Technical Report: Answering Queries on Streaming Data Series

Xiaoyan Liu
Ohio State University, Department of Computer and Information Science
liuxia@cis.ohio-state.edu

Hakan Ferhatosmanoğlu
Ohio State University, Department of Computer and Information Science
hakan@cis.ohio-state.edu

ABSTRACT

Data streams are common in many recent applications, e.g. stock quotes, e-commerce data, system logs, sensor networks, network traffic management, etc. A wide range of query scenarios emerge naturally in these applications. Compared with traditional databases, streaming databases pose new challenges for query processing due to both the streaming nature of data which constantly changes over time and the wider range of queries proposed by the user. In the past literature, most of the research has been focused on designing a new technique for a specific query scenario. In order to adapt to different query scenarios and handle them efficiently, we take the first step in this direction. 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 first define various classes of queries useful for different scenarios in streaming applications. we propose efficient index structures and algorithms for various models of k nearest neighbor (k-NN) queries on multiple data streams, and extend them to handle all kinds of queries including range queries, aggregate queries and so on. 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 in terms of both approximate search and exact search. The proposed techniques are update-efficient and dynamic adaptations of VA-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, 9, 2, 11, 17, 4, 10]. 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 [9, 11, 10], 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 databases instead of static databases 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 multi-dimensional data sets [16, 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 propose an index structure which can accommodate the changing dimensionality of data objects. Since scalar quantization is performed on each dimension independently, the access structure based on such an idea is 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 six 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 is given. We then motivate the need for an adapted version of VA-file and VA$^+$-file, which is specifically used to tackle the problems imposed by dynamic streaming data. At last, the proposed techniques, called VA-Stream and VA$^+$-Stream, are presented in this section. In Section 5, an extensive performance evaluation of proposed techniques is given which shows that significant improvements are achieved by the proposed technique. Fi-

nally, Section 6 concludes the paper with a discussion.

2. STREAMS AND QUERIES

In this section, we start with analyzing all kinds of interesting queries in streaming databases. We then introduce a general stream processing model. Finally we discuss several important scenarios in streaming databases followed by precise definitions of their related queries.

2.1 Streams

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. In the context of databases, we would like to think each stream as a multi-dimensional data object, though the dimensionality is changing. Each sequentially arriving value for every stream is treated as a one-dimensional value for the streaming data object. Hence, the streaming database can be matched to a multi-dimensional data space with changing dimensionality. See Table 1 for notations used to illustrate data streams in this paper.

The data streams are continuous, and equivalently 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. The nature of data streams leaves us with a challenge of trying to get accurate query results from a huge database with time constraints. Moreover, in most cases the users would expect fast response time for queries. If exact results are desired for the queries, it implicitly requires unbounded disk space, and even unbounded memory for faster response, which are not realistic. Hence approximate answers make more sense in most cases, though exact answers are meaningful in certain queries, eg. sliding window queries, which are discussed in Section 2.2.

2.2 Queries in Streaming Databases

To support efficient query processing in streaming databases, it is also necessary to develop an effective index structure for streaming databases with very efficient update costs, so that query results can be obtained in a tolerable amount of time. Before we design such an index structure to expedite the query response time in streaming databases, we first need to examine all possible queries of the practical importance for databases of multiple continuous data streams, and then classify them.

On the top level of our classification scheme, the queries can be issued continuously or only once [1]. Continuous queries are the new type of queries proposed for streaming databases. If the database is static, it is meaningless to issue the same query again and again. For static or traditional databases, one-time queries are usually the case. One-time queries can also occur in streaming databases, though it is of less significance.

On the second level, the queries can be further divided into two groups: predefined queries and online queries. Predefined queries are defined in advance regardless the status of the streaming databases. Online queries are issued by the users on the fly, and no knowledge about the queries can be known by the query processor in advance. Hence, at least predefined queries can be answered in the same way as the online queries. Moreover, the advance knowledge about the predefined queries implies that there possibly exists a more efficient way of dealing with them.

On the third level, the queries can be classified based on the weights put on the past history records of data streams by the user. This level still affects the underlying structure,e.g. the synopsis of data streams, used to support the queries. As we will show in Section 3.4, the weights of different queries on this level affects how the synopsis is constructed and updated. Several most important queries concerned here include: sliding-window queries, timestamp queries, and aged queries. For sliding-window queries, the user is only interested in the data stream segment within a specified sliding window size up to the current time point. The weight is constant 0 for all history data older than one sliding window period, and constant 1 for all history data within the sliding window. For timestamp queries, the user is only interested in the data stream segment after a certain timestamp up to the current time point. The weight is constant 0 for the data before the timestamp and constant 1 after the timestamp. For example, a query involves all the history data

Notation	Meaning
x,y	streaming data objects in database
$x[i,j]$	slices of x between time position i and j
$x[t,i]$	slices of x from timestamp t 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 streams 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 i-th dimension data
g_i	the weight of dimension i in an aged query

Table 1: Frequently used notations.

up to now, and this query is a special case of timestamp queries. The timestamp for this query is the time when the data first came into being. Actually this query can also be treated as a special case of sliding window queries, where the size of sliding window is infinite. Another interesting type of query in a streaming database is the aged query [14], which is of practical significance too. When the history data of the data have been assigned a different role for evaluating the similarity, a different weight value should be assigned to the data at different time points. 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 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 the recent events. 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 the λ-aging data stream, the weight of the data at a certain time position is decreasing exponentially with time. The study of this exponential smoothing effect is of practical interest. Depending on the applications, there are also other types of aging streams besides λ-aging data streams, for example, linear aging streams, where the recent data contributes to the data stream with linearly more weight than the older data.

On the fourth level, the queries can be classified based on what the users are interested in getting from streaming databases. This level does not affect the underlying structure of data streams used to support the queries. Only different search processes based on the underlying structure are needed to answer different categories of queries on this level. The query processing is discussed in Section 4. There are many different queries on this level, which are more than what we can list here. One popular example is k nearest neighbor (k-NN) queries. Besides k-NN queries, there are many other different types of queries which are of particular interest to people and often used on multiple data streams. Using stock time series[24] as an example, we summarize and define them as follows:

- Range queries: each data stream is evaluated so that its data values always fall into a certain range over a sliding window. One example in stock data streams would be "Find all the stocks between \$20 and \$200 over the last 20 minutes", "Find all the stocks within 2% of the 52-week high"
- Similarity joins: every pair of streams are evaluated

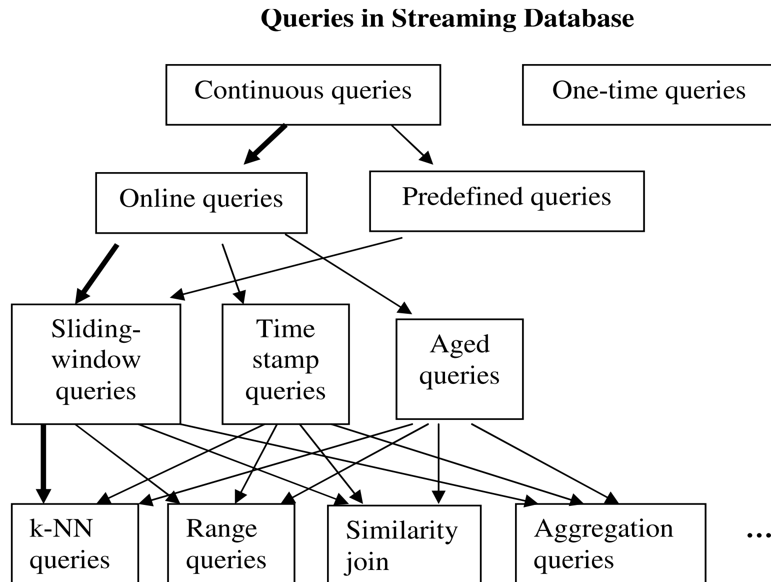


Figure 1: Query Classification

for their similarity over a sliding window, and the pairs most similar to each other are selected. For example, a similarity join query in stock data streams would be "Find all the stock pairs whose stock prices have a similarity of ϵ in the last week".

- Aggregation queries: such a query is based on the aggregation value of each data stream, such as the moving average within a sliding window. For example, aggregation queries in stock data streams would be "Find all the stocks with the average price over $100 in the last month", "Find all stocks whose average price reaches the history record in the last month", "Find all stocks which have been above their 200-day moving average" and so on.

- Momentum queries: this type of queries is used to examine the trend of the data streams. For example, "Find all stocks that have been moving up in the 20 minutes".

- Volatility queries: such a query is based on the variance of each data stream over a sliding window. For example, one variance query in the case of stock data streams would be "Find the stock which has the least fluctuation in the last month", or "Find the stock whose spread between the high tick and the low tick over the past 30 minutes is greater than 3% of the last price.

- Advanced queries: one more advanced form of queries over data streams can be obtained by combining the above query types. For example, "Find all the stocks between $20 and $200 whose monthly average has moved down 20% in the last 6 months" is a combination of the range query and the aggregation query; "Find all stocks whose monthly average has been always increasing in the last year" is a combination of the momentum query and the aggregation query.

Please see Figure 1 for the summary of the classification. The bold arrows indicates the most popular queries in streaming database. One-time queries are not the focus of this paper.

2.3 Stream Processing Model

Figure 2 shows our general architecture for processing continuous queries over streaming databases. 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. At

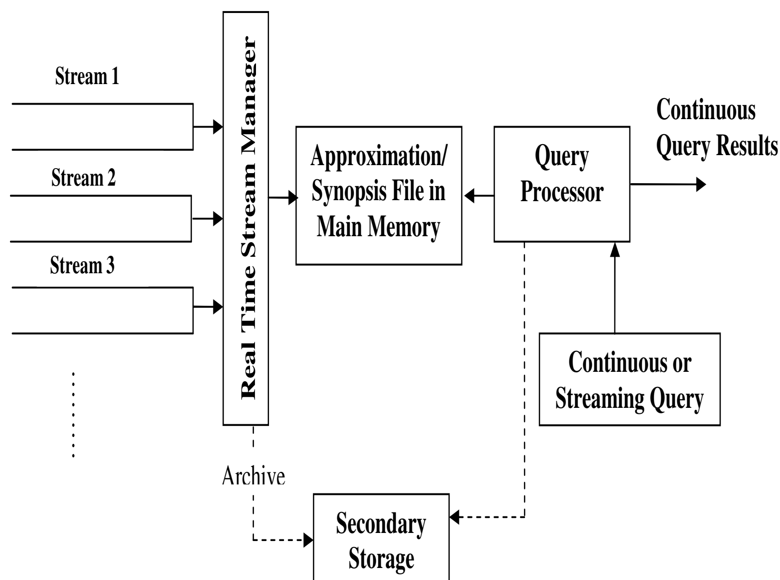


Figure 2: Stream Processing Model

this time, approximate answers to the queries can be obtained by only scanning the approximation file. If the user wants exact results to the query, real data, if archived in the secondary storage, need to be accessed. Before going to the secondary storage, the processor will further eliminate as many disqualified candidates as possible first from the whole pool of data streams based on approximation. 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 exact answers to the queries. The processor can read each candidate from the archived data in the secondary storage. Since the disk space is theoretically limited, exact answers seem to be more meaningful for sliding window queries. We assume the archiving process will automatically discard the data older than a certain period and add new data in. Hence all data required for obtaining exact answers can fit into limited disk space. The key element in our processing model is the Real Time Stream Manager. It should be able to generate a synopsis which can give approximate results of high quality to the queries, and can also be used to prune the data efficiently, so that the secondary storage access time will be greatly minimized if accurate answers are desired, e.g. for sliding window queries. Moreover, the 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 the Real Time Stream Manager in an efficient stream processing model.

We also look at continuous queries with a sliding window, continuous timestamp queries, and continuous aged queries. For those 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. Exact answers are obtainable by accessing the limited amount of archived data. However, for timestamp queries, the situations are different. Take the queries with an infinite window as one extreme example. In this case, only approximate results are meaningful. Though it can be solved in a way similar to those with the fixed-size window, the approximation accuracy will get sacrificed if the size of the synopsis stays the same. A more practical query type for non-sliding window query would be the aged query where the data streams are aging. That's actually what usually happens in real world, where people put more emphasis on more recent activities. If the streams are aging at a very quick rate, e.g.

exponential, near-exact answers to the queries are meaningful since the query processor can access secondary storage where only those not-too-aged data are archived.

As always, it is about the tradeoff between the quality of the results and the response time. If the quality of the query results is not the main concern for the users, the query processor only needs to look at the synopsis and generate approximate results. As a result, the query response time would be more satisfying for the users. If the user is not concerned with the query response time, and accurate results are desirable, in our stream processing model the query processor can further access the secondary storage to generate results of higher quality.

2.4 Definitions of Queries of Practical Significance

In this section, we are defining some queries of practical significance in streaming databases on the basis of our classification structure. Traderbot [1, 24], a web-site which performs various queries on streaming stock price series, offers a variety of queries similar to the models discussed here.

2.4.1 *Nearest Neighbor (NN) Queries*

NN Online Sliding Window Query: Given a streaming database D, let q be the streaming query object issued at time position t, the nearest neighbor of q for the last T time period would be r if $dis(q[t-T,t], r[t-T,t]) \leq dis(q[t-T,t], s[t-T,t]),\ \forall s \in D$, where T is the size of the sliding window.

This 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, hourly, or even more frequently to include the newer quotes. In this case a query may be posed continuously to find the most similar company to a given company in terms of the stock price in the last two months. In our classification, this is a continuous online k-NN query with a sliding window when $k = 1$.

NN Online Timestamp Query: Given a streaming database D, let q be the streaming query at time position t, and the timestamp be t_0, then the nearest neighbor query of q would be r if $dis(q[t_0,t], r[t_0,t]) \leq dis(q[t_0,t], s[t_0,t]),\ \forall s \in D$.

We are not interested in only the past data which are collected within a certain amount of time period, but in all of the data collected from a certain timestamp up to now. For example, we want to find the company most similar to a given company in terms of all the history stock prices, where the timestamp might be the first time when these stock prices were recorded in the database or a historical moment for the given stock. This scenario often occurs when a long-term plan or strategy is concerned, and belongs to continuous online timestamped k-NN queries.

NN Predefined Sliding Window Query: 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 $dis(q, r[t-len(q),t]) \leq dis(q, s[t-len(q),t]),\ \forall s \in D$.

In the category of online queries, the query objects are usually streaming too, and come from the streaming databases. But for predefined queries, the query objects are often given with a fixed length. For example, we have an interesting stock pattern in the bear market, and a regular check against the streaming database would find us those companies whose stock price fluctuations match this pattern most. This is also a scenario of practical importance, and belongs to continuous predefined k-NN queries with a sliding window.

All the above definitions can be easily generalized for k-NN queries. All these discussed queries are usually characterized by their continuity, i.e., 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.

3. SCALAR QUANTIZATION TECHNIQUES

3.1 Limitations of Existing Techniques

To answer all kinds of continuous queries classified in Figure 2.2, especially those with practical significance defined in Section 2.4, 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 repeatedly 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.

There are also a large body of literatures which addressed some specific scenarios in data streams by specific algorithms. In [9, 11, 10], they considered a static time series database with an incoming data stream as the query object. The concerned query in their paper is actually a continuous online timestamp query in our classification except that their database is static instead of streaming. They showed that the search performance for k-NN queries were improved by their methods. However, the time series database considered in these three papers is static, and this greatly limits the application of their methods to more general scenarios with databases formed by streaming time series.

Some research work has also been done on approximate query answering in streaming databases by using data summarization technique. Most of them are designed for answering aggregate queries [14, 6, 12]. But similarity search in streaming databases is an area rarely touched upon.

More literature review will be added here to show the limitations of current methods.

Hence, efficient index structures are needed to assist in answering different queries by accommodating 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 databases. 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 databases.

In this paper, we propose scalar quantization based indexing techniques 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 Online and Predefined Sliding Window Query, and Online Timestamp Query will be discussed in Section 3.4.

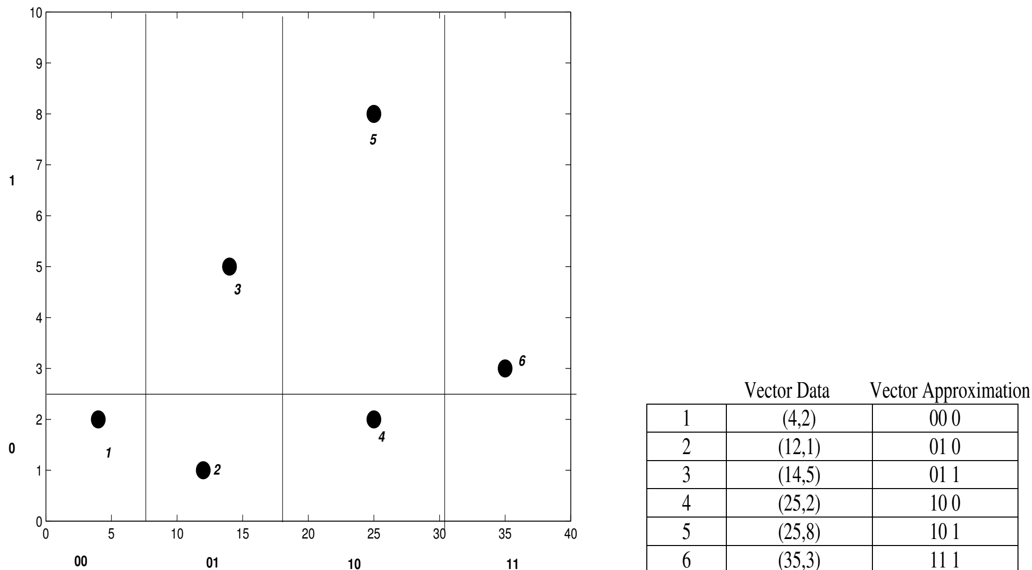
Scalar quantization technique is a way to quantize each dimension independently so that a summary of the database is efficiently captured. It also serves as an index structure for efficient point, range, and k nearest neighbor queries. By observing that there is no cross-interference among dimensions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

3.2 Indexing Based on Vector Approximation

We will first briefly review a scalar quantization technique used in traditional database, Vector-Approximation file (VA-file) [26], to introduce some background knowledge.

Since conventional partitioning index methods, e.g. R-trees and their variants, suffer from dimensionality curse, VA-file was proposed as an approach to overcome the curse and support efficient similarity search in high-dimensional spaces. The VA-file is actually a filter approach based on synopsis files. It divides the data space into 2^b rectangular cells where b is the total number of bits specified by the user [26]. 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 3 shows an example of one VA-file for a two-dimensional space when $b = 3$.

As an example of all kinds of queries, the nearest neighbor queries are discussed in details here regarding how the



	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

Figure 3: A two-dimensional example for VA-file

VA-file efficiently supports them. The exact nearest neighbor search in a VA-file has two major phases [26]. 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 point are computed. The real distance can not be smaller than the distance between the query point 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 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 exact nearest neighbor search.

For the approximate nearest neighbor search, only the first phase of the exact search will be enough to serve its purpose. The simplest way is to use the lower bound to approximate the real value [25], and find the vector with the smallest lower bound value as the nearest neighbor. The quality of the approximate search depends on the approximation file itself and also the measure used to approximate the real values. A detailed discussion of approximate search will be made in Section 5.2.1.

3.3 VA$^+$-file for Non-Uniform Data Sets

In the VA-file approach, although there is always an option of the non-uniform bit allocation among dimensions, no specific algorithms for that option were proposed in [26]. Moreover, each dimension i is divided into 2^{b_i} intervals of either equal sizes, or equal populations, which are the two

simplest partitionings that only suit a uniformly distributed data set. Some problems might occur with the efficiency of VA-file if the data set is not uniformly distributed, especially when it is highly correlated or clustered. VA^+-file [7] is proposed to target non-uniform data sets, and can lead to more efficient searching.

The partitioning performed by the VA-file approach can be viewed as a simple scalar quantization, except that the approach does not care about the representative values for the cells. In a more refined sense, a scalar quantizer [13] 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. It is shown in [7] that a scalar quantization designed by directly aiming for the least possible reproduction error (i.e. the least average Euclidean distance between data points and their representatives) 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 [21, 20], 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) [18] domain, so that the data can be successfully decorrelated. The goal in VA^+-file is to approximate a given data set with the minimum number of bits but the maximum accuracy. Therefore, it is crucial in VA^+-file to analyze the dimensions and then 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 theories, a well-known rule is: if for any two dimensions i and $j, \exists\ k \geq 0$, st. $\sigma_i^2 \geq 4^k \sigma_j^2$, then $b_i \geq b_j + k$ [13] and it is shown to be a good heuristic for allocating bits based on the significance of each dimension [7]. 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 [22].

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 ith 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 values for the intervals should be carefully made in order to avoid getting stuck in the local optima. Equally-populated intervals will be a good choice.

3.4 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. As we have mentioned, the classification of sliding window queries, timestamp queries, and aged queries is based on the weight values assigned to the data at each time point. Hence, the significance of each dimensional data should be weighted by its corresponding weight value, implying a different bit allocation algorithm is needed in each case to build the index structure in support of efficient query processing. We will show in this section our technique can adapt to all three queries.

3.4.1 Sliding Window Queries

With the use of scalar quantization techniques, 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. 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 sliding 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 both online and predefined sliding window queries, which have 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 suitable for data sets with uniform-like distribution. 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 underlying 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 and optimally with its assigned bits by achieving 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 $b_i \geq b_j + k$ [13]. 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 the new dimension with all the 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 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 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 jth 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 as 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 jth 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, s.t. $\sigma_s^{'2} < \sigma_t^{'2}$ 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 ith dimension in the sequence.

Now assume the $(n+1)^{th}$ new dimension comes. Let its variance be σ_{n+1}^2. 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. $\sigma_2^2 \geq \sigma_3^2 ... \geq \sigma_{n+1}^2$. 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 $\sigma_1^2 \geq \sigma_2^2 ... \geq \sigma_{j-1}^2 \geq \sigma_{s_{n+1}}^2 \geq \sigma_{j+1}^2 ... \geq \sigma_n^2$.

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, $\sigma_1^2 \geq \sigma_2^2 ... \geq \sigma_n^2$, 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 windows, and suitable for processing both online and predefined sliding window queries.

3.4.2 Timestamp Queries

If there is an infinite window, i.e., all available dimensions are involved in queries after a certain time point, 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 online timestamp queries. 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 online timestamp queries.

ALGORITHM 4 VA$^+$-Stream:

1. Starting at the pre-specified timestamp, 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 jth 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 Online Timestamp Query, a similarity search method has to consider all dimensions after a certain timestamp 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 the simple sequential scan [26]. It also challenges the plain way of rebuilding VA-File or VA$^+$-file from scratch. For sliding window queries, this plain approach might get lucky when a small window size for the data set is specified by the user. But for timestamp queries, the query building time becomes intolerable with new data continuously coming in if we rebuild the VA-file or VA$^+$ from scratch for involved dimensions after a certain timestamp, as the performance study in Section 5 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 the 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.

3.4.3 Aged Queries

In order for the proposed VA$^+$Stream to be able to work for the aging data streams, 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 $b_i \geq b_j + k$ [13]. The rest of the algorithms remain to be the same. Intuitively, this heuristic rule has taken care of the weights assigned to different dimensions by assigning more bits to those with larger weights. Hence, our approach can be easily adapted to index aging data streams.

3.5 Other Discussions

In a streaming database, the number of streams may also change. Our approach can handle this scenario trivially too. For sliding window queries, if the new data streams pop up, we will wait to collect the data from the new streams until the size matches the sliding window. Then we quantize all new data streams collected so far, and add them to the VA$^+$file which we have maintained incrementally so far. Now the new data streams get synchronized with the existing data streams. On the other hand if some data streams drop out, we just simply delete the corresponding approximations in our VA$^+$file. For timestamp queries, 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 an infinite window size, just delete their approximations 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. Also in this paper, we only consider multiple synchronous data streams. Asynchronous data streams are not studied in this paper.

4. QUERY PROCESSING

As is shown in Section 3.4, with VA$^+$-Stream technique an efficient synopsis of the streaming data can be built based on the category of the given query on the third level of the classification in Figure 1. However, the categories of the queries on the fourth level affect how the search is conducted on the basis of the synopsis. Also, whether the user wants exact results or approximate results affects the search process. In Section 3.2, both exact and approximate search for k-NN queries have been discussed in details. Now we want to show how other queries are being processed on the basis of the synopsis.

4.1 Approximate Search

The meaningfulness of approximate search while the response time is concerned in the high-dimensional spaces has been addressed in [25, 8]. In streaming databases, approximate search is further preferred over exact search due to the latter's unbounded memory requirements, and the desirability of approximate search in streaming databases was discussed

in [27, 14, 1] .In most cases, approximate answers with high accuracy are acceptable as an alternative of exact answers. Hence, a group of experiments were set up to evaluate the the accuracy of the approximate answers obtained through our approach for streaming databases. The design of our experiments also aims to show the advantage of the proposed approaches in terms of both query response time and index building time. The experiments were run for sliding window queries, timestamp queries and also aged queries.

The vector approximation files obtained from VA$^+$-Stream technique are used as the synopses or summaries of the data sets. Though some research work has been done on approximately answering aggregate queries in streaming databases by using data summarization technique, no work has been done to answer the similarity queries in streaming databases. Our approach can be viewed as a general approach to answer all kinds of queries, including similarity queries.

Based on the vector approximation file obtained through our approach, the quality of the approximate results depends on how to approximate the real values using the approximation vector. One simple way is to use lower bound to approximate the real values. Another way is to use upper bound instead. A third way is to use the average of these two bounds to approximate the real values. A fourth way is to use the representation values of each interval obtained at the end of Lloyd's algorithm for optimal partitionings. Hence, the experiments were also set up to compare which choice is the best one to approximate the real values and results highest-quality solution.

4.2 Exact Search

As is discussed in Section 2.1, exact search in streaming databases is only meaningful for sliding window queries. Practically, the size of the window should be small enough to make the stored data to be within the bound the disk space.

5. PERFORMANCE EVALUATION

5.1 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 research on high dimensional indexing and similarity searching [23, 15, 5]. The fourth data set is *Weather Time Series (Weatherdata)*, a time series data set containing 366 temperature movements of 5009 places in year 2000. In the context of streaming databases, 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.

5.2 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 ef-

ficient approximate search for all kinds of continuous queries in streaming databases. Then we will show that our approach can support exact search efficiently for sliding window queries too.

5.2.1 Query Performance for Approximate Search

The first group of experiments was done to evaluate the quality of the approximate search for k-NN queries based on the approximation file obtained by VA+Stream. In order to measure the quality of approximate k-NN search results, two performance metrics were used: precision/recall, and an error metric D which is defined as the relative approximation error, given by $\frac{\sum_{i=1}^{k} d_f(q,a_i)}{\sum_{i=1}^{k} d_f(q,r_i)}$ for k-NN queries, where q is the query, d_f is the similarity measure, or distance function, $(a_1, a_2, ..., a_k)$ and $(r_1, r_2, ..., r_k)$ are approximate results and exact results, respectively. Since for k-NN queries, we only retrieve k data points, so the precision and the recall are the same. Hence, only the precision metric was used.

For the purpose of searching approximately, we dropped off the second phase during similarity search, four different criteria were chosen respectively to approximate the real values. They are the lower bound values, the upper bound values, the average of the above two bounds and also the representation values, obtained in the first phase, The vectors with the k smallest lower values in each criterion are the approximate results for the corresponding criteria. 50 randomly generated 30-NN queries are performed on *Stockdata* and *Weatherdata*. For each data set, we used VA$^+$Stream to build four different vector approximation files with an average 3 bits, 4 bits, 5 bits, and 6 bits per dimension, respectively. For each vector approximation file, we ran the 50 30-NN queries with the above four different criteria, respectively. We also tested the performance of approximate VA-file [25]. Table 2, 3 and 4 show the number of false hits, the precision, and the error metric D and for 10 30-NN queries in *Stockdata*. Table 5, 6 and 7 show the same for *Weatherdata*. Each value in the tables is an average over 50 queries. As expected, the results show that the the quality of approximate search is getting higher with more bits used for approximation. It is also observed that the approximate search with the representation values as the approximation criterion in VA+-Stream is shown to be the most accurate one in every case. The search with the lower bound as the approximation criterion is the second best one in every case. They both far outperform the approximate search with lower bound as the approximation criterion in VA-file [25]. The precision results for both approaches are almost higher than 90% in every case. The use of the upper bound to approximate real values seems to be a bad choice by having the worst performance almost in every test cases. The approximate search performance from using the average of lower bounds and upper bounds as the approximation criterion indicates that it is not a good choice either, though its performance is getting much better when more bits are available for approximation. In some cases, it even performs better than the approximate search in VA-file.

As we have discussed in Section 2.4, besides k-NN queries many other types of queries are of particular interest to people and often used on multiple data streams, .

Experiments on these novel types of queries are desirable in order to show the proposed method VA$^+$Stream can be extended to work well for different queries besides k-NN queries.

The second group of experiments was then done to evaluate the quality of the approximate search for range queries based on the approximation file obtained by VA+Stream. 50 randomly generated range queries are performed on *Stockdata* and *Weatherdata*. As in the first group of experiments, The criteria chosen respectively to approximate the real values are the lower bound values, the upper bound values, the average of the above two bounds and also the representation values, obtained in the first phase.

The third group of experiments was then done to evaluate the quality of the approximate search for similarity joins based on the approximation file obtained by VA+Stream. The similarity join queries are performed on *Stockdata* and *Weatherdata*. As in the first group of experiments, The criteria chosen respectively to approximate the real values are the lower bound values, the upper bound values, the average of the above two bounds and also the representation values, obtained in the first phase.

Approaches	3 bits/dimension	4 bits/dimension	5 bits/dimension	6 bits/dimension
VA$^+$Stream, lower bound	20.3	8.9	3.5	1.7
VA$^+$Stream, upper bound	22.1	10.8	3.6	1.8
VA$^+$Stream, average of lower and upper bound	22.0	10.2	2.5	1.4
VA$^+$Stream, representation values	20.2	7.9	2.2	1.3
Approximate VA-file	16.2	7.2	2.7	1.8

Table 2: False Hits of 50 30-NN Queries by VA-file Approximate Search and Approximate Search Based on Four Criteria for *Stockdata*

Approaches	3 bits/dimension	4 bits/dimension	5 bits/dimension	6 bits/dimension
VA$^+$Stream, lower bound	0.323333	0.703333	0.883333	0.943333
VA$^+$Stream, upper bound	0.263333	0.640000	0.880000	0.940000
VA$^+$Stream, average of lower and upper bound	0.266667	0.660000	0.916667	0.953333
VA$^+$Stream, representation values	0.326667	0.736667	0.926667	0.956667
Approximate VA-file	0.460000	0.760000	0.910000	0.940000

Table 3: Precision of 50 30-NN Queries by VA-file Approximate Search and Approximate Search Based on Four Criteria for *Stockdata*

Approaches	3 bits/dimension	4 bits/dimension	5 bits/dimension	6 bits/dimension
VA$^+$Stream, lower bound	1.932189	1.102510	1.007677	1.001523
VA$^+$Stream, upper bound	1.956730	1.130027	1.008190	1.002331
VA$^+$Stream, average of lower and upper bound	1.953650	1.126013	1.004229	1.001354
VA$^+$Stream, representation values	1.932114	1.105713	1.003464	1.001467
Approximate VA-file	1.343858	1.055122	1.004847	1.002625

Table 4: Error Metric D of 50 30-NN Queries by VA-file Approximate Search and Approximate Search Based on Four Criteria for *Stockdata*

Approaches	3 bits/dimension	4 bits/dimension	5 bits/dimension	6 bits/dimension
VA$^+$Stream, lower bound	3.1	1.7	1.4	0.9
VA$^+$Stream, upper bound	9.9	5.1	3.1	2.1
VA$^+$Stream, average of lower and upper bound	8.2	3.7	2.2	1.0
VA$^+$Stream, representation values	2.6	1.2	0.6	0.4
Approximate VA-file	5.3	3.3	2.0	1.3

Table 5: False Hits of 50 30-NN Queries by VA-file Approximate Search and Approximate Search Based on Four Criteria for *Weatherdata*

Approaches	3 bits/dimension	4 bits/dimension	5 bits/dimension	6 bits/dimension
VA^{+}Stream, lower bound	0.897	0.943	0.955	0.969
VA^{+}Stream, upper bound	0.670	0.829	0.898	0.930
VA^{+}Stream, average of lower and upper bound	0.725	0.877	0.926	0.966
VA^{+}Stream, representation values	0.914	0.961	0.979	0.985
Approximate VA-file	0.824	0.891	0.935	0.955

Table 6: Precision of 50 30-NN Queries by VA-file Approximate Search and Approximate Search Based on Four Criteria for *Weatherdata*

Approaches	3 bits/dimension	4 bits/dimension	5 bits/dimension	6 bits/dimension
VA^{+}Stream, lower bound	1.014	1.004	1.002	1.001
VA^{+}Stream, upper bound	1.127	1.045	1.009	1.005
VA^{+}Stream, average of lower and upper bound	1.090	1.029	1.005	1.001
VA^{+}Stream, representation values	1.010	1.003	1.001	1.000
Approximate VA-file	1.126	1.044	1.016	1.004

Table 7: Error Metric D of 50 30-NN Queries by VA-file Approximate Search and Approximate Search Based on Four Criteria for *Weatherdata*

The fourth group of experiments was then done to evaluate the quality of the approximate search for aggregate queries based on the approximation file obtained by VA+Stream. Several generated range queries are performed on *Stockdata* and *Weatherdata*. As in the first group of experiments, The criteria chosen respectively to approximate the real values are the lower bound values, the upper bound values, the average of the above two bounds and also the representation values, obtained in the first phase.

5.2.2 Query Performance for Exact Search On Sliding Window Queries

Since exact search for sliding window queries is meaningful, a group of experiments were set up to evaluate the performance of the proposed techniques for obtaining exact answers to sliding window queries. We compare the proposed approach with other possible approaches for sliding window query processing. The design of our experiments aims to show the advantage of the proposed approaches in terms of query response time for exact serach. Since the index building capability has been show in previous section, no experiments were done regarding indexing building. The data sets used in approximate search were used here.

To show that the generated vector approximation file can support exact search efficiently for continuous sliding window queries in streaming databases, we will demonstrate the advantage of our approach over the 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 besides the well-known dimensionality curse, which make other choices like R-tree and its variants out of the question for supporting efficient k nearest neighbor search.

A group of experiments was first set up to evaluate the performance of the vector approximation file generated by our approach for some snapshots 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

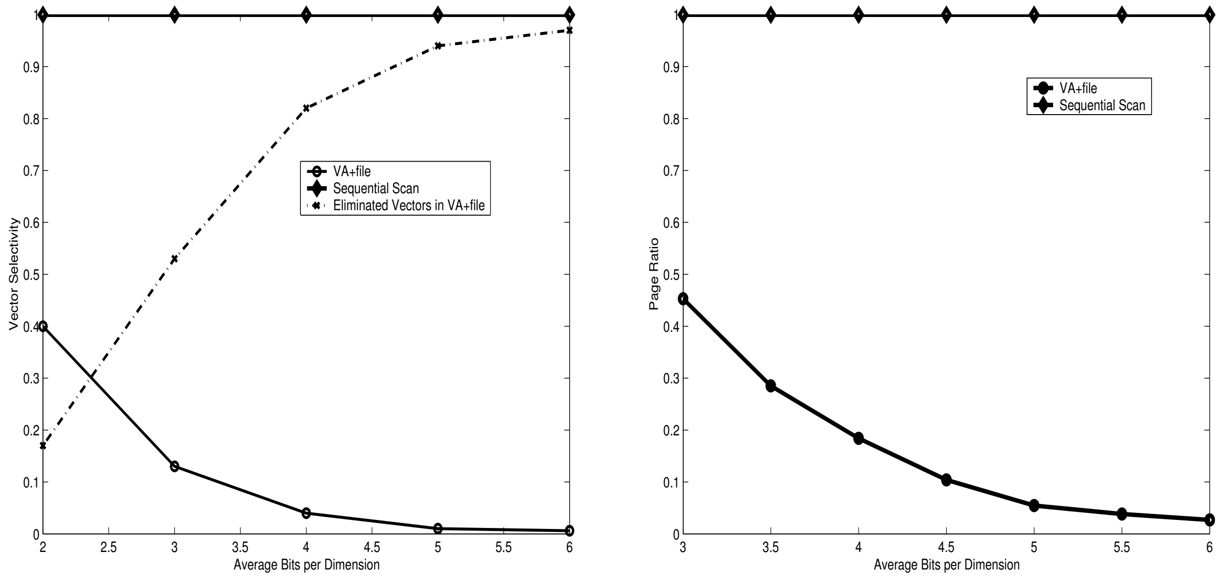


Figure 4: Vector Selectivity (left) and Page Ratio (right) versus Average Bits per Dimension in *Stockdata*

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 first 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 4 (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 first as a function of average number of bits per dimension too. Figure 4 (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 the vector approximation file is only around 10% of that in the sequential scan. Even when we consider the fact that the sequential scan will not invoke any random access which might actually contribute to a factor of 5 for performance improvement [26], the vector approximation file approach still shows advantage.

To show the impact of window sizes on the query performance, we also varied the window size in *Stockdata*. Figure 5 shows the results. We can positively conclude that our approach performs consistently well no matter how large the window size is.

To show the impact of data sizes on the query performance, we also varied the number of data points in *Satellite Image Data Set*. Figure 6 shows the results. We can conclude that our approach performs consistently well no matter how large the data size is.

5.3 Experiments on Index Building

The previous section has demonstrated that the vector approximation file generated by the proposed approach in this paper can support efficient continuous queries in streaming databases. We now want to show that the proposed approach can also support dynamic data streams in terms of

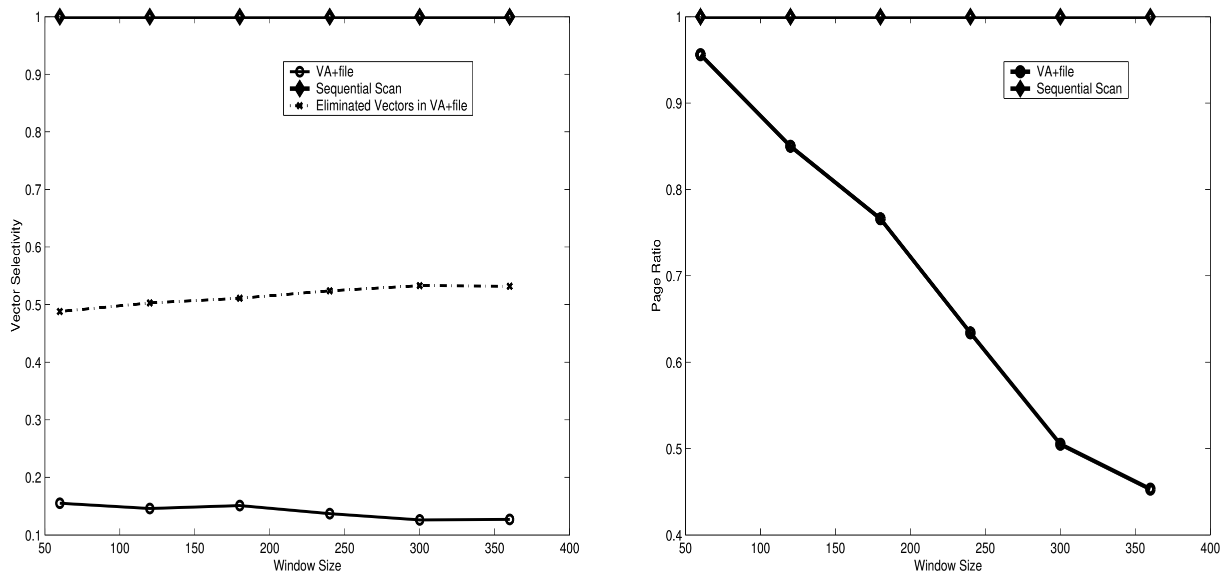


Figure 5: Vector Selectivity (left) and Page Ratio (right) versus Window Size in *Stockdata*

building such a vector approximation file dynamically 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 to build an index structure from scratch each time the new dimension 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 databases, 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 position while 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 databases, 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 sets. 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 tests were done for it.

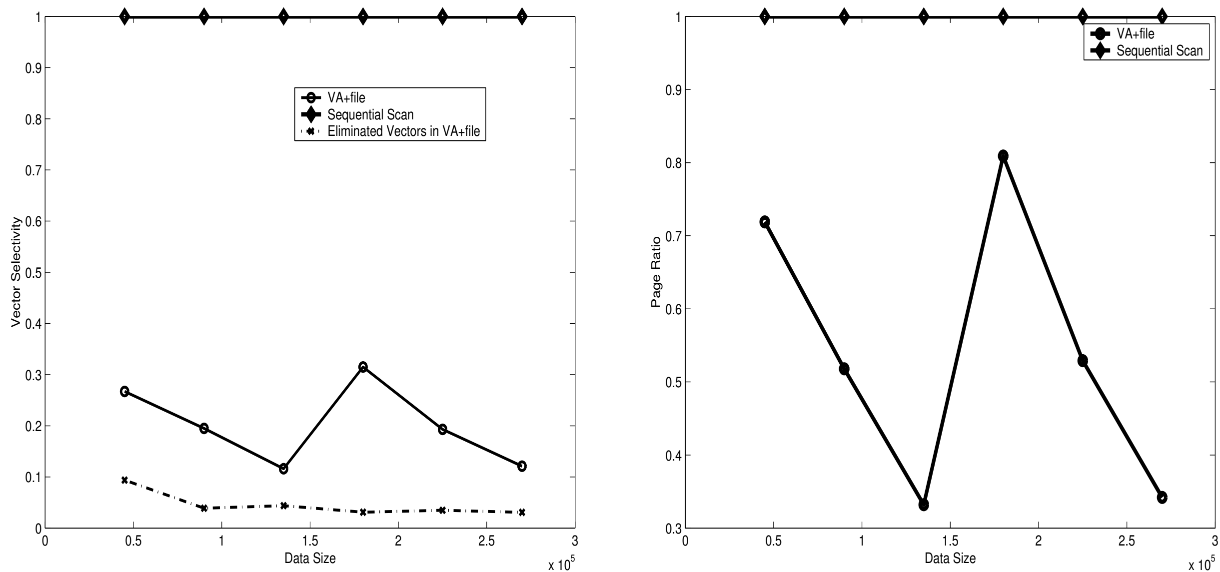


Figure 6: Vector Selectivity (left) and Page Ratio (right) versus Data Size in *Satellite Image Data Set*

The algorithm for building 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 the 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 7 (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 7 (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 8 shows the comparison results of the index building time for highly variant stock time series data and satellite image data set, respectively. 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 sizes for all three data sets.

We ran the test to compare the different implementations of VA$^+$-Stream too. Figure 9 (left) shows the comparison result of indexing building time by Algorithm 3b and Algorithm 3c.

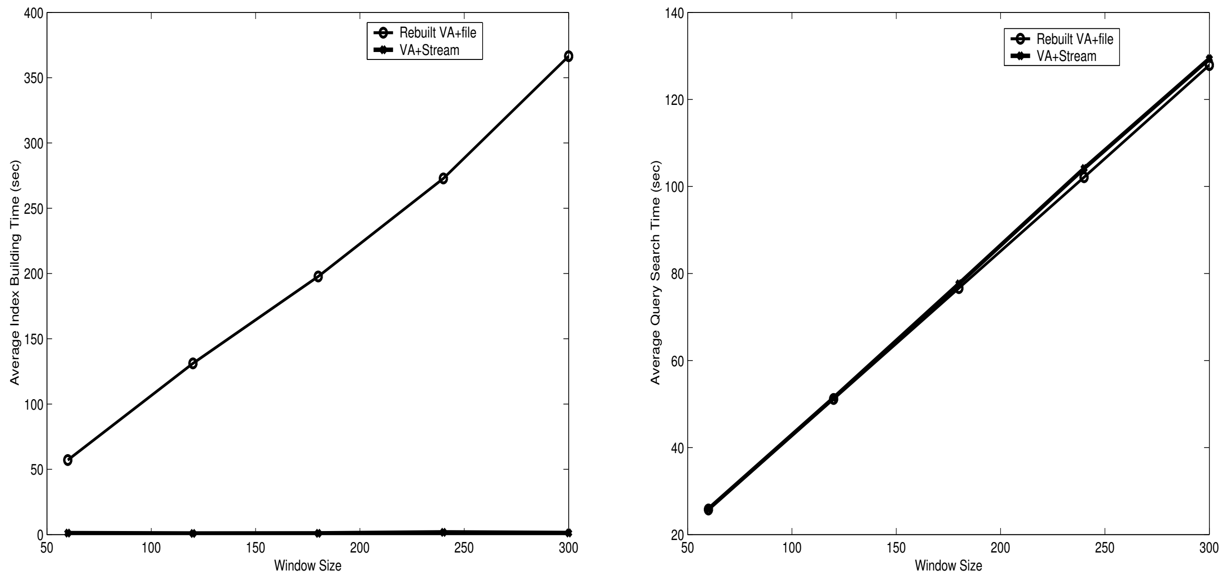


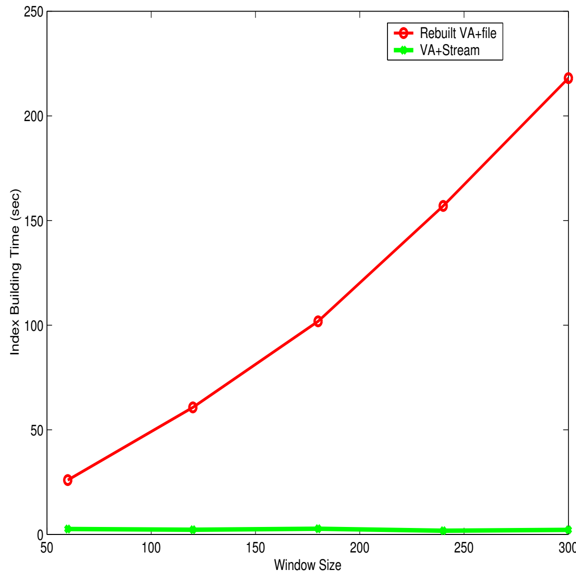
Figure 7: Comparison of Index Building Time (left) and Query Response Time (right) between VA$^+$-file and VA$^+$-Stream for QUERY 1 (i.e. sliding windows) in *Stockdata*

An experiment was also set up for testing ALGORITHM 4 over QUERY 2, which has an infinite window size 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$^+$-file 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 9 (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.

5.4 Experiments on Real-time Performance

Many real-time applications involve multiple data streams. For example, around 20,000 sensors are telemetered once per second in mission operations for NASA's space shuttle at Johnson Space Center, Houston [19]; and there are about 50,000 securities trading in the United States, and every second up to 100,000 quotes and trades are generated [27]. An ideal index structure for streaming time series should be able to process the frequently updated streams in a real-time fashion. We want to demonstrate that the proposed approach can process the data streams on the fly. A group of experiments was designed for this purpose.

The first experiment in this group was used to show how the index building time of the proposed approach changes with the number of data streams increasing. It was tested on *Stockdata* with a window size of 300. Figure 10 (left) shows that the index building time is growing almost linearly for both the proposed approach and also the VA$^+$file approach. However, the proposed approach seems to definitely have a much larger capacity for handling incoming data streams in terms of efficient index building. For example, for 3900 data streams, the proposed approach only needs 1 second while the VA$^+$file approach needs around 126 seconds. More clearly, in Figure 10 (right), we find the difference between the proposed approach and the VA$^+$file approach quite outrageous. For example, within 50 seconds,



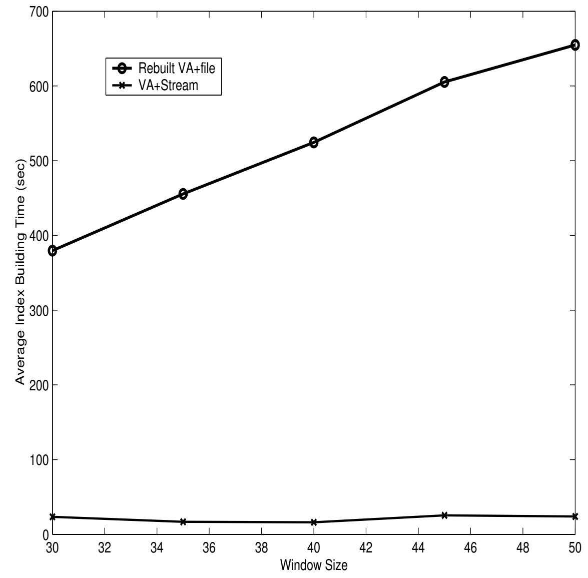
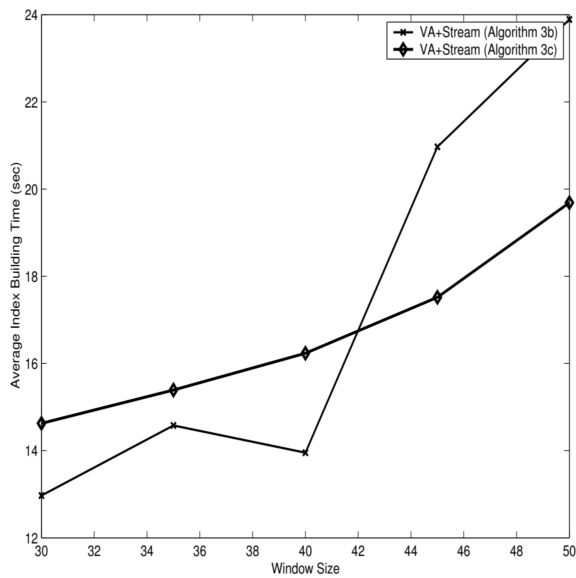


Figure 8: Comparison of Index Building Time between VA$^+$-file and VA$^+$-Stream for QUERY 1 (i.e. sliding windows) in *HighVariant* (left) and *Satellite Image Data Set* (right)



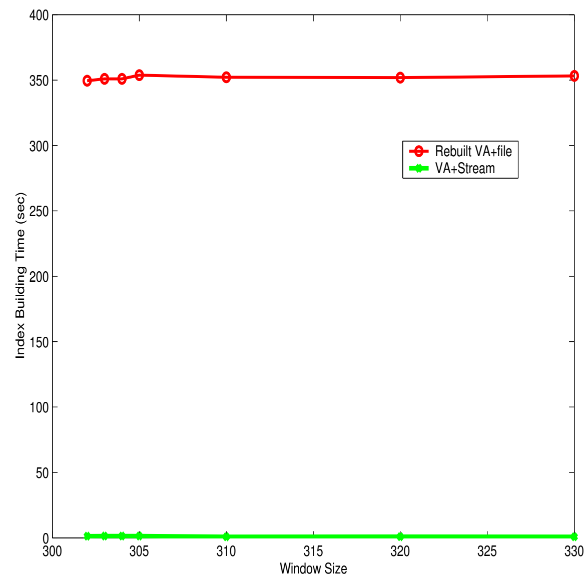


Figure 9: Comparison of VA+Stream (Algorithm 3b) and VA+Stream (Algorithm 3c) in *Satellite Image Data Set* (left) and Comparison of Index Building Time between VA$^+$-file and VA$^+$-Stream for QUERY 2 (i.e. no sliding window) in *Stockdata*

the VA^+file can only handle 1300 data streams, which implies a frequency of 26 streams/second is the capacity limit for VA^+file. However the proposed approach can approximately handle around 200000 data streams within 50 seconds, which means it can handle up to 4000 streams/second and will be able to perform well in those less time-demanding real-time scenarios.

The second experiment in this group was set up to study how the index building time is changing against the query response time. Figure 11 shows that while the query response time is increasing, the indexing building time is also getting longer for both approaches. However, the proposed approach needs much less time for building an index structure.

6. CONCLUSIONS

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

By showing that the scalar quantization technique is a natural choice for handling streaming databases, it reminds us of other techniques which might be good potential choices for indexing streaming databases. As is known, R-tree and its variants are not suitable for streaming databases. Because they are based on the spatial containment relationship between data objects, once new dimension comes it completely destroys 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 need to further split the cells to adapt new data without destroying the previous structure. This indicates it might be a choice for indexing streaming databases too. An effective splitting strategy is needed to keep the fan-out of each cell balanced.

7. 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.

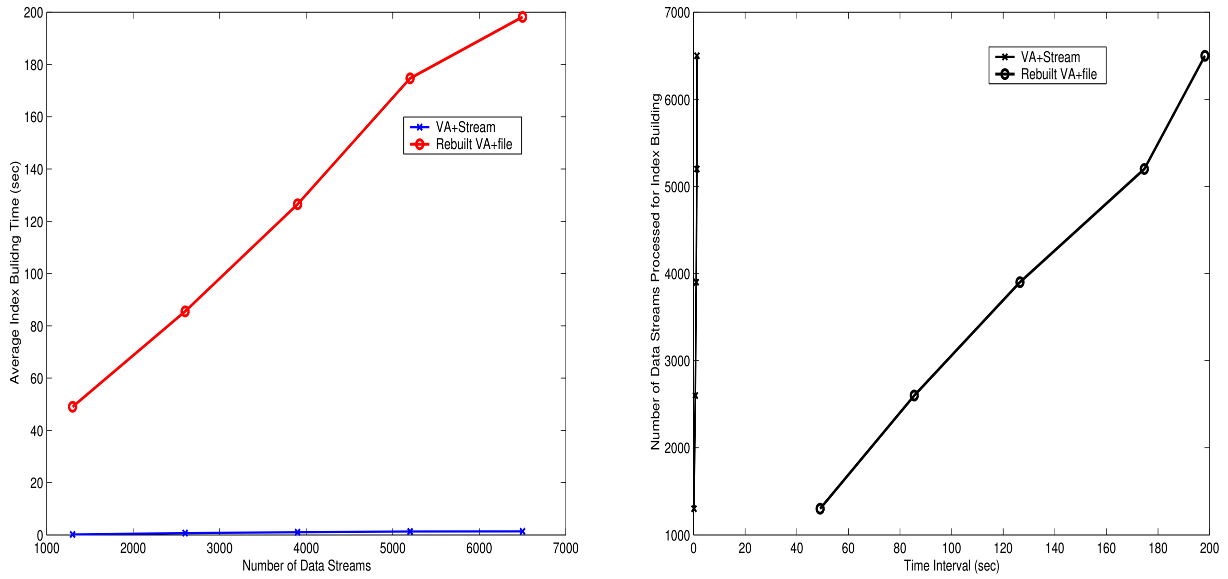


Figure 10: Comparison of Real-time Performance of VA+Stream and Rebuilt VA+file in Stockdata (Window size =300)

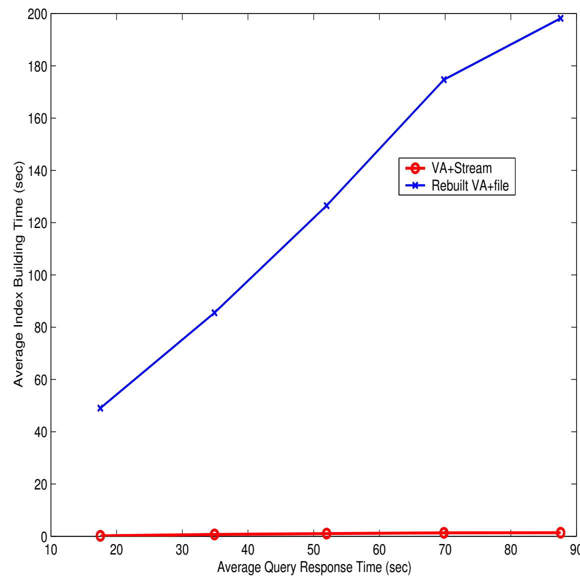


Figure 11: Query Response Time vs Index Building Time for VA+Stream and Rebuilt VA+file in *Stockdata* (Window size =300)

[6] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Madison, Wisconsin, June 4–6 2002.

[7] 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.

[8] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Approximate nearest neighbor searching in multimedia databases. In *Proc of 17th IEEE Int. Conf. on Data Engineering (ICDE)*, pages 503–511, Heidelberg, Germany, April 2001.

[9] 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.

[10] 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.

[11] 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.

[12] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, May 2001.

[13] A. Gersho. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Boston, MA, 1992.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] H. Karhunen. Uber lineare methoden in der wahrscheinlich-keitsrechnung. *Ann. Acad. Science Fenn*, 1947.

[19] E. Keogh and P. Smyth. A probabilistic approach to fast pattern matching in time series databases. In *Proceedings of the Third Conference on Knowledge Discovery in Databases and Data Mining*, 1997.

[20] Y. Linde, A. Buzo, and R. M. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications*, 28:84–95, January 1980.

[21] S. P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:127–135, March 1982.

[22] 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.

[23] B. S. Manjunath. Airphoto dataset. http://vivaldi.ece.ucsb.edu/Manjunath/research.htm, May 2000.

[24] Traderbot. http://www.traderbot.com.

[25] R. Weber and K. Bohm. Trading quality for time with nearest-neighbor search. In *Proc. Int. Conf. on*

Extending Database Technology, pages 21–35, Konstanz, Germany, March 2000.

[26] 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.

[27] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the Int. Conf. on Very Large Data Bases*, Hong Kong, China, August 2002.