Querying Time-Series Streams

Vivekanand Gopalkrishnan
Nanyang Technological University
50 Nanyang Avenue
Singapore
asvivek@ntu.edu.sg

ABSTRACT

Index trees created using distance based indexing are difficult to maintain online since the distance function involved is often costly to compute. This problem is intensified when the database we are dealing with, is frequently updated, as only limited time is available to perform the maintenance. In this paper, we propose a novel tree maintenance mechanism for the problem of answering approximate k-Nearest Neighbor queries with a probabilistic guarantee on time-series streams. When the underlying data change, we may choose to defer updating the tree as long as the probabilistic guarantee of answering queries is high. To prolong such deferment, we present innovative techniques that maintain the utility of the tree by migrating its pivots and by partially reconstructing it. As the probabilistic guarantee decays with time and crosses the minimum guarantee threshold, all of the deferred updates are performed. In essence, our work offers an elegant compromise between the accuracy guarantee of query results and the cost of providing them. With extensive empirical studies, we also show the flexibility and efficiency of our approach.

1. INTRODUCTION

Indexing is traditionally deployed in databases to speed up query performance. If a total ordering can be established, objects in the database are sorted based on some set of attributes, and then an index, usually a tree, is built on the objects. When a query arrives, the index is consulted to reduce the search space as far as possible, resulting in a shorter response time. Where a total ordering cannot be established among them, the objects may be indexed based on their distances to other objects. Ideally we would like to store the distance between every pair of objects in the database, however this is infeasible since it requires high amount of storage, and distance computations are often costly to compute. This leads to the idea that, instead of computing the distance between each pair of objects, we could compute the distance of each object to only a few distinguished objects (*pivots*) selected from the database. One of the earliest data structures based upon this idea is the *metric* tree [23, 24], which is a binary tree that partitions sets of objects into two smaller subsets at each node. In this way, exhaustive search of the whole database can be avoided in most cases.

Numerous types of metric trees have been proposed in the last decade. In *vantage point* tree (vp-tree) [31], each node partitions the data set into inner and outer subsets based on the distance from one pivot. This partitioning method is usually called *ball-partitioning*. Those objects whose distance is more than a specific threshold are included in the *outer* subset, while the rest are included in the *inner* subset. A variation of vp-tree is the *multi* vantage point tree [3], which uses more than one pivot per node in order to increase the fan-out. Fu et al. [12] introduced a *dynamic* vp-tree which supports update. A different partitioning method is used in *generalized hyperplane* tree (gh-tree) [24]. Two pivots are selected on each node and the data are partitioned based on the closest pivot. A variant is introduced in [4], where the number of pivots in a node depends on the cardinality of the data set it partitions. This results in different number of pivots in different levels of the tree.

Pivot selection is an important issue in metric trees, as their performance is dramatically affected by how the pivots are selected [5]. In general, good pivots can eliminate more false candidates very early, e.g., in a balanced binary search tree, approximately half of the remaining search candidates are eliminated on each node. Poorly selected pivots, on the other hand, may eliminate nothing at all, and instead only add to the computation time required for a query.

Another important issue is of tree maintenance, which becomes significant when the underlying database changes frequently. This occurs in several domains, such as in a stock exchange, where the database records transactions and stock movements, or in the telecom sector, where a database may record users' calls and billing statements. Data in such applications is said to be *streaming*, and each record/object is treated as a data stream. Values of these objects may change with time, and in most data stream applications, existing objects are usually decayed after some period of time, rather than being removed entirely from the database.

The sliding window model [1] is used to ensure that only the most recent N elements of the data stream are used when answering queries. This is also known as discounting stale data in data stream applications. Babcock et al. [2] addressed the problem of maintaining variance by providing a continually updated estimate of the variance of the last N values in a data stream.

EDBT'08, March 25–30, 2008, Nantes, France.

Xia et al. [30] proposed a technique for maintaining two separate index structures - one R-tree and one QuadTree. This technique separates objects into slow or fast state of motion, storing them into the R-tree and QuadTree respectively. Such an indexing method is proposed based on the observations of physical moving objects, implying that there's a physical constraint on the value of the next point (location) of the object. Hence approaches developed for indexing moving objects are not directly applicable in general time series data streams, where such constraints cannot be justified. Tao et al. [22] address the problem of solving probabilistic range search in uncertain databases by proposing the U-tree. The U-tree computes some auxiliary information for each object, which is used to disqualify the object or to validate it as a result without having to determine its appearance probability. This differs from the proposed approach in this paper, which looks into the accuracy of the entire tree as a whole, instead of an individual object or node in a tree.

Existing metric tree maintenance approaches only support object-wise addition, removal and updates. In this paper, we present a novel metric tree maintenance mechanism, which addresses the unique characteristics of data streams, i.e., continuous and simultaneous updates of all data streams. We use data streams – represented as time series data – as the objects to be indexed in the database. We use distance based indexing with Edit distance with Real Penalty (ERP) [7] as distance function.

1.1 Problem Definition

We focus on answering approximate k nearest neighbours (k-NN) queries over multiple streams of time series data – each data stream as one object – at once. By definition, the result of a k-NN query for an *object* (data point or a sequence) are the k *closest* objects (with respect to a given distance measure) to the query object. Figure 1 depicts samples of time series data streams with a sliding window over which queries are posed. Here, a query (and the corresponding results) is a time-series sequence from one of the streams ($DS1 - DS10$), and the results are similar subsequences among them. The figure also shows gh-tree (described in detail in Section 2.3) which is used to improve access on the underlying sequences. This is a more challenging problem than problems addressed earlier in the sense that we process queries on several streams of unknown patterns. Wu et al. [29] addressed k-NN queries over financial data streams but they only considered one data stream at a time. Saltenis et al. [21] addressed continuously updating objects all at once, but restricted the domain to location coordinates, and did not address the *time series* as the object. Vlachos et al. [26] worked on multiple time series at once, but they assumed large static time series data instead of real-time changing time series data.

Although there have been several recent works in streaming time series, or in approximate querying, to the best of our knowledge there has been no study on approximate k-NN search over several streams of time series data. Currently, existing literature with regards to time series only looked into the aspect of indexing static time series data. Similarly, existing literature with regards to data streams indexing are performed only in a specified time instance, i.e., a single time point for a set of different data streams. Our work approach the problem of indexing different time series data in a streaming environment. Instead of taking a single time point for indexing, we index subsets of data corresponding to a given window size from the various streaming time series data. In our problem, several constraints about the number of dimensions of data, and the number of data streams are relaxed.

The major contributions of this paper are as follows:

- We address the problem of answering approximate k-NN queries among several streaming time series data. To that end, we build an index that enables efficient querying. These queries are ad-hoc and dynamic.
- We introduce a tree maintenance technique for metric trees even when insertions and existing object updates change the overall object distribution and thereby render the initially chosen pivots as potentially ill candidates. We show that our approach requires little number of distance computations, while maintaining high query accuracy. While our approach is applicable to metric trees in general, for the sake of simplicity, we base our method and analysis on the generalized hyperplane trees with two pivots.
- We provide pivot selection and pivot reselection techniques suitable for our tree maintenance method. Our approach focuses on supporting the tree to maintain its accuracy as long as possible, thus reducing the frequency with which the tree needs to be changed or rebuilt. We demonstrate that our approach requires less execution time to achieve this goal.
- We establish a general framework of pivot maintenance for metric trees based on our probabilistic approach. This includes the pivot migration policy, pivot reselection policy, and the decaying confidence metric tree model.

The rest of this paper is structured as follows. We present a detailed study of the background of the problem, and discuss limitations of some related work in section 2. Then, we present a probabilistic analysis of object migration in metric trees in section 3, based on which we introduce a novel tree maintenance approach in section 4. We evaluate this approach with a real dataset, and analyze the experimental results in section 5. Finally, we conclude in section 6 with directions for further researches.

2. BACKGROUND AND RELATED WORK

2.1 Similar works in other domains

The problem of monitoring continuously changing data objects in real time has been addressed in the moving object databases domain. Cai et al. [6], address continuous monitoring queries on certain ranges. To answer such queries, each mobile object updates the database when it is outside a predefined known range. Each object maintains information about nearby queries to enable it to know when to update the database efficiently. This approach is only applicable only when the ranges being monitored are known in advance, and are static. Wolfson et al. [28] address location queries with imprecise answers along with its deviation. To maintain the answers within some a threshold (deviation), objects are responsible to report their position once they deviate above the deviation threshold. In this paper, we have

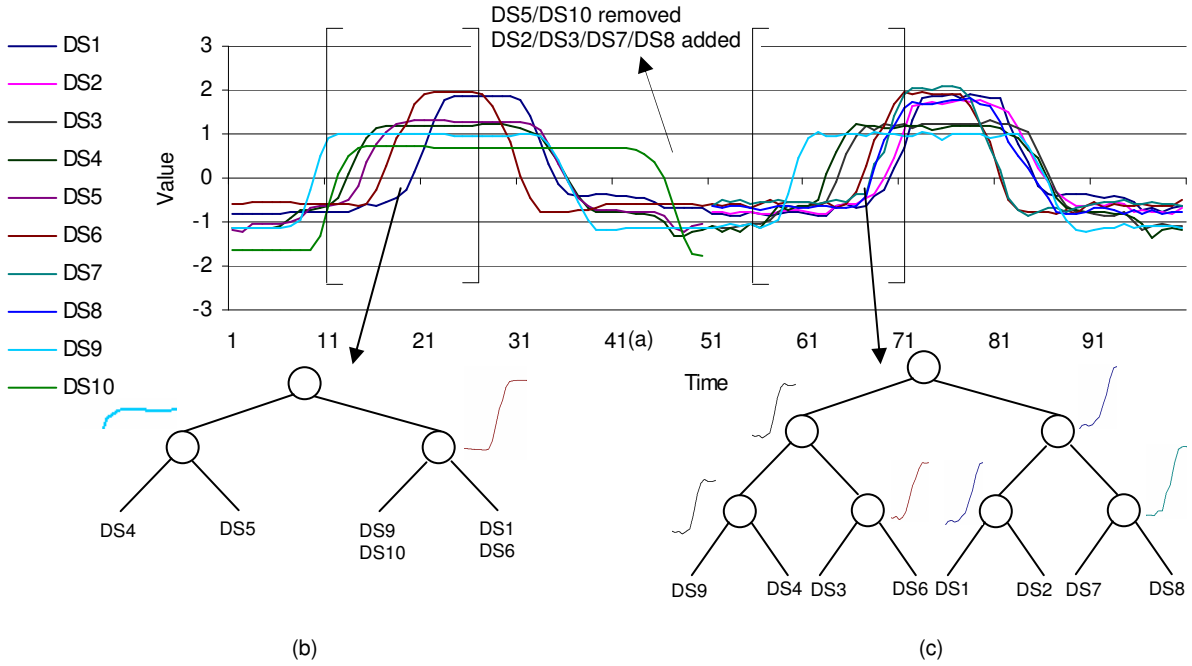


Figure 1: (a) Samples of Time Series Data Streams. (b), (c) gh-tree representations of two windows.

no such assumptions about the ranges, and these ranges dynamically change depending on the actual values of the data in the time series stream. We also do not impose any additional burden for the streams to monitor their deviation.

To the best of our knowledge, [8] is the only work dealing with a generic data model specially for probabilistic querying. It provides a data model that is domain independent and is used to answer a query with some probabilistic guarantee. This problem they tackle is similar to what we are trying to address in our paper. Their data model is based on four concepts:

1. Uncertainty Region U_i, a region where an object may possibly be located.

2. $f_i(location, time)$, a probability density function of object location at a particular time. This *pdf* can be determined by knowing certain motion properties, e.g. velocity.

3. $P_i(r)$, probability of an object being located within range r from a query point.

4. $pr_i(r)$, probability density function of $P_i(r)$.

Even though this data model is generic, it is not suitable for the general time series domain because it is computationally inefficient to find U_i, f_i, $P_i(r)$ and $pr_i(r)$. This is due to the fact that moving object databases maintain point coordinate data while time series databases maintain trajectory data of points. Furthermore, in most cases we don't have time series motion properties to determine f_i.

Hence to the best of our knowledge, while our problem is similar to the one in moving objects domain, it has not been addressed before in the time series domain. Furthermore, approaches in moving objects domain are not extendable to time series domain. This paper proposes an approach specifically for the time series domain for answering queries with a probabilistic guarantee of the result.

2.2 Approximate k-NN

Finding k-nearest neighbors (k-NN) is a classical database problem, and it has been well studied in several contexts. There are several recent methods [10, 18, 19] that tackle this problem when the data is in a high dimensional space. In [20], the authors looked into the problem of searching time series data through a lattice structure integrated into a R-tree indexing scheme. Gao and Wang [13] consider the problem of continuously finding the nearest neighbor to a streaming time series. In their approach, they assume that the database of predefined patterns is in secondary memory. Vlachos et al. [25] addressed the problem of indexing multi-dimensional time series. In their method they propose a technique that is based on the Longest Common Subsequence (LCSS) model and Dynamic Time Warping (DTW). The purpose of this technique is to efficiently organize trajectories on disk, so that they can quickly answer k-NN queries. Another interesting work yielding efficient query filtering for streaming time series is proposed in [27]. Their main objective is to monitor streaming time series for a set of predefined patterns and their problem is analogous to that of query filtering for discrete valued data [11]. Jagadish et al. [16] approaches the problem of k-nearest neighbours search in high-dimensional metric space through a method known as iDistance. In [17] the authors propose an approach to solve the e-approximate k-NN (also called ekNN problem) over single data streams. They develop a technique called DISC to solve k-NN with a guaranteed error bound and to achieve the best accuracy under a given memory constraint.

2.3 Generalized hyperplane tree

As we have discussed earlier, each node in the gh-tree uses two pivots, and partitions the data set into those closer to

the first pivot and those closer to the second pivot. Formally, we say that a node N in gh-tree partitions a set S_N into two subsets S_{N_1} and S_{N_2} using pivots P_{N_1} and P_{N_2}, such that:

$$S_{N_1} = \{x \in S | d(x, P_{N_1}) \leq d(x, P_{N_2})\}$$

$$S_{N_2} = \{x \in S | d(x, P_{N_1}) > d(x, P_{N_2})\}$$

In the gh-tree described using equation above, we say that subset S_{N_1} is *represented* by pivot P_{N_1}, while subset S_{N_2} is *represented* by pivot P_{N_2}. The two disjoint subsets S_{N_1} and S_{N_2} can also be described using alternative equation:

$$S_{N_1} = \{x \in S | d(x, P_{N_1}) - d(x, P_{N_2}) \leq 0\}$$

$$S_{N_2} = S \backslash S_1$$

We call $\psi(x, P_{N_1}, P_{N_2}) = d(x, P_{N_1}) - d(x, P_{N_2})$ the *discrimination* of object x with respect to pivots P_{N_1} and P_{N_2}. Observing the alternative equation above, the resulting S_{N_1} and S_{N_2} can be of different sizes, thus reducing the effectiveness of the node. A recommended partitioning [4] of the gh-tree is based on threshold value m, such that $S_{N_1} = \{x \in S | d(x, P_{N_1}) - d(x, P_{N_2}) \leq m\}$ and $|S_{N_1}|$ is roughly equal to $|S_{N_2}|$ [23, 15]. m, which is user-defined and domain dependant, allows boundary objects to be included in S_{N_1}.

For the rest of the paper, if node N is a node in the gh-tree, we denote P_{N_i} as the i-th pivot of node N, and S_{N_i} as the subset represented by P_{N_i}. S_N refers to $S_{N_1} \cup S_{N_2}$, which is the original set before it is divided into two. If the subset S_{N_i} is to be further partitioned by another node, that particular node is denoted as N_{C_i}.

The general search algorithm for gh-tree uses a Depth First Search approach. For a query q, an empty list with size k is initially created to hold the current k-NN result. During the tree traversal, when a node is visited, the distances between the query and the two pivots P_{N_1} and P_{N_2} are computed. A decision is made whether we need to visit the subsets of this node by observing the computed distance. Given a range query with query radius ε, the subset S_{N_1} needs to be visited if and only if $d(q, P_{N_1}) - d(q, P_{N_2}) \leq 2\varepsilon$, and subset S_{N_2} needs to be visited if and only if $d(q, P_{N_2}) - d(q, P_{N_1}) \leq 2\varepsilon$. The subsets S_{N_1} and S_{N_2} are visited in order of the distances that P_{N_1} and P_{N_2} have from q. Suppose this node has threshold m, S_{N_1} is visited first only when $d(q, P_{N_1}) - d(q, P_{N_2}) \leq m$. In other cases, S_{N_2} is visited first over S_{N_1}.

M-tree [9], which is based on ball-partitioning method, uses an overflowing approach to allocate new nodes when a new object is inserted into the tree. The algorithm finds the leaf node where the object should be placed into, and then if the addition of the new object causes the leaf node to overflow, a new node with new pivot is created and the objects are repartitioned among the two nodes. This splitting may cause the parent to overflow, thus the change may cascade up to the root of the tree. A variant of the M-tree, Slim-tree [13], optimizes the object insertion and node splitting process of the M-tree.

2.4 Limitations of existing methods

A major problem with the approaches discussed above is that they do not specifically handle situations when an update causes existing objects to change their value. Because of such changes, the object may then be placed in the incorrect subset. The naïve way of solving this problem is to first remove the object from the tree, and then insert the updated object back into the tree. However, while this approach may suffice in static databases, it is infeasible in a data stream environment. This is because all objects may potentially change on update of every stream, and so changes to the tree are frequent.

We introduce a novel tree maintenance approach that addresses the problems we discussed above. The main purpose of tree maintenance is to keep the tree updated with the objects that it indexes. With respect to data streams, we identify two major events that lead to the need of tree maintenance: (a) addition of new objects to the tree, and (b) change of existing object's value with the new update. Our approach focuses on reducing the amount of distance computations required to maintain the tree without sacrificing too much query accuracy. With regard to this, we provide a probabilistic analysis of the accuracy with which the approximate k-NN query can be answered by our tree for the two events mentioned above. We establish a decay model for accuracy guarantee provided by the metric tree, and offer algorithms to extend this guarantee by performing tree maintenance according to the model. We use a slightly modified representation of gh-tree in our approach to accommodate the peculiarities of data streams. We store the data objects only on the leaf nodes of the tree, similar to the approach used in M-tree [9]. In our tree, we use a metric distance function, Edit distance with Real Penalty (ERP) [7] for computing distances between time series. Detailed discussion of the metric and implementation can be found in [14].

3. PROBABILISTIC ANALYSIS OF ERP-BASED METRIC TREE

In this section, we provide mathematical analysis of the metric tree which uses gh-tree with ERP as distance function. A similar analysis can be made when other metric functions are used with other trees. In gh-tree, each node contains two pivots which are used to separate objects into the corresponding two partitions. We're interested in determining the probability of an object migrating into other partition in a node, i.e., the probability of an object becoming nearer to the other pivot after it has changed. We observe this probability behaviour against amount of change of the object (Δ) to get an estimation. This probability estimation will be further used to estimate amount of change of nodes of the tree and thereby the confidence level of the tree.

ERP can be seen as the minimum sum of area-differences between two time series which are split into parts. This minimum sum is achieved by aligning parts of each time series after adding gaps into them [7]. This is why ERP values vary around the absolute area difference between the two series. Absolute area-difference between two time series R and S is given by $|Area(R) - Area(S)|$, where Area(R) (resp: S) is the area of time series R (resp: S) plotted in cartesian coordinates. Note that this is actually the L1-norm of two scalar values, i.e. the 1-dimensional Euclidean distance.

Following this, we propose that probability behaviour of an ERP-based metric tree should be similar to that of 1-dimensional Euclidean distance-based metric tree. We show that while the probability plot of the 1-dimensional Eu-

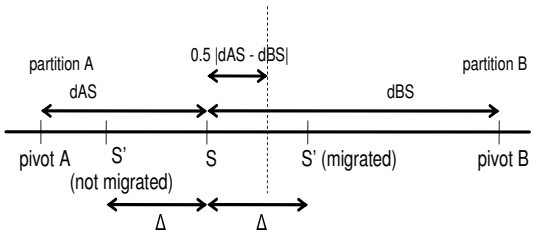


Figure 2: Partitioning in 1-dim. Euclidean space

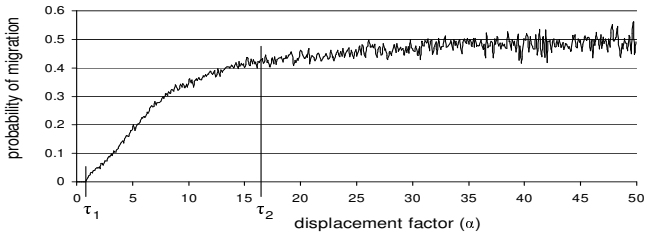


Figure 3: Simulation results

clidean distance-based metric tree is a unit step function, that of the ERP-based metric tree has a smooth transition instead of a step.

Let the two pivots in a node of the gh-tree be A and B, and let the distances of object S to them be dAS and dBS respectively. S belongs to partition A iff $dAS - dBS \leq m$ for a certain threshold value m, otherwise S belongs to partition B. Figure 2 shows an example of gh-tree partitioning in 1-dimensional Euclidean space.

Here S initially belongs to partition A. After update, S changes to S', and the amount of change (distance between S' and S) is Δ. From Figure 2, we see that the object migrates partitions with probability of 0.5, only when $\Delta > 0.5(dBS - dAS + m)$, m is as described in Section 2.3. In general, the probability of migration of an object S is given by the step function:

$$p = \begin{cases} 0, & \Delta \leq \frac{m+|dAS-dBS|}{2} \\ 0.5, & \Delta > \frac{m+|dAS-dBS|}{2} \end{cases}$$

As in [15], we assume in the following that $m = 0$; the following equations can be generalized for other values of m.

ERP behaves similarly to the 1-dimensional Euclidean distance, however, for a given Δ, there is a range of locations for S'. Moreover, in order to analyse the probability of migration in a data-independent manner, it is necessary to have a normalised scale, therefore we propose to observe the probability distribution against the following *displacement factor* α:

$$\alpha = \frac{\Delta}{0.5|dAS - dBS|} \cdot min\left(\frac{dAS}{dBS}, \frac{dBS}{dAS}\right),$$

where $min\left(\frac{dAS}{dBS}, \frac{dBS}{dAS}\right)$ indicates the relative closeness of the object to its current pivot and is always < 1. Just as the probability of migration in the 1-dimensional Euclidean space rises to 0.5 when $\Delta = 0.5|dAS - dBS|$, the probability plot against α is expected to rise from 0 when α nears 1, and approach 0.5 when α increases beyond a threshold.

We propose that the probability will be as follows :

$$p(S, S', A, B) = \begin{cases} 0, & \alpha < \tau_1 \\ f(\alpha), & \tau_1 \leq \alpha \leq \tau_2 \\ 0.5, & \alpha > \tau_2 \end{cases} \quad (1)$$

where $f(\alpha)$ is a monotonically increasing continuous function from 0 to 0.5. The reason we have $f(\alpha)$ instead of a step function is that ERP values vary due to alignment of parts of the time series, i.e., for a given value of Δ, the location of S' is a range instead of a fixed point.

In order to determine the probability behaviour of migration against α, which would demonstrate correctness of the above conjecture, we performed a Monte-Carlo simulation. Initial values were assumed to be uniformly distributed. This assumption allows the probabilistic analysis to be domain-independent. The simulation also provides us with reasonable values of the thresholds τ_1, τ_2 and $f(\alpha)$. Figure 3 shows results of the simulation indicating cutoffs of τ_1 and τ_2. These results are used when objects change and we need to estimate to estimate the confidence level of the tree (section 4.3).

From our experimental runs [14] , we observe that $f(\alpha)$ is nearly linear, with slope approximated as $\frac{1}{32}$. Based on the simulation results, it is approximated that $\tau_1 = 0.8$, $\tau_2 = 16.8$ and $f(\alpha) = \frac{1}{32}(\alpha - 0.8)$.

We use Equation 1 while maintaining our tree in order to determine the probability of an object actually migrating to another node of the tree (c.f. Section 4.3).

4. TREE MAINTENANCE

In this section, we describe our procedures and algorithms to perform tree maintenance. Before we proceed, we portray the terms we use frequently in our explanation.

Let M and N be nodes in a metric tree. M is said to be an *ancestor* of N if and only if $S_N \subseteq S_{M_1} \vee S_N \subseteq S_{M_2}$. If M is an *ancestor* of N, then N is a *descendant* of M. We denote $A(N)$ as the set of ancestors of node N, and $D(N)$ as the set of descendants of node N.

Definition 1. (Terminal node) The *terminal node* $T(o)$ of an object o is the leaf node that contains object o.

Definition 2. (Route) Route $R(o, N)$ is the set of nodes in the subtree rooted at N that contains o.

Definition 3. (Correctly-partitioned) Let m be the threshold of node N, and $\psi(x, P_{N_1}, P_{N_2}) = d(x, P_{N_1}) - d(x, P_{N_2})$. R_{CP} is a relation whose tuples (o,N) satisfy the following condition $(o \in S_{N_1} \wedge \psi(o, P_{N_1}, P_{N_2}) \leq m) \vee (o \in S_{N_2} \wedge \psi(o, P_{N_1}, P_{N_2}) > -m)$. An object o in S_N is *correctly-partitioned by node* N if and only if tuple $(o, N) \in R_{CP}$.

Definition 4. (Correctly-routed) An object o is *correctly-routed* by node N if and only if node N and all the nodes included in $R(o,V)$ correctly partition o. We define a relation R_{CR} whose tuples (o,N) satisfy the condition $(o, N) \in R_{CP} \wedge ((o, K) \in R_{CP}, \forall K \in R(o, V))$, where V is the root node of the metric tree.

Definition 5. (Node correctness) The *correctness of node* N is defined as the ratio of objects in S_N that are correctly-partitioned by node N to the cardinality of S_N. We denote the correctness of node N as $E(N) = |Y|/|S_N|$, where $Y = \{o|\exists(o, N) \in R_{CP}\}$. It can be easily shown that $E(N)$ is bounded between 0 and 1.

Definition 6. (Node confidence) The *confidence of node* N is defined as the ratio of the number of nodes correctly-routed by N to the cardinality of S_N. Each node has a confidence value. Mathematically, we define the confidence of node N as $F(N) = |W|/|S_N|$, where $W = \{o|\exists(o, N) \in R_{CR} \wedge o \in S_N\}$. The value of $F(N)$ is also bounded between 0 and 1.

Another way to look at $F(N)$ is the probability that an object o randomly chosen in N_S will be correctly-partitioned by all nodes in $R(o,N)$. Therefore, the value of $F(N)$ will be influenced by the correctness of node N and the other nodes contained in $R(o,N)$. We approximate the value $F(N)$ as

$$F(N) = E(N)[F(N_{C_1})|S_{N_1}| + F(N_{C_2})|S_{N_2}|]/|S_N| \quad (2)$$

A special case occurs when the node is balanced and the confidence of the two children nodes are approximately the same. In this case, the expression simplifies to $F(N) = E(N)F(N_{C_1})$

LEMMA 1. *The confidence value of a node N is upper-bounded by the value of correctness of node N.*

PROOF. We rewrite Equation 2 into $F(N) = E(N)Z(N)$, where $Z(N) = (F(N_{C_1})|S_{N_1}| + F(N_{C_2})|S_{N_2}|)/|S_N|$. Let us assume that $F(N_{C_1}) \geq F(N_{C_2})$. From the definition of S_N, we know that $|S_N| = |S_{N_1}| + |S_{N_2}|$. Substituting the value of $|S_{N_2}|$ by $|S_N| - |S_{N_1}|$, the equation becomes $Z(N) = \frac{|S_{N_1}|}{|S_N|}(F(N_{C_1}) - F(N_{C_2})) + F(N_{C_2})$. Because the fraction $|S_{N_1}|/|S_N|$cannot exceed 1, therefore the value Z(N) also cannot exceed $F(N_{C1})$. Consequently, $F(N) \leq E(N)$. A similar proof can be derived for the case $F(N_{C_1}) < F(N_{C_2})$ by substituting the value of S_{N_1}. □

LEMMA 2. *The confidence of a node N is upper-bounded by the highest confidence value of its children, i.e., $F(N) \leq \max(F(N_{C_1}), F(N_{C_2}))$.*

PROOF. Let j be the index, such that $F(N_{C_j}) = \max(F(N_{C_1}), F(N_{C_2}))$. From Lemma 1, we see that $Z(N) \leq F(N_{C_j})$. Because $0 \leq E(N) \leq 1$, then it follows that $0 \leq F(N) \leq F(N_{C_j})$. □

The confidence value of a metric tree's root node can be used as measuring the confidence of the tree itself, since it summarizes the confidence information of all the nodes. We use confidence of the metric tree thoroughly when we decide whether an update to the tree can be deferred at present time.

4.1 Construction of Metric Tree

Before we can construct a metric tree for a set of objects S, we need to set the minimum node utilization parameter μ_u. μ_u is defined as the minimum number of objects that node N should have ($|S_N|$), before N may have children. It follows that all non-leaf nodes M in the tree must have $|S_M| \geq \mu_u$. However, $|S_M| \geq \mu_u$ does not imply that M is a non-leaf node. The value of μ_u must insure that each child has at least two objects, viz. its pivots. In this work, we set μ_u to be 8.

The metric tree is constructed by recursively partitioning the set of objects O. For node N, two pivots are selected from O, and the threshold value m is determined from the median of the distance distribution. Each object in O is placed in either S_{N_1} or S_{N_2}. Based on the size of O, the algorithm may decide to stop the partitioning process or to assign a new node to further partition the subsets. Our full algorithm for metric tree construction is shown in *Algorithm* 1.

Algorithm 1: CONSTRUCT

```
Input: A set of objects O, and level of the tree
Output: A node N that partitions the set O
1  begin
2    Create a new node N
3    Select 2 Pivots from O setting them as the Pivots of
     node N
4    for each object o ∈ O do
5      compute the difference in distance between the
       object and both pivots of node N, diffDist
6    Insert object o into distribution of N
7    Set threshold m as the median of diffDist
8    for each object o ∈ O do
9      if diffDist ≤ m then
10       place object in subset 1 of node N, S_N1
11     else
12       place object in subset 2 of node N, S_N2
13   if number of objects in O ≥ μ_u then
14     call CONSTRUCT for node S_N1 at level+1
15     call CONSTRUCT for node S_N2 at level+1
16   Set confidence F(N) value to 1.0
17   return N
18 end
```

To create the whole metric tree, the construction algorithm above is invoked with 0 as its *level* parameter. Our algorithm, however, assumes that the number of objects in O is as least 2.

LEMMA 3. *The resulting metric tree created using the* CONSTRUCT *algorithm consists of between $\frac{2|O|}{\mu_u - 1} - 1$ and $|O| - 1$ nodes.*

PROOF. A node may contain up to (μ_u-1) objects, without the need of further partitioning. A full metric tree with n leaf nodes has (n-1) non-leaf nodes. As there are $|O|/(\mu_u$-1) leaf-nodes, the minimum number of nodes is $\frac{2|O|}{\mu_u - 1} - 1$. The maximum number of nodes is obtained when each node contains exactly 2 objects. Using a similar argument, the maximum number of nodes is $|O| - 1$. □

4.2 Insertion of New Object

When a new object o is inserted, it is first examined at the root of the tree. The root node V computes the value of $\psi(x, P_{V_1}, P_{V_2})$ and routes the object to the next node according to that value. This process is repeated until it reaches its *terminal node*. After the insertion, the object o is guaranteed to be correctly-routed, because on every node N passed by o, the value of $\psi(x, P_{N_1}, P_{N_2})$ is always examined before deciding whether the next node is N_{C_1} or N_{C_2}. Due to this fact, we have to update the confidence value of each node passed by object o.

We rewrite the equation for confidence of node N to be $F(N) = E(N)Z(N)$ as in the proof of Lemma 1. To know the new value $F(N)$', we have to know $E(N)$' and $Z(N)$'.

Algorithm 2: InsertObject

Input: o, N, *level*
begin
  compute new $E(N)$' value
  compute the difference in distance between the object and both pivots of node N, *diffDist*
  if *diffDist* $\leq m$ **then**
    place object into 1st subset of node N, S_{N_1}
    if *node N is not a leaf node* **then**
      call InsertObject for object o, into node N_{C_1} at *level*+1
  else
    place object into 2nd subset of node N, S_{N_2}
    if *node N is not a leaf node* **then**
      call InsertObject for object o, into node N_{C_2} at *level*+1
  UpdateConfidence of node N
end

Algorithm 3: UpdateConfidence

Input: The node N whose confidence is to be updated
Output: The confidence $F(N)$
begin
  if S_N *changes since last computation* **then**
    compute new $F(N)$' value
    Store new $F(N)$' value and return $F(N)$'
  else
    Return stored $F(N)$ value
end

The new value $E(N)$' can be derived from $E(N)$. From Definition 5 we know that $|Y| = E(N)|S_N|$. We want to find out the value $E(N)' = |Y'|/|S'_N|$ Because a new object is inserted into node N and it is guaranteed to be correctly-partitioned, therefore $|S'_N| = |S_N| + 1$ and $|Y'| = |Y| + 1$. The equation for the new confidence becomes $E(N)' = \frac{E(N)|S_N|+1}{|S_N|+1}$.

Computation for the new $Z(N)$' value is not as direct as $E(N)$', because $Z(N)$' is involved with two separate confidence values of the children. Recall from Lemma 2 that the confidence of a node is upper-bounded by the highest confidence value of its children. Moreover, only one of the child's confidences is changing, because the other child does not change. Let us say that i, j are indexes such that the new object is inserted into N_{C_i} and not N_{C_j}. Therefore, $|S'_{N_i}| = |S_{N_i}| + 1$ and $|S'_N| = |S'_{N_i}| + |S'_{N_j}| = |S_{N_i}|+1+|S_{N_j}| = |S_N|+1$. Consequently, the $Z(N)$' equation becomes $Z(N)' = \frac{F(N_{C_i})'(|S_{N_i}|+1)+F(N_{C_j})|S_{N_j}|}{|S_N|+1}$. From the variables involved in the equation, only $F(N_{C_i})$' is not known previously, but it can be easily found out by recursively calling the confidence value computation on the child node i. The full algorithm to insert a new object o into the metric tree is given in *Algorithm* 2 and the full algorithm to update the confidence values accordingly is in *Algorithm* 3.

Since the value of μ_u is set as 8, node splits occur when $|S_M| \geq 8$. Let us now consider two possible scenarios of node splits. The first scenario occurs when the data sets are equally split. This gives us a balanced tree with the same number of nodes on either side of the tree, and each node contains an equal number of objects. The second scenario is when we have a *bad split*, i.e. the tree is unbalanced and consequently inefficient. If this continues, the branch of the tree with nodes having larger number of objects quickly lose confidence and thereby bring the entire tree confidence down. This results in a reconstruction of the tree.

4.3 Modification to Existing Object

Let N be a node, o an object in S_{N_i}, and S_{N_j} refer to the other subset of node N. When o changes to o', there is a probability that o should be placed in S_{N_j} instead of S_{N_i} (refer to Section 3 for the probability analysis). Hence, modification to existing objects influences the correctness and confidence of a node.

In a node N with specific threshold m, we know from Equation 1 that the probability that an object o which changes to o' will migrate from its original pivot P_{N_1} to P_{N_2} is given by $p(o, o', P_{N_1}, P_{N_2})$, where the displacement factor is

$$\alpha = \frac{d(o,o')}{0.5|\psi(o, P_{N_1}, P_{N_2})|} \cdot min\left(\frac{d(o,P_{N_1})}{d(o,P_{N_2})}, \frac{d(o,P_{N_2})}{d(o,P_{N_1})}\right),$$

Therefore the *decrease in correctness* of node N when an object o changes to o' is given by

$$p(o, o', P_{N_1}, P_{N_2})/|S_N| \quad (3)$$

We can see that the correctness potentially reduces with increase in discrimination of the object with respect to the pivots. The modification of o affects the correctness of not only its terminal node, but also the other nodes in the route of o from the root node. We employ a recursive approach (Algorithm 4) to update the affected values from the root to the terminal node. Only the correctness and confidence values of the affected nodes are updated in this process, but the changed object is not updated. This approach is much more efficient than a delete-insert approach since no distance computations are required to be performed between o' and all the nodes, while maintaining the accuracy of the tree. Experimental results (Section 5) show that the tree confidence is a lower bound of the actual k-NN accuracy provided.

4.4 Pivot Selection

Various pivot selection methods have been proposed for use in metric trees. The easiest and most straightforward way is to randomly select the pivots from the set of objects. This is a very naïve method, as it does not attempt to choose pivots that have some properties that may benefit the tree.

Our pivot selection method (*Algorithm* 5) focuses on reducing the decrease in confidence when any object in the tree changes. This effectively allows us to keep the original tree unchanged, thus maintaining a balanced tree as discussed in the previous subsection.

Let us now consider events that affect the confidence of a node. Insertion of a new object partly increases the confidence of the nodes, because the new object is guaranteed to be correctly-routed. Construction of a new tree restores the confidence of all the nodes in the tree to its maximum value. Modification of existing objects in the tree decreases the confidence of all nodes along the route of that particular object. Among these events, only the last event adversely

Algorithm 4: ObjectChanged

```
Input: o, N, lev, d
begin
    compute new E(N)' value
    if N is non leaf node then
        compute the difference in distance between the object and both pivots of node N, diffDist
        if diffDist ≤ m then
            call ObjectChanged for object o, in node N_{C_1} at lev+1 with change of d
        else
            call ObjectChanged for object o, in node N_{C_2} at lev+1 with change of d
        UpdateConfidence of node N
    else
        let F(N)' = E(N)'
end
```

affects the confidence. Therefore, we focus our pivot selection analysis to reduce the decrease of confidence associated with that event.

From Equation 3 and Lemma 1, the decrease of *confidence* of node N when an object o changes to o' is at most $p(o, o', P_{N_1}, P_{N_2})/|S_N|$. Ignoring the $min()$ term, this decrease is dependant on three parameters, viz., the distance of o from o', the number of objects in S_N, and the initial value of $\psi(o, P_{N_1}, P_{N_2})$. Only the last parameter, i.e., initial value of $\psi(o, P_{N_1}, P_{N_2})$, is directly affected by the selected pivots $\{P_{N_1}, P_{N_2}\}$. For a given dist(o, o'), the decrease of confidence is higher if $\psi(o, P_{N_1}, P_{N_2})$ is closer to the threshold value m.

Therefore, if the node must lose less confidence, we must maximize the difference of the function $\psi(o, P_{N_1}, P_{N_2})$ from m for object o. However, finding pivots $\{P_{N_1}, P_{N_2}\}$ that maximize this function value for each object o in the set is an NP-hard problem. Hence, we adopt a heuristic approach such that pivots having the maximum spread of the objective function $|m - \psi(o, P_{N_1}, P_{N_2})|$ are selected.

In some cases, we may opt to reselect the pivots. Reselection here means that pivots P_{N_1} and P_{N_2} have been selected before, but due to reconstruction to part of a tree, we need to do pivot selection again. Reconstruction is required to prolong deferment period when only a certain part of the tree has low confidence that affects the overall confidence of the tree. This is beneficial to perform when the changes are concentrated to one specific part of the tree.

In reselection, we only allow one of the pivots to change (*Algorithm* 6). First, we choose p closest objects to each P_{N_1} and P_{N_2}. We pair each of the p closest objects to P_{N_1} with P_{N_2} and examine the approximate spread of the objective function. Let object P'_{N_1} have the best spread from the p objects closest from P_{N_1}. During reselection, we consider p closest objects to P_{N_2}, and take P'_{N_2} which is the object with the best spread. Then we compare the spread of P'_{N_1} and P'_{N_2}, and choose the one with best spread.

4.5 Overall algorithm

We offer two algorithms to perform the tree maintenance when an object changes. The first algorithm focuses on avoiding full tree reconstruction by performing partial reconstruction of the tree. The goal is to minimize the number of distance computations required to maintain our model at a specific accuracy. To choose which part of the tree needs to be reconstructed, we keep a list of nodes whose confidence has changed since the last maintenance procedure. An approximation of the benefit achieved if we reconstruct this node N is given by the equation $(1 - F(N)) \cdot |S_N|$. We call this algorithm the *continuously-migrate policy* (*Algorithm* 7).

The second algorithm, called the *lazy-update policy* (*Algorithm* 8), focuses to maximize the accuracy of the queries,

Algorithm 5: SelectPivots

```
Input: The set of objects O
Output: The pivots and a set of p objects closest to them
begin
    Set bestSpread = 0, candidate = null, reselectCandidate = null
    while O ≠ NULL do
        Select two random objects C_1, C_2 ∈ O
        for each object o ∈ O do
            compute the object's objective function |m − ψ(o, C_1, C_2)|
        spread(C_1, C_2) = σ(objective functions)
        if spread(C_1, C_2) > bestSpread then
            bestSpread = spread(C_1, C_2)
            candidate = C_1 and C_2
            let reselectCandidate = p closest objects to each C_1 and C_2
    return candidate, reselectCandidate
end
```

Algorithm 6: ReselectPivots

```
Input: The node N
Output: The new pair of pivots
begin
    let L_1, L_2 denote the set of p closest objects to P_{N_1}, P_{N_2} resp.
    set bestSpreadA = bestSpreadB = 0
    for each object in L_1 do
        let spread denote the estimated spread of object in L_1, P_{N_2}
        if spread > bestSpreadA then
            bestSpreadA = spread
            set P_{N_1}' = object in L_1
    for each object in L_2 do
        let spread denote the estimated spread of object in L_2, P_{N_1}
        if spread > bestSpreadB then
            bestSpreadB = spread
            set P_{N_2}' = object in L_2
    if bestSpreadA > bestSpreadB then
        return P_{N_1}' and P_{N_2}
    else
        return P_{N_1} and P_{N_2}'
end
```

Algorithm 7: ContinuouslyMigratePolicy

```
Input: The original object o and the modified o'
1  begin
2      call ObjectChanged with object o, in node V at
       level 0 with change d between object o and o'
3      let L denote the set of nodes whose confidence has
       changed
4      set bestBenefit = 0, targetNode = null
5      for each N ∈ L do
6          compute the benefit of N
7          if benefit > bestBenefit then
8              bestBenefit = benefit
9              set targetNode = N
10     if targetNode ≠ null then
11         Reconstruct tree rooted at targetNode
12 end
```

Algorithm 8: LazyUpdatePolicy

```
Input: The original object o and the modified o'
1  begin
2      call ObjectChanged with object o, in node V at
       level 0 with change d between object o and o'
3      let L denote the set of nodes whose confidence has
       changed
4      if F(V) < minimum query confidence then
5          Call Construct with all objects O
6      else
7          let bestBenefit = 0, targetNode = null
8          for each N ∈ L do
9              compute the benefit of N
10             if benefit > bestBenefit then
11                 bestBenefit = benefit
12                 set targetNode = N
13         if bestBenefit < minimum benefit then
14             set targetNode = null
15         if targetNode ≠null then
16             Reconstruct tree rooted at targetNode
17 end
```

even if a full tree reconstruction needs to be performed. This is a strict approach in which once the confidence of the tree crosses the minimum confidence level, the algorithm requests for a full tree reconstruction. Otherwise, it calculates the benefit of partially reconstructing the tree using the same technique as in the first algorithm. The difference to the previous policy is that, it may decide not to partially reconstruct the tree if the algorithm finds that the benefit is small. The reason is that with the assumption that the decrease of confidence is approximately equal in every update, if the benefit of updating now is low, there is a higher chance that the confidence will drop below the minimum accuracy threshold with the next update. Therefore, a full tree reconstruction is initiated anyway.

We compile the algorithms and approaches we have discussed into the combined maintenance algorithm (*Algorithm* 9). Note that the combined algorithm shown below is with the assumption that we use the *continuously-migrate policy* when an existing object changes. The combined maintenance algorithm using *lazy-update policy* is obtained by replacing the call to ContinuouslyMigratePolicy(*o*,*o*') function with LazyUpdatePolicy(*o*,*o*').

Algorithm 9: CombinedMaintenance

```
1  begin
2      if object o changes to o' then
3          call ContinuouslyMigratePolicy with
           original object o, and modified object o'
4      else if object o is inserted then
5          call InsertObject with object o and node V
6          if E(V)  < minimum query confidence then
7              Call Construct with all objects O
8  end
```

5. EXPERIMENTAL RESULTS

5.1 Environment and Dataset

We performed experiments on data obtained from Standard and Poor's 500 Index (S&P 500). Each data stream in our experiments represents the stock price of one company within the S&P 500 Index. The entire length of a stream consists of the company's stock prices from January 1985 till December 2005. Companies were chosen at random, and the prices were normalised to between 0 and 1000.

Experiments were conducted on a 1.66MHz PC with 3GB memory and 500MB (restricted) disk space. The data were stored as text files and varied in two aspects: the number of different time series data streams, and the window size of the streams.

5.2 Simulation of Streaming Data

We simulate the streaming nature of the data by restricting the visible window of the data to the program. This is done by the use of a window of predefined size. At any point of time, the program only has information on the data of the time series within the window which is of a predefined size. Any data which exists beyond the time series data set is known to be future data thus simulating streaming data. At every *update*, we shift the window of each stream and notify the change to the tree maintenance procedure. This acts as inputting new streaming data of the various time series into the program. The performance of our probabilistic metric tree is examined after all the changes in the current update have been notified to the maintenance procedure.

5.3 Performance Evaluation

We compare the performance of the original gh-tree with our tree maintenance approaches in two aspects: the tree confidence level and the actual accuracy of the k-NN query when using the tree. In the experiment, we set the value of k to be 10% of the number of data streams. Further, we also compare the maintenance cost required in two policies we described in terms of number of distance computations required. In our figures, we denote continuously-migrate policy as *cm-policy*, and lazy-update policy as *lu-policy*. As we can see from Figure 4, our tree confidence is independent of the window size used in the stream. In general, we notice that the lazy-update policy shows superior perfor-

mance in terms of true k-NN accuracy compared to others, as shown in Figure 5. For all the different window sizes, the continuous-migrate policy is able to hold the query accuracy much longer than gh-tree. We observe that the benefit in accuracy of lazy-update policy to gh-tree decreases as the window size grows larger. This may be caused by the fact that with a larger window size, the time series actually experiences lesser change in every update. We expect that gh-tree would perform reasonably well in datasets with small number of data streams and would experience performance degradation as the number increases. However, we find in Figure 6 that the situation is opposite of what we expected. We may explain this with the fact that with smaller number of data streams, we set smaller value of k. This actually tightens the bound of the query, thus demanding a more accurate answer. We can see in Figure 7 that the gh-tree in general is not able to maintain query accuracy of 80% after 1700 updates to the streams. Using the continuous-migrate policy, we are able to double this period, while lazy-update policy is able to consistently produce at least 99% query accuracy.

An interesting observation regarding the continuous-migrate policy is also noted in this experiment. We take note that the chosen node to be updated does not always improve the tree confidence significantly. This is largely because a node is not considered as candidate if it does not change during the last update. The consequence of this is that nodes that have changed before, but did not change in the last update are not considered to be candidates. We establish this restriction to limit the search space we need to explore when generating the candidates for node reconstruction.

In terms of maintenance cost, both the lazy-update policy (LU) and continuously-migrate policy (CM) incurs lesser distance computations compared to the updated gh-tree (Ugh). The updated gh-tree refers to a naïve manner of updating the gh-tree whenever an object changes, thus producing a 100% accuracy in terms of k-NN queries. Here we present the maintenance cost in terms of number of distance computations performed by each approach. Actual run-time efficiency is dependent on other factors like i/o and caching strategies, and can be optimized as well. Details of this are provided in [14]. From Table 5.3, we notice that gh-tree (gh) has the lowest maintenance cost. This could be explained as the gh-tree never updates itself when an object changes, thus incurring only the distance computations due to initial construction of the tree. The lazy-update policy incurs more distance computations to maintain the tree as compared to continuously-migrate policy. One major reason is that lazy-update forces the whole tree to be reconstructed as soon as the tree confidence crosses the minimum accuracy threshold. Therefore, a full tree reconstruction may be initiated even if during the last update all the changes were located in a specific part of the tree, simply because the low confidence level of that part of the tree drags the whole tree confidence down. This may actually be desirable if we want to treat the tree confidence as a rough lower bound for the query accuracy.

In nearly all cases when we track the maintenance cost, continuously-migrate appears to incur less than 50% of the maintenance cost of lazy-update. While both policies incur significantly lesser distance computations compared to an updated gh-tree which gives a 100% accuracy on true k-NN queries. There appears to be a tradeoff between the maintenance cost and the query accuracy we can achieve. The choice between the two policies should be made on which aspect is primarily more important in the application.

	50 Streams			75 Streams		
	ws20	ws35	ws50	ws20	ws35	ws50
CM	1304	845	420	3180	979	1090
LU	1997	1776	1345	3857	3668	2913
Ugh	11600	8750	6200	10050	6000	4200
gh	50	50	50	75	75	75

Table 1: Average maintenance cost for 50 Data Streams and 75 Data Streams

6. CONCLUSION

In this paper, we addressed the problem of efficiently answering approximate k-NN queries. Our domain was in multiple streaming time series data, over which we perform k-NN queries. Since streaming data are seldom deleted from the database, query results may be present in older sequences of the streams. We noticed that there was no prior work addressing this concern. In order to solve our problem, we proposed an indexing technique based on pivots that could be maintained in order to answer our queries with a probabilistic guarantee. The model represented by our tree displays its confidence of answering a query for which some error is tolerated. This approach allow us to maintain the query accuracy of the original tree without the need to recalculate distances with respect to the updated object. We presented two algorithms to maintain this tree, one emphasized accuracy of the query results, while the other tried to balance the cost of tree maintenance by deferring updates to the tree. We also demonstrated that our technique is correct within reasonable error, is efficient with respect to number of distance calculations, scales well and is robust to changes in parameters. Experimental results showed that there are significant reductions to the number of distance calculations as compared to a delete-insert approach. For future work, we plan to explore the possibility of performing a k-NN query of size smaller than the window size. Another interesting improvement we're working on is computing a tighter lower bound on the confidence decay of a node, taking into account the confidence decay of its siblings.

7. REFERENCES

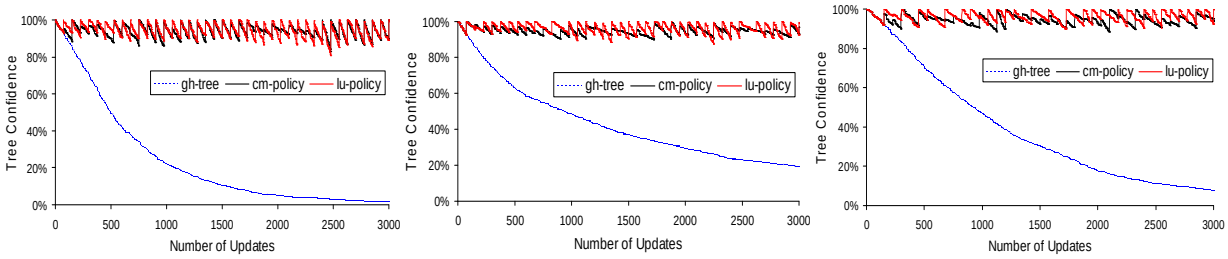
[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[2] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *PODS*, pages 234–243, 2003.

[3] T. Bozkaya and Z. M. Özsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Trans. Database Syst.*, 24(3):361–404, 1999.

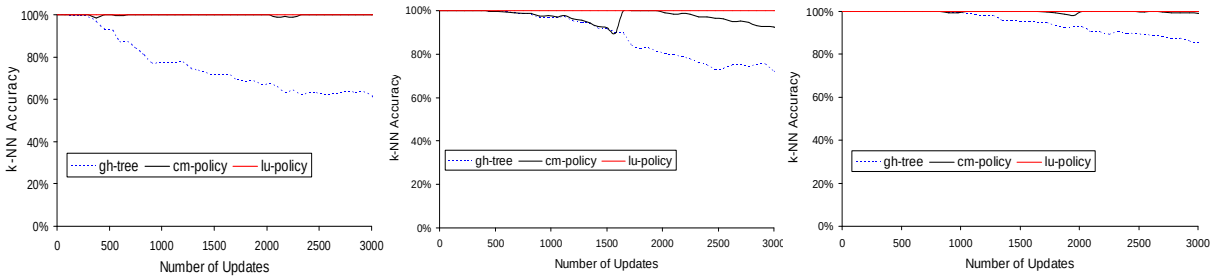
[4] S. Brin. Near neighbor search in large metric spaces. In *VLDB*, pages 574–584, 1995.

[5] B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognition Letters*, 24(14):2357–2366, 2003.



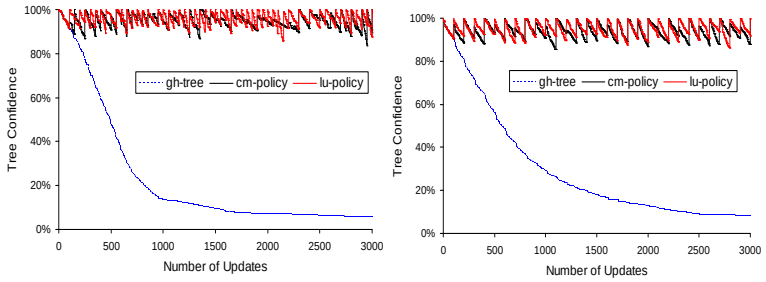
(a) Window size of 20 (b) Window size of 35 (c) Window size of 50

Figure 4: Tree confidence for 75 Data Streams with varying window sizes



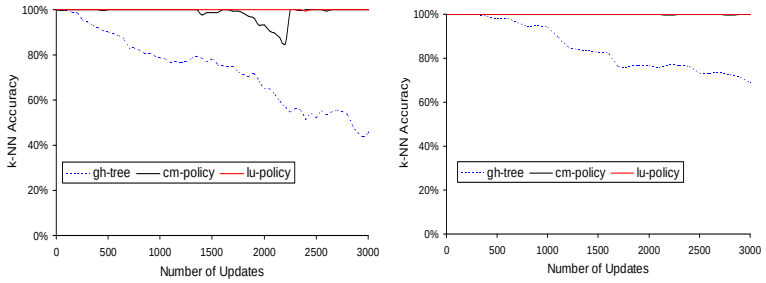
(a) Window size of 20 (b) Window size of 35 (c) Window size of 50

Figure 5: True k-NN query accuracy for 75 Data Streams with varying window sizes



(a) 50 Data Streams (b) 100 Data Streams

Figure 6: Tree confidence for (a) 50 Data Streams, (b) 100 Data Streams with window size of 20



(a) 50 Data Streams (b) 100 Data Streams

Figure 7: True k-NN query accuracy for (a) 50 Data Streams, (b) 100 Data Streams with window size of 20

[6] Y. Cai, K. A. Hua, G. Cao, and T. Xu. Real-time processing of range-monitoring queries in heterogeneous mobile databases. *IEEE Trans. Mob. Comput.*, 5(7):931–942, 2006.

[7] L. Chen and R. T. Ng. On the marriage of Lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.

[8] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Querying imprecise data in moving object environments. *IEEE Trans. Knowl. Data Eng.*, 16(9):1112–1127, 2004.

[9] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[10] B. Cui, B. C. Ooi, J. Su, and K.-L. Tan. Contorting high dimensional data for efficient main memory processing. In *SIGMOD Conference*, pages 479–490, 2003.

[11] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[12] A. W.-C. Fu, P. M.-S. Chan, Y.-L. Cheung, and Y. S. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDB J.*, 9(2):154–173, 2000.

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

[14] V. Gopalkrishnan, P. Chairunnanda, and A. Najib. Efficient index maintenance for answering approximate queries in a volatile environment. Technical Report TR/VG/07/03, CAIS, Nanyang Technological University, Singapore, 2007.

[15] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[16] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

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

[18] Y.-N. Law and C. Zaniolo. An adaptive nearest neighbor classification algorithm for data streams. In *PKDD*, pages 108–120, 2005.

[19] B. Liu, W.-C. Lee, and D. L. Lee. Distributed caching of multi-dimensional data in mobile environments. In *Mobile Data Management*, pages 229–233, 2005.

[20] W. P. M. Polly and M. H. Wong. Efficient and robust feature extraction and pattern matching of time series by a lattice structure. In *CIKM*, pages 271–278, 2001.

[21] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD Conference*, pages 331–342, 2000.

[22] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *VLDB*, pages 922–933, 2005.

[23] J. K. Uhlmann. Metric trees. *Appl. Math. Lett.*, 4(5):61–62, 1991.

[24] J. K. Uhlmann. Satisfying general proximity/ similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[25] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. J. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *KDD*, pages 216–225, 2003.

[26] M. Vlachos, C. Meek, Z. Vagena, and D. Gunopulos. Identifying similarities, periodicities and bursts for online search queries. In *SIGMOD Conference*, pages 131–142, 2004.

[27] L. Wei, E. J. Keogh, H. V. Herle, and A. Mafra-Neto. Atomic wedgie: Efficient query filtering for streaming times series. In *ICDM*, pages 490–497, 2005.

[28] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[29] H. Wu, B. Salzberg, and D. Zhang. Online event-driven subsequence matching over financial data streams. In *SIGMOD Conference*, pages 23–34, 2004.

[30] Y. Xia and S. Prabhakar. Q+Rtree: Efficient indexing for moving object database. In *DASFAA*, pages 175–182, 2003.

[31] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, pages 311–321, 1993.