in-memory version of DISC, DISC is also applicable for secondary storage. The experiments are performed on a desktop computer with Pentium IV, 2.6G CPU and 1G RAM. In our study we employed both synthetic and real data sets. We generated exponentially and normally distributed data sets of varying dimensionality. Figure 7 shows 2-dimensional images of the two data distributions. The real data set contains 2-dimensional records
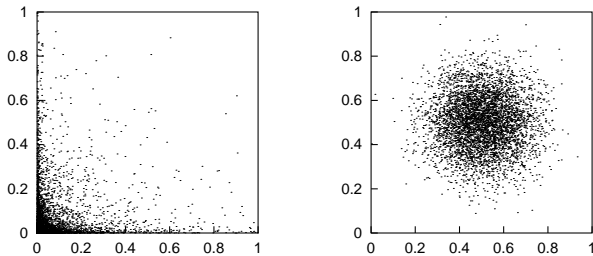

(a)Exponential distribution   (b) Normal distribution

Figure 7: Data distributions

extracted from netflow IP data logs. Such logs were aggregated temporally and ekNN queries were issued using the total number of bytes and associated packet rate attributes. All the data are normalized in the range of [0,1]. By default, we let K equal 20 and we set the order of the Z-curve as 10, which implies an error bound of 0.00138 in a 2-dimensional space. For the in-memory B*-tree, we used a default node size of 1024 bytes. First, we focus our experiments on a 2-dimensional space examining DISC's memory usage and accuracy and compare the two cell merging algorithms. Then we examine the behavior of DISC on higher dimensions.

5.1   Memory Usage of DISC

In a first series of experiments, we study the memory usage of DISC as data stream by. No existing structures or algorithms were proposed to process (approximate) kNN queries over streams as discussed in the related work. Therefore we would compare DISC with the R*-tree [5] indexing under our general scheme to see which one is more efficient. Figures 8(a), (b) and (c) present the memory used by DISC and the R*-tree as a function of the observed data stream size (in number of points) on 2-dimensional exponentially distributed
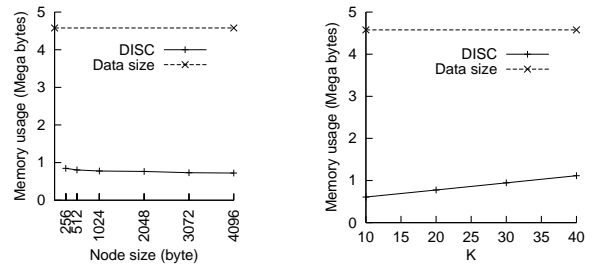
(a)Effect of Node Size   (b) Effect of K

Figure 9: Effect of node size and K

and normally distributed data sets and the real data set.

As the data continually arrive and their cumulative size increases, the memory usage of DISC increases also at first, but the increase slows down soon as more data arrive. At first, all the cells are empty and therefore all of the data are stored as footprints. But as more data come in, more and more cells become full (having contains K points) so that memory usage almost keeps constant. When 600K data points have arrived, the memory used by DISC is 10~25% of the size of the data. Using Equation 5, we calculate, for this setting that the amount of memory needed for the array based method is 41943040 bytes, which is more than 8 times the data size. These results show that DISC does adapt to different data distributions because it only stores necessary cells and in each cell, necessary points to guarantee the error bound, while the array-based method suffer from the static memory allocation greatly. The huge space cost of the array-based method make it not applicable in stream applications. In all the following experiments, the array-based method always needs at least several times the space of the original data to operate, therefore we will not compare DISC with it again. We also observe that the memory usage of the R*-tree is always a little higher than DISC. This is because while the R*-tree also allocates space only to the points requiring explicit storage, the leaf node utilization rate of the R*-tree (about 73%) is lower than that of the B*-tree (about 85%).

To see how how some parameters such as the node size and K affect the memory usage of DISC, we varied the node size and K respectively while keep other parameters constant. The memory usage for different node sizes when 600K netflow data points have arrived is presented in Figure 9 (a). The memory usage decreases as the node size increases. This is because for larger nodes, higher node utilization rate can be achieved. However, the effect of node size is small compared to the total data size. In other experiments, we used 1024 as the default node size.

Figure 9 (b) presents memory usage as a function of K for netflow data. Memory usage increases as K increases in an almost linear fashion, according to expectation. This demonstrates that DISC handles the allocation of the available memory in a space efficient fashion. Experiments over the synthetic data sets show
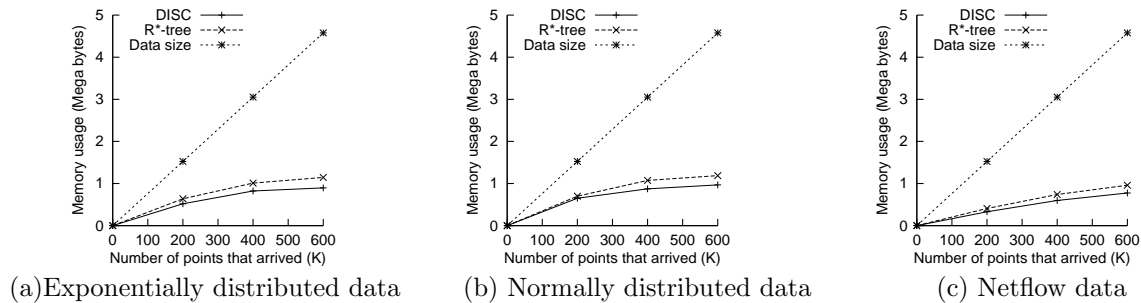
812

(a)Exponentially distributed data (b) Normally distributed data (c) Netflow data

Figure 8: Memory Usage of DISC

(a)Accuracy vs. Arrived Data Size (b)Accuracy vs. Order of the Z-curve (c)Memory Usage vs. Accuracy
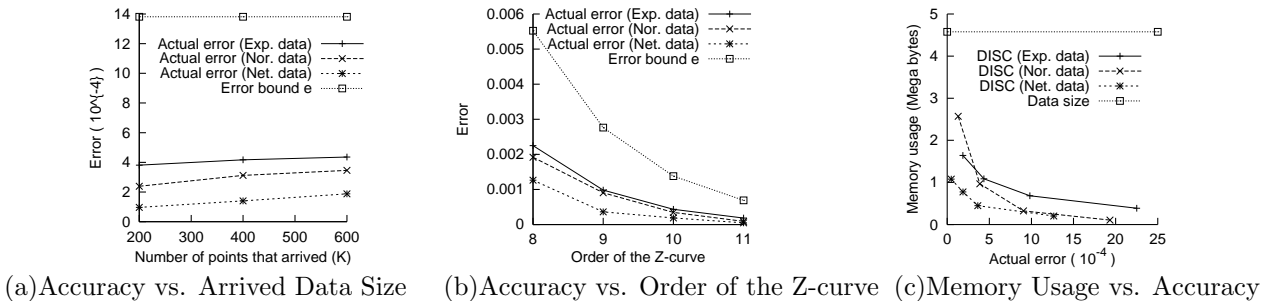
Figure 10: Accuracy of DISC

similar behavior. It is expected that the memory usage of DISC would reach the size of the stream data if K is too large, but it will not be too much beyond the stream size. In the worst case that K is infinite, all the stream data are maintained. The memory usage of DISC would be a little more than the stream size considering the space utilization of the B*-tree, but it will not grow excessively as the array-based method, which may use many times the size of the stream. In many applications, tens of nearest neighbors are enough and K can be determined from domain knowledge or query history. In these cases, DISC is still quite useful. In other experiments, we have used 20 as the default value of K, which is a reasonable number used in data mining applications.

5.2 Accuracy of DISC

While DISC can guarantee a theoretical error bound of e, we run experiments to assess the actual errors. We generated 200 queries following the same distribution as the data. We scan the original data to find the exact kNN to each query and also employ DISC to identify the $ekNN$. We then compare the exact kNN distance and the ekNN distance to obtain the actual error. The results are presented as averages over the 200 queries in Figure 10 (a). The figure shows the comparison between the error bound e and the actual error for the (exponentially distributed, normally distributed and netflow) data streams as the data arrive. We observe that the average actual errors are less than one third of the theoretical error bound. These results demonstrate the accuracy of DISC. In all our experiments, we have also observed that the maximum actual

errors are smaller than the theoretical error bounds, which further confirms the effectiveness of DISC.

In our next experiment we evaluate the impact of the order of the space-filling curve on our scheme. We vary the order of the Z-curve from 8 to 11 and see how it affects the actual errors. The error bound and actual errors for different orders of the Z-curve are shown in Figure 10 (b). As the Z-curve order increases, the error bound e and the actual errors also decrease, while the actual errors are always much smaller than e.

To see the relationship between the memory usage and the accuracy, we present for different error bounds, their corresponding memory usage versus the corresponding actual errors when 600K data points have arrived in Figure 10 (c). The memory usage increases as actual errors decrease. This shows that DISC can easily trade error for memory space by suitably setting the order of the Z-curve.
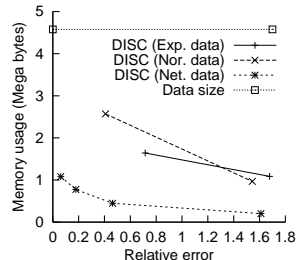

Figure 11: Memory Usage vs. Relative Error

To show that the above absolute errors are reasonably small, we also present the relative kNN distance errors they correspond to in Figure 11. For the netflow data, $ekNN$ has a relative error of 5% when the memory usage is about 1MB, which is less than 1/4 of
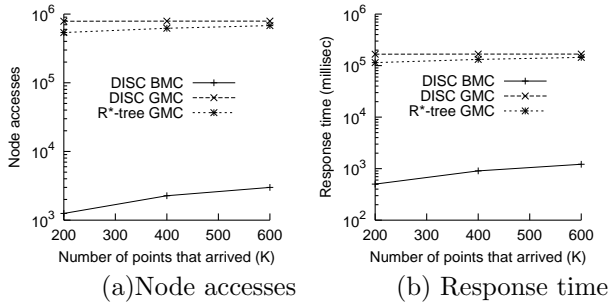
(a)Node accesses   (b) Response time

Figure 12: GMC vs. BMC

Figure 13: Update and Query Cost

the original data size. Even when the memory usage is only 200KB, which is less than 5% of the original data size, $ekNN$ has a relative error of 1.6. For the exponentially and normally distributed data sets, $ekNN$ also has small relative errors while use much less memory size than the data size.

5.3 GMC vs. BMC

In this experiment, we evaluate the two merge-cell algorithms. We have implemented the GMC algorithm for both DISC and the R*-tree. We also implemented the BMC algorithm, which only applies to DISC. We trigger the Merge-cell operation when 200K, 400K and 600K data points have arrived. (In fact, the merge-cell operation should be invoked in the case of the error minimization problem only when available memory runs out. Here we call it explicitly to observe its behavior under varying data size.) We calculate the number of node accesses and response time as measures of their performance. The results for the real data set are shown in Figure 12. We can see that under the DISC scheme, GMC needs much more node accesses than BMC (about 300 to 600 times). This is because in GMC, we need to traverse the tree for each new cell. To support a reasonably small error bound, usually the order of the Z-curve is large, which is 10 in our experiments. So we have to traverse the tree $2^{9 \times 2} = 262144$ times, and each traversal incurs several node accesses (descend the tree and locate the points to the new cell). While in BMC, we only scan all the leaf nodes once (which ranges from hundreds to a few thousand in our experiments). GMC for the R*-tree turns out to be marginally better than its DISC counterpart. This is because in the R*-tree, when some points are discarded from a cell, we are not required to move the remaining points together while in the B*-tree this is necessary. The response time has similar trend. In the experiments, the GMC algorithm takes several minutes to finish while the BMC algorithm takes only 1 or 2 seconds. So clearly, only the BMC algorithm is applicable in practice. This is an additional reason that makes DISC preferable over other approaches. Despite its efficiency, we can still perform incremental cell merging with BMC as described in Section 4.2 in case the memory is very large and the system load is heavy.
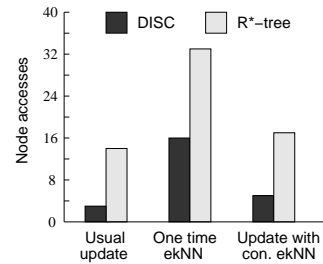
5.4 Updates and Query Processing

To evaluate the update and query processing performance of DISC, we measured the number of node accesses of updates, one-time ekNN query and continuous ekNN query processing for DISC and the R*-tree. The cost of a continuous ekNN query consists of the cost of the initial one-time ekNN query and the cost of maintaining the $ekNN$ set continuously. The maintenance cost is the possible search cost when a point in W_s arrives as described in the continuous ekNN algorithm. Specifically, maintaining the $ekNN$ set involves possible kNN search during the insertion of new points. Therefore, the update cost with continuous ekNN queries running is expected to be higher than the usual update cost.

In our experiments, the query costs of the one-time ekNN queries are averaged from 200 queries which follow the same distribution as the real data set. For continuous ekNN queries, we use the same queries but run 10 continuous queries simultaneously each time. The update costs are averaged from the 600K points inserted. K is still set as 20. The results on the netflow data set are shown in Figure 13. First we observe that for all the operations, DISC has much lower node access cost than the R*-tree. The reason is that in DISC we only store the Z-value as the key, but in the R*-tree we need to store $2d$ values as keys so the fan-out of the tree is lower and hence the height of the tree larger. In addition, there are overlaps between the MBRs of the R*-tree, which also incurs more node accesses. We also notice that the query processing cost is not large in terms of node accesses. This is largely due to the Z-order keeping the proximity of the spatial points and the efficient WindowQuery() algorithm. In addition, in two-dimensional space, the points are dense. For skewed data, most points are clustered at a relatively small region and so do the queries. So for most queries, after locating the cell the query belongs to, we need only a few number of node accesses to retrieve near points. The cost of the continuous ekNN is mainly expressed in the additional part of the update cost. We can see that, update with continuous ekNN queries running costs a little more than the usual update, but the increase is not great. Therefore, the continuous ekNN query processing is still quite efficient.

5.5 DISC on Data Sets of Other Dimensions

We study the behavior of DISC when the number of underlying streams increases (and as a result the dimensionality of stream elements increases as well). Due to the limitation of space, we only present the results on 3-dimensional synthetic data sets. Figure 14 (a) shows the memory usage of DISC as 3-dimensional synthetic data stream by. We still set K as 20 and the order of the Z-curve as 10, which corresponds to an error bound of 0.00169 in 3-dimensional space. The results are sim-
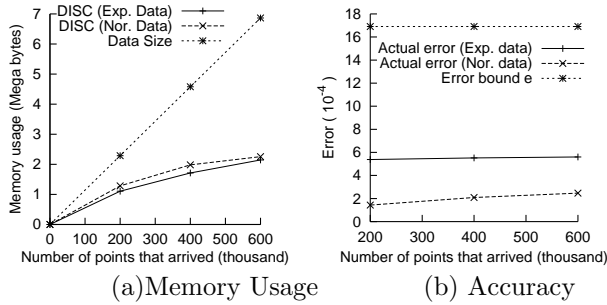

(a)Memory Usage   (b) Accuracy

Figure 14: DISC on 3D data sets

ilar to those of 2-dimensional data (compare with Figures 8 (a) and 8 (b)). DISC uses much less memory compared to the original data size and its memory usage does not increase significantly as the number of arriving data elements increases. As dimensionality increases, DISC tends to occupy more memory than in the 2-dimensional case; this is expected as in higher dimensions, points become relatively sparse and therefore distributed in more cells, which have to be maintained. Similar to the experiments on 2-dimensional data, the average actual errors are much lower than the error bounds as shown in Figure 14 (b).

6 Conclusion

We investigated the k nearest neighbors problem in the data stream model. We introduced the *e*-approximate *k* nearest neighbors (ekNN) problem and presented a structure called DISC to address it over data streams. DISC achieves the goals of memory optimization given an error bound or adjusts itself to achieve the best accuracy to answer ekNN queries when a memory constraint is given. At the same time, DISC retains efficient update and query processing which is a common requirement for data stream applications. Extensive studies on both synthetic and real data showed that the memory usage of DISC is small and the actual errors are much lower than the theoretical error bounds that the structure guarantees.

References

[1] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *SODA*, 1994.

[2] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *JACM*, 45(6):891–923, 1998.

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

[4] R. Bayer. The universal B-tree for multidimensional indexing: General concepts. *World-Wide Computing and Its Applications 97*, pages 10–11, 1997.

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

[6] S. Berchtold, C. Böhm, H. Jagadish, H.-P. Kriegel, and J. Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *ICDE*, 2000.

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

[8] M. Bern. Approximate closest-point queries in high dimensions. *Information Processing Letters*, 45(2):95–99, 1993.

[9] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using hamming norms (how to zero in). In *VLDB*, 2002.

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

[11] J. Goldstein and R. Ramakrishnan. Contrast plots and p-sphere trees: Space vs. time in nearest neighbour searches. In *VLDB*, 2000.

[12] G. Hjaltason and H. Samet. Ranking in spatial databases. In *SSD*, 1995.

[13] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

[14] N. Katayama and S. Satoh. The sr-tree: an index structure for high-dimensional nearest neighbor queries. In *SIGMOD*, 1997.

[15] D. E. Knuth. *The Art of Computer Programming, Volume 3*. Addison Wesley, 2002.

[16] F. Korn, S. Muthukrishnan, and D. Srivastava. Reverse nearest neighbor aggregates over data streams. In *VLDB*, 2002.

[17] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *STOC*, 1998.

[18] C. Li, E. Y. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *TKDE*, 14(4):792–808, 2002.

[19] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS*, 1984.

[20] W.-G. Teng, M.-S. Chen, and P. Yu. A regression-based temporal pattern mining scheme for data streams. In *VLDB*, 2003.

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

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

[23] R. Zhang, B. C. Ooi, and K. L. Tan. Making the pyramid technique robust to query types and workload. In *ICDE*, 2004.