2004

Change Tolerant Indexing for Constantly Evolving Data

Reynold Cheng

Yuni Xia

Sunil Prabhakar
Purdue University, sunil@cs.purdue.edu

Rahul Shah

Report Number:
04-006

CHANGE TOLERANT INDEXING FOR CONSTANTLY EVOLVING DATA

Reynold Cheng
Yuni Xia
Sunil Prabhakar
Rahul Shah

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #04-006
February 2004
(Revised July 2004)

Change Tolerant Indexing for Constantly Evolving Data
Technical Report CSD TR# 04-006

Reynold Cheng Yuni Xia Sunil Prabhakar Rahul Shah
Department of Computer Science, Purdue University.
West Lafayette
IN 47907-1398, USA
Email: {ckcheng,xia,sunil,rahul}@cs.purdue.edu

Abstract

Index structures are designed to optimize search performance, while at the same time supporting efficient data updates. Although not explicit, existing index structures are typically based upon the assumption that the rate of updates will be small compared to the rate of querying. This assumption is not valid in streaming data environments such as sensor and moving object databases, where updates are received incessantly. In fact, for many applications, the rate of updates may well exceed the rate of querying. In such environments, index structures suffer from poor performance due to the large overhead of keeping the index updated with the latest data, move in a well behaved, but restrictive manner (e.g. in straight lines with constant velocity). In this paper, we propose and develop an index structure that is explicitly designed to perform well for both querying and updating. We present techniques for altering the design of an index in order to optimize for both updates and querying. The paper is developed with the example of R-trees, but the ideas can be extended to other index structures as well. We present the design of the Change Tolerant *R-tree, an experimental evaluation.*

1 Introduction

Index structures are used to improve query performance by limiting the amount of data that needs to be examined in order to generate an answer. Static index structures such as the ISAM file format [11] are not designed to handle updates to the data very well and can lead to poor query performance as a result of updates. Dynamic index structures, such as the B-tree and R-tree, are designed to adapt the index structure as data is updated so as to continue to provide good query performance. Existing (dynamic) index structures perform satisfactorily for traditional database applications where updates are infrequent in comparison to queries.

Emerging applications such as sensor-based streaming databases, represent a drastic shift from this traditional behavior. These applications are characterized by virtually constant updates to the data, and relatively infrequent querying. In this setting, existing index structures are compelled to expend large amounts of resources in simply keeping the index updated with the latest values of the data. The cost of updating the index dominates the advantage of improved query performance through the use of the index. One feasible solution is to reduce the need for updates to the index. Recent efforts at indexing moving object data reduce the need for index updates by assuming that objects will move in a well behaved, but restrictive manner (e.g. in straight lines with constant velocity) [12]. This solution is not generally applicable since the assumption is not reasonable for many applications.

In this paper, we address the problem of efficient index update where update rates are high. We drop the traditional approach of processing updates with the goal of improved query performance. Instead, we propose and develop index structures that are explicitly designed to perform well for both querying and updating. We begin by observing that most index structures inherently tolerate some change in the data values being indexed. The first step is therefore to exploit this "tolerance" to avoid an index (without making any restrictions on the nature of change of the data). Next, we present techniques for altering the design of the index in order to optimize for both updates and queries. This is achieved by balancing the need for efficient search (the common criterion for index design) with the cost of updates.

As we shall see, the two goals of improved query performance and improved update performance are directly opposed to each other: improving update performance is typically at the cost of query performance (and vice versa). The paper presents an index structure that is designed for

high update environments – achieving significantly better update performance at the cost of slightly poorer query performance – and superior overall performance as compared to existing methods. The paper is developed with the example of R-trees, but the ideas can be extended to other index structures as well.

The main contributions of this paper are:

1. The introduction of *Change Tolerant* index structures that optimize for frequent updates and queries and the design and development of change tolerant R-trees.
2. An experimental evaluation and validation of the performance, and adaptability of these index structures.

The rest of this paper is organized as follows. In Section 2 we discuss the inherent tolerance of index structures to updates and study how this can be exploited to avoid index updates. In Section 3 the design of a change tolerant R-tree is discussed. Section 4 presents experimental results. Section 5 discusses related work and Section 6 concludes the paper.

2 Change Tolerance of Indexes

The main motivation for our change tolerant indexes comes from data which changes slowly but constantly with respect to time for most periods of time, followed by short periods of time when the data may show a major variation. In nature (e.g., weather systems), these major variations are likely to be caused by some underlying events, which are relatively infrequent.

Consider an index over people in a city. For most of the time a large fraction of these people are inside a building. They may change their locations but these variations are not big. They are confined to limited range of space for a long time. Then, sometimes, when they are on the road, the changes in their locations are rapid. However, this happens for relatively shorter periods of time for most people.

The situation can also be extended to sensor data. Consider temperature and pressure sensors. The index contains temperature and pressure values of many different places. For each place, the variation in these parameters against time is not rapid for most of the time. However, during evenings or during special events like thunderstorms, they can change rapidly. They finally settle around their new values.

We can exploit this property of changing data to build better indexes. In some of the models for changing data, the data variations are modeled as a smooth straight line with constant rate of change. For example, indexes based on kinetic data structures [5] assume mobility of objects in straight lines with some velocity. Our model does not assume data changes are well behaved. The changes are random, but they are restricted in small range of values and in only a few moments rapid changes occur. The rapid changes are followed by another set of small changes – again the changes are confined and random.

2.1 Tolerance to Change

Many index structures are inherently tolerant to the changes in data values without requiring a change in the index structure. Consider the case of an R-tree index [7]. The R-tree is a height balanced tree which can be seen as a generalization of the B-tree for indexing objects in multidimensional space. Each node of the R-tree (internal as well as leaf node) represents a hyper-rectangle in d dimensions. The leaf level rectangles contain objects, and the rectangles for internal nodes contain the rectangles at one level below. The boundaries of the rectangles are made as tight as possible. There is an object on each boundary face (hyperplane in d dimensions) of each of these rectangles. These rectangles are called *Minimum Bounding Rectangles* or MBRs. Unlike the B-Tree, the MBRs of nodes at the same level in an R-Tree are allowed to overlap. Hence searching an object may involve traversing several paths in this tree. When a node becomes overfull it undergoes a split. Efficient heuristics and pruning are used to reduce the expected number of paths visited by subsequent searches.

Given any specific entry in a leaf node of the tree, the Minimum Bounding Rectangle (MBR) of the entry for that leaf node in its parent node represents the "tolerance" of the index to changes in the values of the objects pointed to by the leaf node. In particular, if an object's location remains within this MBR, the index is correct without requiring an update. Under normal R-tree operations, such an update is processed by searching the index and updating the location of the object. In order to avoid this expense for each update, it is desirable to be able to perform a cheap update in cases where the index does not change.

The R-tree is very often used as an index on spatial coordinates. Typical updates on R-trees are insertions and deletions. While performing a deletion operation on the space attribute, the object is first searched (based upon its spatial coordinates) and then deleted. However, if the deletion operation directly provides a pointer to the page in which the object is stored, then the cost for searching in the R-tree can be saved. For example, if a deletion is by a different (non-spatial) attribute, say object identifier (*id*), we can maintain a secondary index on *id*. This secondary index stores, for each *id*, the pointer to the page containing the corresponding object in the R-tree.

When the R-tree is used to index constantly evolving data such as the locations of mobile objects, the types and the frequencies of the updates can be very different. For example, most updates can be of the form—object with id i moves from its current location (x_1, y_1) to new location (x_2, y_2). This can be handled in an R-tree by first deleting

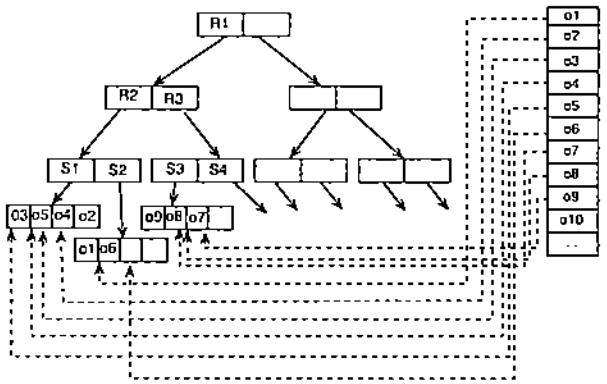


Figure 1. Secondary hash-index structure

this object from its current location and then re-inserting it in the new location. However, if the new location is in the same MBR, the change tolerant property of the R-tree can be exploited. Additionally, the secondary index on *id* can be used to reduce the search cost associated with deletion and insertion.

Hence, in conjunction with the R-tree, we maintain a secondary hash index on *id* for handling updates. This is the basic idea applied in the lazy-update R-trees [10]. Figure 1 shows an example of this secondary index structure. The secondary index (on the right) is simply an array of pointers to leaf pages of the R-tree with one entry for each object ordered by *id*. Thus, all the updates where the new location is in the same MBR as the old location can be accomplished with a constant number of I/Os. Note that the R-tree structure does not change due to such updates (only the location of the updated object is changed in the corresponding leaf node). This kind of secondary structure is essential when updates are frequent. If most objects remain within their MBRs, most updates can be handled through the secondary hash index while the R-tree index is used to process spatial queries.

2.2 Optimizing for Updates

In the previous subsection, we saw that the available tolerance of an index to data change can be used to improve update performance with no impact on search performance. In this section we explore the possibility of altering the design of the index structure to increase the available tolerance of an index while balancing the potential increase in the cost for querying. Again, we focus on R-trees as the running example.

Given a set of data, the structure of an R-tree index for this data is determined by two critical parameters: the node size, and the order of inserts and deletes. The node size is chosen to be a multiple of disk blocks. The structure that results is largely determined by split of an overfull node into two nodes. The R-tree (like other index structures) attempts to find a split of the children of the overfull node in order to achieve balance (each of the split nodes has roughly the same number of children), and improve search performance. It is assumed that the area of the resulting MBR of each child is proportional to the number of queries that will access the corresponding node. Consequently, the goal is to minimize this area. Other structures such as R*-trees use a slightly more complicated decision process to determine the split, but with the same goal of minimizing the expected number of queries that will intersect with the resulting nodes. In either case, the impact of the split on future updates is not taken into account. For example, the split may result in a situation wherein objects frequently cross from one MBR to another – thereby resulting in a high update cost.

In the traditional R-tree, the MBR is tight (i.e. it is the smallest rectangle that contains all underlying objects). This implies that there is at least one object touching each side of the MBR (otherwise it would shrink further). Having a small MBR improves search performance and pruning. In situations where the objects move constantly, these boundary objects are likely to move in and out of the MBR very frequently. Each time an object leaves the MBR, it has to be re-inserted (either into a different MBR or stays in the same MBR after expansion). Note that the use of lazy updating through the secondary index discussed above does not eliminate this cost. Thus, MBR boundaries being tight to the objects improves the search performance but can result in a high update cost. The concept of having slightly larger MBRs than needed (that is, the MBR is no longer a *minimum* bounding rectangle) is explored in [10]. We shall call this structure the α-tree, which is essentially an R-tree with "loose" MBRs. The idea is that whenever an MBR needs to be expanded, we expand it by α% more than its minimum size. Thus, the boundary objects get some leeway to move and stay within the same MBR. Naturally, this implies poorer query performance.

The intuition behind these indexes is as follows: The design of the MBRs of the index should not be governed solely by the current values of the data being indexed. Instead, the MBRs should be designed based upon the nature of changes to data values. For example, if changes from one particular value to another are very common, the index structure should tolerate this change with minimal cost. Naturally, this may lead to increased query cost. Therefore, the cost savings for updates should be balanced against cost increases for queries. We will discuss how this can be done in details in the next section.

3 CT-R-tree–the change tolerant index

The *CT*-R-tree we develop is an extension of the R-tree that is tolerant to frequent data changes. The structure of this index is based on R-tree, where the data is hierarchically arranged in bounding rectangles (MBRs). The key idea is to design the MBRs such that updates that cross MBR boundaries are not common. While the future updates (or queries) cannot be predicted, we assume that the past behavior is a good indicator of events in the future.[1] With this in mind, our algorithm utilizes the history of updates to create a *CT*-R-tree, in order to facilitate future updates. In this section, we first describe how the index is created, followed by a discussion of index maintenance operations.

3.1 Creating a *CT*-R-Tree

The creation procedure of *CT*-R-tree can be summarized by four steps:

1. Identification of MBRs (called *quasi-static regions (qs-regions)*) that maximize the "tolerance" of the index to update. A qs-region is simply a range of the domain which encloses numerous updates. Updates that change the value from one qs-region to another should be relatively infrequent (since these are expensive updates). For the case of moving objects, these are regions of space in which objects tend to remain for a long period of time. qs-regions are generated by consulting the history of updates received from each object (Section 3.1.1).
2. Using qs-regions found in step 1, construct a structure called the *update graph*, which depicts traffic among qs-regions (Section3.1.2).
3. The update graph is used to merge the qs-regions (Section 3.1.3).
4. Creation of an "empty" R-tree structure using the identified qs-regions as MBRs at the leaf level, and insertions of current data values to generate the *CT*-R-tree (Section 3.1.4).

Let us now investigate these steps in further details.

3.1.1 Phase 1: Identifying object qs-regions

This phase results in the identification of rectangular regions of the domain that are small and enclose several updates of an object. These rectangles are essentially *qs-regions*, since they represent ranges of values where the data changes constantly in a confined space. We begin by dividing the update trail of each object into pieces that do not have very large changes over a short period of time. As an example, consider Figure 2(a), where some individual object trails are segmented into qs-regions. The connected bold lines show the update trails of objects. The dashed boxes represent the bounding rectangles for initial qs-regions. For ease of exposition, we use an example of mobile objects in two-dimensional space to describe the scenario. However, the algorithms presented here are applicable to the general case of any multidimensional data where the movement of an object represents the change in data value.

Formally, let $O_1, O_2, \ldots, O_n$ be n moving objects. Let H_i denote the trail history of object O_i. Then H_i is a set of points $\{(x_{i,1}, y_{i,1}, t_{i,1}), \ldots, (x_{i,k}, y_{i,k}, t_{i,k}), \ldots, (x_{i,|H_i|}, y_{i,|H_i|}, t_{i,|H_i|})\}$, where $t_{i,k}$ is the time when the kth location update $(x_{i,k}, y_{i,k})$ occurs, and $|H_i|$ is the the total number of samples in H_i. Let $B_i(j,k)$ be the bounding rectangle (MBR) for O_i which encloses $\{(x_{i,j}, y_{i,j}), \ldots, (x_{i,k}, y_{i,k})\}$ in H_i. Let $A_i(j,k)$ be the area of $B_i(j,k)$. Further, let $d_i(j,k)$ be the diameter (i.e. diagonal) of $B_i(j,k)$. We assume that H_i is ordered by increasing values of $t_{i,k}$'s. Figure 3 describes the algorithm for this phase.

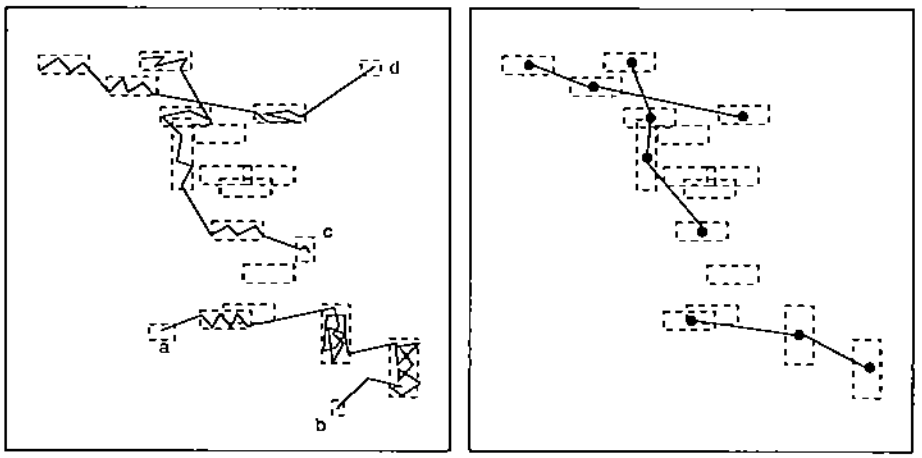
Input: H_i
Output: $B_{i,l}, \tau_{i,l}$
1. $j \leftarrow 1, l \leftarrow 1$
2. $B_{i,j}(1,1) \leftarrow (x_{i,1}, y_{i,1})$
3. **for** $k = 2$ to $|H_i|$ **do**
   A. Let $B_i(j,k)$ be the MBR after expanding $B_i(j,k-1)$ to include $(x_{i,k}, y_{i,k})$
   B. **if** $d_i(j,k) > T_{dist}$ **and** $\frac{d_i(j,k) - d_i(j,k-1)}{t_k - t_{k-1}} > T_{rate}$ **then**
      a. **if** $t_{k-1} - t_j > T_{time}$ **and** $A_i(j,k) < T_{area}$ **then**
         i. $B_{i,l} \leftarrow B_i(j,k-1)$
         ii. $\tau_{i,l} \leftarrow t_{k-1} - t_j$
         iii. $l \leftarrow l + 1$
      b. **else** Discard $B_i(j,k-1)$
      c. $j \leftarrow k$
      d. $B_i(j,j) \leftarrow (x_{i,k}, y_{i,k})$

Figure 3. Identifying *qs*-regions for object O_i (Phase 1).

The algorithm "grow"s MBRs to enclose the samples while tracing the history records, and if an MBR satisfies certain criteria, it is "frozen" and qualified as a *qs*-region for O_i. We maintain a list of qualified MBRs for each object O_i, where we denote the lth MBR of this list by $B_{i,l}$. Let $A_{i,l}$ be the area of $B_{i,l}$, and $\tau_{i,l}$ the time object O_i spent in $B_{i,l}$.

Step 1 introduces the variable j, which indicates the time t_j at which the oldest sample is included in the lth MBR

[1] Note that the design of existing index structures is based upon a prediction of future queries under the assumption that queries are uniformly distributed (i.e. the area of a MBR is a rough indicator of how often it will be accessed by queries).



(a) (b)

Figure 2. (a) Initial qs-regions from object trails. (b) Object update graph.

$(B_{i,l})$. Both j and l are set to 1, and the first MBR, $B_{i,1}$, contains only the first sample, $(x_{i,1}, y_{i,1})$ (Step 2).

Step 3 scans the trail of the object in increasing order of time, identifying *qs*-regions on the way. In Step 3(A), $B_{i,l}$ is expanded to include the kth sample of H_i. Step 3(B) decides if $B_{i,l}$ should be *frozen* as a *qs*-region, based on the following conditions:

$$d_i(j.k) > T_{dist} \tag{1}$$

$$\frac{d_i(j.k) - d_i(j.k-1)}{t_k - t_{k-1}} > T_{rate} \tag{2}$$

That is to say, after expanding $B_i(j.k)$ to some particular threshold diameter T_{dist}, if $B_i(j.k)$ grows at the rate faster than T_{rate}, we stop it from growing further. This relies on the fact that after the initial growth of the rectangle, if there is a sudden increase in growth rate of the region, the object has started moving faster and thus should not be considered as lying in a qs-region. As long as these two conditions are not violated, $B_{i,l}$ continues to grow to enclose more samples.

Steps (a) to (d) in 3(B) take care of the situation when $B_{i,l}$ ceases to grow. First, we decide whether $B_{i,l}$ should be considered as a *qs*-region (steps (a) and (b)). $B_{i,l}$ is only qualified as a qs-region when

1. $t_{k-1} - t_j$ is larger than T_{time}. This verifies O_i has stayed long enough in $B_{i,l}$. Singleton rectangles, such as those labeled 'a', 'b', 'c', and 'd', in Figure 2(a), are also eliminated.
2. The area of $B_{i,l}$, i.e., $A_{i,l}$, is smaller than T_{area}. This removes rectangles that are too large, whose dead space may lead to poor query performance.

in which case we "freeze" $B_{i,l}$ (step (a)(i)) and calculate $\tau_{i,l}$, which is the time spent by the object in $B_{i,l}$ (step (b)(ii)). Steps (c) and (d) create a new MBR$(B_{i,l+1})$, which only contains the kth sample. The whole process is repeated again until all the samples in H_i are exhausted, at which time we obtain a sequence of *qs*-regions for O_i. For the sake of convenience, let C_i denote the number of *qs*-regions generated from H_i.

3.1.2 Phase 2: Creating an update graph

1. **for** $i \leftarrow 1$ to n **do**
   A. **while** $\exists j.k \in [1..C_i]$ **such that**
   $\tau_{i,j}/A_{i,j} < (\tau_{i,j}+\tau_{i,k})/(A_{i,j,k})$ and
   $\tau_{i,k}/A_{i,k} < (\tau_{i,j}+\tau_{i,k})/(A_{i,j,k})$ and $A_{i,j,k} < T_{area}$ **do**
      a. Expand $B_{i,j}$ to include $B_{i,k}$
      b. Replace common links of $B_{i,j}$ and $B_{i,k}$ by a single link, and update the weight of the link
      c. $\tau_{i,j} \leftarrow \tau_{i,j} + \tau_{i,k}$

Figure 4. Merging qs-regions (Phase 2).

We can represent the sequence of rectangular qs-regions just generated as a chain graph with the set of MBRs $B_{i,l}$ as vertices and link between each consecutive rectangles in this sequence (initially each edge is assumed to have a weight 1). Figure 2(b) shows this chain graph for the example histories shown in Figure 2(a) (note that not all nodes and edges of this graph are shown for the purpose of clarity).

We now discuss how to cluster the chain graph of each object to obtain the *object update graph*, where the clustering is based on grouping subsets of vertices (i.e., rect-

angular qs-regions). Figure 4 illustrates the details of how the graph is formed for each object. Define the term "resident density", which is the total amount of time that objects spends inside the qs-region ($\tau_{i,j}$), divided by the area of the qs-region. We see that Step 1(A) chooses any j and k in $[1..C_i]$ such that the following conditions hold:

$$\tau_{i,j}/A_{i,j} < (\tau_{i,j}+\tau_{i,k})/(A_{i,j,k}) \quad (3)$$

$$\tau_{i,k}/A_{i,k} < (\tau_{i,j}+\tau_{i,k})/(A_{i,j,k}) \quad (4)$$

$$A_{i,j,k} < T_{area} \quad (5)$$

where $A_{i,j,k}$ denote the area of the new rectangle that tightly encloses $B_{i,j}$ and $B_{i,k}$. These three conditions enforce the rule that the pair of rectangles are merged only when the resulting "resident density" of the resulting rectangle is greater than each of the "resident densities" of the individual rectangles. Moreover, rectangles are only merged when there is sufficient overlap.

When all these conditions are satisfied, $B_{i,j}$ is expanded to include $B_{i,k}$ (Step (a)). Further, the links that are destined to the same qs-region from $B_{i,j}$ and $B_{i,k}$ are replaced by a single link (Step (b)), with the weight of the new link updated as the sum of the weights of the links being replaced. The time value $\tau_{i,j}$ is then assigned to be the sum of all the individual time values of the merging rectangles (Step (c)). Notice that the algorithm merges the rectangles in arbitrary order, until none of them satisfies the above criteria. This process is repeated for every object (Step 1).

Once the update graphs for all objects are generated, we take the union of all these graphs. A merging procedure similar to Step 1(A) in Figure 4 is applied to this unified graph. This merging gives a us a set of qs-regions as rectangles and a graph on it called the *update graph*. The time value of each rectangle gives the total amount of time that objects spent in that rectangle, and the weight of link (i, j) between two rectangles i and j in the update graph gives the total number of updates between B_i and B_j. Finally, we scale down all the edge weights by the factor of t_{Tot}, where $t_{Tot} = \max(t_{i,|H_i|}$ (i.e., the longest duration of the trail histories). Each edge weight now reflects the number of updates between two qs-regions per unit time.

3.1.3 Phase 3: Merging qs-regions via update graph

In the previous phase, merging occurs only when qs-regions have reasonable amount of overlap. In other words, two rectangles that do not overlap will not be merged by the above phase. However, there could be two unmerged rectangles between which a large number of objects move. In such a situation, it is reasonable to merge these rectangle to form a single MBR and save update cost. In this stage, we use the update graph to detect such occurrences, and merging qs-regions if necessary.

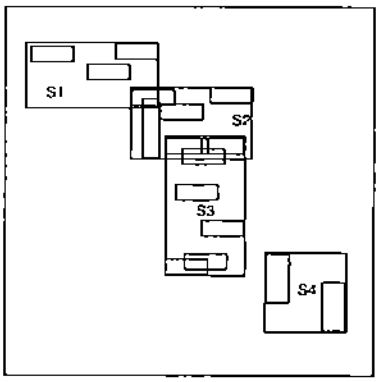


Figure 5. Merged qs-regions.

The high volume of traffic between qs-regions by itself cannot guarantee a good merge. This is because these rectangle can be far apart, in which case the merging of these qs-regions into a single MBR will result in a very large MBR, with lots of dead space. If this happens, many queries will hit this MBR unnecessarily, resulting in higher query cost. Thus there is a trade-off between merging qs-regions and query cost.

We capture the effect of the various factors that contribute to this trade-off. Let ΔA be the increase in area due to the merging operation, A be the total area spanned by the structure, and r_q be the query arrival rate. Then, we expect that $r_q \Delta A/A$ queries per unit time will hit the dead space. This represents the loss due to this merge. On the other hand, if the weight of the edge between these two qs-regions in the update graph is w, then w is the rate of updates caused by not merging these rectangles. Let C_q and C_u be the scaling factors for queries and updates respectively. Then we merge two qs-regions if the following criterion holds:

$$C_u w > C_q r_q \frac{\Delta A}{A} \quad (6)$$

Figure 5 shows the qs-regions as a result of these merging steps for the example history.

3.1.4 Phase 4: Creating a structural R-tree

Given the set of qs-regions identified in the earlier phases, we first create an R-tree index on these qs-regions. This is achieved by inserting the qs-regions into an empty R-tree. This forms a *Structural R-tree*, where the leaf level of this R-tree contains the qs-regions. Note that bulk loading techniques [3] for R-tree can be applied here with appropriate modifications, but since this is not the focus of this paper, we choose repeated insertions, a simpler method. We are not concerned here with the cost of constructing the index.

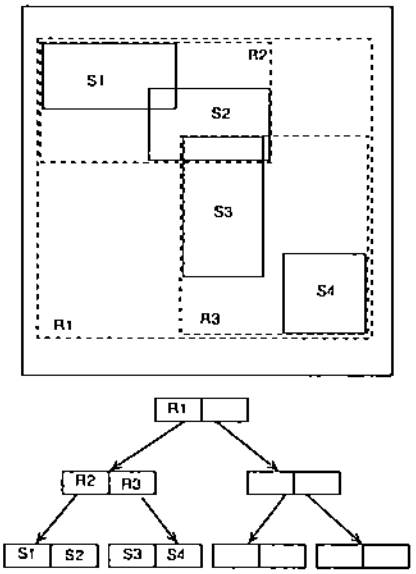


Figure 6. Structural R-tree over qs-regions

since index construction is seen as an offline process. We are more interested in the online query and update performance of the index. Figure 6 shows the structural R-tree that results for our running example.

Using the structural R-tree, we create the change tolerant R-tree (*CT*-R-Tree) over current data. The structural R-tree does not index data – it indexes qs-regions. We begin by inserting the current data values into the structural R-tree, treating the leaf level nodes of this index as one level above the leaf for the *CT*-R-Tree. The qs-regions in the leaves of the structural R-tree serve as the parent MBRs for the data being inserted. Although these MBRs serve a similar purpose as MBRs in a regular R-tree, they are treated specially in two respects: (i) they are never removed from the index (i.e. they are allowed to be underfull – in fact they are all empty at the beginning of the *CT*-R-Tree construction)[2] and (ii) they are not split when overfull – this avoids the high cost for updates. Thus there is a possibly unlimited overflow buffer (which can span multiple pages) attached to these MBRs, as in the *X*-tree [6].

We also attach a linked list of overflow buffers to each internal (non-leaf) MBR. When an object's new position does not fall in any of the qs-region MBRs (MBRs at leaf level), it is stored in the lowest internal node whose MBR contains the new location. The objects which are stored in the internal node buffers are likely to be those whose values are

[2]For now we assume the movement patterns of objects is never unchanged. If a qs-region becomes useless due to movement pattern changes, it is possible to remove the qs-region from the *CT*-R-tree. We will discuss this in Appendix A.

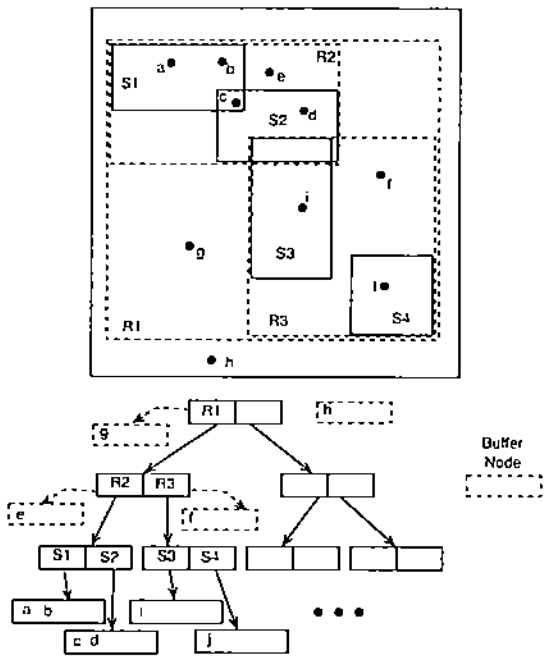


Figure 7. The Change Tolerant R-tree

changing rapidly. Usually, there are relatively fewer objects of this kind unless the movement patterns of objects change significantly. In case any linked list overflow buffers becomes too large, it is converted to an α-R-tree. This issue will be addressed again in Appendix A.

To conclude, objects can be stored in the internal nodes, and each MBR (leaf or internal) has a special pointer to its set of buffer pages. Figure 7 shows the structure of *CT*-R-tree for our example. This index has four levels as opposed to the three levels of the structural R-tree of Figure 6. Examples of data points are shown in the top figure of the domain. The nodes shown in dashed lines are either linked lists of overflow buffers or α-R-trees for the internal nodes. The data objects are inserted at the new leaf level of this tree.

Along with this structure we also maintain a secondary hash-index. Each entry in this hash-index consists of two fields: (1) object id and (2) a pointer to the page in R-tree which contains its location. This structure is the same as the secondary index described in Section 2.1. Figure 1 shows the structure. When we insert an object into the CT-R-Tree, it is also simultaneously inserted into the hash-index and the pointer in its corresponding entry in the hash index is set to the page in the CT-R-tree where it is stored. More details on insertions and other dynamic operations are presented in the next subsection.

3.2 Dynamic operations

Once the index structure is created for rectangular qs-regions, they are usually not deleted, even if they are empty. Thus the structure of the index is basically intact even when objects are inserted or deleted. Query processing is similar to that of the R-tree while updates, insertions and deletions are handled differently. We now describe how these operations are supported. Although all these operations are described in terms of a two-dimensional space structure, they can be extended to multiple dimensions.

Insert(o). Insert object o with location $(o.x, o.y)$ into the index. Determine all the leaf level MBRs (qs-regions) that contain this point. If multiple MBRs contain the point, we choose the one with minimum area (to optimize query performance). The object is inserted into the first non-full page of this MBR. If all pages are full, a new page is allocated and the object is inserted into it. If none of the leaf-level MBRs contain the point, a lowest level MBR that contains this point is chosen. If more than one such MBRs exist, the one with minimum area is chosen. Note that the overflow buffer associated with an internal node can be in the form of either a linked list or an α-R-tree. If the number of pages of the linked list is less than T_{list} after insertion, the point is inserted to the linked list. Otherwise, an α-R-tree is created, to which all data in the linked list are moved. The α-R-tree is then attached to the internal node. Subsequent insertions to the internal node will be directed to the α-R-tree. Finally, the entry for o in the hash-index is updated to point to the page which contains o.

Delete(o). Search the hash-index for o. Delete o from the page and deallocate the page if it is empty. Set the hash-index entry for o to null.

UpdateLoc($o, (x_1, y_1), (x_2, y_2)$). Consult the hash index for o. Set $o.x = x_2, o.y = y_2$. If (x_2, y_2) does not belong to the same MBR, perform *Delete*(o) and *Insert*(o).

Search(x, y). Searching for point (x, y) follows the search pattern of R-tree. Since objects can also be stored in the internal nodes, the search visits the set of buffer pages at each internal node. If the overflow buffer is a linked list, the search checks all the pages since the data in the linked list is unordered. If it is an α-R-tree, an R-tree range search is performed.

RangeSearch($(x_1, y_1), (x_2, y_2)$). This is similar to *Search*. Each MBR which intersects with the rectangle (lower left (x_1, y_1) and upper right (x_2, y_2)) qualifies.

As long as traffic patterns do not change, the qs-regions discovered by our algorithms remain valid, and our index behaves well. However, when the pattern of movement changes, previously undiscovered qs-regions may appear. Many objects may not fall into a qs-region, and they are accumulated in the α-R-trees of internal nodes. We can detect which MBRs of these α-R-trees which show stability, change them into qs-regions, and insert them to the main structure of the *CT*-R-tree. Details can be found in Appendix A.

4 Experimental Results

We performed an extensive simulation study on the performance of change-tolerant indexing. We implemented the *CT*-R-tree, and compared its performance with three variants of R-trees. A study of the sensitivity of the *CT*-R-tree to various parameters was also conducted. Below we discuss the simulation model, followed by the experimental results. The experiment results for changing traffic patterns can be found in Appendix A.

4.1 Simulation Model

Our experiments are based upon data generated by the City Simulator 2.0 [8] developed independently at IBM. The City Simulator simulates the realistic motion of up to 1 million people (N_{obj}) people moving in a city. The input to the simulator is a map of a city. We used the sample map provided with the simulator that models a city containing 71 buildings, 48 roads, six road intersections and one park. Each building is three-dimensional and contains a number of floors. The simulator models the movement of objects within the building and on the roads and park. To generate reasonable movement and occupation of buildings, the simulator keeps track of two conditions based on parameters T_{fill} and T_{empty}: The simulator ensures that the fraction of people at the ground level lies between T_{fill} and T_{empty}.

Each object reports its location to the server at an average rate of λ_u. Before recording the simulation results, the simulator enters a warm-up phase, where at most N_{relax} samples for each object are generated, or at least T_{start} of the population are in the ground level of buildings. Next, 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. The trace file serves as the data source for our experiments. It captures, for each object, a total of $N_{hist} + N_{update}$ location updates. We use the first N_{hist} updates as the history profile. The first $N_{hist} - 1$ records are used to generate an R-tree composed of qs-regions. The N_{hist}-th sample is then inserted to the R-tree to produce the *CT*-R-tree. Once the *CT*-R-tree is built, the remaining N_{update} samples are modeled as dynamic updates to the *CT*-R-tree, as well as other R-tree variants. At the same time, range queries are generated at an average rate of λ_q. Each range query has the shape of a square, with central point chosen randomly within the city area and size equal to a fraction f_q of the city area. It should be noted that the city map is used only by the City Simulator to generate realistic

movement of objects -- it is not used for the generation of the *CT*-R-tree index structure.

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. We do not distinguish between sequential page I/Os and random page I/Os – each page is treated equally. This is likely to be a disadvantage for the *CT*-R-tree since its node buffer pages may often be multiple pages long, unlike the other trees for which the nodes are always the same size. Each page has a size of S_{page} with a fan-out of N_{entry}. The secondary index of the *CT*-R-tree (i.e., the hash table) with size S_{hash}. We assume all tree structures and the hash table are stored on disk.

The City Simulator is implemented in Java and run under Windows XP. The programs for generating the *CT*-R-tree are written in C++ and Java, and the testbed is run on a UNIX server. Although we focus on the performance of dynamic updates and queries, it is worth notice that the time required to generate the CT-R-tree using the history profiles is usually less than ten minutes. Also, since this process can be done in an offline fashion, it does not interrupt the processing of online updates. Table 1 shows the parameters of the simulation model, the parameters of the *CT*-R-tree, as well as their corresponding values.

4.2 Results

Here we present the simulation results of the *CT*-R-tree. Four index structures are evaluated in our experiments: (i) the traditional R-tree [15]; (ii) the traditional R-tree augmented with lazy updating using the secondary index structure shown in Figure 1. We call this *lazy*-R-tree; (iii) the α-tree which is essentially an R-tree with lazy updating and expanded MBRs (i.e. the MBRs are not minimal, but widened by a factor of α (we used $\alpha = 0.1$ in our experiments); and (iv) the *CT*-R-tree.

4.2.1 Effect of Update/Query Ratio

We begin by studying the relative performance of the various index structures as the number of queries and updates is varied. Figure 8 shows the total number of page I/Os performed for query and update for the R-tree, the *lazy*-R-tree, the α-tree and the *CT*-R-tree. The performance is measured under the same query generation rate but different update arrival rates. To generate a slower update rate, some location samples are skipped. It should be noted that this graph uses a Log-scale on both axes. As the ratio of update rate over the query rate (abbreviated as update/query ratio) is increased from 10^{-2} to 10^{3}, all four 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 *CT*-R-tree takes about 2 times as many as I/Os than the other R-tree variants. Recall that the R-tree and the *lazy*-R-tree uses MBRs, which are tight bounds over the enclosed objects' values. On the other hand, the *CT*-R-tree employs qs-regions that do not necessarily enclose as tightly as MBRs. When a query is executed, its query region potentially has less overlap with the R-tree's MBRs than with qs-regions. This results in fewer searches and better performance. With an α of 0.1, the expanded MBR of the α-tree is slightly larger than the other R-trees. Thus it also suffers the same problem as the *CT*-R-tree and its performance is worse than the R-trees. The advantage of using 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.

Towards the right end of the graph, when the update workload dominates the query workload, the *CT*-R-tree registers a significant improvement over other R-tree variants. In fact, once the update/query ratio crosses over 5.6, the number of I/Os needed by all three R-trees increases sharply, whereas the *CT*-R-tree gracefully handles the high update burden. When updates are much more frequent than queries, which is a typical scenario in sensor and moving object databases, 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 α-tree improves further over the *lazy*-R-tree since it can handle more updates through the secondary index on account of its more lax MBR. The *CT*-R-tree clearly outperforms the other indexes in this high update environment since its structure is inherently designed to maximize tolerance to changes in object values. The advantage of better update performance more than compensates for the slightly poorer query performance.

The *CT*-R-tree works the best under high update rates because it is aware of the presence of qs-regions, and uses them to cluster the search space. Further, these qs-regions are not split further into smaller units. Therefore, when an object moves inside the qs-region, no matter how frequently it reports its value, only the secondary index is consulted and the current value is directly updated in the leaf node. 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 *CT*-R-tree is only 1/4th that of the α-tree, 1/7th that of the *lazy*-R-tree, and 1/27th that of the R-tree.

Param	Default	Meaning
Simulation parameters		
λ_u	5,000	Location update rate (sec^{-1})
T_{start}	0.15	Start threshold
T_{fill}	0.09	Fill threshold
T_{empty}	0.5	Empty threshold
N_{obj}	10^5	# of moving objects
N_{relax}	2000	Max samples skipped before recording
N_{hist}	110	# of historic samples (per object)
N_{update}	20	# of online updates (per object)
λ_q	50	Query arrival rate (sec^{-1})
f_q	0.1	Query size (% of the city area)
***CT*-R-tree parameters**		
T_{dist}	25	Distance threshold in Eqn 1 (m)
T_{rate}	1	Max growth rate of qs-region (m/sec)
T_{time}	300	Min time objects in qs-region (sec)
T_{area}	22500	Max area of qs-region (m^2)
C_q	1	Query scaling factor (Eqn 6)
C_u	1	Update Scaling factor (Eqn 6)
S_{page}	4096	Size of a page (bytes)
N_{entry}	20	# of entries (per page)
S_{hash}	8	Size of secondary index (Mbytes)

Table 1. Parameters and baseline values.

4.2.2 Effect of Query Size

Since the *lazy*-R-tree maintains tighter bounding rectangles than the α-tree and the *CT*-R-tree, it is expected to outperform them for querying. In this experiment, we examine more precisely how well the *lazy*-R-tree outperforms the two indexes by measuring the ratio of the query I/Os of two trees over the query I/Os for the *lazy*-R-tree. Note that the *lazy*-R-tree and the traditional R-tree have identical query performance. Figure 9 shows the ratios over different query sizes. The query size is varied from 0.1% to 2% of the domain. We observe that both the α-tree and the *CT*-R-tree require more query I/Os than the R-tree. Also, the *CT*-R-tree needs more query I/Os than the α-tree. As the query size increases, their performance 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. In that case, searching a qs-region in the *CT*-R-tree is even more effective than searching in the R-tree, because a qs-region does not limit how many objects are stored inside. On the other hand, MBRs need to be split when they are over-full, so that more access paths are necessary. Thus the performance of *CT*-R-tree improves over large query size.

Although the *CT*-R-tree does not perform as well for queries as the other two indexes, we can see from Figure 10 that it is the clear winner in terms of overall performance (total number of I/Os). The *CT*-R-tree is designed for databases with more updates than queries. Its loss in query performance is compensated with a significant gain in update performance, resulting in three-fold improvement over the α-tree, and four-fold improvement over the lazy-R-tree, consistently over all query sizes considered.

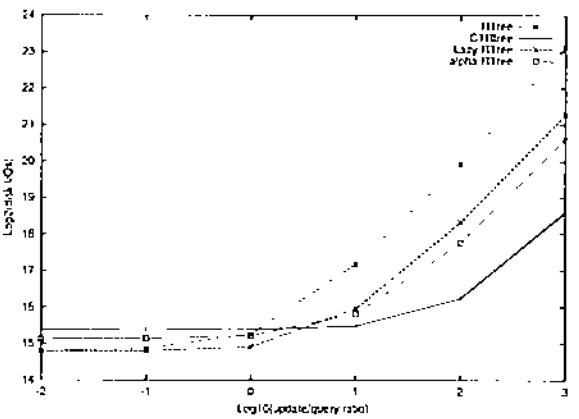


Figure 8. Total I/O vs. Update/Query Ratio

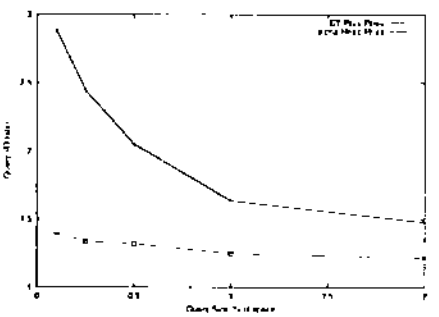


Figure 9. Query I/O ratio vs. Query Size

4.2.3 Scalability of *CT*-R-tree

In this experiment, we study the scalability of the *CT*-R-tree. The number of I/Os for the *lazy*-R-tree and the *CT*-R-tree are reported for up to 500K objects (Figure 11). We observe that the *CT*-R-tree performs better than the *lazy*-R-tree as the number of objects is increased from the baseline value (100K). This shows that the *CT*-R-tree scales with the number of objects. A closer look at the graph reveals that the performance gap between the two indexes *widens* with increasing number of objects. The rationale is two-fold: First, when more objects are maintained in the system, more update requests are generated. As discussed in 4.2.1, the performance of the R-tree degrades more than that of the

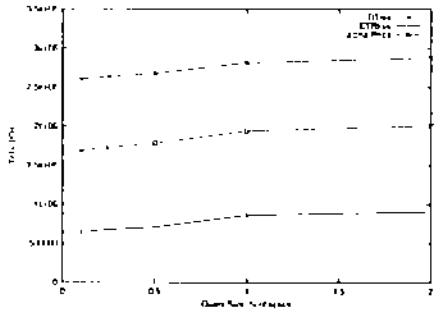


Figure 10. Total I/O vs. Query Size

CT-R-tree. Second, the city plan is fixed. Injecting more objects to the city implies a higher population density. Many objects are close to each other, so that they have a higher chance of being clustered to the same MBR. As a result, an MBR gets full easily, and more splits are necessary to maintain the R-tree. A *CT*-R-tree does not have to perform any split operations, even when the density of objects is high. It therefore requires fewer I/Os.

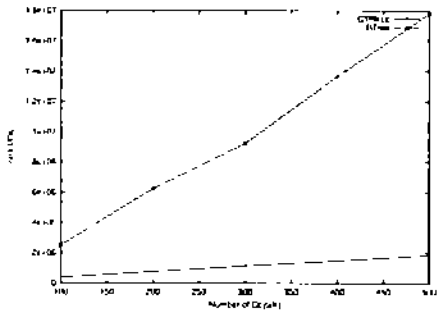


Figure 11. Total I/O vs. Number of objects

4.2.4 Sensitivity to Parameter Values

This set of experiments studies the sensitivity of the *CT*-R-tree to its parameter values, namely T_{dist}, T_{rate}, T_{area}, and T_{time}. These parameters are used in the first step of identifying qs-regions, so their values can be critical to the performance results. We examine the I/O performance of the *CT*-R-tree over a wide range of values for these parameters. The results for T_{rate} and T_{area} are shown in Figures 12(a) and (b) respectively. The results for T_{dist} and T_{time} showed trends very similar to those for T_{rate} and the graphs are omitted due to space constraints. Each graph plots the number of page I/Os for query and update for the *CT*-R-tree as a function of the respective parameter.

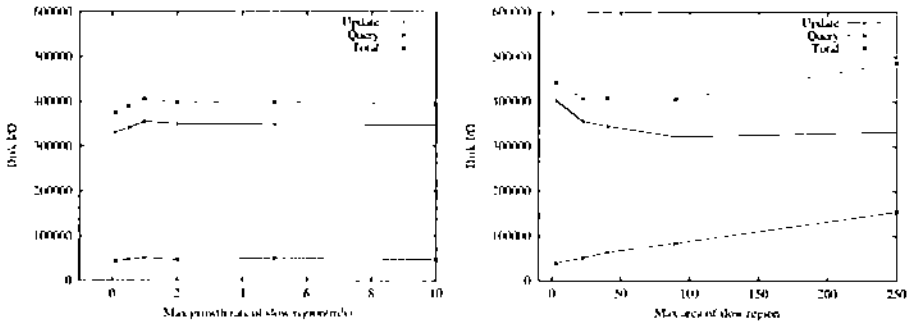
In general, these graphs illustrate flat curves for update, query and overall I/O performance, over a wide range of values. This indicates that the *CT*-R-tree is not sensitive to these parameters and therefore it is not critical to choose precise parameter values for the *CT*-R-Tree to work efficiently. As long as the parameter values are "reasonable", the *CT*-R-tree behaves well. Special care needs to be taken in choosing a value for T_{area}, though. In particular, one needs to avoid choosing a value that is too small, otherwise the number of qs-regions may be too small, or qs-regions may tend to be smaller than they should be. Many objects that should be in a qs-region may then not be able to hit one of these small qs-regions. They are forced to be placed in the overflow pages of the internal nodes, leading to poor performance.

We also studied the effect of changing traffic patterns on α-R-tree experimentally. Their results are shown in Appendix A.

5 Related Work

Developing an efficient index structure for constantly evolving data is an important research issue for databases. Most works in this area so far focus on moving object environments, where the positions of objects keep changing. 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 and numerous update operations.

To reduce the number of updates, many approaches describe a moving object's location by a linear function, and the index is updated only when the parameters of the function change, for example, when the moving object changes its speed or direction. Saltenis et al. [12] proposed the time-parameterized R-tree (TPR-tree). In this scheme, the position of a moving point is represented by a reference position and a corresponding velocity vector. The MBRs of the tree vary with time as a function of the enclosed objects. When splitting nodes, the TPRtree considers both the positions of the moving points and their velocities. Later, Tao et al [13] presented TPR*-tree, which extends the idea of TPR-trees by employing a different set of insertion and deletion algorithms in order to minimize the query cost. Kollios et al. [9] proposed an efficient indexing scheme using partition trees. Tayeb et al. [14] introduced the issue of indexing moving objects to query the present and future positions and proposed PMR-Quadtree for indexing moving objects. Agarwal et al.[1] proposed various schemes based on the duality and developed an efficient indexing scheme to answer approximate nearest-neighbor queries. The problem of all these techniques is that there hardly exists a good function for describing the objects' movements in reality. In many applications, the movement of objects is complicated and non-linear. In such situations, the approaches based on a linear function cannot work efficiently– the function changes-



(a) (b)

Figure 12. Performance for (a) T_{rate} and (b) T_{area}

too often. Approximation technique using threshold such as maximal velocity has been proposed to reduce the update cost. However, this approximation technique can decrease the efficiency of the index.

In the computational geometry community, kinetic data structures [5] were introduced for mobile data. These are main memory structures that assume that the objects move in a rectilinear motion with certain velocities. The updates are in the form of change in velocity or direction of an object. A kinetic event occurs when objects change their velocities or directions or when the combinatorial structure changes e.g. when two points cross each other. The idea is that the structure only needs to be updated when such a kinetic event occurs. These data structures were applied to solve geometry problems like closest pair, convex hull and voronoi diagram problems efficiently while objects are moving continuously. Kinetic space partitioning tree (or cell-trees) were introduced by [2]. Based on this notion of kinetic data structures, Agarwal et al. [1] proposed kinetic version of kd-tree, where the medians are dynamically maintained. However, most works have been in the main memory data structures. For external memory, Agarwal et al. [1] applied this idea to external range trees [4] and bounds on query performance are proved.

6 Conclusion and Future Work

Traditionally, index structures are optimized for improved query performance in the presence of less frequent updates. For environments such as sensor and moving object databases where data is constantly evolving traditional index structures give poor performance. We introduced the notion of *Change Tolerant* indexing for these high update environments. Change tolerant indexes optimize for both query and update performance. We developed the algorithms for creation and use of a change tolerant R-tree index. Experimental results showed the superior performance of the proposed index structure. The proposed *CT*-R-tree trades slightly poorer query performance for much superior update performance resulting in better overall performance. The performance was also found to be robust with regards to number of objects and queries, and query sizes. We observe that the generic idea of change tolerant indexing can be applied to other index structures. Preliminary ideas for extensions to other structures were outlined. In future work, we will study change tolerant versions of these other index structures in more detail.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Sym. on Principles of Database Systems*, pages 175–186, 2000.

[2] P. M. Agarwal, J. Erickson, and L. J. Guibas. Kinetic binary space partitions for intersecting segments and disjoint triangles. In *Symposium on Discrete Algorithms*, pages 107–116, 1998.

[3] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1):104–128, May 2002.

[4] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. of the ACM Sym. Principles of Database Systems,*, pages 346–357, 1999.

[5] J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. *Symposium on Discrete Algorithms*, 1997.

[6] S. Berchtold, D. A. Keim, and H. P. Kreigel. The X-tree: An index structure for high-dimensional data. In *22nd. Conference on Very Large Databases*, pages 28–39, Bombay, India, 1996.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proc. of the ACM SIGMOD Int'l. Conf.*, 1984.

[8] J. Kaufman, J. Myllymaki, and J. Jackson. IBM City Simulator 2.0, http://www.alphaworks.ibm.com/tech/citysimulator.

[9] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Sym. on Principles of Database Systems*, pages 261–272, 1999.

[10] D. Kwon, S. J. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update R-tree. *3rd International Conference on Mobile Data Management*, Jan 2002.

[11] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2000.

[12] S. Saltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the position of continuously moving objects. *Proc. of ACM SIGMOD*, 2000.

[13] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. *Proceedings of the 29th International Conference on Very Large Databases(VLDB)*, pages 790–802, 2003.

[14] J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, pages 185–200, 1998.

[15] University of California, Riverside. Spatial index library version 0.44.2b (java). http://www.cs.ucr.edu/ marioh/spatialindex/.

Appendix A: Adaptation to Changing Patterns

Recall that we build the *CT*-R-tree by consulting history records of the objects. The structure of the *CT*-R-tree, once built, is basically unchanged. In essence, we assume future changes of data follow the discovered patterns (in the form of qs-regions). This assumption may not hold, however, if the patterns *do* change. For example, a party of people may gather around for a few hours and dismiss afterwards. The qs-regions discovered is then be no longer useful. Similarly, new qs-regions can be created after the *CT*-R-tree is constructed. To handle these problems, we may rebuild the *CT*-R-tree periodically, running as a background process, and then switch to the new tree once it is built. But since the cost of construction is high, we cannot afford to rebuild it very often. In this section, we discuss how to change the *CT*-R-tree temporarily to handle unexpected traffic pattern changes.

We described in Section 3.2 that the overflow buffer is switched from the linked list to the α-R-tree when the linked list is longer than T_{list}. This is the first measure to handle movement pattern changes. Usually the portion of items that need to be placed in the overflow buffer is little (as verified by our experiments), and thus a linked list suffices. However, if traffic pattern changes, the linked list may grow indefinitely and degrade index performance. This is why an upper bound T_{list} is placed on the length of the linked list, and an α-R-tree, an adaptive structure, is used to replace the linked list when it is excessively long.

A.1 Discovering new qs-regions online

Another purpose of using the α-R-tree as the overflow buffer is that it facilitates discovery of new, albeit approximate, qs-regions. The MBR of the α-R-tree is actually $(1+\alpha)$ larger than its actual size, and is thus more tolerant than the MBR of the R-tree. We may thus treat the MBR of the α-R-tree's leaf node as an approximate qs-region if the objects located there illustrate some properties of a qs-region. The identified MBR can then be migrated to the *CT*-R-tree as its new leaf node.

In order to detect if a leaf-node MBR X_i of the overflow α-R-tree behaves like a qs-region, we store the following information in the node:

- The time qs-region behavior is observed, t_i. Initially, t_i is ∞.
- The number of objects in the leaf node, n_i, with an initial value of 0.

When an insertion to X_i is made at time t, n_i is incremented. Then we perform additional checks on the following conditions:

1. $n_i > T_{leaf\_min}$

2. Area of the MBR of the leaf node $< T_{area}$

where T_{leaf_min} is the minimum number of objects in X_i, and T_{area} is the area constraint defined before in Section 3.1.1. If these conditions are satisfied, t_i is set to t, and insertion is completed.

On subsequent insertions, conditions (1) and (2) are checked again. If any of them are not satisfied, then t_i is reset to ∞, indicating that the node does not behave like a qs-region. Otherwise, the following additional condition is checked:

3. $t - t_i > T_{leaf-time}$

Here $T_{leaf-time}$ denotes the minimum amount of time that (1) and (2) are satisfied. That is, we require (1) and (2) to hold over a period $T_{leaftime}$.

If condition (3) is satisfied, X_i (and its associated objects) is removed from the α-R-tree and re-inserted to the structural R-tree as a new qs-region. No change to the hash table is necessary. We remark that the qs-regions so discovered may only approximate the true qs-regions. They are only used as temporary measures when a complete analysis of qs-regions is not feasible.

A.2 Deleting a qs-region

When a qs-region is no longer useful due to a change in traffic pattern, it may be removed to improve query performance. Let the upper bound of the number of times an object is removed from each qs-region be T_{remove} per unit time. We observe that every time an object is removed from a qs-region, the object has violated the supposed stability of the qs-region. When the removal rate is greater than T_{remove}, it indicates that the qs-region is not qualified for holding objects, and it cannot save updates. Thus, we can check the removal rate of a qs-region every time an object gets deleted, and remove the qs-region if necessary. All items in the qs-region are re-inserted to the *CT*-R-tree.

Notice that even if a qs-region is not used now, it does not necessarily mean that the qs-region is not used again. In particular, if there is a periodic pattern, e.g., the office is occupied between 9-5 every day, we may retain the qs-region representing the office space in the tree for future use. Whether deleting a qs-region is beneficial depends on the requirements.

A.3 Rebuilding a *CT*-R-tree

Since the MBRs in the α-R-tree are not true qs-regions, and some qs-regions may not even be discovered, the scheme we just proposed can only approximate the performance of an actual *CT*-R-tree. However, it can be used as a temporary measure to adapt to changing patterns, and is much cheaper than the cost of constructing the whole *CT*-R-tree. We still need to rebuild the *CT*-R-tree if its structure changes too much. For example, we may start the rebuilding process if the number of qs-regions being deleted or inserted is too high. New history records that are not used for constructing the tree can be used. The rebuilding process should be run in background, with no interference to the current index. Once the rebuilding is completed, the new index is used immediately.

A.4 Experimental Results

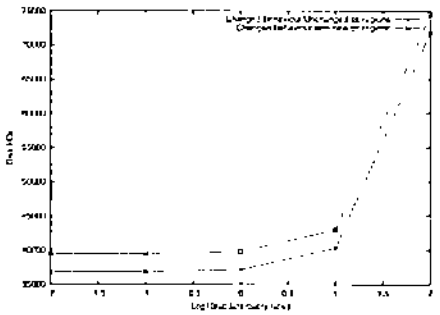


Figure 13. Total I/O vs. Update/Query Ratio

We experimentally study the effect of changing traffic patterns on the performance of the *CT*-R-tree, using the basic settings mentioned in Section 4.1. A *CT*-R-tree is first built based on their movement records in the city plan. Then we generate a set of movement records based on a new city plan, with five buildings removed and five buildings created. Since an object now cannot enter the regions where buildings are destroyed, but they can enter buildings which originally do not exist, some qs-regions are no longer valid, while new qs-regions are created.

The index created based on the first set of records is used to test its efficiency in storing the locations of objects which move around in the second city. Its performance in shown in the curve "Changed Behavior/Unchanged qs-regions" in Figure 13. The second curve "Changed Behavior/New qs-regions" illustrates the performance of the index when we apply the approximate qs-region detection algorithm mentioned in this section. As we can see, over a large range of update/query ratios, the *CT*-R-tree performs consistently better after the qs-region detection algorithm is applied. We thus show experimentally that the *CT*-R-tree can adapt to changing traffic patterns.