Relaxed Space Bounding for Moving Objects: A Case for the Buddy Tree

[1]Shuqiao Guo [1]Zhiyong Huang [2]H. V. Jagadish [1]Beng Chin Ooi [1]Zhenjie Zhang

[1]Department of Computer Science
National University of Singapore, Singapore
{guoshuqi, huangzy, ooibc, zhenjie}@comp.nus.edu.sg

[2]Department of EECS
University of Michigan, USA
jag@eecs.umich.edu

ABSTRACT

Rapid advancements in positioning systems and wireless communications enable accurate tracking of continuously moving objects. This development poses new challenges to database technology since maintaining up-to-date information regarding the location of moving objects incurs an enormous amount of updates. There have been many efforts to address these challenges, most of which depend on the use of a minimum bounding rectangle (MBR) in a multi-dimensional index structure such as R-tree. The maintenance of MBRs causes lock contention and association of moving speeds with the MBRs cause large overlap between them. This problem becomes more severe as the number of concurrent operations increases. In this paper, we propose a "new" simple variant of the Buddy-tree, in which we enlarge the query rectangle to account for object movement rather than use an enlarged MBR. The result is not only elegant, but also efficient, particularly in terms of lock contention. An extensive experimental study was conducted and the results show that our proposed structure outperforms existing structures by a wide margin.

1. INTRODUCTION

Rapid advancements in positioning systems, sensing technologies, and wireless communications, make it possible to track accurately the movement of thousands of mobile objects. This has led to an urgent need to develop techniques of efficient storage and retrieval of moving objects. Indeed, this topic has received significant interest in recent years [8, 12, 7, 15, 13].

Mobile objects move in (typically two or three-dimensional) space. As such, traditional index techniques for multidimensional data are a natural foundation upon which to devise an index for moving objects. A standard technique for indexing objects with spatial extent is to create a *minimum bounding rectangle* (MBR) around the object, and then to index the MBR rather than the object itself. Many multidimensional index structures, including in particular R-tree and its derivatives [5, 1], follow such an approach.

Moving objects, even if they are modeled as points, are in different locations in space at different times. In an index valid over some period of time, if we wish to make sure to locate the moving object, we can do so by means of a bounding rectangle around the locations of the object within this period of time. (Most spatio-temporal indices also have explicit notions of object velocity, and make linear, or more sophisticated, extrapolations on object position as a function of time. But an MBR is still required to make sure that a search query does not suffer a false dismissal). See our discussion of the popular TPR-tree structure [13] in Section 2.3 for more detail.

The key observation we make is that these MBRs can become quite large, particularly as objects have high velocity or as index structures are used to model the world for longer periods of time. Because of this, too many false dismissals may be observed. If an R-tree like structure is used to hierarchically organize these MBRs, we may find large overlaps between non-leaf MBRs, causing search to proceed on multiple paths down the tree.

A fix to the above problem is to have the tree reflect only a very short period of time. (This is actually a fix only with respect to queries regarding current position. Reducing the time interval of index validity actually makes more difficult predictive queries dealing with future positions of objects.) But this means objects have to be removed and inserted from the tree very frequently, as soon as they move more than a little. Traditional multidimensional index structures have not been designed to support such high update rates. There is always the possibility of a node split upon an insertion and this could back up all the way to the root, requiring conservative choices in concurrency control.

In this paper, we attempt to address these difficulties by redefining the problem of indexing mobile objects. Instead of embedding the velocity information within the index, we attempt to capture it in the query. Now, instead of point objects ballooning into large MBRs, we will have point queries being turned into rectangular range queries. On the surface, this appears to make no difference in terms of performance – so one wonders why bother to make this equivalence transformation?

It turns out that the benefit we get is that we can now build much simpler indices – we only need to consider static objects rather than mobile objects. So we can choose a multidimensional structure with good update properties. In particular, we propose a simple indexing structure based on the Buddy-tree [14] – the Buddy*-tree. The bounding rectangles in the internal nodes are not minimum, and are based on the pre-partitioned cells. This turns the Buddy-tree to a multi-level like grid file [11], in that the union of the lower level bounding spaces span the bounding space of the parent.

To allow concurrent modifications, we adapt the concurrency control mechanism of the R-link tree [9]. Since the Buddy*-tree is a space partitioning-based method, it does not suffer from the high-update cost of the R-tree, and due to the decoupling of velocity information from bounding rectangles, it does not suffer from the overlap problem of the TPR-tree. Experimental studies were conducted, and the results that the Buddy*-tree is much more efficient than the TPR*-tree [17] (an improved variant of the TPR-tree) and the B^+-tree [3] based B^x-tree [6].

The rest of this paper is organized as follows. Section 2 surveys previous index techniques for moving objects and concurrency control for index trees. Section 3 and 4 describe the structure and algorithms of the Buddy*-tree. Experimental evaluation is described in Section 5. Finally, Section 6 concludes the paper.

2. RELATED WORK AND ANALYSIS

2.1 Index for Moving Objects

There is a long stream of research on the management and indexing of spatial and temporal data, which eventually led to the study of spatio-temporal data management. MOST [15] is one such early effort. By treating time as one dimension, moving objects in a d-dimension space can be indexed in $(d+1)$-dimensions. Thus, the predicted near future state of an object can be queried.

The *TPR-tree* (the Time Parameterized R-tree) [13] is a popular R-tree based index conceptually similar to MOST. Velocity vectors of objects or MBRs as well as the dynamic MBRs at current time are stored in the tree. At a non-leaf node, the velocity vector of the MBR is determined as the maximum value of velocities in each direction in the subtree. A good study of the performance of TPR-tree is in [17]. The *TPR*-tree* [17] was proposed to improve the TPR-tree by employing a new set of update algorithms. For insertions, TPR*-tree maintains a QP (priority queue) to record the candidates paths which have been inspected. By visiting the descendant nodes, TPR*-tree extends the paths in QP until a global optimal solution is chosen, while TPR-tree only chooses a local optimal path.

The B^x-tree [6] is a B^+-tree structure that indexesg moving objects after performing a transformation into single-dimensional space using a space filling curve. Objects are partitioned based on time, but indexed in the same space.

Indices based on hashing have been proposed to handle moving objects [16], [2]. Combinations have also been proposed. For instance, in [4], hashing on the grid cells is used to manage hot moving objects in memory, while the TPR-tree is used to manage cold moving objects on disk, as a way to provide efficient support for frequent updates.

2.2 Concurrency in the B-Tree and R-tree

The B-link tree [10] was proposed to provide efficient concurrent traversal and update of the B^+-tree. Every node keeps a right link pointing to the right sibling node in the same level. When a search process without lock-coupling goes down in the tree, it will learn of any splits racing with it by comparing keys. It is then able to visit the new split node along the right link chain before the new node is installed into the tree.

The R-link tree [9] employs a similar modification for the R-tree. The main difference between the R-tree and B^+-tree is that keys in R-tree are not ordered. Therefore, LSN (logical sequence number) is introduced to each node and kept in each entry of the internal nodes. Comparison of these LSNs is used to discover node splits. A right link chain is again used to locate newly split nodes.

2.3 MBR Expansion Due to Velocity

Let $x_i^l(0), x_i^u(0)$ be the lower bound and upper bound of some MBR respectively on dimension i at time 0, and $\vec{\boldsymbol{u}}_i^l, \vec{\boldsymbol{u}}_i^u$ be the minimum and maximum velocity of the points in the MBR on dimension i . After t time units, the volume of this MBR is $V = \prod_{i=1}^{d}(x_i^u(t) - x_i^l(t))$. Since $x_i^l(t) = x_i^l(0) + \vec{\boldsymbol{u}}_i^l \cdot t$ and $x_i^u(t) = x_i^u(0) + \vec{\boldsymbol{u}}_i^u \cdot t$, the volume of MBR can be rewritten as $V = \prod_{i=1}^{d}[(x_i^u(0) - x_i^l(0)) + (\vec{\boldsymbol{u}}_i^u - \vec{\boldsymbol{u}}_i^l) \cdot t]$. Therefore,

$$\frac{\partial V}{\partial t} = \sum_{i=1}^{d}\{(\vec{\boldsymbol{u}}_i^u - \vec{\boldsymbol{u}}_i^l) \cdot \prod_{i'=1, i' \neq i}^{d}[(x_i'^u - x_i'^l) + (\vec{\boldsymbol{u}}_i'^u - \vec{\boldsymbol{u}}_i'^l) \cdot t]\}$$

That is, $\frac{\partial V}{\partial t}$ is $O(t^{d-1})$.

The probability of any MBR being accessed by a random point search query, assuming uniform distributions, is proportional to the volume of the MBR. Therefore the expected number of MBRs accessed at any level of the index tree is proportional to the sum of their volumes. The rate of the increase on the expected number of MBRs to be accessed at some level l is $O(t^{d-1})$, where t is the elapsed time and d is the dimensionality. This shows that traditional indexing methods for moving object deteriorate greatly due to the overlap problem if there is no update for a long time.

3. OVERALL INDEXING STRUCTURE

3.1 Indexing Snapshots

In this paper, we adopt the basic assumption of maximum update interval of the moving objects. Under such an assumption, a moving object must update its movement at least once in every T_{ui} timestamps.

Thus, a series of snapshots indices are constructed on different reference timestamps. The reference timestamp of the ith index is on $T_{ui}(i - 0.5)$. There is a Buddy*-tree indexed at these reference timestamps for the moving objects. The detail of Buddy*-tree will be covered later in this paper. The lifespan of the ith index tree is between $T_{ui}(i-1)$ and $T_{ui}(i+1)$. Thus, there are two indexing trees maintained at the same time in our system. Figure 1 shows an example of the indexing trees.

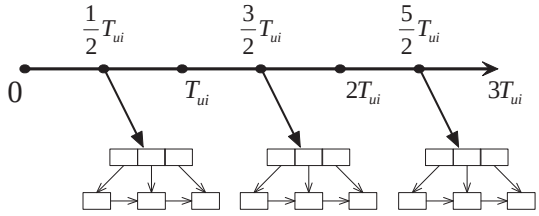


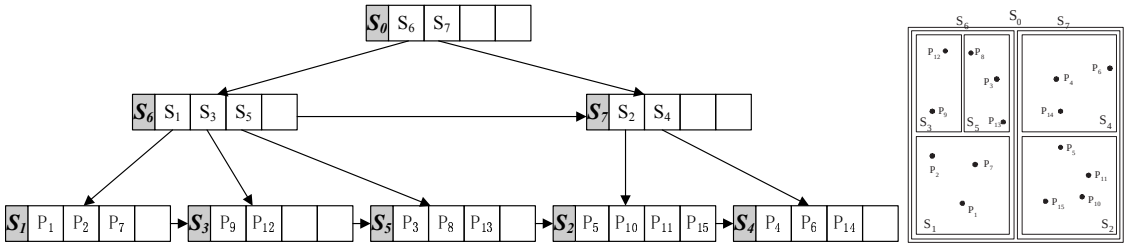
Figure 1: The indices on the snapshots

In our system, every object is indexed by one Buddy*-tree at a single timestamp. Assume an object updates its motion at timestamp t_u, it will be received and updated by the $(\lfloor t_u/T_{ui} \rfloor + 1)$th Buddy*-tree, according to the velocity and the position of the object at the reference time of the Buddy*-tree. If the object is previously indexed by the ith Buddy*-tree but the new update occurs on the $i+1$th tree, the object will be deleted from the ith tree as well as inserted into $i+1$th tree. For example, if $T_{ui} = 120$ and an object updates at $t_u = 110$, the update will be conducted on the first Buddy*-tree at timestamp 60. If it updates again at $t_u = 130$, the first tree will delete the object, while the second tree at timestamp 180 will insert the object into itself. It is guaranteed that the ith index tree must be empty at the end of its lifespan, since every object will update at least once between timestamp $T_{ui}i$ and $T_{ui}(i+1)$.

If the user issues a predictive range query q on timestamp t_q, our system will transform the queries to both the index trees currently maintained. Since every object is stored in at least one tree, the union of the range query result on these two trees must be a complete and valid result for the original query. The detail of the query processing on the Buddy*-tree will be covered later.

3.2 Buddy*-Tree

Given that we regard the moving objects as static points to index in each snapshot, and given the importance of fast update, we choose the Buddy-tree [14] as the basic structure of our proposed index. The index tree is constructed by cutting the space recursively into two subspaces of equal size with hyperplanes perpendicular to the axis of each dimension. Each subspace is recursively partitioned until the points in the subspace fit within a single page on



(a) The Buddy*-Tree

(b) A planar representation

Figure 2: An Example of the Structure of Buddy*-Tree

disk.

We make several alterations to this basic Buddy-tree structure to suit our needs. We call the new index structure Buddy*-tree. A traditional Buddy-tree creates tight bounding rectangles around the data points in each node. Since such tight MBRs are costly to update, we choose instead to use loose bounding. We call this a *Loose Bounding Space* (LBS) associated with the index tree node. We also store the maximum and minimum velocities in a node for all of the objects stored in the subtree of the node.

To support high degree concurrent operations on Buddy*-tree, we absorb the idea of right links among each level from B-link tree [10] and R-link tree [9]. Thus, at any given level all nodes are chained into a singly-linked list. In [9], the authors extends the R-Tree by assigning an additional parameter LSN as the timestamp to each node, which is used to detect the split and determine where to stop when moving right along the right link chain. However, this structural addition is not necessary in the Buddy*-tree since we are guaranteed not to have overlaps between nodes. Instead, we can simply detect the uninstalled node split by comparing the current LBS of the nodes with the old LBS. Figure 2(a) shows an example of the Buddy*-Tree in 2-dimensional space, with the corresponding data space illustrated in Figure 2(b). The first capital letter in a node denotes the LSB of it, followed by the entries with key LSB (expected LBS for the child node) or points.

3.3 Query Expansion on Buddy*-Tree

On every snapshot Buddy*-tree index, we use query expansion instead of MBR expansion for query answering. Our central idea is that movement of objects can be handled by expanding queries rather than actually perturbing objects in the index.

As in so many other moving object index structures, we use linear interpolation to estimate object position at times other than t_{ref}. The position of an object at time t can be calculated by the function $\boldsymbol{x}(t) = \boldsymbol{x}(t_{ref}) + \vec{\boldsymbol{v}} \times (t - t_{ref})$.

Based on this, we can suitably enlarge a query as follows: Suppose the query is q with query window [qx_i^l, qx_i^u] ($i = 0, 1, ...d-1$,where d is dimension of the space), and the query time is t_q, the enlarged query window [eqx_i^l, eqx_i^u] is obtained as:

$$eqx_i^l = \begin{cases} qx_i^l + \vec{\boldsymbol{u}}_i^l \cdot (t_{ref} - t_q) & if t_q < t_{ref} \\ qx_i^l + (-\vec{\boldsymbol{u}}_i^u) \cdot (t_q - t_{ref}) & otherwise \end{cases}$$

$$eqx_i^u = \begin{cases} qx_i^u + \vec{\boldsymbol{u}}_i^u \cdot (t_{ref} - t_q) & if t_q < t_{ref} \\ qx_i^u + (-\vec{\boldsymbol{u}}_i^l) \cdot (t_q - t_{ref}) & otherwise \end{cases}$$

where $\boldsymbol{u}_i^l$ and $\boldsymbol{u}_i^u$ are the minimum and maximum velocities respectively of objects inside the query window in dimension i. Note that we would ideally have liked to enlarge the query by precisely the velocities of the objects included in the query. Although we do not have idea about which objects are in the query result, the maximum and minimum velocity stored in the nodes is enough to help us find the correct result. The following theorem is straightforward and used in next section.

THEOREM 1. *Given a query q and a MBR node N, the enlarged query q' must overlap N if N contains at least one object in the result of q.*

4. BUDDY*-TREE OPERATIONS

In this section, we provide the detailed algorithm on the operations of Buddy*-tree, including querying, insertion and deletion. We also discuss how to achieve high concurrency on Buddy*-tree.

There are three kinds of locks on the nodes of Buddy*-tree, read lock, write lock and mark lock. If a node is read locked, this node can be read by other threads but can not be written. If a node is write locked, it can not be read or written by any other thread. If the node is mark locked, all read and write operations, except merge, can be run on it. In the following, we use r_lock (r_unlock), w_lock (w_unlock) and m_lock (m_unlock) to denote the locking (unlocking) operations for read lock, write lock and mark lock respectively.

4.1 Querying

Algorithm 1 Range_Search(r, N, l)

/* Input: r is the query window. N and l are the pointer of the node to be examined and its LBS obtained from its parent node, respectively*/

```
1: r_lock(N)
2: if N is mark locked by this thread then
3:    m_unlock(N)
4: for each entry e in N that e.LBS overlaps R, obtained by enlarging r
   according to the time difference and extremal velocities in e.node do
5:    if N is not a leaf node then
6:       Insert (e.node, e.LBS) into tobeVisited
7:       m_lock(e.node)
8:    else
9:       output qualified points in e
10: r_unlock(N)
11: while tobeVisited is not empty do
12:    (N', l') = GetFirst(tobeVisit)
13:    Range_Search(r, N', l')
14: if N.LBS is not equal to l then
15:    traverse the right link chain starting at N to first node whose LBS is
       not contained in l
16:    for each node M along the chain except the last one do
17:       r_lock(M)
18:       l' := M.LBS
19:       r_unlock(M)
20:       Range_Search(r, M, l)
```

In Algo 1, we provide the detail of the recursive range search over Buddy*-tree. The searching process is similar to other tree-

indexing structure. For a non-leaf node in Buddy*-tree, the algorithm retrieves all the children with overlap with the expanded query, and put all these children into the list for nodes to be visited later (line 5 to line 13). Then, the algorithm further visits and search the new nodes created by other threads on the right link chain of the current node (line 14 to line 20).

By Theorem 1, the querying algorithm can not miss any node containing at least one point in the query result. Therefore, it can always output the correct result in all cases.

The read lock is exerted on the node when the range search algorithm tries to retrieves the children node. Such a lock is used also during the process of searching uninstalled nodes on the right chain to get the correct LBS.

4.2 Insertion

To insert a point into Buddy*-tree, our system first computes the location of the point at the reference time of the tree. Then, the simple location searching operation is invoked to locate the leaf node in Buddy*-tree whose LBS covers the point. In Algo 2, we present the recursive implementation of how to insert an entry into a Buddy*-tree node. Given the node to insert the point, the algorithm first finds the correct node if uninstalled split nodes exist (line 1 to line 4). If the node to insert still has room for a new entry, the algorithm directly inserts it into it (line 5 to line 7). Otherwise, some split operations are invoked, which is followed by recursive insertion operation on the parent of the node (line 9 to line 18).

Algorithm 2 Insert_Entry(s, N)

/* Input: s is the entry containing a point or a branch to install into node N. Nodes N as input is write-locked and it is unlocked after the procedure, */

```
1: while N.LBS doesn't cover s.LBS do
2:    N' := N's right neighbor on the chain.
3:    w_lock(N') and w_unlock(N)
4:    N := N'
5: if there is an empty entry e in N then
6:    put s in e
7:    w_unlock(N)
8: else
9:    newN = Split_Node(N) /* Split N into two nodes N and newN */
10:   Choose one from N and newN to insert s
11:   if N is not root then
12:      P := N's parent node
13:      w_lock(P) and w_unlock(N)
14:      e := the entry containing newN
15:      Insert_Entry(e, P)
16:   else
17:      Construct a new root and insert N and newN into it
18:      w_unlock(N)
```

The write lock is exerted on a node in two cases. The first case happens when the algorithm is trying to insert an entry to the node. The second case happens when the algorithm finds an uninstalled split, it moves the cursor as well as the write lock to the right nodes on the chain.

4.3 Deletion

The deletion operation is similar to insertion operation. In the first step, the leaf node containing s is first located, followed by Algo 3 to remove the entry and merging its parent if necessary.

In Algo 3, we present the detailed process of deleting an entry from the tree. The algorithm first locates the correct node by traversing on the right chain (line 1 to line 4). The entry containing the point is directly removed if it is found in the correct node (line 5 to line 8). Once the resulting node has too few entries, merging operations is invoked if the right neighbor on the right chain is its buddy in the Buddy*-tree. The merging process is finished after the parent of the two nodes removes one of the buddies (line 9 to line 19). We note that the merge operation can be delayed due to the mark lock exerted by querying operation.

Algorithm 3 Delete_Entry(s, N)

/* Input: s is the entry containing a point or a branch to install into node N. Nodes N as input is write-locked and it is unlocked after the procedure, */

```
1: while N.LBS doesn't cover s.LBS do
2:    N' := N's right neighbor on the chain.
3:    w_lock(N') and w_unlock(N)
4:    N := N'
5: if find the entry s in N then
6:    delete s from N
7: else
8:    w_unlock(N) and report error
9: if there are too few entries in N then
10:   M := N's right neighbor on the right chain
11:   w_lock(M) /* delayed if M is mark locked */
12:   if M and N are buddies and the total number of entries in M and
      N below maximum entry bound then
13:      P := N's parent node
14:      w_lock(P)
15:      Merge M into N
16:      e :=the entry containing M
17:      Del_Entry(e, P)
18:   else
19:      w_unlock(M)
20: w_unlock(N)
```

Write lock is exerted on at most three nodes in the algorithm, including the node N containing s, N's right neighbor M and their parent P.

4.4 Deadlock Analysis

There can be no deadlock between any two threads in the system. We analyze the situations based on the operations of the threads as follows.

Two threads of the same type of operations must be safe from deadlock, since the threads must lock the nodes on the same direction. For one querying thread and one insertion thread, there can be no deadline, since the querying node can read lock at most one node at the same time, while the insertion node can wait for the release of it. For one insertion thread and one deletion thread, they are also deadlock free, since both threads work from bottom to top on the tree. For one querying thread and one deletion thread, on the first hand, the read lock from querying thread can not have deadlock with the write lock from deletion thread. On another hand, the mark lock is released after the querying thread exerting the read lock, so it can not block deletion thread either.

5. EXPERIMENTAL EVALUATION

We implemented the Buddy*-tree, and compared its performance to that of the TPR*-tree and B^x-tree. All of these structures were implemented in C. All experiments were conducted on a single CPU 3G PentiumIV Personal Computer with 1 G bytes of memory.

We ran two sets of experiments, one with a single thread of activity, and another with multiple concurrent threads. In both sets of experiments we use synthetic uniform datasets. The position of each object in the data set is chosen randomly in a 1000×1000 space. Each object moves in a randomly chosen direction with a randomly chosen speed ranging from 0 to 3. We constructed the index at time 0. The parameters used in the experiments are summarized in Table 1, and the default values are highlighted in bold.

Parameter	Setting
Page size	4K
Max update interval	60,**120**,180,240
Max predictive interval	120
Query window size	**10**,20,...,100
Number of queries	200
Dataset size	100K,...,**500K**,...1M
Number of threads	2,4,8,...,**64**,128,256
Number of operations per thread	200

Table 1: Parameters and Settings

5.1 Storage Requirement

In a Buddy*-tree internal entry, the space partition is kept for the child node (at least 8 Bytes for 2-dimension space). As for B^x-tree, each entry contains a 64bit key (8 Bytes). However, a TPR*-tree internal node stores MBRs and VBRs for each child entry (24 Bytes for 2-dimensions). The storage requirement of the indices is shown in Figure 3. As anticipated, TPR*-tree requires more than twice storage space of the others, which are comparable.

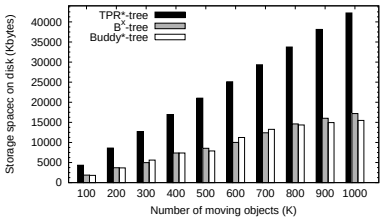
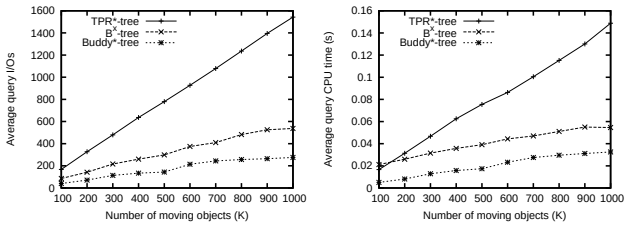


Figure 3: Storage Requirement

5.2 Single Thread Experiments

5.2.1 *Effect of Dataset Size*

First, we study the range query performance with different sizes of dataset. 200 window queries with size 10 are issued after the index running for 120 time units (an entire maximum update interval). The predictive intervals of the queries are randomly chosen in the range from 0 to 120. Figure 4 shows the average cost of I/O operation and CPU time per query for the three inspected indices.



(a) I/O cost (b) CPU time

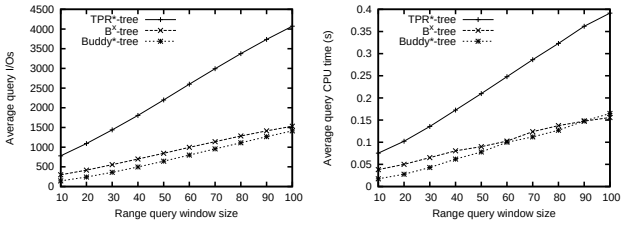
Figure 4: Effect of Dataset Size on Range Query Performance

As expected, the results show that the window query costs of all the indices increase with the number of objects. However, The increasing speed of TPR*-tree is much higher than that of the others. When there are 1M objects in the dataset, the cost of the TPR*-tree is nearly 3 times over that of B^x-tree and more than 5 times over that of Buddy*-tree. This is due to the fact that, the performance of TPR*-tree highly depends on the ratio of overlap, while that of B^x-tree and Buddy*-tree is related only to the result sizes of the queries.

5.2.2 *Effect of Query Size*

We next investigate the performance of the indices with respect to query size.

In the experiments we vary the query window size from 10 to 100 on a dataset of size 500K. The predictive intervals of queries are set as same with last experiments. As shown in Figure 5, query costs increase with the query window size. This behavior is straightforward, since a larger window contains more objects and accordingly, more index nodes will be accessed. The TPR*-tree degenerates considerably over the other indices. This behavior is attributed to the overlap problem of TPR*-tree. The costs of B^x-tree and Buddy*-tree increase at the same rate.

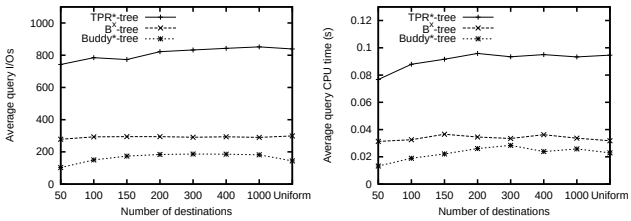


(a) I/O cost (b) CPU time

Figure 5: Effect of Query Window Size on Window Query Performance

5.2.3 *Effect of Data Distribution*

This experiment uses the network dataset to study the effect of data distribution on the indexes. The dataset is generated by an existing data generator, where objects move in a road network of two-way routes that connect a given number of uniformly distributed destinations [13]. The dataset contains 500K objects, that are placed at random positions on routes and are assigned at random to one of three groups of objects with maximum speeds of 0.75, 1.5, and 3. Objects accelerate as they leave a destination, and they decelerate as they approach a destination. Whenever an object reaches its destination, a new destination is assigned to it at random.



(a) I/O cost (b) CPU time

Figure 6: Effect of Data Distribution on Range Query Performance

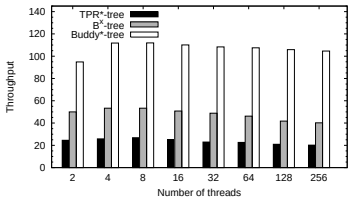
Figure 6 summarizes the average range query costs of the three indexes when the number of destinations in the simulated network of routes is varied. Decreasing the number of destinations adds skew to the distribution of the object positions and their velocity vectors. As is shown, increased skew leads to a decrease in the range query cost in the TPR*-tree, since when there are more objects with similar velocities, they are easier to be bounded into rectangles. The performance of the B^x-tree is not affected by the data skew because objects are stored using space-filling curves, which is not sensitive to density. Observe that the range query cost of the Buddy*-tree firstly increases with the number of destinations and after the point that the number of destination is 300, the cost descends.

5.3 Concurrent Operations

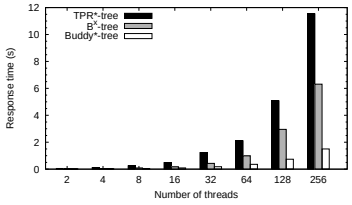
We implemented B-link for B^x-tree, and simply locked the whole TPR*-tree [1]for concurrency control. In the following experiments, each thread issues 200 operations, and the workload of each thread contains the same number of queries and updates.

5.3.1 Effect of Thread Number

First, we investigate the effect of the number of threads. Figure 7 shows the throughput and response time for the indices by varying the thread number from 2 to 256.



(a) Throughput



(b) Response time

Figure 7: Effect of Threads on Concurrent Operations

All the indices reach the highest throughput at around 8 threads and thereafter show deteriorating performance as the number of threads is increased. Measuring the decline as we go from 8 to 256 threads, we find this decrease to be only 6.5% for Buddy*-tree, but 24.5% and 24.6% for B^x-tree and TPR*-tree respectively. Since Buddy*-tree has been designed for high concurrency, its superior performance with multiple threads validates our design. The decline in performance of TPR*-tree is also to be expected. The surprise is the decline in performance of the B^x-tree in spite of the use of B-link chain for high concurrency. The main reason for this is that a lot of "jumps" in the B^x-tree for range query increase the number of accesses of and locks on internal nodes.

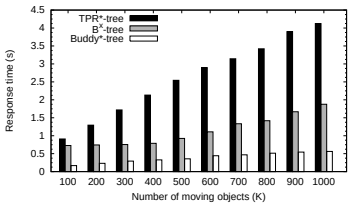
5.3.2 Effect of Dataset Size

As shown in Figure 8, the performance of all indices reduces with the increasing number of moving objects. This is straightforward, since the larger the dataset is, the more nodes an index contains and the more I/O operations a query or update brings. However the Buddy*-tree outperforms the other indices for both throughput and response time.

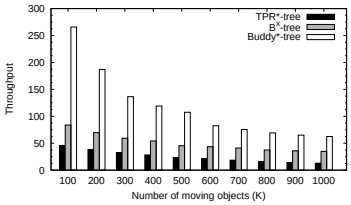
6. CONCLUSION

In this paper, we proposed a space partitioned based index structure Buddy*-tree, a generalization of Buddy-tree, for indexing mobile objects. An adaptive query expansion technique is used for predictive range query over object motion, while only static snapshots indexed, with a right link structure to achieve higher concurrency.

[1]Since TPR*-tree employs different update algorithms from TPR-tree (e.g. remove and reinsert a set of entries in split algorithm), it can not grantee RR even we implement R-link for it.



(a) Throughput



(b) Response time

Figure 8: Effect of Data Size on Concurrent Operations

7. REFERENCES

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *the Proceedings of ACM SIGMOD*, 1990.

[2] H. D. Chon, D. Agrawal, and A. E. Abbadi. Using space-time grid for efficient management of moving objects. In *the Second ACM international workshop on Data engineering for wireless and mobile access*, 2001.

[3] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[4] B. Cui, D. Lin, and K. L. Tan. Towards optimal utilization of main memory for moving object indexing. In *Proceedings of DASFAA*, 2005.

[5] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *the Proceedings of ACM SIGMOD*, 1984.

[6] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *Proceedings of VLDB*, 2004.

[7] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *Proceedings of the International Workshop on Spatio-Temporal Database Management*, 1999.

[8] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. pages 261–272, 1999.

[9] M. Kornacker and D. Banks. High-concurrency locking in r-trees. In *Proceedings of VLDB*, pages 134–145, 1995.

[10] P. Lehman and S. Yao. Efficient locking for concurrent operations on b-trees. *ACMTODS*, 6(4), Dec. 1981.

[11] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An adaptable, symmetric multikey file structure. *ACM TODS*, 9(1):38–71, 1984.

[12] B. C. Ooi, K. L. Tan, and C. Yu. Frequent update and efficient retrieval: an oxymoron on moving object indexes? In *Proceedings of International Web GIS Workshop*, 2002.

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

[14] B. Seeger and H. P. Krieger. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proceedings of VLDB*, pages 590–601, Aug. 1990.

[15] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *the Proceedings of ICDE*, pages 422–432, 1997.

[16] Z. Song and N. Roussopoulos. Hashing moving objects. In *the Proceedings of the 2nd International Conference on Mobile Data Management*, 2001.

[17] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *the Proceedings of VLDB*, 2003.