Supporting Frequent Updates in R-Trees: A Bottom-Up Approach

Mong Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, Keng Lik Teo

April 16, 2004

TR-6

A DB Technical Report

Title	Supporting Frequent Updates in R-Trees: A Bottom-Up Approach
	Copyright © 2004 Mong Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, Keng Lik Teo. All rights reserved.
Author(s)	Mong Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, Keng Lik Teo
Publication History	April 2004. A DB Technical Report

Abstract

Advances in hardware-related technologies promise to enable new data management applications that monitor continuous processes. In these applications, enormous amounts of state samples are obtained via sensors and are streamed to a database. Further, updates are very frequent and may exhibit locality. While the R-tree is the index of choice for multi-dimensional data with low dimensionality, and is thus relevant to these applications, R-tree updates are also relatively inefficient. We present a bottom-up update strategy for R-trees that generalizes existing update techniques and aims to improve update performance. It has different levels of reorganization—ranging from global to local—during updates, avoiding expensive top-down updates. A compact main-memory summary structure that allows direct access to the R-tree index nodes is used together with efficient bottom-up algorithms. Empirical studies indicate that the bottom-up strategy outperforms the traditional top-down technique, leads to indices with better query performance, achieves higher throughput, and is scalable.

1 Introduction

Innovations in primarily wireless technologies and positioning technologies are combining to enable applications that rely on the tracking of the locations of mobile objects such as vehicles, users of wireless devices, and deliveries. A wide range of other applications beyond moving-object applications also rely on the sampling of continuous, multidimensional variables. This class of monitoring applications is characterized by large volumes of updates, which occur when the applications strive to maintain the latest state of the continuous variables being monitored.

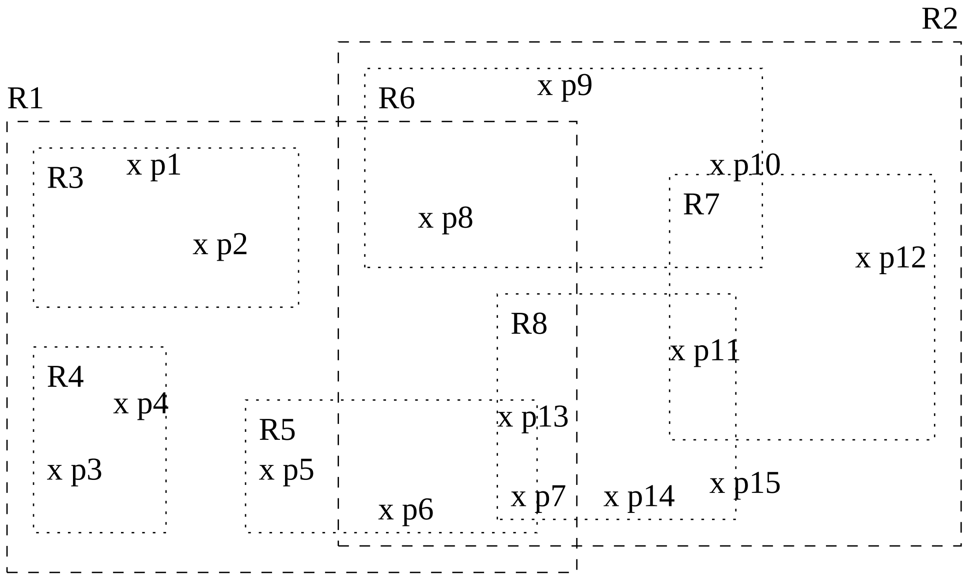
The provision of high performance and scalable data management support for monitoring applications presents new challenges. One key challenge derives from the need to accommodate very frequent updates while simultaneously allowing for the efficient processing of queries. This combination of desired functionality is particularly troublesome in the context of indexing of multidimensional data. The dominant indexing technique for multidimensional data with low dimensionality, the R-tree [3] (and its close relatives such as the R*-tree [1]), was conceived for largely static data sets and exhibits poor update performance.

Existing R-tree update procedures work in a top-down manner. For each update, one index traversal is needed to locate and delete the data item to be updated. Depending on the amount of overlap among the bounding rectangles in the index nodes, this traversal may well follow more than a single partial path from the root towards the leaf level of the index. Then a separate index traversal is carried out to insert the new data item. While this top-down procedure makes for a quite adaptable index structure with good query performance, it is also costly.

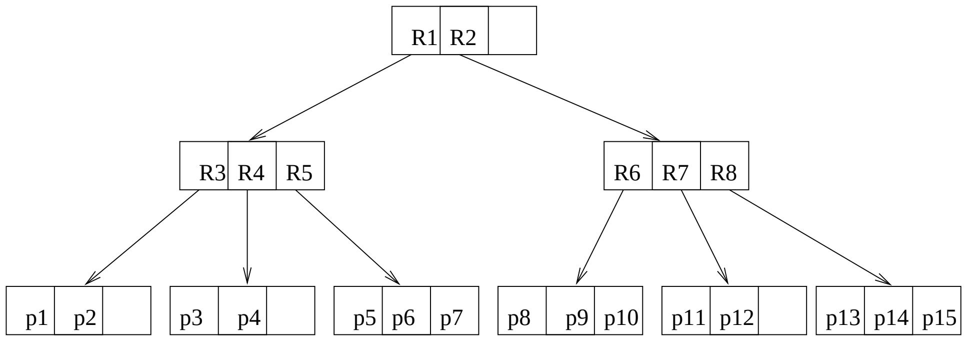
With the recent interest in moving objects, several techniques for indexing the past as well as the current and anticipated future positions of moving objects have been proposed that are based on the R-tree, most often the R*-tree, that are efficient for querying [4, 6, 7, 9, 12, 14, 16, 17]. These techniques typically process updates as combinations of separate deletion and insertion operations that operate in a top-down manner. They are related to this paper's contribution in the sense that the update techniques proposed here may well improve their performance if applied to them.

Kwon et al. [7] advocates lazy updates for R-trees to reduce update cost. When locality is present in updates, the proposal is to enlarge leaf-level bounding rectangles equally in all directions so that the new location for an object remains within the same bounding rectangle as the old location. However, this somewhat preliminary proposal suffers from expensive maintenance of parent pointers, and query performance deteriorates with the increased overlaps caused by the enlargement of leaf-level bounding rectangles.

Understanding the tradeoffs between update and query performance in index structures will become increasingly important in the future. Motivated by the class of locality-preserving monitoring applications, by the importance of indexing, and by the deficiencies of R-trees, we take a first step in this direction by offering concrete insight into this tradeoff for the R-tree, which was originally designed with primarily



(a) Planar Representation of R-Tree



(b) Directory of R-Tree

Figure 1: Example R-Tree

efficient querying in mind. In this paper, we propose bottom-up update techniques for R-trees. These techniques exploit a compact, easy-to-maintain main-memory summary structure that provides direct access to index nodes. Experimental results indicate that the proposed bottom-up techniques offer better update performance than does the standard top-down approach, while simultaneously resulting in indices with better query performance. The techniques presented can be easily integrated into R-trees as they preserve the index structure.

The remainder of the paper is organized as follows. Section 2 briefly describes the R-tree. Section 3 discusses the shortcomings of the top-down approach and presents the bottom-up techniques, covering the concepts, algorithms, optimizations, and the summary structure. Section 4 examines the cost of top-down and bottom-up updates. Section 5 presents a thorough experimental evaluation of the bottom-up update approach, and Section 6 concludes.

2 R-Tree Based Indexing

Much research has been conducted on the indexing of spatial data and other multidimensional data. The commercially available R-tree [3] is an extension of the B$^+$-tree for multi-dimensional objects, and remains a focus of attention in the research community. It is practical and has shown itself to support the efficient processing of range and more advanced queries on spatial and other low-dimensional data.

Assuming that we consider spatial objects embedded in two-dimensional space, the spatial extent of each data object is represented by a Minimum Bounding Rectangle (MBR). Leaf nodes in the R-tree contain entries of the form $(oid, rect)$, where *oid* is a pointer to the object in the database and *rect* is the MBR of the object. Non-leaf nodes contain entries of the form $(ptr, rect)$ where *ptr* is a pointer to a child node in the tree and *rect* is the MBR that bounds all the MBRs in the child node.

Figure 1 shows a set of data rectangles and how they are indexed by an R-tree with a fanout of 3. The bounding rectangles at each level of the R-tree are allowed to overlap. Thus, any range query on the R-tree may result in multiple complete or partial paths being followed from the root to the leaf level. The more the overlap, the worse the branching behavior of a query. This is in contrast to the B-tree.

A number of variations of the initial R-tree exist, including packed R-trees [11], the R$^+$-tree [15], the R*-tree [1], and the Hilbert R-tree [5]. Most recently, several extensions of R-trees have been proposed specifically for moving objects, including the TPR-tree [14], the STAR-tree [10], and the R$^{\mathrm{EXP}}$-tree [13]. These R-tree variants all process updates as combinations of separate top-down deletion and insertion operations. The techniques proposed in this paper may be applied to each of these.

3 Bottom-Up Update Strategies

A traditional R-tree update first carries out a top-down search for the leaf node with the index entry of an object to be updated, deletes the entry, and then executes another and separate top-down search for the "optimal" location in which to insert new the entry for the object. The first search may descend, either completely or partially, from the root to several leaf nodes, which may be costly. In addition, node splits and reinsertion of index entries may occur.

While this strategy makes for a very dynamic index structure that adapts well to the data being indexed, it is often sub-optimal, one reason being that the separate descents probably contain some overlap. While a main-memory cache may prove useful, a large amount of varying updates may be expected to render a cache relatively ineffective. In addition, a cache does not reduce the CPU costs incurred by the two descents.

Top-down update is inherently inefficient because objects are stored in the leaf nodes, whereas the starting point for updates is the root. This and the observation that many applications exhibit locality-preserving updates motivate the bottom-up concept.

3.1 Localized Bottom-Up Update

An initial bottom-up approach is to access the leaf of an object's entry directly. This requires a secondary index on object IDs—see Figure 2.

If the new extent of the object does not exceed the MBR of its leaf node, the update is carried out immediately. Otherwise, a top-down update is issued. Initial experiments on a dataset with one million uniformly distributed points reveal that this simple strategy fails to yield the improvements to be expected, as a large percentage of the updates (82%) remains top-down.

Kwon et al. [7] allow leaf MBRs to expand by some ϵ in each direction in each dimension so that more objects can remain in their leaf nodes, thus reducing the need for expensive top-down updates. In order to preserve the R-tree structure, the expansion of a leaf MBR is bounded by its parent MBR. This requires access to the parent node, and a parent pointer must be stored in the leaf node. The ϵ is varied to find a

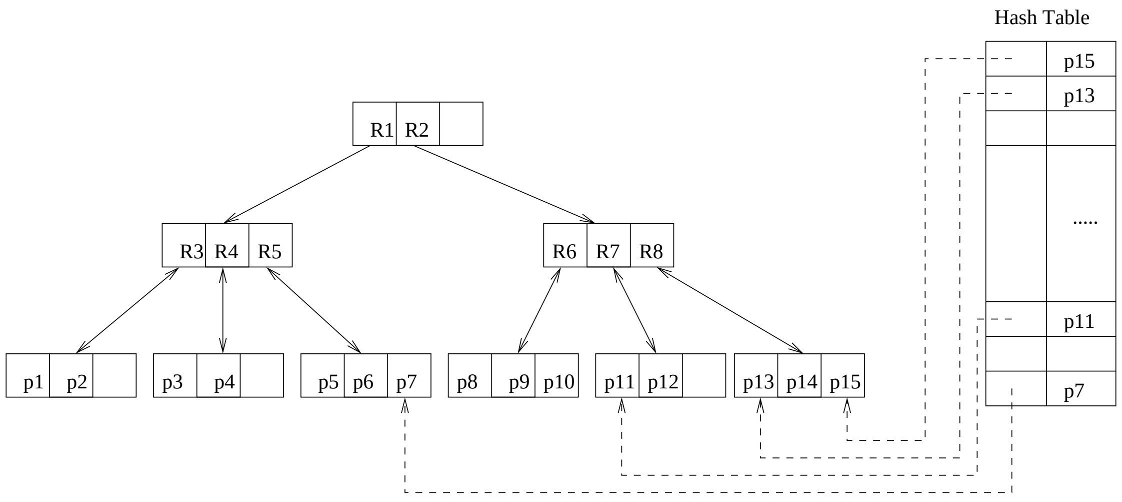


Figure 2: Secondary Index Access to R-Tree Leaves

compromise between update and query performance. A large ϵ will lead to decreased query performance; and update performance may not improve much, as the enlargement of a leaf MBR is limited by its parent's MBR. A small ϵ yields little update improvement.

An alternative and complementary strategy is to place an object being updated in some sibling node. When we allow the new value to be shifted to a (non-full) sibling, we are considering a range of leaf nodes, leading to a less localized update. Intuitively, shifting is desirable since this reduces the need for enlargements of bounding rectangles. However, it costs more disk accesses, as siblings have to be inspected to be sure they are not full.

Applying the two optimizations to this bottom-up approach, we obtain the Localized Bottom-Up Update Algorithm (Algorithm 1). In this algorithm, if the new location of an object is outside the leaf bounding rectangle, the bounding rectangle is enlarged by ϵ. If the new location remains outside (after the enlargement), then we identify a suitable sibling to contain the new location. Otherwise, a top-down update is issued.

The Localized Bottom-up Update Algorithm gains the most when updates preserve locality, so that the majority of updates are concentrated on the leaf level and its parent level. However, this approach results in a dip in query performance due to the enlargement of leaf MBRs. Further, the need to maintain parent pointers at the leaf level reduces fanout and increases the maintenance costs during node splits.

To support dynamic data more efficiently, a generalized bottom-up strategy is proposed that relies less on the locality-preserving property and is able to ascend to higher levels without the need to maintain parent pointers.

3.2 Generalized Bottom-Up Update

In the generalized bottom-up algorithm, the basic R-tree structure is kept intact, and a compact main memory summary structure is introduced. This structure consists of

1. a direct access table that covers the non-leaf nodes of the R-tree, and
2. a bit vector that indicate whether each leaf node is full or not.

Figure 3 shows the summary structure for our example R-tree. The single MBR captured in an entry of the direct access table bounds all MBRs stored in the entries of the corresponding R-tree index node. All

Algorithm 1 Localized Bottom-Up Update (oID, newLocation, oldLocation)

```
Locate via the secondary object-ID index (e.g., hash table) the leaf node with the object;
if newLocation lies within the leaf MBR then
    Update the location of the object in the leaf node;
    Write out leaf node; return;
Retrieve the parent of the leaf node;
Let eMBR be the leaf MBR enlarged by $\epsilon$;
if eMBR is contained in the parent MBR and newLocation is within eMBR then
    Enlarge the leaf MBR;
    Update location of object in leaf node;
    Write out leaf node; return;
if Deletion of the object from the leaf node leads to underflow then
    Issue a top-down update; return;
Delete old index entry for the object from leaf node;
Write out leaf node;
if newLocation is contained in the MBR of some sibling node which is not full then
    Retrieve sibling node;
    Insert an index entry for the object with newLocation into sibling node;
    Write out sibling node; return;
/* Issue a standard R-tree insert. */
Insert (root, oID, newLocation);
```

the entries are contiguous, and are organized according to the levels of the internal nodes they correspond to.

The maintenance cost for the main-memory summary structure is relatively low. We only need to update the direct access table when there is an MBR modification or a node split. The MBR of an entry in the direct access table is updated when we propagate an MBR enlargement from the bottom of the R-tree. When an internal node is split, a new entry will be inserted into the direct access table. We observe that since most of the node splits occur in the leaf level due to the high node fanout, inserting a new entry into the direct access table will be very infrequent. Note that the direct access table only keeps information about the internal nodes of the R-tree.

The size of each entry in the direct access table of the summary structure is a fraction of the size of the corresponding R-tree node. The average size ratio of a table entry to corresponding R-tree index node is 20.4%. This savings is achieved because the individual MBRs of the child nodes in the R-tree are excluded in the direct access table. Given a 4 KB page with a fanout of 204 and 66% node utilization, the ratio of the number of entries in the table, i.e., the number of internal nodes, to the number of nodes for an R-tree is 0.75%. Overall, the space consumption of the direct access table is 0.16% of that of the associated R-tree.

The generalized bottom-up strategy (Algorithm 2) aims to offer a comprehensive solution to support frequent updates in R-trees. As in the previous section, we use an existing secondary identity index such as a hash table for access to leaf nodes. The direct access table in the summary structure facilitates low-cost access to parent nodes. Depending on how much the object has moved, the algorithm adaptively determines whether to enlarge the leaf MBR, to place the new object in a sibling leaf node (if the object has moved outside its leaf MBR), or to ascend the index to look for a less localized solution.

With hybrids of the various bottom-up alternatives available, we limit the use of leaf MBR enlargement and extend it more intelligently to minimize the negative impact on query performance. The algorithm only enlarges an MBR in the directions the object is moving and only enough to bound the object while limiting the enlargement to at most ϵ.

Algorithm 2 Generalized Bottom-Up Update(oID, newLocation, oldLocation)

```
Access the root entry in direct access table;
if newLocation lies outside rootMBR then
    Issue a top-down update; return;
Locate via the secondary object-ID index (e.g., hash table) the leaf node that contains the object;
if newLocation lies within leafMBR then
    Update location of object in leafNode;
    Write out leafNode; return;
node = leafNode; /* Leaf level */
parentNode = FindParent(node, oldLocation);
iMBR = iExtendMBR(leafMBR, newLocation, $\epsilon$, parentMBR);
if newLocation lies within iMBR then
    leafMBR = iMBR;
    Update location of object in leafNode;
    Write out leafNode;
    Update leafMBR in parentNode;
    Write out parentNode; return;
if Deletion of the object from the leaf node leads to underflow then
    Issue a top-down update; return;
Delete index entry for object from leafNode;
Write out leafNode;
if newLocation is contained in the MBR of some sibling node that is not full then
    Retrieve sibling node;
    Insert index entry for object with newLocation into sibling node;
    Write out sibling node;
    return;
ancestor = FindParent(parentNode, newLocation);
/* Issue a standard R-tree insert at the ancestor node */
Insert (ancestor, oID, newLocation);
```

Algorithm 3 FindParent(node, location)

```
$l$ = node.lev + 1; /* start from parent level of the node*/
while $l <$ root level do
    for each entry at level $l$ do
        if some child offset matches node offset then
            if MBR contains location then
                return(entry offset);
    $l$++;
return(root offset);
```

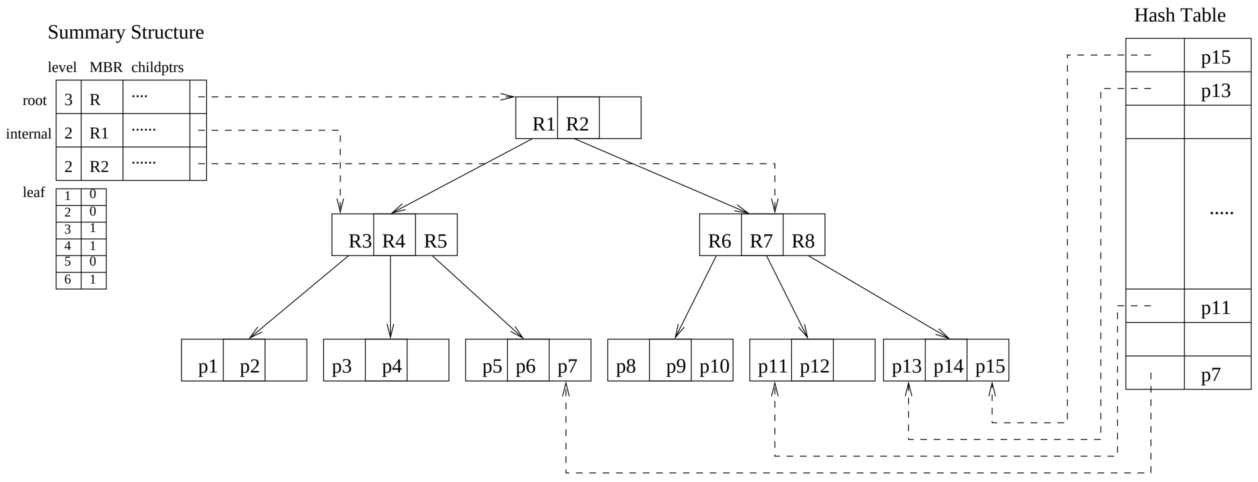


Figure 3: Summary Structure for an R-Tree

Algorithm 4 iExtendMBR (leafMBR, newLocation, ϵ, parentMBR)

```
Let parentMBR by given by $(p_1, q_1, p_2, q_2)$;
Let leafMBR be given by $(x_1, y_1, x_2, y_2)$;
Let newLocation be given by $(x, y)$;
Compute iMBR by enlarging leafMBR: /* Extend only in direction moved. */
if $x < x_1$ then
    Enlarge $x_1$ to min($x_1 - \epsilon, p_1$)
else if $x > x_2$ then
    Enlarge $x_2$ to min($x_2 + \epsilon, p_2$);
if $y < y_1$ then
    Enlarge $y_1$ to min($y_1 - \epsilon, q_1$)
else if $y > y_2$ then
    Enlarge $y_2$ to min($y_2 + \epsilon, q_2$);
return(iMBR);
```

For shifting to sibling nodes, the bit vector for the R-tree leaf nodes in the summary structure indicates whether sibling nodes are full. This eliminates the need for additional disk accesses to find a suitable sibling. After a shift, the leaf's MBR may be tightened to reduce overlap, thus possibly leading to improved query performance.

If the update cannot be carried out in the leaf level, then we call Algorithm 3, $\mathrm{FindParent}()$, to find the lowest level ancestor node that bounds the new position, and re-insert the object with the new location into the subtree rooted at that ancestor node.

By having a more general bottom-up strategy that can cater to different types of updates, the effectiveness of bottom-up update is preserved, even if there is a shift from local towards global. Further, we expect that the generalized bottom-up update strategy will outperform top-down update with a cache, since the page access requirement is usually lower. Indeed, performance studies demonstrate that it offers significant improvements in update performance over the localized bottom-up and top-down methods.

At the same time, we can exploit the summary structure to perform queries more efficiently. We first check for overlap with the root entry in the direct access table and then proceed to the next level of internal node entries, looking for overlaps until the level above the leaf is reached. Equipped with knowledge of which index nodes above the leaf level to read from disk, we carry on with the query as usual. The savings are expected to be significant when the index has received large numbers of updates (overlaps increase), and when the tree height and fanout are high.

3.2.1 Optimizations

We utilize several tuning parameters and optimizations to make the generalized bottom-up strategy more adaptive, the objective being to further improve its performance.

a. Epsilon ϵ: This parameter limits the amount of MBR enlargement. It is set to some small value relative to the average leaf MBR size. The enlargement is specific in the direction of the object's movement. Intuitively, if the object moves Northeast, we enlarge the MBR towards the North and East only. This parameter is already incorporated in Algorithm 4.

b. Distance threshold θ: We track the current speeds of the moving objects. The speed of an object is indicated by the distance moved in-between consecutive updates. A fast-moving object (distance moved is greater than θ) requires a less localized solution, so shifting to a sibling is considered before attempting to extend the MBR. On the other hand, if the distance moved is less than θ, we will try to extend the leaf MBR first before looking at the siblings.

c. Level threshold L: This restricts the number of levels to be ascended from the leaf level. If L is 0, the generalized bottom-up algorithm is reduced to an optimal localized bottom-up, suitable for updates that exhibit high locality. Parameter L is set to the maximum possible (the height of the R-tree -1), as this offers flexibility in the index organization when less localized updates occur. In the performance studies, we use different settings for L to gain insight into the properties of the paper's proposal.

d. Choice of sibling: We pick a suitable sibling from the large set of candidate siblings by first eliminating those that are full. Then we consider the siblings with MBRs that contain the object. When a sibling is chosen, we not only shift one object, but piggyback other equally mobile objects over and thus redistribute objects between the two leaves to reduce overlap.

The performance studies in Section 5 examine the effects of different settings of the parameters discussed.

3.2.2 Concurrency Control

Concurrent access in R-trees is provided by Dynamic Granular Locking (DGL) [2]. DGL provides low overhead phantom protection in R-trees by utilizing external and leaf granules that can be locked or released. The finest granular level is the leaf MBR. Natively, DGL supports top-down operations.

Bottom-up updates fit naturally into DGL as well. Since a top-down operation needs to acquire locks for all overlapping granules in a top-down manner, it will meet up with locks made by the bottom-up updates, thus achieving consistency. The DGL protocol is also applicable to the proposed summary structure. We associate each entry in the direct access table and the bit vector with 3 locking bits for DGL to support the different types of locks.

4 Cost Analysis

We now proceed to analyze the cost of updating the R-tree top-down and bottom-up. Specifically, we compare the update performance of the proposed generalized bottom-up strategy under its worst case to that of the traditional top-down approach under its best case. We assume that the entire data space is normalized to the unit square, and data is uniformly distributed. The distance an object can move is thus bounded by $\sqrt{2}$.

4.1 Cost of Top-Down Update

Lemma 1. Let w be a window of size $x * y$ over the entire data space, where $y \leq x$. Then the probability of a point belonging to w is $x * y$.
Lemma 2. Let $w1$ be a window of size $x1 * y1$ and $w2$ be a window of size $x2 * y2$ over the data space. Then the probability of $w1$ overlapping $w2$ is $min(1, (min(1, x1 + x2)) * (min(1, y1 + y2)))$.

Proof: Suppose we construct a rectangle $w3$ with the dimensions $(x1 + x2) * (y1 + y2)$. Within $w3$, the two windows $w1$ and $w2$ will definitely meet (see Figure 4). Hence, the probability of $w1$ overlapping with $w2$ is given by $min(1, (min(1, x1 + x2)) * (min(1, y1 + y2)))$.

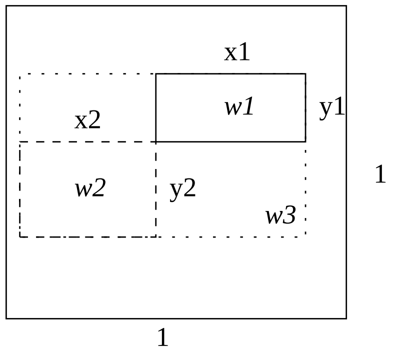


Figure 4: Probability of Two Windows Overlapping

Theorem 1. Let the height of an R-tree be h. Let N_l be the number of nodes at level l, and n_i^l be the i^{th} node of the R-tree at level l. Let $x_i^l * y_i^l$ be the size of the MBR of n_i^l. For a query window of size $x * y$, the expected number of disk accesses is given by

$$D = \sum_{l=1}^{h} \sum_{i=1}^{N_l} (min((x_i^l + x) * (y_i^l + y), 1))$$

Hence, the total cost for a top-down update of R-tree is given by $T = 2 * (D + 1)$. Note that we have added the additional I/O required to write the leaf page to disk to the total cost.

4.2 Cost of Bottom-Up Update

Given a point P whose location is to be updated, let the largest distance from the previous position to its current position be d. The distance moved is a random number from 0 to d. The direction of movement is also random. The area reachable from the point P is $\pi * d^2$. Figure 5 shows the movement of point P during an update. There are three possible cases.

Case 1: The new location of the object remains within the MBR of the leaf node.

The cost incurred is 3 I/O: one read and one write of the leaf node and an additional I/O to read the hash index that provides direct access to the leaf node.

The probability that the new location of an object is still within the leaf MBR is

$$p_1 = \frac{x_1 * y_1}{\pi * d^2} \quad ,$$

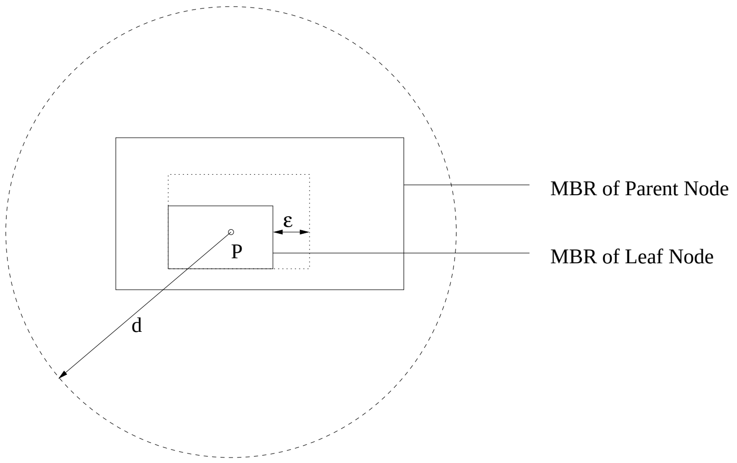


Figure 5: Movement of a Point for Cost Analysis

where $x_1 * y_1$ is the area of leaf MBR; 1 is the leaf level, given that the root level is h.

Case 2: The new location of the object can be accommodated by extending the MBR of the leaf node by ϵ. Note that the actual extension ϵ' is less than ϵ since the leaf MBR is extended intelligently, i.e., only in the necessary direction.

The cost incurred is equivalent to the cost to extend the leaf MBR, which is computed as follows:

1. Read parent MBR to check whether extension of leaf MBR exceeds parent's MBR.
2. If it does not, then we extend leaf MBR by ϵ.

Total cost = 1 (hash index) + 2 (R/W parent) + 2 (R/W leaf node) = 5 I/Os.
The probability that the leaf MBR can be extended is given by

$$p_2 = \frac{(x_1 + \epsilon) * (y_1 + \epsilon) - x_1 * y_1}{\pi * d^2} \quad ,$$

where x_1 + $\epsilon < x_2$ and y_1 + $\epsilon < y_2$; x_2 and y_2 denote the dimensions of the parent MBR.

Case 3: The new location of the object has to be inserted at some sibling node.

The cost incurred is based on the following procedure to determine the nearest sibling node to insert the object.

1. One level above leaf:
   Total cost = 2 (R/W hash index) + 1 (R parent node) + 2 (R/W leaf node) + 2 (R/W sibling node) = 7 I/Os.
2. Recursively traverse up the tree to level l_s, where $l_s \leq h$:
   Total cost = 2 (hash index) + 2 (R/W leaf node) + 2 (R/W sibling node) + $2 * l_s - 1$ (R parent node) $= 5 + 2 * l_s$ I/Os.

Note that we can use the in-memory structure to access the ancestor node which bounds the new location and insert the point from that node. If we use the direct access table in the summary structure to traverse up the tree, then in the worst case, the cost is reduced to a constant that is equal to 2 (R/W hash index) + 2 (R/W leaf node) + 2 (R/W sibling node) + 2 (R parent nodes, assuming that they are different) = 8 I/Os.

The probability that the object has to be inserted into a sibling node is given by

$$p_3 = \frac{x_2 * y_2 - (x_1 + \epsilon) * (y_1 + \epsilon)}{\pi * d^2} + \sum_{i=2}^{l_s - 1} \frac{(x_{i+1} * y_{i+1}) - (x_i * y_i)}{\pi * d^2} ,$$

where $x_1 + \epsilon < x_2$ and $y_1 + \epsilon < y_2$.

Hence, the total cost for a bottom-up update of an R-tree is:

B = Probability(Object moves within MBR) * Cost of inserting into leaf node +
Probability(Leaf MBR can be extended) * Cost of extending leaf MBR and inserting into leaf node +
Probability(Object has to be inserted into sibling node) * Cost of inserting into sibling node
$= 3 * p_1 + 4 * p_2 + 8 * p_3$

With the direct access table in the summary structure, the cost is reduced to a constant of 8 I/Os in the worst case. Since the sum of all the possibilities is 1, we observe that the cost for bottom-up update can be bounded by 8. For top-down update, the best case scenario occurs when there is only one path from the root to the leaf, with a cost of $T = 2h + 2 = 12$. Note that we do not consider the cost for MBR modification propagation and node split in both cases.

Since the theoretical analysis given here indicates that the upper bound for bottom-up update is smaller than the lower bound for top-down update, the former can be expected to offer better performance in practice. The experiments in the next section show that on average, bottom-up outperforms top-down for an R-tree of height 4.

5 Performance Studies

We evaluate the performances of the traditional top-down approach (TD), and two bottom-up approaches: a localized bottom-up version based on Algorithm 1 (LBU) and a generalized version based on Algorithm 2 (GBU). We implemented these algorithms and the original R-tree with re-insertions in C and carried out experiments on a Pentium 4 1.6GHz PC with 512 MB RAM running on Windows XP Professional. We consider a range of tuning parameters, sensitivity and scalability, and throughput. The performance metrics include both disk I/O and CPU time.

By default, all experiments utilize a LRU (Least Recently Used) buffer [8] that is 1% of the database size. For the GBU algorithm, the space consumption of the direct access table is only 0.16%, which is included in the buffer size. A data generator similar to GSTD [18] is used to generate the initial distribution of the objects, followed by the movement and queries. Each object is a 2D point in the unit square that can move some distance in the range of $[0, 0.15]$. Query rectangles are uniformly distributed with dimensions in the range of $[0, 0.03]$. The number of objects ranges from 1 to 10 million, and the density of objects increases proportionally. The resulting R-tree has 5 levels. The number of updates ranges from 1 to 10 million. The number of queries is fixed at 1 million, which are executed on the R-tree obtained after all the updates.

The workload parameters used are summarized in Table 1. Unless stated otherwise, the default parameter values, given in bold, are used. The page size is set to 1024 bytes for all techniques.

Parameter	Values Used
ϵ	0, **0.003**, 0.007, 0.015, 0.03
θ (distance threshold)	0, **0.03**, 0.3, 3
L (level threshold)	0, 1, 2, **3**
Data distribution	Gaussian, Skewed, **Uniform**
Buffers (percentage of database size)	0%, **1%**, 3%, 5%, 10%
Maximum distance moved	0.003, 0.015, **0.03**, 0.06, 0.1, 0.15
Number of updates	**1M**, 2M, 3M, 5M, 7M, 10M
Database size	**1M**, 2M, 5M, 10M

Table 1: Parameters and Their Values

5.1 Sensitivity Experiments

5.1.1 Effect of ϵ

We begin by investigating how different values for ϵ affect the performance of updates and queries. Figures 6(a) and (c) show that GBU performs best in terms of I/O and CPU for updates. The performance of TD remains unchanged since ϵ is applicable to only LBU and GBU. The update costs of LBU initially decreases when ϵ increases because it can extend the leaf MBR more to avoid TD. However, it cannot exploit a large ϵ since the extension is limited by its parent's MBR. Overall, LBU incurs more disk I/O than TD, even when ϵ is 0 (i.e., LBU inserts into siblings only). The reasons include the maintenance of parent pointers during node splits and reinsertions, the checking of sibling nodes to see if they are full, and that not all MBR extensions are successful because they exceed their parent MBRs. Further, LBU always first attempts an MBR extension in all directions, and only then tries to insert into a sibling. This is not always optimal, as later experiments show.

For GBU, a larger ϵ benefits its update costs, since being able to extend an MBR only in select directions reduces the need to ascend, and thus lowers costs. The CPU cost is also lower, as less top-down updates are made. Top-down updates are expensive in terms of I/O, since more nodes need to be read, and in terms of CPU, since more decisions are being considered.

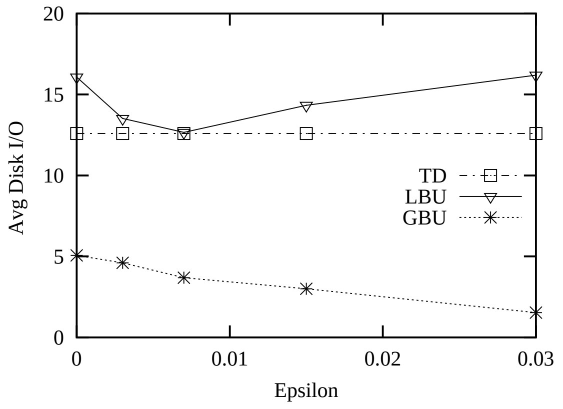
Figures 6(b) and (d) show the query performance. LBU performs slightly worse than TD because the successful MBR extensions increase leaf-node overlaps. GBU performs on par or slightly better than TD if ϵ is small. This is because GBU uses piggybacking when shifting to siblings, which reduces overlap and distributes objects better among leaves, and because it uses the summary structure for better query performance. However, if ϵ is big, query performance degrades significantly. A large ϵ introduces excessive overlaps (not just at the leaf level) in GBU since it can ascend and extend. Hence, a small ϵ (0.003) should be used because it gives GBU excellent update performance and query performance at least on par with TD.

Note that queries are relatively more expensive than updates. This is because the query window size is randomly selected from $[0, 0.03]$ and 1 million objects yields a somewhat high data density.

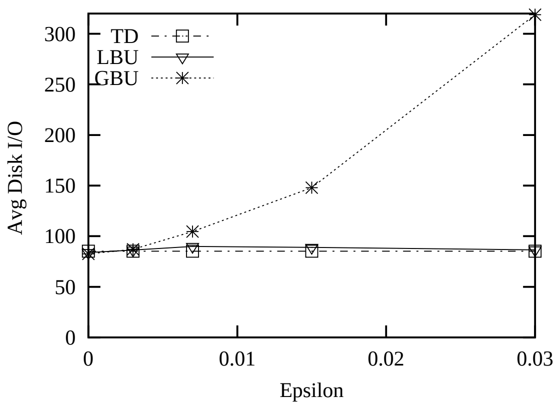
For the rest of the experiments, we omit the CPU graphs as the disk access graphs show similar relative results for both update and queries.

5.1.2 Effect of θ (Distance Threshold)

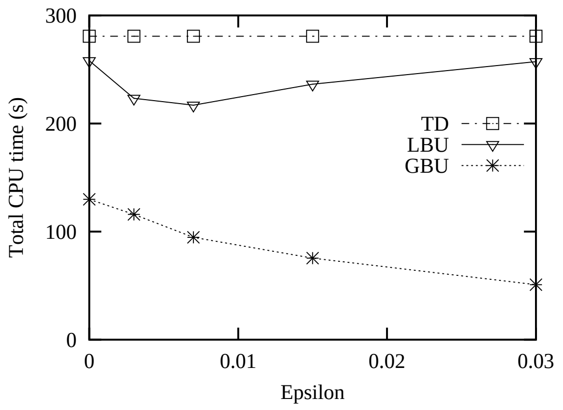
Based on the value of the parameter θ, the GBU algorithm decides whether to use iExtendMBR first or to try to insert into some sibling. A value of 0 implies that shifting will always be attempted first, whereas a large θ implies that GBU tries to execute iExtendMBR first. Figure 7(a) shows that GBU performs best. This is due to the optimizations mentioned earlier. The update performance of GBU increases very slightly



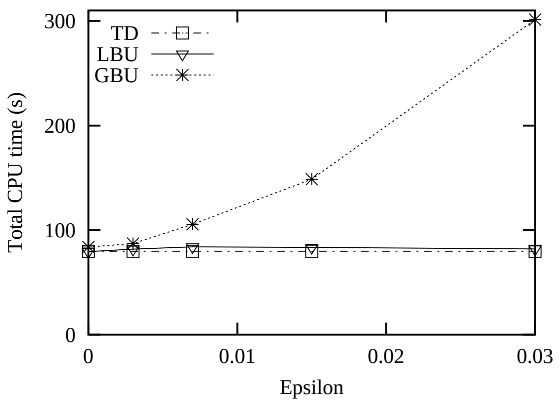
(a) Varying ϵ: Average Disk I/O, Update



(b) Varying ϵ: Average Disk I/O, Querying

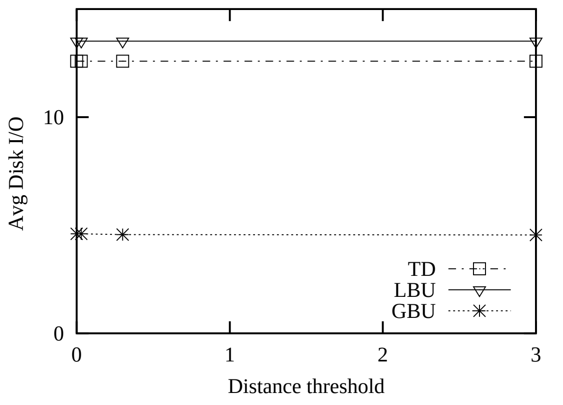


(c) Varying ϵ: Total CPU Cost, Update

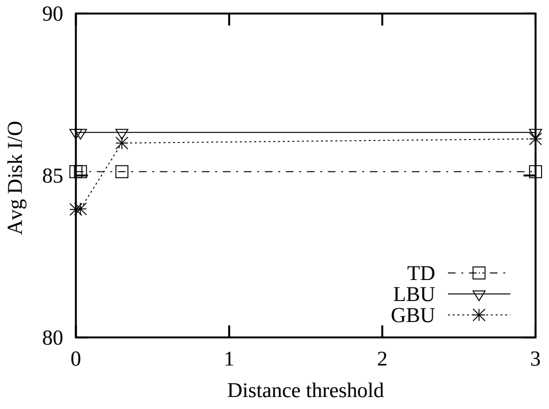


(d) Varying ϵ: Total CPU Cost, Querying

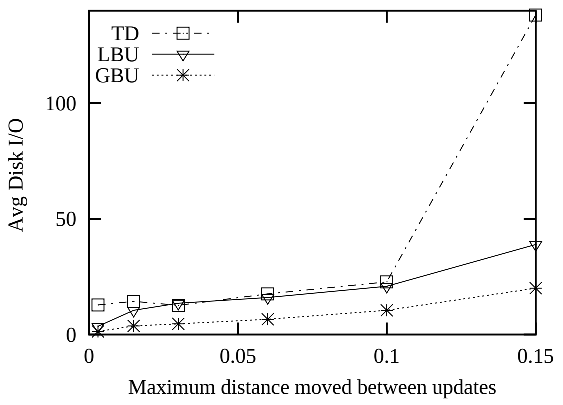
Figure 6: Performance Results I



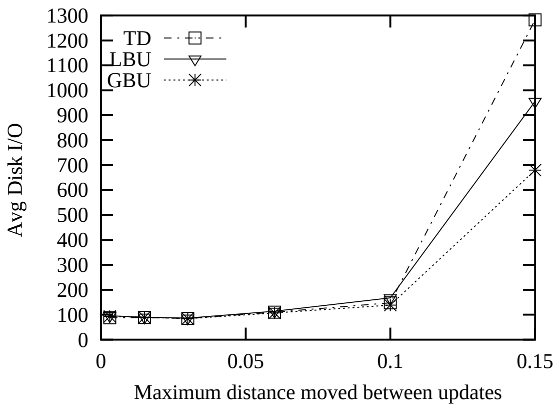
(a) Varying θ, Update



(b) Varying θ, Querying



(c) Varying Maximum Distance, Update



(d) Varying Maximum Distance, Querying

Figure 7: Performance Results II

when θ is large, i.e., when iExtendMBR is favored. This is in line with the expectations. The parameter θ does not affect LBU and TD, and thus their results are constant.

For query performance (Figure 7(b)), GBU performs better than the rest if θ is small and slightly worse than TD if θ is large. This is so because the shifting into a sibling in effect reduces the overlap among leaves and not just prevents enlargements of MBRs. On the other hand, iExtendMBR enlarges MBRs, which introduces more overlaps. From the results, we set θ to be 0.03 since it offers good query and update performance.

5.1.3 Effect of Maximum Distance Moved

We vary the range of the maximum distance moved between updates to investigate how this affects GBU, LBU, and TD. This parameter intuitively gives a measure of how fast the objects move. In Figures 7(c)–(d), the update performances of all techniques deteriorate when the maximum distance increases, since the R-tree index is essentially expanding outwards. This is most pronounced for TD when the maximum distance is 0.15, as a result of increased rate of reinsertion (due to TD deletion) and node splits (due to TD insertion). LBU performs better than TD when objects do not move too fast, as extensions of MBRs can be used more frequently. It is also better when objects move very fast because MBR extension and shifting into sibling can help reduce the rate of reinsertion and node splits due to top-down updates. GBU has the best update performance for similar reasons and because iExtendMBR and shifting into sibling can be done more frequently, as it can ascend higher up the tree when both methods fail. Higher up the tree, a parent MBR is larger, and thus a more global decision can be made.

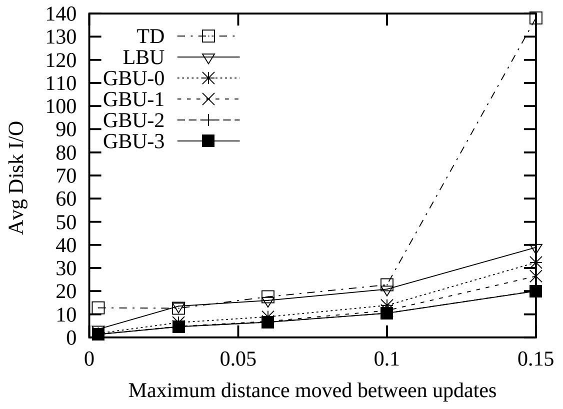
As for query performance, all techniques perform similarly until the maximum distance increase to 0.15. The poor query performance of TD is due to the fact that dead space is increased when objects beyond the root MBR are inserted, and that all updates are handled in the same manner, i.e., top-down. For LBU, updates have little flexibility since they must choose between localized updates at leaf level and global top-down. GBU has the lowest query costs increase when maximum distance is 0.15 because the use of the summary structure for querying and piggybacking when shifting into siblings are able to offset the effects of increases in sizes of MBRs. Overall, GBU has the flexibility to handle localized updates at lower levels and to ascend the tree to apply more global strategies when necessary.

5.1.4 Effect of L (Level Threshold)

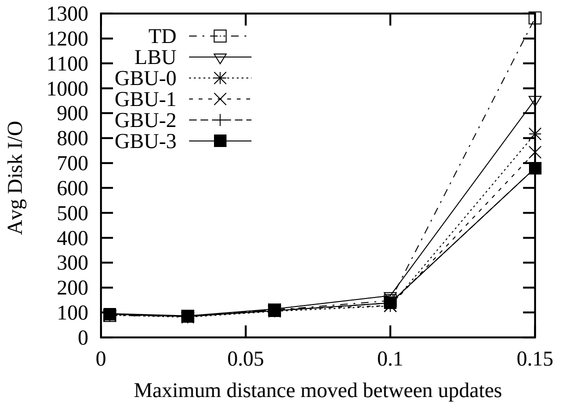
Next, we vary the maximum number of levels L that GBU can ascend from 0 (GBU-0) to 3 (GBU-3) to examine the effect on update and query performance. We also vary the maximum distance moved between updates because the impact of different values for L are not significant for small values of maximum distance.

We obtain two interesting results. First, the update performance (Figure 8(a)) of GBU-0 is better than that of LBU as a result of improved optimizations. Second, GBU-3 (GBU-2 is almost equivalent) performs the best. This is because for the updates that cannot be carried out using iExtendMBR or shifting into a sibling at a lower level, we can ascend up the tree. Note that ascending up to the level below the root is still much cheaper than top-down since bottom-up strategies are only used when there is no risk of split or underflow (reinsertion) that may propagate. For the same reasons as given in the previous experiment, the cost of TD increases sharply when maximum distance is 0.15.

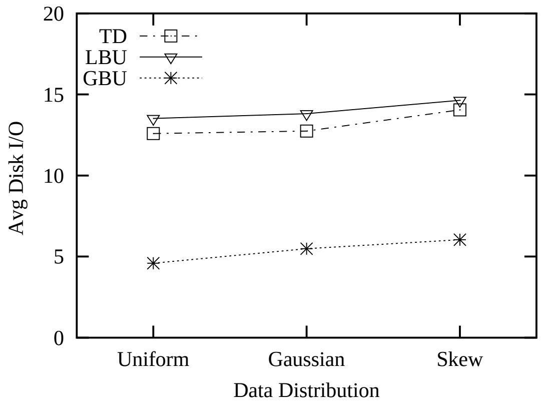
In terms of query performance (Figure 8(b)), GBU-3 incurs the lowest cost since we only ascend the tree if iExtendMBR and shifting into a sibling fails for the current level. This ensures that movement of objects is handled at a lower level in the tree, i.e., locally. For larger movement or persistent movement according to a trend, we ascend up the tree to find a better solution. By having a robust way of handling updates of different nature, the R-tree index remains well organized for querying.



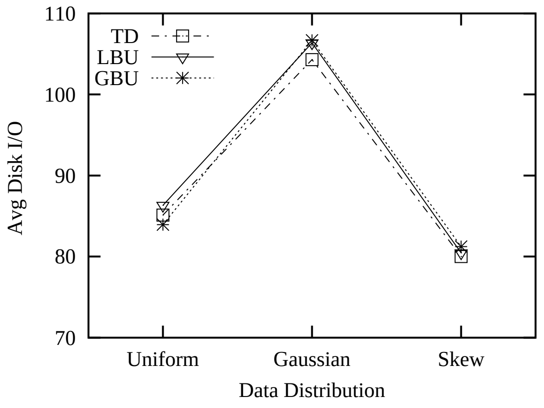
(a) Ascending the R-Tree, Update



(b) Ascending the R-Tree, Querying

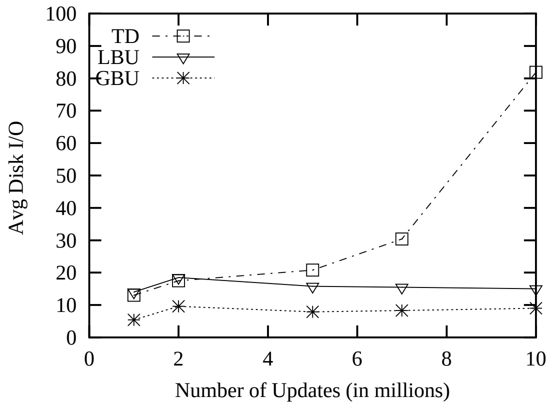


(c) Varying Data Distributions, Update

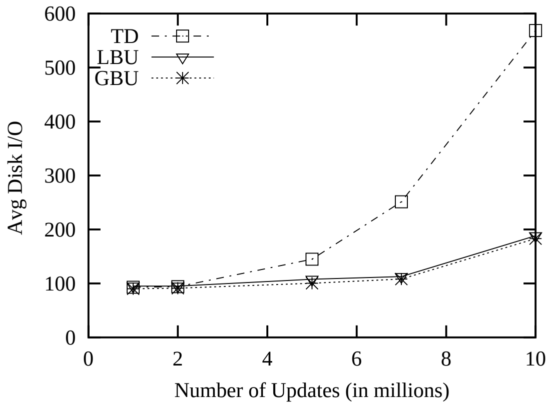


(d) Varying Data Distributions, Querying

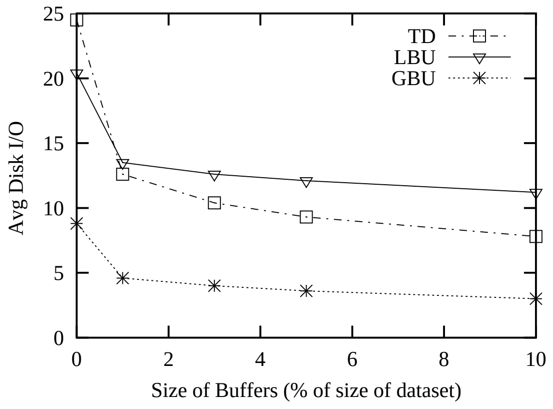
Figure 8: Performance Results III



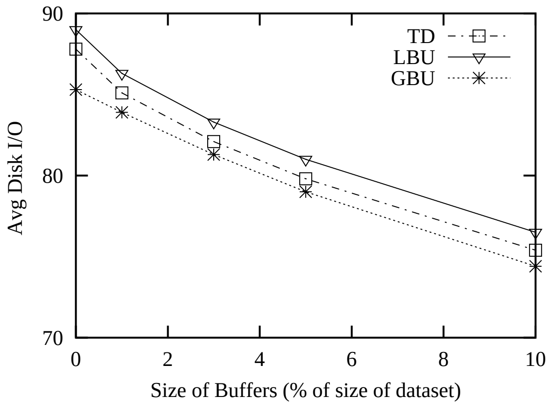
(a) Varying Amounts of Updates, Update



(b) Varying Amounts of Updates, Querying



(c) Varying Buffer Size, Update



(d) Varying Buffer Size, Querying

Figure 9: Performance Results IV

5.1.5 Effect of Data Distribution

Till now, the experiments are run with a uniform distribution. Here, we consider data distributions that are initially Gaussian and skewed. As expected, the update performance is generally best for all techniques when the distribution is uniform (Figure 8(c)). A skewed distribution increases the costs of all techniques, and Gaussian increases the cost for LBU and GBU. With the skewed and Gaussian distributions, data is initially more clustered. As the data objects start to move, more node splits and reinsertions (due to underflow) are likely to occur, which yields an increased cost.

Figure 8(d) shows the query performance. The query rectangles are uniformly distributed across the data space. The techniques perform better for skewed distribution as most of the space is empty, and poorer for uniform and Gaussian because the data are spread out. The query performance of GBU is better for uniform distribution, but slightly worse than the others for the skewed and Gaussian initial distributions.

5.1.6 Effect of Updates

In this experiment, we vary the number of updates from 1 million to 10 million and examine how the R-trees perform after millions of updates. For example, for 1 million updates, we perform the updates followed by the queries. From Figure 9(a), we see that the costs increase as the number of updates increase. This is because the objects would have moved quite far from their original positions, causing the R-tree index to expand with more frequent node splits and reinsertions, particularly after 10 million updates. Overall, GBU, with its robust bottom-up strategies, has the lowest update cost, followed by LBU and then TD. The small spike for TD and LBU at 2 million updates is possibly due to the random nature of the data and movement.

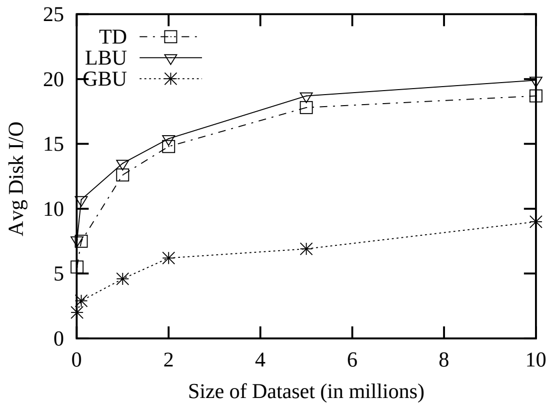
Query-wise (Figure 9(b)), costs increase as more updates are made. As objects move further apart, the amount of dead space in the index increases, and thus false hits are more likely during queries. Again, GBU performs better than does TD. The reasons are similar to those explained earlier, and the results substantiate previous findings. Noting that objects are moving relatively fast and randomly, and we have used numerous updates of 10 million objects, we believe that this is an important finding: TD deteriorates significantly over numerous updates or very fast movement.

5.2 Effect of Buffering

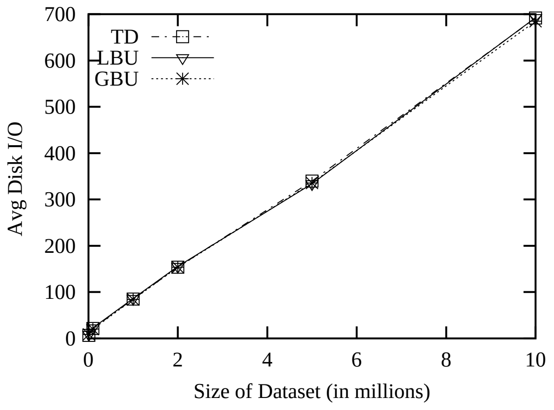
We also investigate both the update and query performance with different amounts of buffer space. We vary the percentage of buffer space to database size from 0% to 10%. From Figures 9(c)–(d), it follows that in the absence of a buffer, LBU performs better than does TD. However, when a buffer is used, its performance drops below that of TD. GBU is significantly better than the rest. For all techniques, update performance improves with increased buffer space, as can be expected. For query performance, all techniques also improve with increased buffer.

5.3 Scalability

In this study, we increase the number of objects from 1 million to 10 million, to determine how scalable the bottom-up strategies are compared to conventional top-down. As we do not expand the data space, we are also effectively investigating the effect of the density of objects. Density increases as the number of objects increases. Looking at Figure 10, update performance decreases with a larger number of objects. Still, GBU performs the best. On a larger scale, the query performance of all techniques is pretty much the same. But we can see that the query costs increase dramatically with more objects (10 million), due to the very high density of objects and, certainly, more node overlaps.



(a) Update



(b) Querying

Figure 10: Scalability

5.4 Throughput

Finally, we study the throughput of bottom-up versus top-down approaches. We employ the Dynamic Granular Locking in R-trees [2] and run the experiments with 50 threads, varying the percentage of updates versus queries. We use window queries within the range of $[0, 0.003]$ with updates. As expected, Figure 11 shows that the throughput for TD and LBU is best when we have 100% queries and worst when we have 0% queries (i.e., 100% updates). The reverse is true for GBU as its optimizations reduce the update costs significantly. The throughput of GBU is consistently better than that of TD, with LBU under-performing TD. From this last set of experiments, we conclude that bottom-up strategies, if properly optimized (GBU), can perform significantly better than top-down updates. As for queries, GBU is still overall better than TD, as it does not degrade as badly under drastic circumstances.

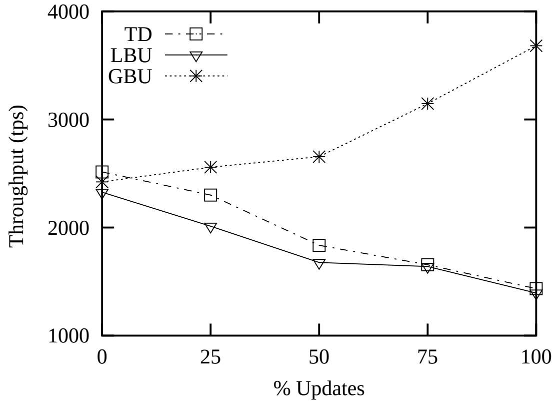


Figure 11: Throughput for Varying Mix of Updates and Window Queries

6 Summary and Research Directions

Motivated by the class of monitoring applications, which are characterized by large volumes of updates, and the increasingly important role of indexing, this paper proposes a generalized bottom-up update strategy for R-trees. This update strategy can easily be applied to the members of the family of R-tree-based indexing techniques, as it preserves the index structure and takes into account concurrency control. The strategy improves the robustness of R-trees by supporting different levels of index reorganization—ranging from local to global—during updates, thus using expensive top-down updates only when necessary.

The paper presents a compact main-memory summary structure along with efficient bottom-up algorithms that reduce the numbers of disk accesses and CPU resources used for update and querying. Empirical studies indicate that the new strategy outperforms the traditional top-down approach for updates in terms of I/O, achieves higher throughput, and is scalable. In addition, indexes that result from the bottom-up updates are more efficient for querying than their top-down counterparts. The query performance for bottom-up indexes does not degrade after even large amounts of updates.

Future research directions include the application of the bottom-update techniques proposed here to other R-tree variants. It may also be of interest to develop a better theoretical and empirical understanding of the apparent, general tradeoff between the global-ness and cost of index update. Briefly, global updates translate into a dynamic index structure that adapts to the data it indexes, which is good for query performance, but also costly in terms of updates. The reverse properties tend to hold for localized updates.

References

[1] N. Beckmann, H-P Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *ACM SIGMOD*, 1990.

[2] K. Chakrabarti and S. Mehrotra. Dynamic Granular Locking Approach to Phantom Protection in R-trees. In *Proc. of ICDE*, 1998.

[3] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of ACM SIGMOD*, 1984.

[4] M. Hadjieleftheriou, G. Kollios, V.J. Tsotras, and D. Gunopulos. Efficient Indexing of Spatio-Temporal Objects. In *Proc. of EDBT*, 2002.

[5] I. Kamel and C. Faloutos. Hilbert R-Tree: An Improved R-Tree Using Fractals. In *Proc. of VLDB*, 1994.

[6] G. Kollios, D. Gunopulos, and V.J. Tsotras. On Indexing Mobile Objects. In *Proc. of PODS*, 1999.

[7] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-Tree. In *Proc. of the Int'l. Conf. on Mobile Data Management*, 2002.

[8] S.T. Leutenegger and M.A. Lopez. The Effect of Buffering on the Performance of R-Trees. In *Proc. of ICDE*, 1998.

[9] D. Pfoser, C.S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *Proc. of VLDB*, 2000.

[10] C. Procopiuc, P. Agarwal, and S. Har-Peled. Star-Tree: An Efficient Self-Adjusting Index for Moving Objects. In *Proc. of ICDE (poster)*, 2002.

[11] N. Roussopoulos and D. Leifker. Direct Spatial Search in Pictorial Databases Using Packed R-Trees. In *Proc. of ACM SIGMOD*, 1985.

[12] S. Saltenis and C.S. Jensen. Indexing of Now-Relative Spatio-Bitemporal Data. *VLDB Journal*, 11(1): 1-16 (2002).

[13] S. Saltenis and C.S. Jensen. Indexing of Moving Objects for Location-Based Services. In *Proc. of ICDE*, 2002.

[14] S. Saltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proc. of ACM SIGMOD*, 2000.

[15] T.K. Sellis, N. Roussopoulos, and C. Faloutos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proc. of VLDB*, 1987.

[16] Y. Tao and D. Papadias. Efficient Historical R-Trees. In *Proc. of SSDBM*, 2001.

[17] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *Proc. of VLDB*, 2001.

[18] Y. Theodoridis, J.R.O. Silva, and M.A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proc. of SSD*, 1999.