Efficient k-NN Search on Streaming Data Series

Xiaoyan Liu[1] and Hakan Ferhatosmanoğlu[1]

Ohio State University,
Department of Computer and Information Science,
1971 Neil Ave., Columbus, OH 43220
{liuxia, hakan}@cis.ohio-state.edu

Abstract. Data streams are common in many recent applications, e.g. stock quotes, e-commerce data, system logs, network traffic management, etc. Compared with traditional databases, streaming databases pose new challenges for query processing due to the streaming nature of data which constantly changes over time. Index structures have been effectively employed in traditional databases to improve the query performance. Index building time is not of particular interest in static databases because it can easily be amortized with the performance gains in the query time. However, because of the dynamic nature, index building time in streaming databases should be negligibly small in order to be successfully used in continuous query processing. In this paper, we propose efficient index structures and algorithms for various models of k nearest neighbor (k-NN) queries on multiple data streams. We find scalar quantization as a natural choice for data streams and propose index structures, called VA-Stream and VA$^+$-Stream, which are built by dynamically quantizing the incoming dimensions. VA$^+$-Stream (and VA-Stream) can be used both as a dynamic summary of the database and as an index structure to facilitate efficient similarity query processing. The proposed techniques are update-efficient and dynamic adaptations of VA-file (vector-approximation file) and VA$^+$-file, and are shown to achieve the same structures as their static versions. They can be generalized to handle aged queries, which are often used in trend-related analysis. A performance evaluation on VA-Stream and VA$^+$-Stream shows that the index building time is negligibly small while query time is significantly improved.

1 Introduction

Data streaming has recently attracted the attentions of several researchers [1, 7, 2, 14, 4, **?**,8]. Many emerging applications involve periodically querying a database of multiple data streams. Such kind of applications include online stock analysis, air traffic control, network traffic management, intrusion detection, earthquake prediction, etc. In many of these applications, data streams from various sources arrive at a central system. The central system should be able to discover some useful patterns based on the specifications provided by the user. Due to the

streaming nature of the involved data, a query is continuously evaluated to find the most similar pattern whenever new data are coming. Immediate responses are desirable since these applications are usually time-critical and important decisions need to be made upon the query results. The dimensionality of the data sets in these applications is dynamically changing over time when new dimensions are periodically appended.

There are many types of scenarios occurring in data stream applications. These scenarios can have either streaming or static queries, and the database either is fixed or consists of data streams. In prior related work [7, 9, 8], only the scenario with streaming queries and fixed time series database is discussed. However, in many important applications, the database itself is formed by streaming data too. For example, in stock market analysis, a database is usually formed by incrementally storing multiple data streams of stock prices. The users are often concerned with finding the nearest neighbors of a specific stock or all pairs of similar stocks. In this scenario, the streaming database is usually sized by a predefined window on the most recent dimensions, e.g. the last 30 days, or the last 24 hours, etc. The user-specified stock can be either taken from the streaming database or a fixed stock pattern. In order to be able to respond to the volatility of the stock market, the user requests need to be analyzed continuously whenever new stock prices arrive. In this paper, the scenarios with streaming database instead of fixed database will be studied in order to address these emerging needs. The queries are either fixed or streaming. Several techniques will be proposed respectively to increase the overall query response time for each scenario.

Data stream applications may involve predefined queries as well as ad hoc and online queries. Prior information on the predefined queries can be used to improve the performance. However, for efficient support of ad hoc and online queries it is necessary to build highly dynamic index structures on the data streams. Hence, we are motivated to propose techniques for both cases, i.e., predefined and online queries, in this paper. Since the dimensionality is changing for stream data, it will be beneficial to dynamically index such data to enable the support for efficient query processing. Although index structures in the literature are designed to handle inserting and deleting of new data objects, we are not aware of any mechanisms that handle dynamic dimensionality.

R-tree based index structures have shown to be useful in indexing multidimensional data sets [13, 3], but they are not suitable for indexing data streams since they are designed for the cases where the dimensionality is fixed. Based on this observation, we are motivated to come up with an index structure which can accommodate the changing dimensionality of data objects. Since scalar (one-dimensional) quantization is performed on each dimension independently, the access structure based on such an idea seems to be a better choice for handling dynamic dimensionality. In this paper, we propose effective scalar quantization based indexing techniques for efficient similarity searching on multiple data streams. The proposed technique can be used both as an index and as a summary for the database, which can produce accurate answers to queries in an incremental way.

Our contributions are as follows: we first study the model for processing data streams, formulate several important data streaming scenarios. Both sliding window and infinite window cases are considered for the completeness. We then present an index structure to support efficient similarity search for multiple data streams. The proposed technique is dynamic, update-efficient, and scalable for high dimensions, which are crucial properties for an index to be useful for stream data. To the best of our knowledge, there has been no techniques for indexing data with dynamic dimensionality such as streams. Our method can also be generalized to support the aged query in trend-related analysis of streaming database. In the context of aged query, the users are more interested in the current data than in the past data, and bigger weights will be assigned to more recent dimensions of the data.

This paper consists of five sections. Section 2 describes a stream processing model and then gives a detailed formulation of three different but equally significant data streaming scenarios. In Section 3, a brief review of scalar quantization techniques VA-file and VA$^+$-file will be given. We then motivate the need for an adapted version of VA-file and VA$^+$-file, which is specifically used to tackle the problem imposed by dynamic streaming data. At last, the proposed techniques, called VA-Stream and VA$^+$-Stream, are presented in this section. In Section 4, an extensive performance evaluation of proposed techniques is given which shows that significant improvements are achieved by the proposed technique. Finally, Section 5 concludes the paper with a discussion.

2 Streams and Queries

In this section, we start with introducing a general stream processing model, and then discuss several important scenarios in streaming database followed by precise definitions of their related queries.

2.1 Stream Processing Model

Figure 1 shows our general architecture for processing continuous queries over streaming database.

The Real Time Stream Manager is responsible for updating the synopsis in real time and archiving the stream data. Whenever there are new data streaming in, the synopsis is changed correspondingly. When a query arrives at Query Processor, the processor will first look at the synopsis in main memory and eliminate as many disqualified candidates as possible from the whole pool of data streams. The details regarding how the elimination procedure works will be discussed in Section 3. After the elimination process, the possible candidates have been narrowed down to a much smaller set. This set can be used to obtain approximate answers to the queries. For exact answers to the queries, the processor can read each candidate from the archived data in the secondary storage. The key element in our processing model is Real Time Stream Manager. It should

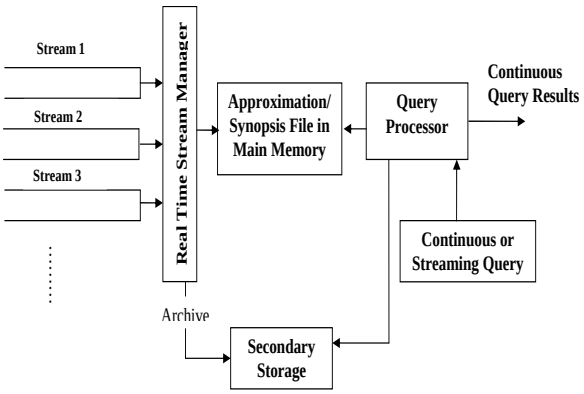


Fig. 1. Stream Processing Model

be able to generate a synopsis which can be used to prune the data efficiently, so that the secondary storage access time will be greatly minimized if accurate answers are desired. Moreover, Real Time Stream Manager should be able to sketch the incoming data streams in a prompt way. The proposed indexing techniques in this paper will be shown to be capable of processing the incoming data streams and generate the synopsis in an efficient way. Hence it can support the implementation of Real Time Stream Manager.

We also look at continuous or streaming queries with either a sliding window or an infinite window. For queries with a fixed-size sliding window, the size of the synopsis stays the same, and the accuracy of this synopsis does not suffer from the evolving dimensionality. However, for queries with an infinite window, though it can be solved in a way similar to those with fixed-size window, the accuracy will get sacrificed if the size of synopsis stays the same. Hence, another approach for queries with an infinite window would be based on an reasonable assumption that the data streams are aging. That's actually what happens in real world, where people put more emphasis on more recent activities. A discussion of aged queries can be found in Section 3.4.

2.2 Queries

We define streaming database as a collection of multiple data streams, each of which arrives sequentially and describes an underlying signal. For example, the data feeds from a sensor network form a streaming database. The dimensionality of each data stream is always increasing in this case. Hence, theoretically the amount of data stored, if stored at all, in a streaming database tends to be infinite. This leaves us with a challenge of trying to get accurate query results from a huge database with time constraints. Moreover, in most cases users would expect fast response time for queries. This makes it necessary to develop an effective index structure for streaming database with very efficient update cost, so that query results can be obtained in a tolerable amount of time. In this paper,

we are considering several important scenarios occurring in streaming database. See Table 1 for notations and Table 2 for a classification of these scenarios.

Notation	Meaning
x,y	streaming data objects in database
$x[i,j]$	slices of x between time position i and j
$x[.,i]$	slices of x up to time position i
q	query object
$len(q)$	the number of dimensions in the query, i.e. the length of q
$dis(x,y)$	the distance function between x and y
D	the streaming database
b	the total number of available bits
b_i	the number of bits assigned to dimension i
σ_i^2	the variance of the data
g_i	the weight of dimension i in an aged query

Table 1. Frequently used notations.

The first scenario occurs when streaming queries are issued to a streaming database. For example, we have a streaming database containing stock quotes of companies, which is updated daily or even hourly to include the new quotes. In this case a query may be posed to find the most similar company to a given company in terms of the stock price in last 2 months. We formally define this scenario as QUERY 1, which is shown below.

QUERY 1: Given a streaming database D, let q be the streaming query object at time position t, the nearest neighbor of q for the last T time period would be r if $dis(q[t-T,t],r[t-T,t]) \leq dis(q[t-T,t],s[t-T,t]),\ \forall s\ in\ D$.

The second scenario is similar to the first scenario, except in a more general sense. In the first scenario, we are only interested in the past data sets which are collected over a certain amount of time period. However, in the second scenario, we are interested in all of the data sets collected up to now. For example, we want to find the company most similar to a given one after considering all the history records of their stock prices. This scenario often occurs when a long-term plan or strategy is concerned, and it is formulated as QUERY 2.

QUERY 2: Given a streaming database D, let q be the streaming query at time position t, the nearest neighbor query of q would be r if $dis(q[.,t],r[.,t]) \leq dis(q[.,t],s[.,t])$,
$\forall s\ in\ D$.

In the above two scenarios, the streaming query objects usually come from the streaming database. The third scenario occurs when queries are predefined over streaming databases. For example, we have a history bull pattern for stocks, and a regular check against the streaming database would find us those companies whose stock price fluctuations match this pattern most. This is an interesting scenario, and we formally define it as QUERY 3.

QUERY 3: Given a streaming database D, let q be the predefined query object at time position t with a fixed length len(q), the nearest neighbor of q would be r if $dis(q, r[t - len(q), t]) \leq dis(q, s[t - len(q), t]),\ \forall s\ in\ D$.

QUERY 1 and QUERY 3 are sized by a sliding window, and QUERY 2 has an infinite window. Their definitions can be easily generalized for k nearest neighbor queries. All these discussed queries are usually characteristics of their continuity, which means the same query will be asked continuously against the database over a period of time and will be evaluated over and over again during that time period. For example, Traderbot [1, **?**], a web-site which performs various queries on streaming stock price series, offers a variety of queries similar to the models discussed here.

Query Type	Query	Database
QUERY 1	Streaming data objects	Streaming database with a sliding window
QUERY 2	Streaming data objects	Streaming database with an infinite window
QUERY 3	Predefined data objects	Streaming database with a sliding window

Table 2. Summary of Queries in Streaming Database.

To answer the above continuous queries occurring in streaming database, the most straightforward method is to do a sequential scan on all data objects with all available dimensions at the moment when the queries are issued. However, the streaming databases are usually not only large but also high-dimensional, hence this method cannot deliver a satisfying performance in terms of overall query response time. Hence, efficient index structures are needed to accommodate the uniqueness of streaming databases, which have (1) high dimensionality (2) dynamically growing/changing dimensions (3) large amount of data. R-tree based index structures can be used to improve the efficiency of similarity searching in multi-dimensional databases, however it is well known that the performance of R-tree degrades as the number of dimensions increases. Moreover, the update operations of R-tree pose another problem for data streams, since the dimensionality is frequently changing in streaming database. Whenever a new dimension comes, it completely destroys the basis used to construct the R-tree, i.e. the spatial containment relationship between levels of the tree structure. Hence, R-tree based structures cannot be used to index streaming database. In this paper, we propose scalar quantization based indexing technique as a natural choice for handling dimensionality changes. Since the scalar quantization is performed independently on each dimension, when a new dimension is added it can be quantized separately without making major changes to the overall structure. Quantization of data can also serve as the summary to answer queries approximately. Depending on the time and space constraints, scalar quantization-based summary of data can be effectively used as an approximation as well as an index for the actual data.

The efficient scalar quantization-based solutions to QUERY 1 and QUERY 2 will be discussed in Section 3.3. For QUERY 3, it can be treated as a special case of QUERY 1 with queries fixed , hence no separate discussions will be made on it.

3 Scalar Quantization Technique

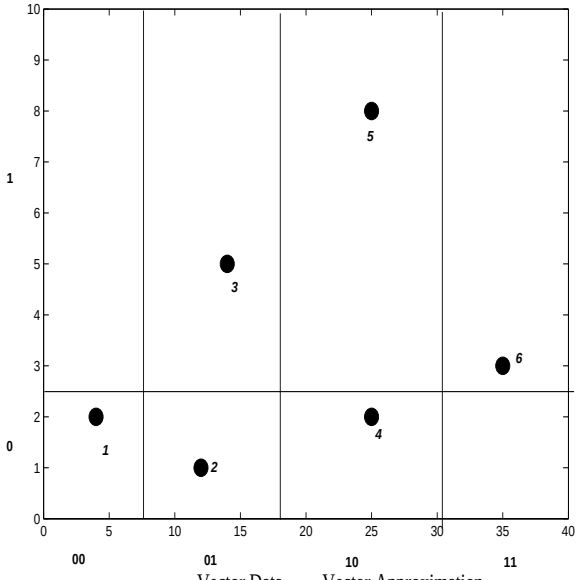
Scalar quantization technique is a way to quantize each dimension independently so that a summary of the database is efficiently captured. It also serves as an index structure for efficient point, range, and k nearest neighbor queries. By observing that there is no cross-interference among dimensions when applying scalar quantization techniques to the traditional databases, we are motivated to apply it to support the similarity search in streaming databases.

3.1 Indexing Based on Vector Approximation

We will first briefly review one scalar quantization technique used in traditional database, Vector-Approximation file (VA-file) [21], to introduce some background knowledge.

Since conventional partitioning index methods, e.g. R-trees, grid files and their variants, suffer from dimensional curse, VA-file was proposed as an approach to overcome the curse and supports efficient similarity search in high-dimensional spaces. The VA-file is actually a filter-based approach of synopsis files. Here is a sketch of how it works. It divides the data space into 2^b rectangular cells where b is the total number of bits specified by the user [21]. Each dimension is allocated a number of bits, which are used to divide it into equally populated intervals on that dimension. Each cell has a bit representation of length b which approximates the data points that fall into this cell. The VA-file itself is simply an array of these bit vector approximations based on the quantization of the original feature vectors. Figure 2 shows an example of one VA-file for a two-dimensional space when $b = 3$.

Nearest neighbor searching in a VA-file has two major phases [21]. In the first phase, the set of all vector approximations is scanned sequentially and lower and upper bounds on the distance of each vector to the query vector are computed. The real distance can not be smaller than the distance between the query vector and the closest point on the corresponding cell. Therefore, the real distance can be lower-bounded by this smallest possible distance. Similarly, the distance of the furthest point in the cell with the query point determines the upper bound of the real distance. In this phase, if an approximation is encountered such that its lower bound exceeds the smallest upper bound found so far, the corresponding objects can be eliminated since at least one better candidate exists. At the end of the first phase, the vectors with the smallest bounds are found to be the *candidates* for the nearest neighbor of the query. In the second phase, the algorithm traverses the real feature vectors that correspond to the candidate set



	Vector Data	Vector Approximation
1	(4,2)	00 0
2	(12,1)	01 0
3	(14,5)	01 1
4	(25,2)	10 0
5	(25,8)	10 1
6	(35,3)	11 1

Fig. 2. A two-dimensional example for VA-file

remaining after filtering. The real feature vectors of the candidates are visited until it is guaranteed that the actual nearest neighbor is found. The feature vectors are visited in the order of their lower bounds and then exact distances to the query point are computed. One of the advantages of traversing the feature vectors in the order of their lower bounds to the query is that the algorithm can stop at a certain point and does not check the rest of the candidates. If a lower bound is reached that is greater than the k-th actual nearest neighbor distance seen so far, then the algorithm stops retrieving the rest of the candidates. It is important to note that the accesses in the second phase are usually secondary storage accesses. It is crucial to decrease the number of vectors visited in the second phase to reduce the number of random I/O that must be incurred during the nearest neighbor search.

3.2 Vector Approximation Based Indexing for Non-Uniform Data Sets

In the VA-file approach, although there is always an option of non-uniform bit allocation among dimensions, no specific algorithm for that option was proposed in [21]. Moreover, in [21] each dimension i is divided into 2^{b_i} cells of either equal size, or equal population, which are the two simplest partitionings that only suits a uniformly distributed data set. Some problems might occur for VA-files if the data set is not uniformly distributed, especially when it is highly correlated or clustered. VA$^+$-file [6] is proposed to target non-uniform data sets, and can lead to more efficient searching.

A scalar quantizer [10] is a VA-file together with representative values assigned to each cell. The target is to achieve the least reproduction error, i.e., the least average Euclidean distance between data points and their representatives. The partitioning performed by the VA-file approach can be viewed as a scalar quantization, except that the approach does not care about the representative values for the cells. It is shown in [6] that a scalar quantization designed by directly aiming for the least possible reproduction error would result in much tighter lower and upper bounds for the distances between the query point and the data points. Since tighter lower bounds mean less number of vectors visited in the second phase of the VA-file algorithm, and tighter upper bounds mean better filtering of data in the first phase, VA$^+$-file uses the reproduction error as its minimization objective in order to increase its pruning ability. An approach for designing a scalar quantizer follows these steps:

1. The total available bits specified by the quota is allocated among the dimensions non-uniformly, based on one bit allocation algorithm, shown below as ALGORITHM 1.
2. An optimal scalar quantizer is then designed for each dimension independently, with the allocated number of bits. The Lloyd's algorithm [17, 16], shown below as ALGORITHM 2, is used here to quantize each dimension optimally. No assumption about data uniformity is needed for this algorithm and data statistics is used instead.

Before the first step, if the user prefers, the data could be first transformed into a more suitable domain, for example Karhunen Loeve Transform (KLT) [15] domain, so that the data can be successfully decorrelated. The goal in VA$^+$-file is to approximate a given data set with a minimum number of bits but maximum accuracy. Therefore, it is crucial in VA$^+$-file to analyze the dimensions and allocate the bits to dimensions, rather than using the simple uniform bit allocation, so that the resulting accuracy obtained from the approximation can be maximized. Non-uniform bit allocation is an effective way to increase the accuracy of the approximations for any data set.

Let the variance of dimension i be σ_i^2, and the number of bits allocated to dimension i be b_i. Assume the quota is b bits, i.e, $b = \sum_i b_i$ always holds. In quantization theory, a well-known rule is: if for any two dimensions i and $j, \exists\ k \geq 0,\ \text{st.}\ \sigma_i^2 \geq 4^k \sigma_j^2$, then $b_i \geq b_j + k$ [10] and it is shown to be a good heuristic for allocating bits based on the significance of each dimension [6]. The significance of each dimension is decided by its variance.

ALGORITHM 1 Bit Allocation Algorithm for VA$^+$-file:

1. Begin with $d_i = \sigma_i^2$, $b_i = 0$ for all i, and $k = 0$.
2. Let $j = argmax_i d_i$. Then increment b_j and $d_j \leftarrow d_j/4$.
3. Increment k. If $k < b$, go to 2, else stop.

Once the number of bits assigned to a given dimension is known, an actual scalar quantizer is designed based on ALGORITHM 2, shown as follows. This algorithm will produce a set of partition points or intervals along each dimension. This algorithm is actually a special case of a popular clustering algorithm, the so-called K-means algorithm in the clustering literature [18].

ALGORITHM 2 Lloyd's Algorithm:

Denote by t_n the value of the nth data point along the dimension currently considered for quantization. Start with a given set of intervals $[c_i, c_{i+1})$ for $i = 1, \ldots, K$, where $K = 2^{b_d}$, $c_1 = \min_n t_n$ and $c_{K+1} = \epsilon + \max_n t_n$. Set $\Delta = \infty$, and fix $\gamma > 0$. Denote by r_i the representative value for the i^{th} interval $[c_i, c_{i+1})$. b_d is the number of bits allocated to the current dimension, ϵ and γ are small positive integers.

1. For $i = 1, \ldots, K$, compute the new representative value r_i as the center of mass of all data points in the interval $[c_i, c_{i+1})$, i.e.,

$$r_i = \frac{1}{N_i} \sum_{t_n \in [c_i, c_{i+1})} t_n \ ,$$

   where N_i is the total number of data points in the interval $[c_i, c_{i+1})$.
2. Compute the new intervals by $c_i = \frac{r_{i-1} + r_i}{2}$ for $i = 2, \ldots, K$.

3. Compute the total representation error as

$$\Delta' = \sum_{i=1}^{K} \sum_{t_n \in [c_i, c_{i+1})} (t_n - r_i)^2 \ .$$

If $\frac{\Delta - \Delta'}{\Delta} < \gamma$, then STOP. Otherwise set $\Delta = \Delta'$, and go to step 1.

ALGORITHM 2 always converges, but suitable choice of initialization value for the intervals should be carefully made in order to avoid being stuck in local optima. Equally-populated intervals will be a good choice.

3.3 VA-Stream Technique for Indexing Streaming Database

As we have noticed, both VA-file and VA$^+$-file are targeted towards traditional databases, in which the dimensionality of data objects is fixed. In order to handle dynamic streaming databases, the approaches should be customized for streaming databases. We call the customized approaches as VA-Stream and VA$^+$-Stream in this paper. Just as VA-file or VA$^+$-file can be viewed as a way to generate the synopsis or summarization of traditional databases, VA-Stream or VA$^+$-Stream is also a way to generate dynamic synopsis or summarization for streaming databases with dynamic updates. Since VA-Stream and VA$^+$-Stream are capable of taking a snapshot of any moment of the dynamic streaming databases, a preliminary analysis can be made on the current snapshot so that the data set can be preprocessed and a better performance for efficient similarity searching can be achieved. The proposed approach is an incremental way to update the VA-file or VA$^+$-file to reflect the changes in the databases, and it can eliminate the need to rebuild the whole index structure from scratch. Hence it enables faster query response time.

With the use of scalar quantization technique, if the data is indexed with a sliding window on recent dimensions, the previously quantized first dimension will be replaced by the quantized new dimension. This is the case for QUERY 1. It is a plain idea but it might get complicated with a need for bits reallocation and optimal quantization. The complexity of bit reallocation algorithm for different query types will depend on whether it is based on VA-file or VA$^+$-file. As we all know, the bit allocation strategy in regular VA-file approach is quite simple, with the same number of bits equally assigned to each dimension. Hence, for an initial vector approximation file and the dynamic incoming streams, the steps to restructure a new vector approximation file based on the previous one will be relatively simpler too. The algorithm used to build a new VA-file in order to capture the up-to-date snapshot of the streaming databases is called VA-Stream, and shown as ALGORITHM 3a.

ALGORITHM 3a VA-Stream:

1. Assume the window size for the database is k. Begin with the initial data set of k dimensions, build the original VA-file for this k-dimensional data set, where the total number of available bits b is equally assigned to each dimension. Hence, $b_i = \lfloor \frac{b}{k} \rfloor + 1,\ \forall i$ in $[1, mod(b,k)]$; and $b_i = \lfloor \frac{b}{k} \rfloor,\ \forall i$ in $(mod(b,k), k]$. When new dimension data is coming, let it be j and $j = k + 1$.
2. Let $b_j = b_{(j-k)}$ and $b_{(j-k)} = 0$. Compute the vector approximation for j^{th} dimension, and replace the data for $(j-k)^{th}$ dimension in the VA-file with the newly computed approximation.
3. If there is still new dimension coming, increment j and go to 2, else wait.

The above algorithm can be used to handle QUERY 1, since it has a fixed-size sliding window for the data sets. It uses VA-file as the index structure. The idea is simple by always keeping the same number of dimensions in VA-file with the new dimension replacing the oldest dimension. However, the power of VA-Stream is limited since it is only suitable for uniform data sets. Hence, a more general scheme is needed in order to handle non-uniform data sets. For this purpose, VA$^+$-file can be used as the basis index structure. Therefore, the following ALGORITHM 3b is proposed. Its bit reallocation algorithm is more complicated than the one in VA-Stream due to the following facts: (1) VA$^+$-file allocates different number of bits to different dimensions in order to deal with the non-uniformity of the data set, and (2) VA$^+$-file quantizes each dimension independently with its assigned bits in order to achieve the least reproduction error.

For non-uniform data sets, when a new dimension comes, we first evaluate its significance. Thereafter, a bit reallocation scheme should be applied in order to justify the significance of each dimension. The number of bits allocated to each dimension should be decided by its variance. The same quantization principle applies here : if $\sigma_i^2 \geq 4^k \sigma_j^2$, then $b_i \geq b_j + k$ [10].

The following ALGORITHM 3b illustrates the steps to build an up-to-date VA$^+$-file for streaming databases, and we call it VA$^+$-Stream. This algorithm assigns those extra bits contributed by the oldest dimension based on comparing the variance of new dimension with all other remaining dimensions. When no extra bit is left, this algorithm will continue to check if the new dimension deserves more bits. The detailed algorithm is shown below.

ALGORITHM 3b VA$^+$-Stream:

1. Assume the window size for the database is k. Begin with the initial data set of k dimensions, build the original VA$^+$-file for this k-dimensional data set, where the total number of available bits b is unevenly assigned to k dimensions based on the data distribu-

tion. When new dimension data is coming, let the new dimension be $j = k + 1$ and let $b_j = 0$ initially.

2. Let $t = j - k$ be the oldest dimension which we need to take out from VA$^+$-file. Now we have b_t bits available for reallocation. Let $\sigma_i^{'2} = \frac{\sigma_i^2}{4^{b_i}}, \forall i$ in $[t, j]$, and it represents the current significance of dimension i after being assigned b_i bits. Let $s = \max_{n=t+1}^{n=j}(\sigma_n^{'2})$, then $b_s = b_s + 1$, $\sigma_s^{'2} = \frac{\sigma_s^{'2}}{4}$, and $b_t = b_t - 1$. Repeat this procedure, until $b_t = 0$.
3. When $b_t = 0$, if $\sigma_j^{'2} > 4 \min_{n=t+1}^{n=j-1} \sigma_n^{'2}$, more bits need to be extracted from other dimensions for use by dimension j. Let $b_j = b_j + 1$, and $b_s = b_s - 1$, where $s = \min^{-1}{}_{n=t+1}^{n=j-1} \sigma_n^{'2}$ (min^{-1} returns the index of the minimum). Also $\sigma_j^{'2} = \frac{\sigma_j^{'2}}{4}$, and $\sigma_s^{'2} = 4\sigma_s^{'2}$. Repeat this procedure, until $\sigma_j^{'2} \leq 4 \min_{n=t+1}^{n=j-1} \sigma_n^{'2}$. When $\sigma_j^{'2} \leq 4 \min_{n=t+1}^{n=j-1} \sigma_n^{'2}$, go to step 4 directly.
4. If $b_j > 0$, quantize the j^{th} dimension based on the number of bits assigned to it, b_j, using ALGORITHM 2.
5. Check if there are any other dimensions whose bits assignments have been changed during the step 2 and step 3. Re-quantize each of those affected dimensions independently using ALGORITHM 2.
6. If there is still new dimension coming, increment j and go back to step 2, else wait.

ALGORITHM 3c is a different way to implement the idea of VA$^+$-Stream. It starts with a different perspective by comparing the variance of the oldest dimension with the newest dimension. It is based on the observation that if the variance of the newest dimension is larger than the oldest dimension, it will at least deserve the same number of bits from the oldest dimension. The detailed algorithm is shown below.

ALGORITHM 3c VA$^+$-Stream:

1. Assume the window size for the database is k. Begin with the initial data set of k dimensions, build the original VA$^+$-file for this k-dimensional data set, where the total number of available bits b is unevenly assigned to k dimensions based on the data distribution. When new dimension data is coming, let the new dimension be $j = k + 1$ and let $b_j = 0$ initially.
2. Let $t = j - k$ be the oldest dimension which we need to take out from VA$^+$-file. Now we have b_t bits available for allocation. Let $\sigma_i^{'2} = \frac{\sigma_i^2}{4^{b_i}}, \forall i$ in $[t, j]$, and it represents the current significance of dimension i.

3. If $\sigma_j^{'2} > \sigma_t^{'2}$ and $b_t > 0$, then $b_j = b_j + 1$, $\sigma_j^{'2} = \frac{\sigma_j^{'2}}{4}$, $b_t = b_t - 1$ and $\sigma_t^{'2} = 4\sigma_t^{'2}$. Repeat until either $b_t = 0$ or $\sigma_j^{'2} < \sigma_t^{'2}$. Go to step 4.

4. If $\sigma_j^{'2} > \sigma_t^{'2}$ and $b_t = 0$, then more bits need to be extracted from other dimensions for use by dimension j.

   case 1: $\sigma_j^{'2} > 4 \min_{n=t+1}^{n=j-1} \sigma_n^{'2}$

   Let $b_j = b_j + 1$, and $b_s = b_s - 1$ where $s = \min^{-1}{}_{n=t+1}^{n=j-1} \sigma_n^{'2}$. Also $\sigma_j^{'2} = \frac{\sigma_j^{'2}}{4}$, and $\sigma_s^{'2} = 4\sigma_s^{'2}$. Go back to step 4.

   case 2: $\sigma_j^{'2} < 4 \min_{n=t+1}^{n=j-1} \sigma_n^{'2}$

   Go to step 5.

   If $\sigma_j^{'2} < \sigma_t^{'2}$ and $b_t > 0$, then let $s = \max_{n=t+1}^{n=j}(\sigma_n^{'2})$, and $b_s = b_s + 1$, $\sigma_s^{'2} = \frac{\sigma_s^{'2}}{4}$, $b_t = b_t - 1$, $\sigma_t^{'2} = 4\sigma_t^{'2}$. Repeat until $b_t = 0$. Go to step 5.

   If $\sigma_j^{'2} < \sigma_t^{'2}$ and $b_t = 0$, then go to step 5.

5. If $b_j > 0$, quantize the j^{th} dimension based on the number of bits assigned to it, b_j, using ALGORITHM 2.
6. Check if there are any other dimensions whose bits assignments have been changed during the step 3 and step 4. Re-quantize each of those affected dimensions independently using ALGORITHM 2.
7. If there is still new dimension coming, increment j and go to step 2, else wait.

Lemma 1 For any two dimensions s and t represented in VA$^+$-stream or VA$^+$-file, $4\sigma_s^{'2} > \sigma_t^{'2}$.

Proof. By contradiction. If there exists s and t, st. $\sigma_s^{'2} < \sigma_t^{'2}$ and also $4\sigma_s^{'2} < \sigma_t^{'2}$, this implies that s should not get its last bit in its last assignment. That will make t deserve that bit. This is contradictory to the current bit assignment. END

Lemma 2 The VA$^+$-file built by ALGORITHM 3b incrementally to reflect the impact of new dimension is the same as the VA$^+$-file built from the scratch for streaming database with new dimensions coming.

Proof. Assume at the beginning, we have a data set of n dimensions, and a VA$^+$-file is built for this n-dimensional data set. Then according to the algorithm for building VA$^+$-file, there must exist a sequence (1,2,...,n), s.t. $b_1 \geq b_2 \geq ... \geq b_n$ and $\sigma_1^2 \geq \sigma_2^2 ... \geq \sigma_n^2$. Here b_i is the number of bits assigned to dimension with sequence index i, σ_i^2 is the variance of each dimensional data. Let $d_i = \frac{\sigma_i^2}{4^{b_i}}$. Let s_i denote the index of the i^{th} dimension in the sequence.

Now assume the $(n+1)^{th}$ new dimension comes. Let its variance be σ^2_{n+1}. If we rebuild the whole VA$^+$-file from scratch, then will get a new sequence (2,3,...,n+1), s.t. $b_2 \geq b_3 \geq ... \geq b_{n+1}$, s.t. $\sigma^2_2 \geq \sigma^2_3 ... \geq \sigma^2_{n+1}$. Since the variance of each of dimension is a constant once the data is fixed, the order for the dimensions from dimension 2 to dimension n should stay unchanged though the assigned number of bit might have been changed. Hence, for the new dimension $n+1$, it should only be inserted into a proper place, denoted by s_{n+1}, at the ordered sequence of the last $n-1$ dimensions. Let j be the sequence index of $(n+1)^{th}$ dimension. Hence $b_{s_{n+1}}$ should satisfy the following conditions $\sigma_{j-1} > \sigma_{s_{n+1}} > \sigma_{j+1}$.

If we use VA$^+$-Stream approach, the same sequence will be produced since it compares the variance of new dimension $n+1$ against those of other remaining dimensions until we find a proper bit assignment, say $b_{s_{n+1}}$, s.t. $b_2 \geq b_3 \geq .. \geq b_{j-1} \geq b_{s(n+1)} \geq b_{j+1} ... \geq b_{n+1}$ and $\sigma^2_1 \geq \sigma^2_2 ... \geq \sigma^2_{j-1} \geq \sigma^2_{s_{n+1}} \geq \sigma^2_{j+1} ... \geq \sigma^2_n$.

Now we need to show the number of bits assigned to each dimension from both will be same. Since we already show there is one and only one order for all dimensions based on the variance of each dimension, $\sigma^2_1 \geq \sigma^2_2 ... \geq \sigma^2_n$, the number of assigned bits to each dimension b_i should make a sequence with the same order, $b_1 \geq b_2 \geq ... \geq b_n$. Now a unique order of all dimensions can be obtained based on the variance. Moreover, Lemma 1 shows that the current variance d_i between any two dimensions can not differ by a factor of more than four. Since $d_i = \frac{\sigma_i^2}{4^{b_i}}$, we will always assign the same number of bits to the dimension for a total of b bit quota for allocation. END

ALGORITHM 3a, 3b and 3c are all used to deal with the streaming data sets with fixed-size sliding, and suitable for processing QUERY 1 and QUERY 3. If there is an infinite window, i.e., all available dimensions are involved in queries, and the index size is kept the same, then some bits need to be extracted from those old dimensions for the new dimension. This is the case for QUERY 2. In this case a restructuring on all involved dimensions is needed. An efficient and effective bit reallocation algorithm should be investigated so that the restructuring work can be kept as least as possible while maximizing the accuracy. The following ALGORITHM 4 is the VA$^+$-Stream approach customized for dynamically building VA$^+$-file for processing QUERY 2.

ALGORITHM 4 VA$^+$-Stream:

1. Begin with the initial data set of k dimensions, build the original VA$^+$-file for this k-dimensional data set, where $b_i (1 \leq i \leq k)$ still represents the number of bits already assigned to the ith dimension, When new dimension data is coming, let it be $j = k+1$ and $b_j = 0$.
2. The following rules will apply.
   Case 1: $\sigma_j^{'2} > 4 \min_{n=1}^{n=j-1} \sigma_n^{'2}$
   Let $b_j = b_j + 1$, and $b_s = b_s - 1$ where $s = \min^{-1}{}_{n=1}^{n=j-1} \sigma_n^{'2}$. Also let

$\sigma_j^{'2} = \frac{\sigma_j^{'2}}{4}$, and $\sigma_s^{'2} = 4\sigma_s^{'2}$. Go back to step 2.

Case 2: $\sigma_j^{'2} \leq 4 \min_{n=1}^{n=j-1} \sigma_n^{'2}$
In this case, it means the new jth dimension doesn't matter too much in answering the query while all dimensions are concerned. Let $b_j = 0$. Go to step 3.

3. If $b_j > 0$, quantize the j^{th} dimension based on the number of bits assigned to it, b_j, using ALGORITHM 2.
4. Check if there are any other dimensions whose bits assignments have been changed during the step 2. Re-quantize each of those affected dimensions independently using ALGORITHM 2.
5. If there is still new dimension coming, increment j and go to step 2, else wait.

For QUERY 2 with an infinite window, a similarity search method has to consider all dimensions up to the current moment. It poses quite a challenge for using any tree-based index structures like R-tree since the dimensionality is too high for R-tree to perform better than simply sequential scan [21]. It also challenges the plain way of rebuilding VA-File or VA$^+$-file from scratch. For QUERY 1, this plain approach might get lucky when a small window size for the data set is specified by the user. But for QUERY 2, the query building time becomes intolerable if we rebuild the VA-file or VA$^+$ from scratch, as the performance study in Section 4 shows. However, all of the algorithms shown in this section have the flexibility to accommodate new dimensions without the need to rebuild either the VA-file or the VA$^+$-file from scratch, and by dynamically changing the VA-file or VA$^+$-file with only modifying a small portion of the index file, it can deliver much faster response to similarity queries. This is because the scalar quantization technique is dealing with each dimension independently while building the index structure. Hence, dimensionwise, the overall index structure can be easily extended without a need for an exhaustive restructuring work when new dimensions come.

3.4 Aging Data Stream

Another interesting query in a streaming database is the aged query [11]. When each dimension of the data set has been assigned a different role for evaluating the similarity, a weight value should be assigned to different dimensions. For example, in the context of network traffic monitoring, a trend-related analysis is made over data streams to identify some kind of access pattern, more emphasis is usually needed to put on the most recent traffic. It is quite reasonable to think the traffic in this week should be more important than the traffic in the last week since a trend analysis should focus more on current. Let $..., d_{-3}, d_{-2}, d_{-1}, d_0$ be a stream of network traffic data, where d_0 means today's data, d_{-1} means yesterday's data, and so on. A λ-aging data stream will take the following form:

$... + \lambda(1-\lambda)^3 d_{-3} + \lambda(1-\lambda)^2 d_{-2} + \lambda(1-\lambda) d_{-1} + \lambda d_0$.

Hence in λ-aging data stream, the weight of the data at a certain time position is decreasing exponentially with time. There are also other types of aging streams besides λ-aging data stream, for example, linear aging stream, where the recent data contributes to the data stream with linearly more weight than the older data. In order for the proposed VA$^+$Stream to be able to work for the aging stream database, we need to make the following modifications regarding the heuristic rule for allocating bits. If a weight g_i is specified for dimension i, the following heuristic rule should be used: if for any two dimensions i and $j, \exists\ k \geq 0$, st. $\sigma_i^2 g_i \geq 4^k \sigma_j^2 g_j$, then $b_i \geq b_j + k$ [10]. The rest of the algorithms remain same.

3.5 Other Discussions

In a streaming database, the number of streams changes too. Our approach can handle this scenario trivially too. For queries with a fixed-size sliding window, if the new data streams pop up, we will wait to collect the new ones until the size matches the sliding window. At the same time, we continue quantizing the new data for existing data streams incrementally. Then we quantize all new data streams collected so far, and add them to the VA$^+$file which we have maintained incrementally so far. On the other hand if some data streams drop out, we just simply delete the corresponding record in our VA$^+$file. For queries with infinite window size, if new data streams pop up, we can simply assume the old data for these new streams don't matter in order to adapt to the infinite window size, and let them be zero then. On the other hand, if new data streams drop out for queries with infinite window size, just delete their records from VA$^+$file. Actually when a large number of data streams come in or drop out , it will affect the existing bit allocation too. If this is the case, we can reallocate the bits from scratch and rebuild the VA$^+$file for once.

By showing the scalar quantization technique as a natural choice for handling streaming database, it reminds us of other techniques which might be good potential choices for indexing streaming database. As is known, R tree and its variant are not suitable for streaming databases. Because they are based on the spatial containment relationship between data points, once new dimension comes it might completely destroy this relationship. However, the grid file poses a d-dimensional orthogonal grid on the universe, resulting many cells with different sizes and shapes. Since when new dimensional data come, it will only need to further spit the cells to adapt new data without destroying the previous structure. This indicates it might be a good choice for indexing streaming database too.

4 Performance Evaluation

In this section, we evaluate the performance of the proposed techniques and compare them with other possible approaches for query processing. The design

of our experiments aims to show the advantage of the proposed approaches in terms of both query response time and index building time.

4.1 Data Sets

For the purpose of performance evaluation, we used several real-life and semi-synthetic data sets from different application domains. The techniques proposed in this paper are for high dimensional streaming data sets, therefore we chose our real-life data sets to have high dimensions and streaming nature. The first data set, *Stock Time-series (Stockdata)*, is a time-series data set which contains 360-day stock price movements of 6,500 companies, i.e., 6,500 data points with dimensionality 360. The second data set *Highly Variant Stockdata (HighVariant)* is a semi-synthetic data set based on *Stockdata*, and is of size 6,500 with 360-dimensional vectors too. The generation of the second data set aims at obtaining a data set with high variance across dimensions. In order to do that, we performed KLT transform on the original *Stockdata*, and as a result the transformed data set tends to have larger variance at the beginning dimensions, and smaller variance at the ending dimensions. Then we randomly re-order the dimensions inside the data set so that the dimensional data with a certain level of variance can be uniformly distributed. The third data set is *Satellite Image Data Set*, which has 270,000 60-dimensional vectors representing texture feature vectors of satellite images. This data set provides challenging problems in high dimensional indexing and is widely used in high dimensional indexing and similarity searching research [19, 12, 5]. In the context of streaming database, we adapt all the above data sets for our purpose by having the data pretend to come into database one dimension after another dimension. For example, in the case of *Stockdata*, each dimension corresponds to daily stock prices of all companies.

When not stated otherwise, VA+Stream in the following experimental study refers to the implementation version of Algorithm 3b and an average of 3 bits per dimension is used to generate the vector approximation file. When not stated otherwise, k is set to be 10 for k-NN queries through our whole experimental study.

4.2 Experiments on Query Performance

As Lemma 2 states that the vector approximation file built by Algorithm 3b is the same as the one built from scratch using VA$^+$-file approach, the first thing we want to show is that the generated vector approximation file can support efficient continuous queries in streaming database. We will show the advantage of our approach over sequential scan. The reason we chose the sequential scan as the yardstick here is because of the infeasibility of well-known techniques for stream data and the well-known dimensionality curse, which make other choices like R-tree and its variants out of the question for efficient k nearest neighbor search. However, while compared with R-tree and its variants, sequential scan can perform much better in high-dimensional space due to its sequential nature of I/O requests.

A group of experiments was first set up to evaluate the performance of the vector approximation file generated by our approach for a snapshot of streaming databases. Two metrics were used to evaluate how the generated vector approximation file can support k-NN queries: vector selectivity and page ratio. Vector selectivity was used to measure how many vectors have been actually visited in order to find the k nearest neighbors of the query. Since vectors actually share pages, the vector selectivity does not exactly reflect the paging activity and query response during the similarity search. Hence, page ratio was adopted to measure the number of pages visited as a percentage of the number of pages necessary for sequential scan algorithm.

In our experiment, vector selectivity was measured as a function of average number of bits per dimension, which actually reflects the quota for the total number of available bits b. Figure 3 (left) shows the results for 360-dimensional *Stockdata*. When the quota for bit allocation is increasing, the vector selectivity, e.g. the percentage of actual visited vectors, is quickly decreasing. The pruning rate of the generated vector approximation file during the first phase of similarity search is also shown as dotted line in this figure, and it is getting higher when the bit quota is increasing. For example, when the average number of bits per dimension is 4, the vector selectivity of the approximation file is only 8%. In contrast, the vector selectivity is always 100% for sequential scan since it needs to visit all the vectors in order to find the k nearest neighbors.

Page ratio was measured as a function of average number of bits per dimension too. Figure 3 (right) shows that the number of visited pages is decreasing quickly in vector approximation file when the number of available bits for allocation is increasing. When the average number of bits per dimension is 4.5, the number of visited pages in vector approximation file is only around 10% of that in sequential scan. Even when we consider the fact that sequential scan will not invoke any random access which might actually contribute to a factor of 5 for performance improvement [21], the vector approximation file approach still shows advantage.

To show the impact of window size on the query performance, we also varied the window size in *Stockdata*. Figure 4 shows the results.

To show the impact of data points on the query performance, we also varied the number of data points in *Satellite Image Data Set*. Figure 5 shows the results.

4.3 Experiments on Index Building

Previous section has demonstrated that the vector approximation file generated by the proposed approach in this paper can support efficient continuous queries in streaming database. We now want to show that the proposed approach can also support dynamic data streams in terms of building such an vector approximation file as an efficient index structure. A group of experiments was set up for this purpose. We compared the time of using VA$^+$-Stream to build an index structure incrementally against using VA$^+$-file technique to build an index structure from scratch each time new data element is streaming into the database. The reason we chose VA$^+$-file as the yardstick here is because of the following

two reasons. First, it is shown that both VA$^+$-file and VA$^+$-Stream can generate the vector approximation file which supports efficient k nearest neighbor search in high-dimensional space, while R-tree and its variants are suffering from the dimensionality curse and can not be used in streaming database. Secondly, to the best of our knowledge, there exists no specific techniques for targeting efficient k nearest neighbor search in streaming database, hence the possible approaches that can be used here for comparison purposes are just direct applications of existing efficient techniques for traditional high-dimensional databases to streaming databases.

The first experiment was set up for ALGORITHM 3b and 3c, which target QUERY 1 with a fixed window size. To show the impact of window size on the performance of the approaches under test, the window sizes were chosen to be 60, 120, 180, 240, and 300 for *Stockdata* and *HighVariant*, respectively; the window sizes were set to be 30, 35, 40, 45, and 50 for *Satellite Image Data Set*. The experiment were set up to process k-NN queries for the streaming database at any time positions after new dimension comes.

The following two types of metrics were used here for performance evaluation: *average index building time* and *average query response time*.

In order to get the average index building time and the average query response time, for each different window size we chose, we processed 20 10-NN queries at each time position after the new dimension arrived. We recorded the index building time and the average query response time over 20 queries at each time position. For practical reasons, we actually sampled 10 continuous time positions for each different window size, and then computed the average index building time over 10 and the average query response time over 200 queries. Since for QUERY 1 and QUERY 2, the query points usually come from the database, the 20 10-NN queries issued at each time position in our experimental study were from the streaming data sets. These 20 queries were randomly chosen from the data set. For testing QUERY 1, *Stockdata*, *HighVariant* and *Satellite Image Data Set* are used. Since QUERY 3 can be treated as a special case of QUERY 1, no separate test was done for it. The algorithm to build the VA$^+$-file from scratch was used as one of the benchmarks with the same setup for the purpose of comparison.

At each sampled time position or dimension, we restructured the VA$^+$-file incrementally using ALGORITHM 3b. For a specific test instance of a certain window size, the total number of bits allocated to the feature vectors were always kept same. We also rebuilt a VA$^+$-file from scratch to make the comparison. We performed queries by asking 10-NN of 20 data points in the data set. The same k-NN search algorithm discussed before was applied to both techniques. The performance of the methods was evaluated in terms of their average index building time and also the average query response time. Since the same search algorithm was used to search the resulting VA$^+$-file, the query response time only depended on the structure of VA$^+$-file. The search algorithm consists of a first-phase elimination, and a second-phase checkup. For all cases, the stated results are mean values.

Figure 6 (left) compares the index building time of two methods, for stock time series data. The index building time does not vary too much for VA$^+$-Stream technique, since it is an incremental method and works almost on only one dimension. But the index building time for completely rebuilt VA$^+$ method is skyrocketing with the window size increasing. This is because when we build a VA$^+$-file from scratch, the amount of efforts is proportional to the number of dimensions or the window size we want to consider for the data set. When the window size is 300, VA$^+$Stream achieves a speedup of around 270. It is not surprising that Figure 6 (right) shows the average query response time of both methods is basically the same for stock time series data since the VA$^+$-files generated by them are actually the same. With the window size increasing, the query response time is getting longer.

Similarly, Figure 7 shows the comparison of the index building time, for highly variant stock time series data and satellite image data set. For highly variant stock time series, VA$^+$Stream achieves a speedup of 100 when the window size is 300. A speedup of around 28 is achieved when the window size is 50 for satellite image data set. It is also observed that the speedup of VA$^+$Stream is increasing with bigger window size for all three data sets.

We ran the test to compare the different implementations of VA$^+$-Stream. Figure 8 (left) shows the comparison of indexing building time of Algorithm 3b and Algorithm 3c for implementing VA+Stream.

An experiment was also set up for testing ALGORITHM 4 over QUERY 2, which has an infinite window and needs to consider all the dimensions up to now. *Stockdata* was used here. For the purpose of testing the performance of algorithms, the experiment was set with an initial data set of 300 dimensions, i.e., we took the first 300 dimensions from *Stockdata* as our initial data set. Therefore each time new dimension comes, the proposed VA$^+$-Stream and the traditional VA$^+$ were run separately to build the index structure. Similarly, 20 10-NN queries were issued at each time position after new dimensions arrived. The index building time and the query response time were then recorded. Figure 8 (right) shows the comparison between VA$^+$-file and VA$^+$-Stream for processing QUERY 2. It is obvious there exists a huge difference between these two methods while the index building time is concerned. Especially, with the number of dimensions increasing in the data set, the index building time for traditional VA$^+$-files tends to increase a little bit, but the index building time for VA$^+$-Stream almost remains same since it is only dealing with one new dimension.

5 Conclusions

In this paper, we presented a scheme to efficiently build an index structure for streaming database, called VA-Stream and VA$^+$-Stream. We motivated the need for an effective index structure in streaming database. Our performance evaluation establishes that the proposed techniques can be in fact used to build the index structure for streaming database in a much shorter time than other avail-

able approaches, especially when the number of dimensions under consideration for building index structure is large.

To the best of our knowledge, the proposed technique is the first solution for building an index structure on multiple data streams. Our technique can work both as an update-efficient index and as a dynamic summary on stream data.

Although multi-dimensional index structure on multiple streams can significantly improve the performance of queries, dynamic nature of dimensions would cause a significant restructuring on a tree-based index such as an R-tree. Neither an efficient method to restructure the tree nor whether the effort is worthwhile has been studied yet. The problem can be possibly attacked by a dual-based approach, e.g., a dual R-tree which is built on dimensions. Hence, when new dimension comes, we only need to consider an insertion problem instead of totally rebuilding the index tree. More investigation is needed to develop a more effective index for dynamic dimensionality.

References

1. Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-First ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, Madison, Wisconsin, June 4–6 2002.
2. Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 30:109–120, September 2001.
3. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R* tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, May 23-25 1990.
4. Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *Proceedings of 28th VLDB Conference*, Hongkong, China, August 2002.
5. P. Ciaccia and M. Patella. PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Proc. Int. Conf. Data Engineering*, pages 244–255, San Diego, California, March 2000.
6. H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Vector approximation based indexing for non-uniform high dimensional data sets. In *Proceedings of the 9th ACM Int. Conf. on Information and Knowledge Management*, pages 202–209, McLean, Virginia, November 2000.
7. Like Gao and X. Sean Wang. Continually evaluating similarity-based pattern queries on a streaming time series. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Madison, Wisconsin, June 2002.
8. Like Gao and X. Sean Wang. Improving the performance of continuous queries on fast data streams: Time series case. In *SIGMOD/DMKD Workshop*, Madison, Wisconsin, June 2002.
9. Like Gao, Zhengrong Yao, and X. Sean Wang. Evaluating continuous nearest neighbor queries for streaming time series via pre-fetching. In *Proc. Conf. on Information and Knowledge Management*, McLean, Virginia, November 4-9 2002.
10. A. Gersho. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Boston, MA, 1992.
11. Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the 27th VLDB Conference*, Rome, Italy, September 2001.

12. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 518–529, Edinburgh, Scotland, UK, September 1999.
13. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.
14. D.V. Kalashnikov, S. Prabhakar, W. G. Aref, and S. E. Hambrusch. Efficient evaluation of continuous range queries on moving objects. In *DEXA 2002, Proc. of the 13th International Conference and Workshop on Database and Expert Systems Applications*, Aix en Provence, France, September 2–6 2002.
15. H. Karhunen. Uber lineare methoden in der wahrscheinlich-keitsrechnung. *Ann. Acad. Science Fenn*, 1947.
16. Y. Linde, A. Buzo, and R. M. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications*, 28:84–95, January 1980.
17. S. P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:127–135, March 1982.
18. J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Math. Stat. and Prob*, volume 1, pages 281–196, 1967.
19. B. S. Manjunath. Airphoto dataset. http://vivaldi.ece.ucsb.edu/Manjunath/research.htm, May 2000.
20. Traderbot. http://www.traderbot.com.
21. R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 194–205, New York City, New York, August 1998.

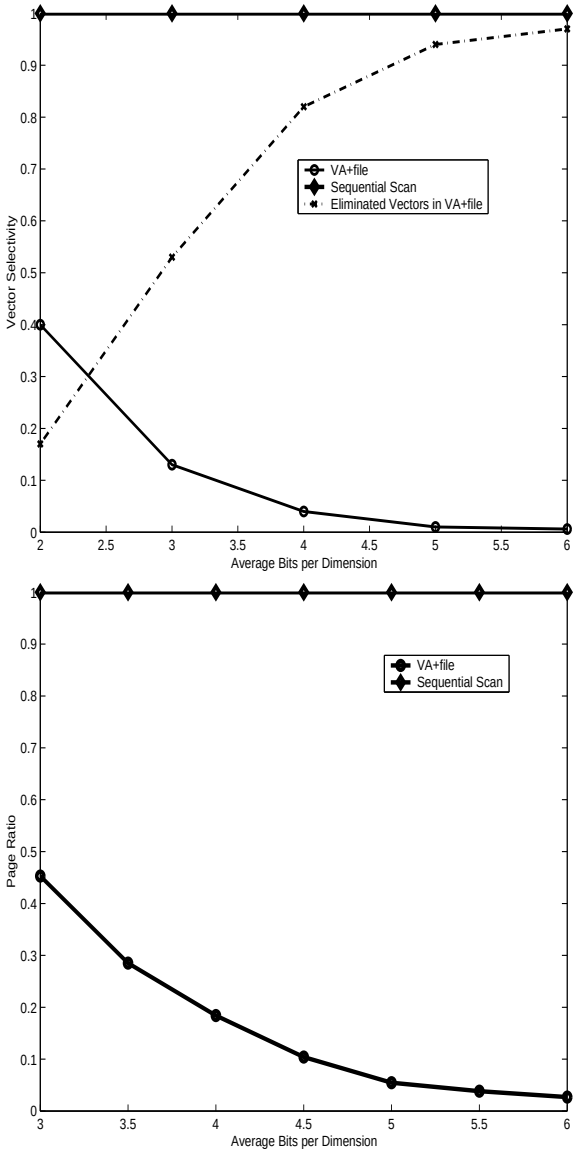


Fig. 3. Vector Selectivity (left) and Page Ratio (right) versus Number of Available Bits in *Stockdata*

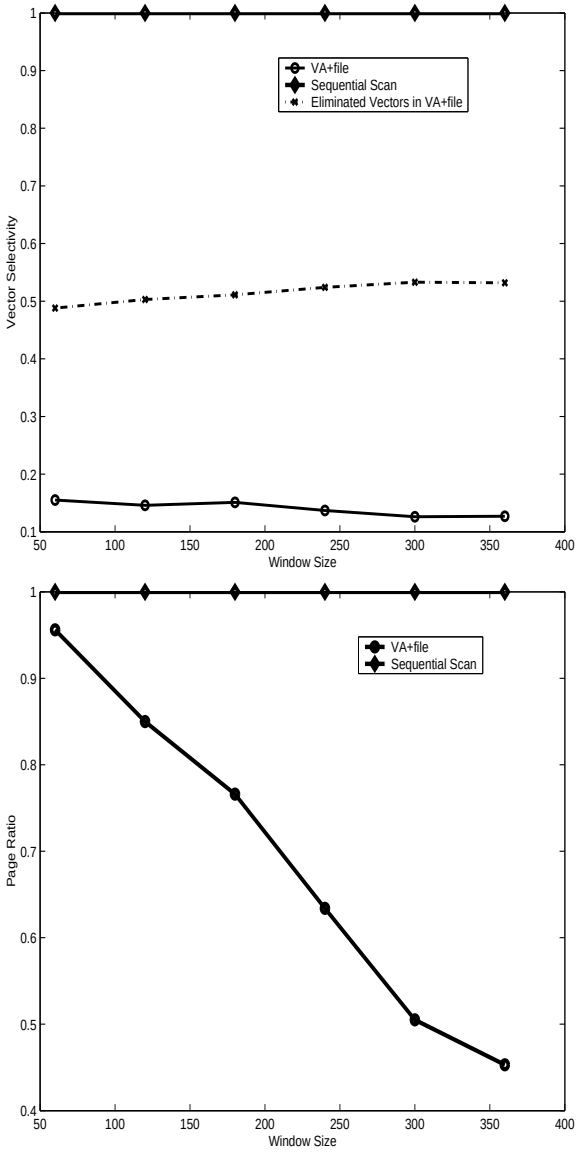


Fig. 4. Vector Selectivity (left) and Page Ratio (right) versus Window Size in *Stockdata*

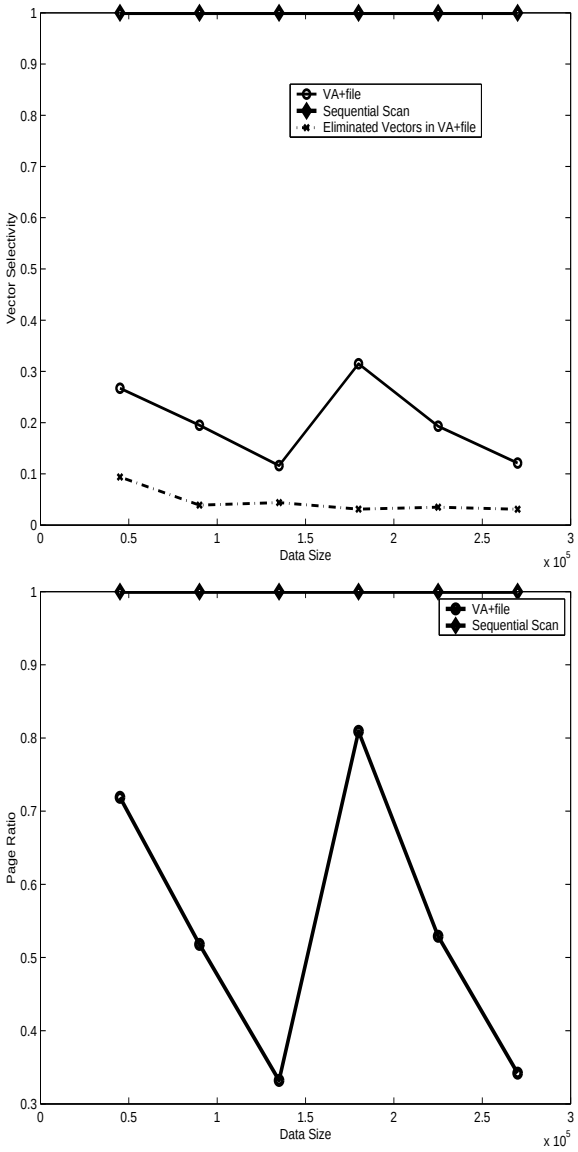


Fig. 5. Vector Selectivity (left) and Page Ratio (right) versus Data Size in *Satellite Image Data Set*

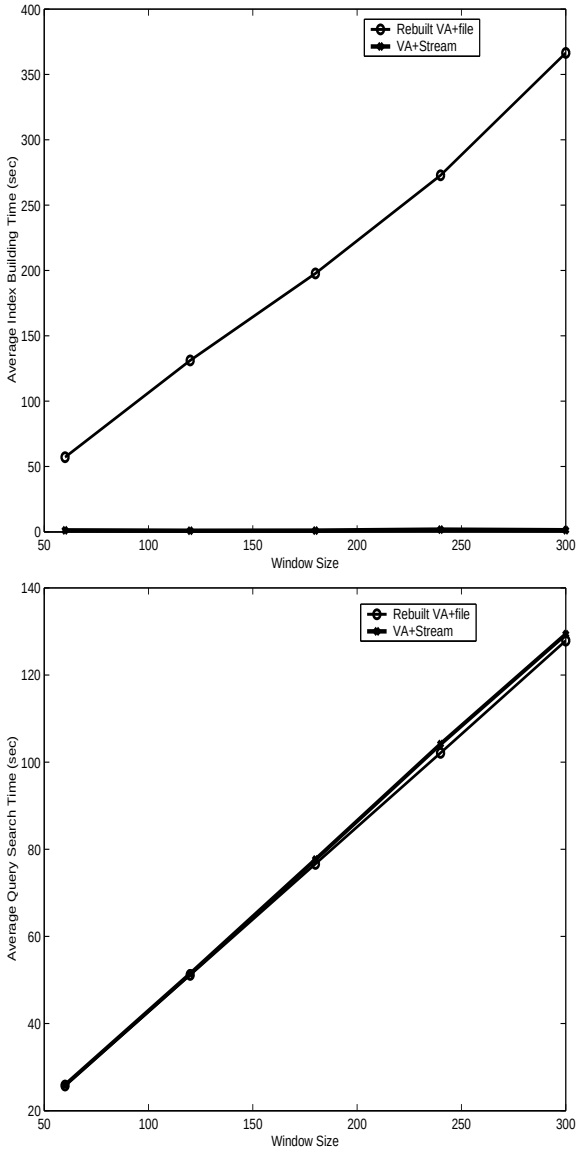


Fig. 6. Comparison of Index Building Time (left) and Query Response Time (right) between VA$^+$-file and VA$^+$-Stream for QUERY 1 (i.e. sliding windows) in *Stockdata*

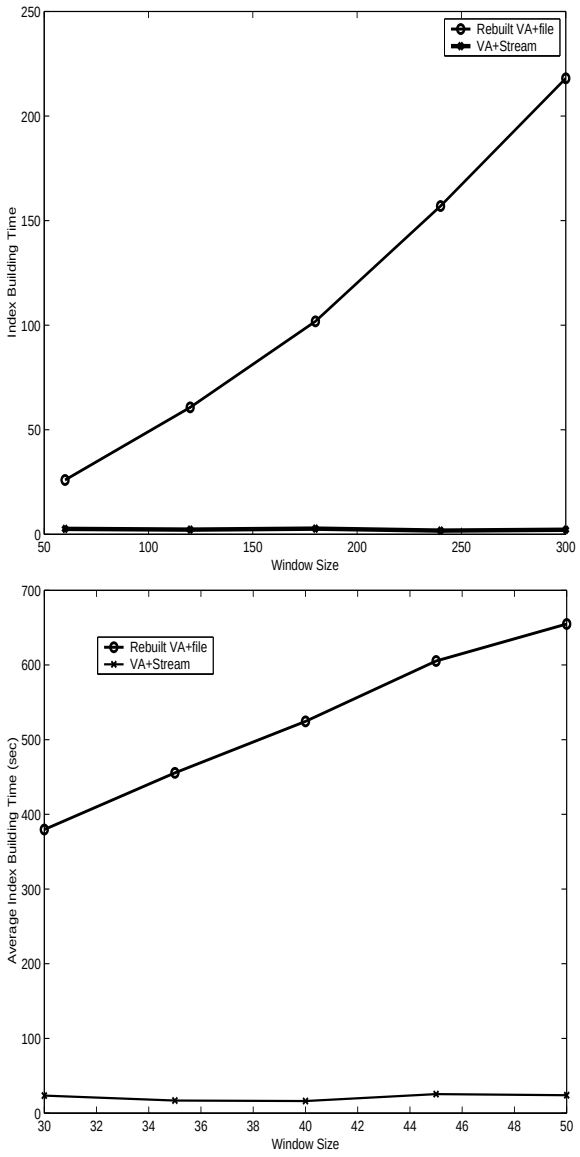


Fig. 7. Comparison of Index Building Time between VA$^+$-file and VA$^+$-Stream for QUERY 1 (i.e. sliding windows) in *HighVariant* (left) and Satellite Image Data Set (right)

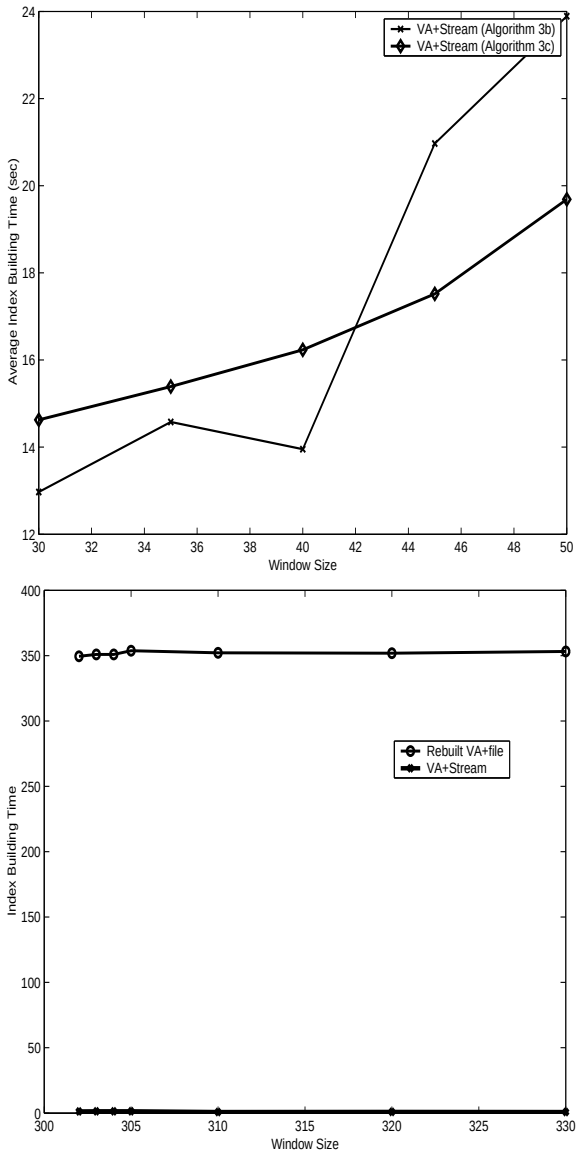


Fig. 8. Comparison of VA+Stream (Algorithm 3b) and VA+Stream (Algorithm 3c) in Satellite Image Data Set (left) and Comparison of Index Building Time between VA$^+$-file and VA$^+$-Stream for QUERY 2 (i.e. no sliding window) in *Stockdata*