Indexing continuously changing data with mean-variance tree

Yuni Xia*

Department of Computer Science,
Indiana University,
Purdue University Indianapolis, USA
E-mail: yxia@cs.iupui.edu
*Corresponding author

Reynold Cheng

Department of Computing,
The Hong Kong Polytechnic University,
Hung Hom, Kowloon, Hong Kong
E-mail: csckcheng@comp.polyu.edu.hk

Sunil Prabhakar, Shan Lei and Rahul Shah

Department of Computer Science,
Purdue University, West Lafayette, USA
E-mail: sunil@cs.purdue.edu
E-mail: leishan3@gmail.com
E-mail: rahul@cs.purdue.edu

Abstract: Traditional spatial indexes like R-tree usually assume the database is not updated frequently. In applications like location-based services and sensor networks, this assumption is no longer true since data updates can be numerous and frequent. As a result these indexes can suffer from a high update overhead, leading to poor performance. In this paper we propose a novel index structure, the Mean Variance Tree (MVTree), which is built based on the mean and variance of the data instead of the actual data values that can change continuously. Since the mean and variance are relatively stable features compared to the actual values, the MVTree significantly reduces the index update cost. The mean and the variance of the data item can be dynamically adjusted to match the observed fluctuation of the data. Our experiments show that the MVTree substantially improves index update performance while maintaining satisfactory query performance.

Keywords: indexing; query and update processing; data streaming.

Reference to this paper should be made as follows: Xia, Y., Cheng, R., Prabhakar, S., Lei, S. and Shah, R. (xxxx) 'Indexing continuously changing data with mean-variance tree', *Int. J. High Performance Computing and Networking*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Yuni Xia is an Assistant Professor of Computer Science in Indiana University – Purdue University Indianapolis. She received BS in Computer Science in 1996 from Huazhong University of Science and Technology in China. She obtained MS and PhD Degrees in Computer Science from Purdue University. Her research areas are database and data stream management.

Reynold Cheng is an Assistant Professor of the Department of Computing in the Hong Kong Polytechnic University. He received BEng in Computer Engineering in 1998 and MPhil in Computer Science in 2000 from the University of Hong Kong. He obtained his MSc and PhD Degrees in Computer Science from Purdue University in 2003 and 2005, respectively. His main research area is database systems, uncertainty management and efficient execution of high-update streaming applications.

Sunil Prabhakar is an Associate Professor of Computer Sciences at Purdue University. He received the Bachelor of Technology in Electrical Engineering from the Indian Institute of Technology, Delhi in 1990, and MS and PhD in Computer Science from the University of California, Santa Barbara in 1998. His research interests are in uncertain databases, sensor and streams databases, data privacy, and biological databases.

Shan Lei is a Software Engineer at Google. He received his MS Degree from Purdue University in 2004. His research areas are distributed systems and mobile computing.

Rahul Shah is a Visiting Assistant Professor at the Department of Computer Science, Purdue University. He received BTech in Computer Science from the Indian Institute of Technology, Bombay in 1997. He obtained MS and PhD Degrees from Rutgers University in 1999 and 2002, respectively. His main areas of interest are algorithms and databases.

1 Introduction

The recent advancement of wireless communication technologies such as cellular network systems, Global Positioning System (GPS), sensor networks and RF-ID enable a wide class of emerging applications. Examples of these applications include location-based services, digital battlefields, telematics and weather monitoring, where data are continuously collected from various sources for querying, analysis and monitoring purposes. A typical feature of these applications is that a large amount of data has to be handled, and the rate of update arrival can be very high. This presents novel problems such as database querying, storage, indexing and mining support where the assumption of low data arrival rate is no longer correct.

In particular, the issue of data indexing needs to be revisited in such an 'unstable' environment. Indexing is a technique where by limiting the amount of data that needs to be examined, query performance is improved. Typically, dynamic indexes (such as R-tree and B-tree) are widely used in traditional database applications since they work well in environments where updates are infrequent in comparison to queries. However, these indexes are not optimised for data streaming applications that are characterised by numerous and frequent updates, which can incur high update overhead. In an R-tree, for example, a data update can trigger expensive node splitting operations. If updates are frequent, the update overhead can present a huge performance bottleneck. A new technique is thus needed in order to prevent a data structure from being changed too much due to frequent updates.

Our solution to the above problem is based on the observation that although the current value of the data might change all the time, its mean and variance are usually relatively stable. We thus propose the Mean Variance Tree (MVTree) which exploits the fact that with high probability the value of the constantly changing data stays within an interval represented by its mean and variance if the mean and variance are properly identified. An index that is built using these more stable mean and variance values instead of the rapidly changing exact values is likely to have a lower update cost. If the item remains within the indexed range, then fewer updates to the index will be required.

The rest of the paper is organised as follows. Section 2 discusses related work. Section 3 explains the index in detail, including index construction, update and query processing. Section 4 presents experimental evaluation of the proposed approach and Section 5 concludes the paper.

2 Related work

Developing an efficient index structure for constantly evolving data is an important database research topic. Most work in this area so far focuses on moving object environments. As a simple approach, multi-dimensional spatial index structures can be used for indexing the positions of moving objects, however, they are not efficient because of frequent update operations.

Many approaches describe a moving object's location by a linear function in order to reduce the number of updates, and only when the parameters of the function change, for example, when the moving object changes its speed or direction, the database is updated. Saltenis et al. (2000) proposed the Time-Parameterised R-tree (TPR-tree). In this scheme, the position of a moving point is represented by a reference position and a corresponding velocity vector. When splitting nodes, the TPR-tree considers both the positions of the moving points and their velocities. Later, Tao et al. (2003) presented the TPR*-tree, which extends the idea of TPR-trees by employing a different set of insertion and deletion algorithms in order to minimise the query cost. Tayeb et al. (1998) introduced the issue of indexing moving objects to query the present and future positions and proposed PMR-Quadtree for indexing moving objects. Kollios et al. (1999) employed dual transformation techniques that represent the position of an object moving in a d-dimensional space as a point in a $2d$-dimensional space and proposed an efficient indexing scheme using partition trees. Agarwal et al. (2000) proposed various schemes based on the duality and developed an efficient indexing scheme to answer approximate nearest-neighbour queries.

The problem with all the above techniques is that the movement of objects is rarely captured accurately by a simple function. In many applications, the movement of objects is complicated and non-linear. Our work differentiates itself by exploiting the nature and pattern of change in values while not imposing any restrictions on the change. In our previous work Cheng et al. (2005), we introduced the notion of Change Tolerant Indexing for the high update environments. A new index structure and algorithms are proposed for optimising both query

and update performance. In our work, no assumptions are made about the amount or rate of change in the data value and it is not necessary for data to change according to well-behaved patterns.

3 The MVTree index

In this section, we first discuss the motivation of the MVTree, then present the details of the MVTree construction, update and query processing.

3.1 Motivation

Traditional index structures for continuously changing data are affected by the huge update overhead that results in unacceptable performance. In this work, we aim to build indexes that are more stable and tolerant to data changes. We observe that for continuously changing data, although the data values change all the time, they usually fluctuate within certain intervals for a long period of time. For example, sensor data such as temperature is very likely to vary continuously, but not significantly, within a small interval for a relatively long time. Similarly, the positions of a large numbers of moving objects also tend to change within a small area such as within buildings over a long period. We propose to use a mean and variance to represent each data item and build an R-tree index for the continuously changing data based on these variance intervals or ranges. The data items are represented by points corresponding to their (mean, variance) coordinates, as shown in Figure 1. As long as the new data value remains within its current interval, no index update is needed. Only when the new value moves out of its current interval, its interval needs be updated, by adjusting the variance, for example. Otherwise we only need to update the corresponding point in the index structure. If the intervals are chosen well, index updates will not occur very often. Therefore, the MVTree is less susceptible to index change and reduces update overhead considerably. The structure of the MVTree is shown in Figures 2 and 3. It is a regular R-tree with a secondary index. Each entry in the index maps a data item to the page number in the R-tree that contains its value. An entry in the leaf nodes of the R-tree contains the current value of the data as well as its mean μ and variance δ. The data value is within the corresponding interval $[\mu - \delta, \mu + \delta]$. If data changes and the new value moves out of that interval, its μ and δ should be adjusted. However, the new value moving out of the current interval is not the only reason that triggers the adjustment of μ and δ, as we will discuss later. μ and δ can be dynamically adjusted to reflect changes in the pattern of the data change. Please note that although the MVTree is built based on the data intervals, it contains the current value of each data item and is an accurate index. The data intervals are only for the purpose of maximising change tolerance and reducing index update cost.

Figure 1 Map constantly changing data into 2-D means/variances points

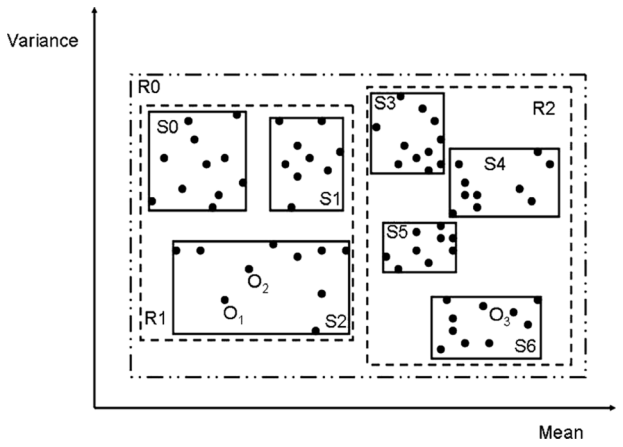


Figure 2 MVTree and secondary index structure

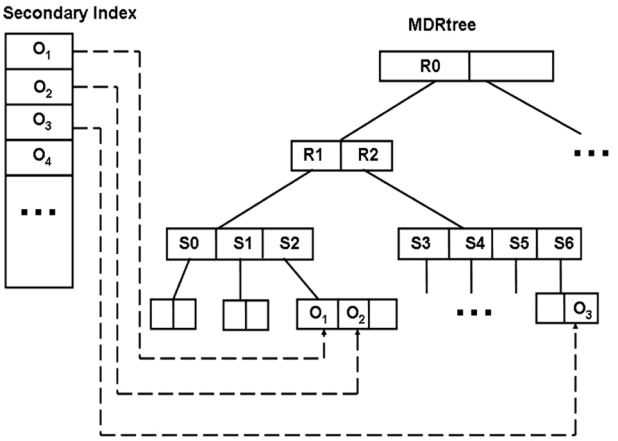
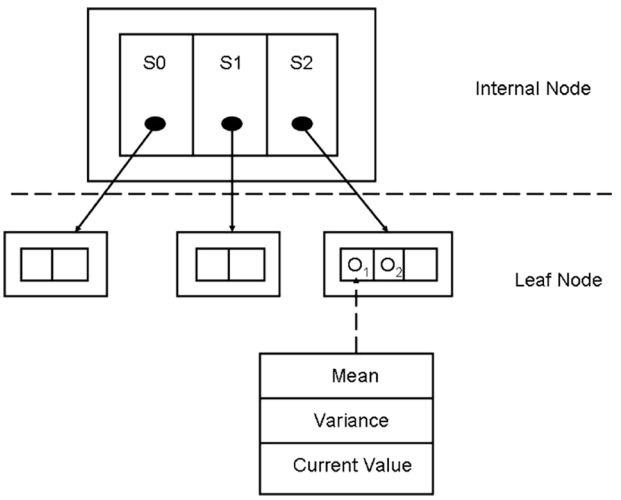


Figure 3 MVTree nodes



3.2 Index creation

The process of building an MVTree consists of two steps:

1 *Converting each data item, X, to a point* (μ_x, δ_x). When the MVTree is first created, the current value of each data is mapped to a point (mean, variance)

using which an R-tree index is created. The point (mean, variance) can be computed the historical values of the item. In case history data is unavailable, we can use the current value as the mean μ and an estimated variance δ_{initial}. The initial mean and variance are very likely to be inaccurate, but that is fine since we will dynamically adjust them later. Consequently, the index will be better tuned and give better performance as time goes on, while the performance of most existing indexes deteriorate as the data changes and need refreshing or rebuilding from time to time.

2 *Build an R-tree over all data points* (μ, δ). After all data are converted into points, an R-tree is built over them. This process is almost identical to the regular R-tree operations. The leaf node entry contains the current values of the item, and its corresponding mean and variance. For the purposes of the R-tree index minimum bounding rectangles, the item is treated as having an extent equal to $[\mu - \delta, \mu + \delta]$. As each item is inserted into the R-tree, an entry is also be inserted into the secondary index which contains the data ID and the R-tree page that the data is inserted into.

3.3 Mean and variance of the data

Since the MVTree is built based on the (mean, variance) of the changing data, it is important that these two parameters are properly identified and adjusted as the data change. Next, we show how to determine and adjust the mean and variance for each data item.

3.3.1 Variance tradeoff

Although representing data by an interval can reduce index update cost to a large extent, it affects the query performance negatively. Index structures improve query performance by limiting the amount of data that needs to be examined. For example, an R-tree index structure recursively clusters neighbouring data nodes into Minimum Bounding Rectangles (MBRs) . For a range query, if it does not overlap with the MBR of an internal node in the R-tree, then the subtree under that internal node is pruned since all the data in that subtree is within the MBR of that node, therefore, they cannot overlap with the range query. Since the MVTree is built based on data intervals, its MBRs are larger than one built based on actual data values therefore it might not be able to perform as much pruning resulting in worse query performance.

In general, an index based on the current value might result in an index structure update every time a data value changes, but it can more efficiently support query processing and has the best query performance. Indexes based on data intervals can reduce index updates since chances are that the data will remain within its interval. The larger the intervals are, the less likely the data values will move out of the intervals and the fewer updates need to be performed on the index structure (the data value in the leaf node still needs be updated). However, the query performance will be worse due to the looseness of the index, which leads to less pruning. If we find, for each data, an interval that is so large that it can always guarantee that the data value changes within it and build an index with these large intervals for each data, then the index structure never needs updating since it gives each data the largest change ranges or intervals they need. No matter how the data changes, they will not move out of the intervals that represent them, thus, the intervals do not need updating and neither does the index structure. However, the query performance of this index is likely to be the worst.

Therefore, there are two extremes in terms of updating the index for continuously changing data: always update it or never update it. Indexes based on the current values which always need updating are one extreme. This extreme has the best query performance and the largest update overhead. Indexes that use large intervals or ranges to represent each data are the other extreme, with the least update overhead, but the worst query performance. Usually, neither extreme is the best solution. We hope to find a point somewhere in the middle that has the best overall performance for both index updating and query processing. Different applications have different data update rates, query rates, and data change intervals and consequently, different optimal points. For example, applications with a high data update rate and a low query rate should use intervals that are relatively larger than applications with a low update rate and a high query rate. This optimal point should be determined with all these factors taken into consideration.

3.3.2 Determining the mean and variance

Assume that a constantly changing data item X has a set of historical values $X_1, X_2, \ldots, X_N$. Its mean is given by $\mu_x = \frac{1}{N}\sum_{k=1}^{N} X_k$. The sample variance may be computed as ${s_N}^2 = \frac{1}{N}\sum_{k=1}^{N}(X_k - \mu_x)^2$. Note that this sample variance is not an unbiased estimator for μ^2. In order to obtain an unbiased estimator for μ^2, it is necessary to define a bias-corrected sample variance ${s_{N-1}}^2 = \frac{1}{N-1}\sum_{k=1}^{N}(X_k - \mu_x)^2$. The square root of the bias-corrected variance is the standard deviation.

Before building the index, if a period of history is available, we may compute the mean μ_x and standard deviation σ_x for each data item x. Let $\delta = m\sigma_x$, we convert each data item to the corresponding point (μ_x, δ) and index these points. m is an adjustable parameter which can be tuned to achieve the best overall performance. We call m the *variance extension factor*. Different data items can have different values of m. m is adjusted when data changes to ensure that the new data value is always within the range $(\mu_x - m\delta, \mu + m\delta)$. If the history is unavailable, as mentioned earlier, we will use the current value as the mean μ_x and a σ_{initial} or σ_{default} as σ_x, and convert each data item to point $(\mu_x, m_{\text{default}}\,\sigma_x)$ to build the index.

After the index is built, as data continuously changes, new values for each data item are received. It is infeasible to store all the history values for all data. Therefore, we

only maintain a sliding window history for each data item. For example, the L most recent values $X_1, X_2, \ldots, X_L$ for X. We store the history window for each data as a queue. When the new value X_{L+1} arrives, it is inserted at the head of the queue and the oldest value X_1 is deleted. If X_L is different from X_1 , we will recompute the new mean μ_{new} and standard deviation σ_{new} based on these L most recent values again. To avoid excessive updates, we do not update the old mean and variance $(\mu_{\text{old}}, m\sigma_{\text{old}})$ that are used in the index with the new ones $(\mu_{\text{new}}, m\sigma_{\text{new}})$ all the time . Only when $(\mu_{\text{new}}, m\sigma_{\text{new}})$ is different from the $(\mu_{\text{old}}, m\sigma_{\text{old}})$ by certain thresholds (T_{mean} and $T_{\text{deviation}}$ respectively), do we update $(\mu_{\text{old}}, m\sigma_{\text{old}})$ with $(\mu_{\text{new}}, m\sigma_{\text{new}})$. By doing this, we avoid unnecessary frequent updates to the index and ensure that the mean and variance in the index contain the latest patterns of the data.

Since the more recent data values indicate the latest data changes, it is reasonable to assign higher weights to the more recent values. Suppose we assign a weight $W_i (i = 1, \ldots, L)$ to each historical value X_i, then

$$\mu_x = \frac{1}{L} \sum_{i=1}^{L} W_i \cdot X_i$$

and

$$s_{L-1} = \sqrt{\frac{1}{L-1} \sum_{i=1}^{L} W_i \cdot (X_i - \mu_x)^2}.$$

Here $\sum_{i=1}^{L} W_i = L$. For example, if we assign each weight to be P times as much as the previous one, then W_i should be

$$W_i = \frac{P^{i-1}(P-1)L}{P^L - 1}, \quad i = 1, \ldots, L.$$

The mean and standard deviation should be:

$$\mu_x = \frac{1}{L} \sum_{i=1}^{L} \frac{P^{i-1}(P-1)LX_i}{P^L - 1}$$

$$s_{L-1} = \sqrt{\frac{1}{L-1} \sum_{i=1}^{L} \frac{P^{i-1}(P-1)L(X_i - \mu_x)^2}{P^L - 1}}.$$

3.3.3 *Dynamically adjusting the mean and variance*

As the data change, whenever a new data value falls outside the interval represented by its mean and variance, we enlarge its variance to contain the new value. Consequently, the variance δ for each data item used in the index can become large, and the pruning effect of the index degrade. As mentioned earlier, there are two conflicting criteria for the variance: on the one hand, we should use as large a variance as possible to avoid updates to the index structure; on the other hand, the variance should be as small as possible so that they represent data more precisely and the index can better prune during query processing. In order to maintain good query performance of the index, the δ for each data item should be reduced from time to time.

When shrinking the variance, different data item can be treated differently. For example, consider two items A and B that both have a high variance. Item A has a high variance because it actually changes significantly, while B has a high variance from one outlier value, which happened a long time ago. Obviously, δ_B should be reduced much more than δ_A. This can be achieved by using the approach of dynamically determining the mean and variance according to the recent history window. The reason is that for A which changes significantly, the variance computed based on the history window is large too, while for B with one extreme value a long time ago, either the extreme value has shifted out of the history window, thus will not affect the variance any more; or even if it is still within the window, it has a very low weight since it is old and will not affect the variance much. As a result, δ_B will drop significantly.

The frequency of adjusting the mean and variance is an important issue. They should not be adjusted too frequently, which results in extra update overhead to the index. However, they should not be adjusted too rarely, either. Both the mean and variance should be adjusted in a timely manner so that they represent the current status and pattern of change of the data. If the mean is out-dated and drifts far from the real mean value, the variance has to be unnecessarily large to contain the data changes, and an unnecessarily large variance degrades query performance. In our scheme, the frequency of adjustment essentially depends on two threshold values: T_{mean} and $T_{\text{deviation}}$. The larger the threshold values, the less frequent the adjustment. Consequently, we need to determine appropriate thresholds based on the data and queries. Please note that the adjustment to the mean and variance need not to be done separately. They are performed 'on-the-fly' when updating the corresponding data values.

In summary, the mean and variance in the index for each data item are adjusted in two cases:

- when the new data value is out of the interval $[\mu - \delta, \mu + \delta]$
- when the newly computed mean or variance differ from the old ones by more than the respective threshold.

The first case is for the correctness of the index while the second is for the accuracy and efficiency of the index.

3.4 *Data update*

When the data changes and a new value is received, first, its corresponding page is found via the secondary index and its current value in that page is changed to the new value. The new mean and deviation for the data will be computed taking the latest value into account. Assume the old mean and deviation are μ_{old} and σ_{old} and the new mean and

deviation are μ_{new} and σ_{new}. μ_{new} and σ_{new} are compared with μ_{old} and σ_{old} and processed as follows:

1 If μ_{new} and σ_{new} are not different from μ_{old} and σ_{old} by more than the thresholds, then the latest data value is checked to see if it still falls in the interval $[\mu_{\text{old}}, \delta_{\text{old}}]$. If it does, no update needs to be done on the index structure at all. The current value is updated in the leaf by following the pointers from the secondary index structure. As discussed earlier, most data changes fall in this category, therefore, the index structure update cost is greatly reduced. In case the new value lies outside the interval $[\mu_{\text{old}}, \delta_{\text{old}}]$, since $\delta_{\text{old}} = m\sigma_{\text{old}}$, m should be increased till the interval contains the latest value. Assume that m is increased to m_{new}. The point $[\mu_{\text{old}}, \delta_{\text{old}}]$ in the index should be updated with the new point $[\mu_{\text{old}}, \delta_{\text{new}}]$, in which $\delta_{\text{new}} = m_{\text{new}}\sigma_{\text{old}}$. Again, the index will be updated in a lazy fashion.

2 If μ_{new} is different from μ_{old} by more than T_{mean}, or σ_{new} is different from σ_{old} by more than $T_{\text{deviation}}$, then update the point $(\mu_{\text{old}}, \delta_{\text{old}})$ with the new point $(\mu_{\text{new}}, \delta_{\text{new}})$. Set δ_{new} to be $m\sigma_{\text{new}}$, m is set to the default value m_{default}, or if the new value does not fall in the interval $[\mu_{\text{new}}, m_{\text{default}}\sigma_{\text{new}}]$, m is increased just enough to contain the new value. The index structure will be updated in a lazy manner (Kwon et al., 2002), that is, if the point $(\mu_{\text{new}}, \delta_{\text{new}})$ is still within the current MBR of $(\mu_{\text{old}}, \delta_{\text{old}})$, then just update $(\mu_{\text{old}}, \delta_{\text{old}})$ to $(\mu_{\text{new}}, \delta_{\text{new}})$ and no further update needs to be done. Only in the case that the new point $(\mu_{\text{new}}, \delta_{\text{new}})$ is out of the current MBR, the old point $(\mu_{\text{old}}, \delta_{\text{old}})$ should be deleted and the new point $(\mu_{\text{new}}, \delta_{\text{new}})$ inserted into the index.

3.5 *Query processing*

Range queries are one of the most common queries in databases. We now explain how the MVTree index supports range queries. A range query (a, b) searches for all data items whose values fall within the interval (a, b). To process a range query, we first check the query (a, b) against the MBRs of the nodes in this tree. A node N is pruned when it is guaranteed that no item in the subtree rooted at N can satisfy (a, b). Let μ_1, μ_2 be the lowest and highest values of mean over all objects in the subtree of N and let δ_1, δ_2 be the lowest and highest values of variance over all the objects in the subtree. Note that the MBR for N is [(μ_1, δ_1): (μ_2, δ_2)]. Let $L = \mu_1 - \delta_2$ and $R = \mu_2 + \delta_2$. If the query (a, b) does not overlap with (L, R) we can say the no data items in the subtree of N overlap with query (a, b) and the subtree of N can be pruned. The reason for the pruning is that for every data item d in the subtree of N, its mean $\mu_d \geq \mu_1$ and its variance $\delta_d \leq \delta_2$, therefore, its lowest possible value $L_d = (\mu_d - \delta_d) \geq (\mu_1 - \delta_2)$, which is L. That means $L_d \geq L$. Similarly, the mean of the every data item $\mu_d \leq \mu_2$ and its variance $\delta_d \leq \delta_2$, therefore, its highest possible value is $H_d = (\mu_d + \delta_2) \leq (\mu_2 + \delta_2)$, which is H. This indicates $H_d \leq H$. Since $L_d \geq L$ and $H_d \leq H$, (L_d, H_d) is contained in (L, R); if (L, R) does not overlap with (a, b), (L_d, H_d) can not overlap with (a, b). As a result, no data items in the subtree of N can fall in (a, b) and satisfy the range query condition (Cheng et al., 2004).

The same pruning approach can be used for other queries such as point queries. For nodes that can not be pruned, if it is a leaf node, all data items contained in the node are compared with the query; if it is an internal node, its child nodes will be read one by one and this query process will run recursively.

4 Experimental results

We perform a set of experiments on the performance of the MVTree. We compare the performance of MVTree with two variants of the R-tree. Studies of the sensitivity of the MVTree to various parameters are also conducted. Below, we discuss the simulation model and the experimental results.

4.1 *Simulation model*

Our experimental data are based upon continuously changing position data generated by the City Simulator 2.0 developed at IBM (Kaufman et al., 2002). The City Simulator simulates the realistic motion of up to 1 million (N_{obj}) people moving in a city. The simulator records the location updates of each object in a trace file which contains the timestamp of the update and the spatial coordinates of the object at that time. For our experiment, we used only one dimension of the spatial coordinates.

We build the MVTree based on the first position data of each object. Once the MVTree is built, the remaining N_{update} samples are modelled as dynamic updates to the MVTree, as well as other R-tree variants. Since these are disk-based index structures, the number of page I/Os is the natural metric for measuring the performance of the indexes. We measure the number of page I/Os for reads and writes of both dynamic updates and queries during the simulation. As is common practice, we assume that the first two levels of the indexes reside in the main memory. Each page has a size of S_{page}, with a fan-out of N_{entry}. The secondary index of the Lazy R-tree and the MVTree is also assumed to be in the main memory. Table 1 summarises the parameters.

Table 1 Parameters and baseline values

Param	*Default*	*Meaning*
	Simulation parameters	
N_{obj}	100,000	# of moving objects
N_{update}	20	# of online updates (per obj.)
N_q	1000	# of queries
	MVTree parameters	
N_{entry}	20	# of entries (per page)
m_{default}	3	Default variance extension
T_{Mean}	1.0	Threshold for mean
$T_{\text{Deviation}}$	1.0	Threshold for deviation

4.2 Index performance

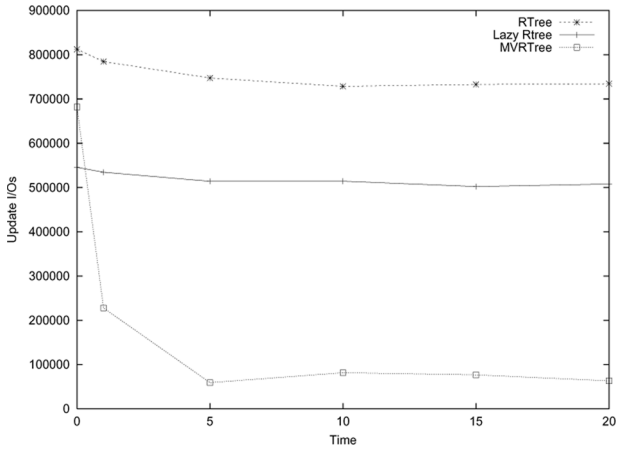
Here we present the index performance results. Three index structures are evaluated in our experiments:

- the traditional R-tree (Hadjieleftheriou, 2003)
- the traditional R-tree augmented with lazy updating using the secondary index structure. We call this the *lazy*-R-tree
- the MVTree. Since there are no constraints on the nature of changes to the data, the data does not change in any well-behaved pattern, consequently, the TPR-/TPR*- trees are inapplicable in these scenarios.

4.2.1 Update overhead

We begin by studying the relative index update performance of the various index structures. Figure 4 shows the number of page I/Os performed for update for the R-tree, the *lazy* R-tree, and the MVTree. Both the traditional R-tree and the lazy R-tree are built based on the actual data values. During each cycle, one tenth of the total 100 K objects change in value.

Figure 4 Update performance

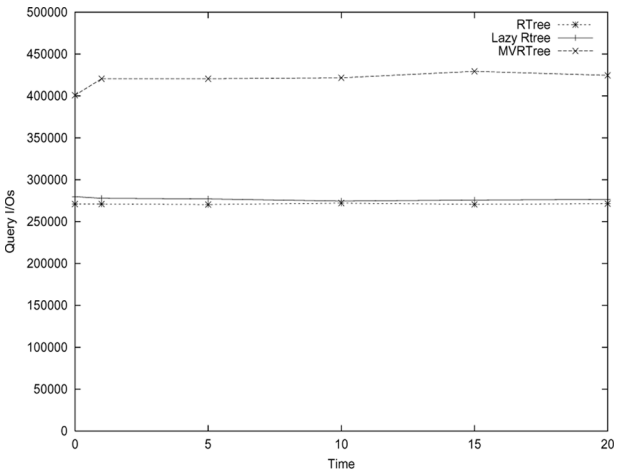


As Figure 4 shows, the traditional R-tree always has the highest update I/O cost. The *lazy*-R-tree reduces the cost by almost one third. This is because the traditional R-tree handles data changes by deleting the data with the old value and inserting one with the new value. While the *lazy*-R-tree uses a secondary index on the data ID, which can help locate the data in the index with only one I/O. Furthermore, when the new data value remains within its current MBR, only the data entry value needs to be changed and no further update needs to be done to the index. Only when the new value is out of the range of its current MBR, the old value is deleted and the new value is inserted. The use of the secondary structure in the *lazy*-R-tree gives it a minor edge over the traditional R-tree since it saves the cost of accessing the R-tree when an updated object remains inside the same leaf node. Therefore, the *lazy*-R-tree helps reduce update I/Os to some extent. The MVTree almost always has the lowest update cost except during the first few cycles. The reason is that in our experiment, we did not make the assumption that any history data is available. When first building the index, we used the default variances, which are usually small. Consequently, during the first several cycles, the true variances are larger than the default variances, which results in many updates to the index. After a few cycles, variances are enlarged and most data are more likely to change within their variances. Thus the number of updates to the index drops dramatically. The update cost is only 1/8 that of the *lazy* R-tree and 1/12 that of the traditional R-tree.

4.2.2 Query and overall performance

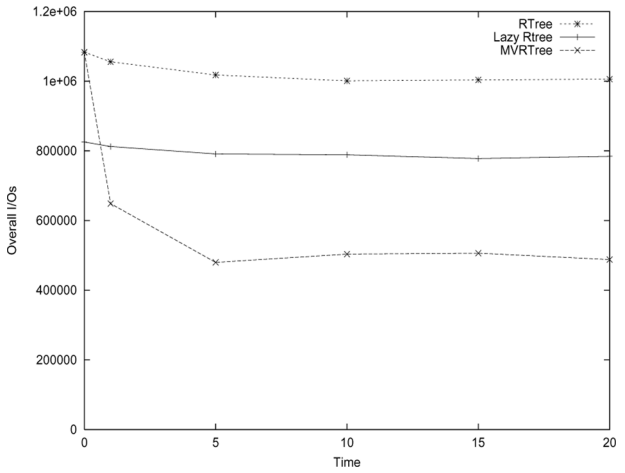
Although the MVTree contains the current data value and is an accurate index, its internal nodes are built based on data intervals and have larger MBRs than the one based only on actual data values. Therefore, the query region potentially has more overlap with the MBRs of the MVTree. This results in less pruning and worse query performance. In this experiment, we examine how the traditional R-tree and the lazy R-tree perform with respect to the MVTree. During each query evaluation cycle, 1000 queries are evaluated. Figure 5 shows the query I/Os for each cycle. The *lazy* R-tree and traditional R-tree have almost identical query performance, while the MVTree requires more I/Os.

Figure 5 Query performance



Although the MVTree does not perform as well as the other two indexes in terms of query cost, we can see from Figure 6 that it is the winner in terms of overall performance, which is the total number of I/Os for both update and query. The MVTree is designed for databases with frequent updates. Its loss in query performance is compensated by a significant gain in update performance, resulting in more than a two-fold improvement over the lazy R-tree and traditional R-tree. As we will discuss later, the higher the update frequency, the more significant the advantage of the MVTree.

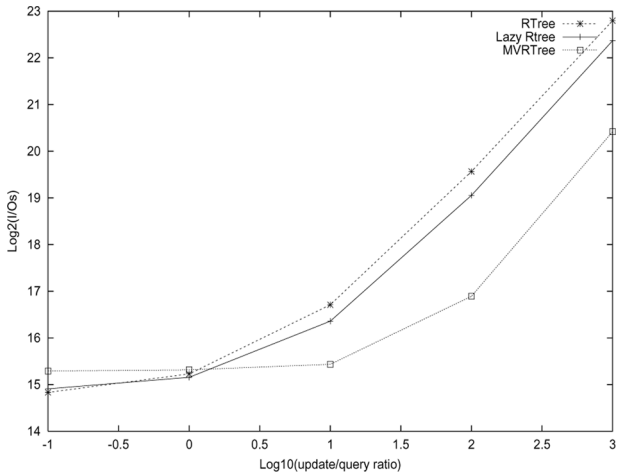
Figure 6 Overall performance



4.3 *Update frequency*

We study the relative performance of the various index structures as the relative numbers of queries and updates is varied. Figure 7 shows the total number of page I/Os performed for query and update for the R-tree, the *lazy* R-tree, and the MVTree. The performance is measured under the same query generation rate but different update arrival rates. It should be noted that this graph uses a Log-scale on both axes. As the ratio of update rate to the query rate (abbreviated as update/query ratio) is increased from 10^{-2} to 10^{3}, all three indexes show an increase in the number of I/Os. This is because increasing the update rate implies more demands on the index, and consequently more I/Os are needed. When the update/query ratio is low, the MVTree takes about 30% more I/Os than the other R-tree variants. The reason is that the R-tree and the *lazy* R-tree uses actual data values, while the MVTree employs data intervals and results in worse query performance.

Figure 7 Disk I/O vs. update/query ratio



Towards the right end of the graph, when the update workload dominates the query workload, the MVTree registers a significant improvement over other R-tree variants. In fact, the number of I/Os needed by all three R-trees increases sharply, whereas the MVTree gracefully handles the high update burden. When updates are much more frequent than queries, which is a typical scenario in sensor and moving object environments, the R-tree suffers from expensive updates. The distinction between the R-tree and the *lazy*-R-tree begins to show in this high update setting as the secondary index yields significant gains from cheaper updates. The MVTree clearly outperforms the other indexes in this high update environment since its structure is inherently designed to maximise tolerance to changes in object values. The advantage of better update performance more than compensates for the slightly poorer query performance. The gains come from identifying data intervals represented by $[\mu - \delta, \mu + \delta]$. The other index structures fail to capitalise on these regions and build indexes based upon current object values with no regard to expected updates. Thus for these indexes objects moving within the intervals constantly switch from one MBR to another, resulting in expensive updates and possibly costly splits.

As the update/query ratio increases, the improvement over R-trees is more obvious. In particular, when the update/query ratio is 1000, the number of I/Os required by the MVTree is only 1/4 that of the *lazy* R-tree, and 1/5 that of the R-tree.

4.4 *Effect of the variance extension*

As discussed earlier, the variance extension is a tradeoff factor for the index. In this set of experiments, we study how the extension of the variance affects the index update and query performance. We examined the performance of the MVTree with the default variance extension varied from 1 to 5, which means the default variance for each data varies from its standard deviation σ to 5σ. Please note that m is only a default variance extension. When the data changes out of the range of the $(\mu - m\sigma, \mu + m\sigma)$, m will be discarded and the variance will be enlarged to so that the new value is still within its interval. Experiments were conducted with different data update frequencies.

Figure 8 shows the index update cost, query cost and total cost for different default variance extensions when the Update/Query ratio is 10. The figures shows that when the default variance extension is small, the query performance is good, while the index update overhead is high. As the default variance extension grows, the index updating cost decreases while the query cost increases. The reason is that with a larger default variance extension m, the default interval for each data item, $[\mu - m\sigma, \mu + m\sigma]$ is larger, therefore, the value is less likely to move out of the current interval which reduces updates to the index. However, with a large interval representing each data, the MBRs in the index are also larger, which results in greater overlaps with the query and less pruning. The upper curve in Figure 8 shows the overall I/Os for different default variance extensions by adding up the index update and query cost. It has a minimum point corresponding to $m = 2$, which represents a default variance of 2σ.

The experiment was repeated with update/query ratio set to 100. The results are shown in Figure 9. Here we see

similar curves which again shows that a large default variance reduces index update overhead but degrades query performance, while a small default variance is good for query, but produces a large index update cost. However, since the update frequency in this experiment is higher than the previous one, the update cost is dominant. The upper curve in the figure is the overall cost. When $m = 1$, the total number of I/Os is more than 200 K. As m gets larger, the total I/Os keep dropping and reach the lowest point of 94,720 when m is around 4. After that, the cost starts growing slowly, and when m is 5, the number of the total I/Os is 96572.

Figure 8 Query I/O vs. query size

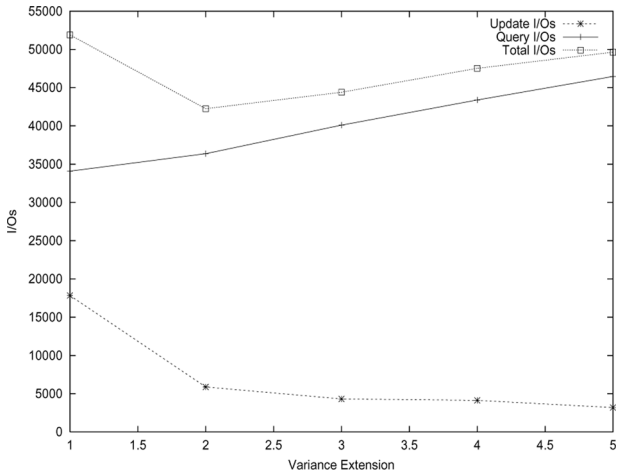
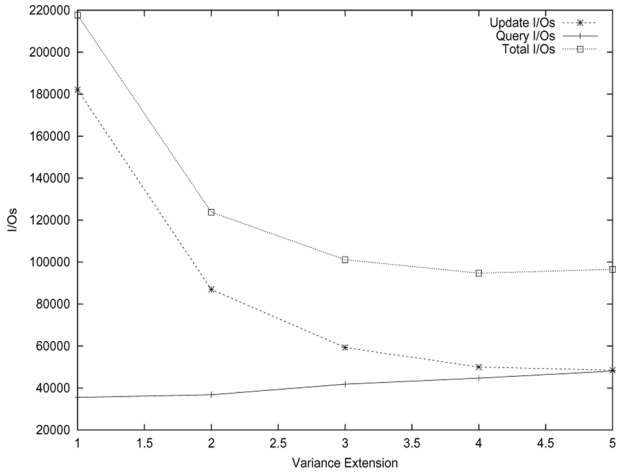


Figure 9 Disk I/Os vs. variance extension (update/query ratio = 10)



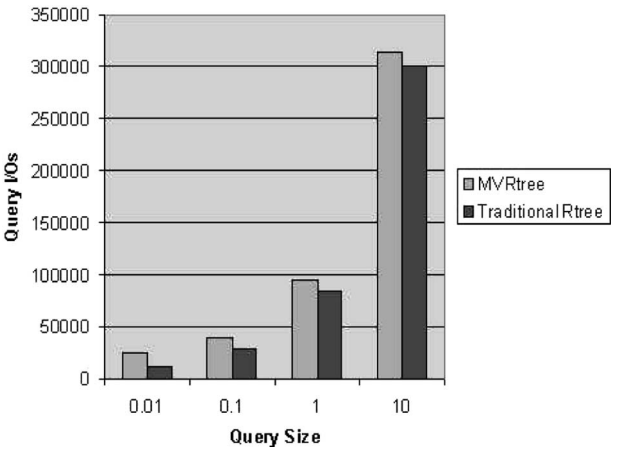
In general, both curves show that the index performance is sensitive to the default variance extension m. Either too large or too small a default variance extension results in poor performance. When m is properly selected, the overall I/Os of the MVTree can reach the optimal lowest point. However, m largely depends on the update/query ratio. When the update/query ratio is low, m should be small since the query performance is dominant in this situation and a small variance extension is efficient for querying. For traditional databases where the update/query is very low, the variance extension should be small. That explains why the traditional R-tree with zero variance extension works well in that scenario. However, as the update/query ratio increases, the update cost becomes more influential, m should also grow since a large variance extension helps reduce index update overhead.

4.5 Effect of query size

In this experiment, we study how query size affects the query performance of the MVTree and traditional R-tree. Note that since the *lazy* R-tree and the traditional R-tree have almost identical query performance, here, we compare the query cost of MVTree with only the traditional R-tree.

Figure 10 shows the query cost for the MVTree and traditional R-tree over different query sizes. The query size is varied from 0.01–10% of the domain. We observe that the MVTree always requires more query I/Os than the traditional R-tree. However, as the query size increases, the performance of MVTree starts to converge to that of the R-tree. The reason is that with a large query area, the probability that a given region will be covered by a query increases. Thus the advantage of having a smaller area MBR reduces. To see this, consider a very large query that covers 95% of the space – it is highly likely that most MBRs will overlap with this query and therefore need to be searched. Thus the advantage of having a small MBR is diminished with larger queries.

Figure 10 Disk I/Os vs. variance extension (update/query ratio = 100)



When the query size is 0.01% of the domain, the traditional R-tree takes only 50% of the query I/Os of the MVTree. This difference between them keeps dropping as queries get larger. When the query size reaches 10% of the space, the query I/Os for traditional R-tree is more than 95% of that for the MVTree.

5 Conclusions

For sensor or moving object applications where data is constantly evolving, traditional index structures give

poor performance since they are optimised for query performance in the presence of less frequent updates. We introduced the MVTree for this high update environment. Since the mean and variance are more stable compared to the data values, the MVTree significantly reduces the index update cost. We developed algorithms for the creation and maintenance of the MVTree, and dynamic adjustment of mean and the variance for the index. Experimental results showed the superior performance of the proposed index structure. The MVTree trades slightly worse query performance for much better overall performance. We observe that the variance extension that should be chosen in order to achieve optimal overall performance is largely dependent on the update/query ratio. In future work, we will develop a mathematical model to determine the optimal variance extension and optimise the query processing for MVTree.

References

Agarwal, P.K., Arge, L. and Erickson, J. (2000) 'Indexing moving points', *Proceedings of the ACM Symposim on Principles of Database Systems (PODS)*, pp.175–186. AUTHOR PLEASE SUPPLY LOCATION.

Cheng, R., Xia, Y., Prabhakar, S. and Shah, R. (2005) 'Change tolerant indexing for constantly evolving data', *Proceedings of the International Conference on Data Engineering (ICDE)*, pp.391–402.

Cheng, R., Xia, Y., Prabhakar, S., Shah, R. and Vitter, J.S. (2004) 'Efficient indexing methods for probabilistic threshold queries over uncertain data', *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp.876–887. **AUTHOR PLEASE SUPPLY LOCATION.**

Hadjieleftheriou, M. (2003) *Spatial Index Library Version 0.44.2b Java*', http://u-foria.org/marioh/

Kaufman, J., Myllymaki, J. and Jackson, J. (2002) *IBM City Simulator 2.0*. AUTHOR PLEASE SUPPLY FULL DETAILS.

Kollios, G., Gunopulos, D. and Tsotras, V.J. (1999) 'On indexing mobile objects', *Proceedings of the ACM Symposim on Principles of Database Systems (PODS)*, pp.261–272.

Kwon, D., Lee, S.J. and Lee, S. (2002) 'Indexing the current positions of moving objects using the lazy update R-tree', *Proceedings of the Conference on Mobile Data Management (MDM)*, pp.113–120. AUTHOR PLEASE SUPPLY LOCATION.

Saltenis, S., Jensen, C., Leutenegger, S. and Lopez, M. (2000) 'Indexing the position of continuously moving objects', *Proceedings of the ACM SIGMOD Internation Conference on Management of Data (SIGMOD)*, pp.261–272. AUTHOR PLEASE SUPPLY LOCATION.

Tao, Y., Papadias, D. and Sun, J. (2003) 'The TPR*-tree: optimized spatio-temporal access method for predictive queries', *Proceedings of the Internation Conference on Very Large Data Bases (VLDB)*, pp.790–802. AUTHOR PLEASE SUPPLY LOCATION.

Tayeb, J., Ulusoy, O. and Wolfson, O. (1998) 'A Quadtree-based dynamic attribute indexing method', *The Computer Journal*, pp.185–200. AUTHOR PLEASE SUPPLY LOCATION.

Query

AQ1: AUTHOR PLEASE CHECK IF ALL FIGURES AND ITS CITATION IS OK.