TOWARDS HISTORICAL R-TREES

Mario A. Nascimento*
CNPTIA – EMBRAPA
P.O. Box 6041
13083-970 Campinas SP BRAZIL
mario@cnptia.embrapa.br

Jefferson R. O. Silva†
IC – UNICAMP
P.O. Box 6176
13083-970 Campinas SP BRAZIL
972147@dcc.unicamp.br

ABSTRACT

R-trees are the "de facto" reference structures for indexing spatial data, namely the minumum bounding rectangles (MBRs) of spatial objects. However, R-trees, as currently known, do not support the evolution of such MBRs. Whenever an MBR evolves, its new version replaces the old one, which is therefore lost. Thus, an R-tree always shows the current state of the data set, not allowing the user to query the spatial database with respect to past states. In this paper we extend the R-tree in such a way that old states are preserved, allowing the user to query them. The proposed approach does not duplicate nodes which were not modified, thus saving considerable storage space. On the other hand, the query processing time does not depend on the number of past states stored.

1 INTRODUCTION

Managing multidimensional data is needed in many application domains, e.g., spatial databases and geographical information systems [13]. In particular, indexing spatial data is of foremost importance in many application domains, and indeed such an issue has been quite well researched. Samet [13] and Gaede and Günther [4] present excellent surveys on the area. However, it is hardly argueable that an structure has been more cited and used as a reference than the R-tree [5]. The R$^+$-tree [14] and the R*-tree [1] are well known R-tree derivatives, where the R*-tree has been shown to be quite efficient. Recently, the Hilbert R-tree [7] and the STR-tree [9] have been shown to have better "packing" capabilities. This is specially useful for sets of data which are not very dynamic in nature. With the exception of the R$^+$-tree, all have the same basic structure. K-dimensional spatial objects are modeled by their Minimum Bounding Rectangles (MBRs) – we assume, without loss of generality, that $K = 2$. Subsets of the indexed MBRs are organized into overlapping subspaces, using a tree hierarchy. They all differ in the way the tree nodes are split (when overflown) and/or MBRs are assigned to subspaces (i.e., nodes in tree).

In this stage of our research we assume an R-tree has already been built by using algorithms from any of the R-tree derivatives, but the R$^+$-tree. Regardless of how it was built, we refer to such an structure as an R-tree, and we assume it obeys the following conditions [5] (where a hyper-rectangle is an MBR):

- Every leaf (non-leaf) node contain between m and M index records (children) unless it is the root;
- For each index record (I, Tid) in a leaf node, I is the smallest hyper-rectangle that spatially contains the object represented by the indicated tuple Tid;
- For each entry (I, Cid) in a non-leaf node, I is the smallest hyper-rectangle that spatially contains the hyper-rectangles in the child node Cid;
- The root node has at least two children unless it is a leaf;
- All leaves appear on the same level.

For simplicity, and proof-of-concept, we base our algorithms for insertion, deletion and updating on those by Gutmann [5]. MBRs can model objects which vary with time, one trivial example is a farm which can be expanded or shrunk by selling or buying land. Similarly new objects can begin or cease to exist. An R-tree indexes only the current MBR for each object. Should any object evolve and have its MBR changed, the R-tree must delete the old MBR and insert the new one (the one corresponding to the new instance of the object). From that point on, no query will ever take into account that past instance of that particular MBR. In other words, R-trees as currently known, allow querying *only* the current state of a spatial database. A trivial way to overcome such shortcoming would be to store all previous states of the R-trees. As we shall see shortly, this is not an acceptable, nor practical, solution.

The problem of indexing non-spatial objects (i.e., regular tuples in a relation) over time has been researched by many researchers in the temporal databases community. A thorough survey can be found in [12]. However, to our knowledge no research has been published regarding the temporalization of the R-tree. That is exactly the kernel of our contribution.

We assume the temporal attribute is transaction time. Transaction time interval is the time an object has been stored in the database [6]. Hence an MBR is considered stored in the spatial database since the time it is input *ad infinitum*, or until the point it is updated or deleted. This special feature prevents one of using the quite simple idea of

*Also an invited lecturer at Institute of Computing of the State University of Campinas (mario@dcc.unicamp.br).

†Supported by CAPES. Alternative email: jeffsilva@writeme.com

considering time as another spatial dimension, thus using the $K+1$ dimensional space to index MBRs varying over time. The problem is that the current MBR versions would have one of its "sides" being extended continously (notice that the current point in time is always moving forward). Even if one could somehow manage this variable side, it would imply a large overlap ratio among the R-trees' subspaces (which does affect negatively the R-tree's performance).

Hence, this paper addresses the problem of querying and maintaing current and past states of R-trees. To accomplish that the paper is organized as follows. The next Section presents an overview of the rationale behind our approach, which we call Historical R-trees. Section 3 discusses how the R-tree's algorithms need be changed to allow the realization of the Historical R-trees. We conclude in Section 4, presenting some issues which are being currently investigated and/or have potential for future research.

2 ENHANCING R-TREES WITH TIME

The technique we propose is inspired by an idea first presented by Burton and colleagues [3, 2]. The authors proposed the use of overlapping trees to manage the evolution of text files. Later, the idea was generalized by [10] to manage temporally evolving B^+-trees in general. The basic idea behind those techniques was to keep current and past states of the B^+-trees by maintaing the original tree and replicating, from state to state, only the root and the branches which reflect any changes. The unchanged branches were not replicated, but rather were pointed to by the nodes in the new branch. The approach we propose is an extension of the overlapping approach originally proposed by Manolopoulos e Kapetanakis [10] for the B^+-trees, to the R-trees. We call such an approach Historical R-tree (H-R-tree for short). In this paper we concentrate on managing it, rather than benchmarking it.

Let us illustrate the rationale supporting the H-R-tree with the following example. Consider the initial R-tree in Figure 1(a) at time T0. Suppose that at time T1 MBR 3 suffers a modification resulting in MBR 3a. Likewise MBR 8 is modified at time T2 yielding MBR 8a. The three states (at T0, T1 and T2) of that particular R-tree are thus those shown in Figures 1(a), (b) and (c) respectively. Note that in that particular example the subtree rooted at node B did not change at all, nevertheless it was replicated in all three states. Moreover, the subtree rooted at C (A1) did not change from T0 (T1) to T1 (T2), but the whole subtree was replicated as well. It should be now clear that duplicating the whole tree at each state is rather unpractical.

Let us now see how the same scenario would be handled by the H-R-tree. We assume an array, called A, indexing the time points where updates occurred. The initial R-tree must be kept full and it is pointed to by A[T0]. From T0 to T1 MBR 3 changes, and as such node A changes as well, after all its contents did change. This update propagates upwards until the root node. At the end only the path {R1, A, 3} needs to be updated, resulting the new path {R2, A1, 3a}. Naturally, the R-tree at T1 is composed by the subtrees rooted at A1, B and C. However, those rooted at B and C did not change at all, and thus need not be replicated. Similarly, at time T2 the R-tree rooted at R3 is composed by those subtrees roted at A1, B and C1. Again, from T1 to T2, the subtrees rooted at A1 and B did not go under any modification and as such need not be replicated. The resulting H-R-tree at time T2 is shown in Figure 2(a). Simple inspection shows that, even in a trivial example like this, the H-R-tree is much smaller than the set of tree R-trees in Figure 1. Figure 2(b) shows the logical view of the resulting R-tree at time T2, which is exactly the same one in Figure 1(c).

It is important to stress that querying *any* version of the R-tree under the H-R-structure is a matter of obtaining the correct R-tree root. Once this is done, using the array A[.], the logical view is that of a standard R-tree, no complications are therefore added by the approach we use to keep the R-tree's history.

3 HISTORICAL R-TREES

In this section we discuss in detail how to modify the R-tree's algorithms in order to realize the H-R-tree. Given the space constraints we omit certain details, which can be found in Guttman's original paper [5].

We draw particular attention to insertion and (logical) deletions of MBRs, modification of MBRs can be accomplished by deleting the old version and inserting the new one. As we argued above, querying any version of the R-tree is straightforward.

The H-R-tree is a structure composed by an array `A` of time values, which in turn point to several logical R-trees (see Figure 2). The H-R-tree structure is very similar to the other R-tree derivatives. The difference, besides the existence of array `A` is that each node contains a timestamp `t`, representing the time that one node was created. Any operation is always performed onto the most current R-tree version, i.e., the one pointed to by `A[t]`, and will yield a new version, which is thus timestamped with `t = now`[1]. From now on, we use the notation presented in Table 1.

Table 1: Notation used in the algorithms.

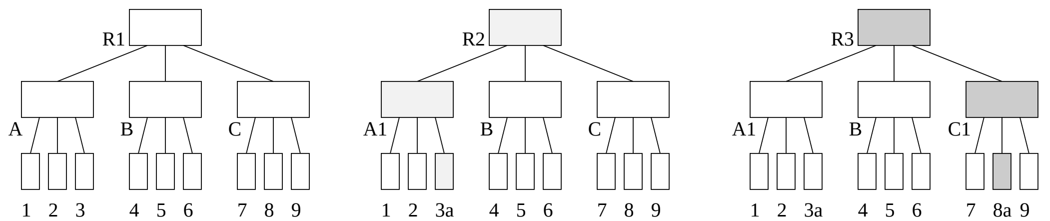
`OR`	a pointer to the H-R-tree
`A`	the H-R-tree's array of time points
`R`	root node of a R-tree
`On`	MBR inserted in the H-R-tree
`Oo`	MBR removed from the H-R-tree
`F`	an entry in a R-tree node
`I`	MBR associated to `F`
`p`	pointer associated to `F`
`N.Nt`	node N's timestamp
`Q`	queue of R-tree nodes
`L,LP,N,NN,NR,P,PP`	pointers to R-tree nodes
`now`	current point in time
`pt`	most recent entry in `A`

3.1 INSERTING AN MBR INTO THE CURRENT R-TREE

The `Insert` algorithm inserts a new MBR `On` into the most recent logical version of the R-tree within the H-R-tree `OR`, thus creating a new R-tree version (whose nodes will be timestamped with `now`).

A new branch of the H-R-tree is created, in which `On` is placed. A new logical R-tree rooted at `NR` is created, and it is pointed to by `A[now]`. The algorithm first invokes `CreateBranch` which descends the current R-tree rooted at `R`

[1]In the temporal database literature (e.g. [15]), `now` is usually regarded as a variable. We, on the other hand, use it only as a shorter name for "the value of the current point in time".



(a) R-tree at T0 (b) R-tree at T1 (MBR 3 changed to 3a) (c) R-tree at T2 (MBR 8 changed to 8a)

Figure 1: An R-tree evolving through time.

(which is pointed to by `A[pt]`) to find the leaf node in which `On` will be placed. `CreateBranch` returns a root `NR` and a leaf node `L`, where `On` is to be inserted. `A[now]` is then updated to `NR`, obtaining the most recent logical version of the R-tree. (`CreateBranch` was adapted from the Guttman's ChooseLeaf algorithm.)

```
Algorithm Insert(On, OR)
1. { create a new state in the H-R-tree }
   if A[pt] < now
      then create a new entry in A indexing now;
2. { create a root NR to insert On }
   invoke CreateBranch to create a new logical
     R-tree rooted at NR. The new logical R-tree
     contains a leaf node L in which to place On;
3. { insert On in L }
   if L has room for another entry
      then insert On in L;
      else remove L created by CreateBranch;
           invoke SplitNode to obtain a new L
             and LP containing On and all the
             other entries of L removed;
4. { propagate split upwards }
   if a split was performed
      then Invoke AdjustTree on L, also passing LP;
5. { grow tree taller }
   if node split propagation caused a root split
      then create a new root NR whose children are
             the resulting nodes resulting from the
             root split;
   adjust the entry in A to point to the new root NR;

Algorithm CreateBranch(On, OR)
1. { initialize }
   set N to be the root pointed to by A[pt] in OR;
   if N.Nt < now
      then create a new node L;
           copy all entries of N into L;
           set L.Nt = now;
           set NR = L;
      else set NR = N;
           set L = N;
2. { leaf check }
   if N is a leaf
      then return NR and L;
3. { choose subtree }
   let F be the entry in N whose rectangle F.I
     needs least enlargement to include On. Break
     ties by choosing the entry with the rectangle
     of smalest area;
4. { create a new node of the new branch and
     descending the tree }
   set LP = L;
   set N to be the node pointed to by F.p;
   if N.Nt < now
      then create a new node L;
           copy all entries of N into L;
           set L.Nt = now;
           adjust the pointer F.p in LP to point
             to L;
      else set L = N;
5. { Loop until a leaf is reached }
   repeat from step 2;
```

The `SplitNode` and `AdjustTree` procedures mentioned above are essentially the algorithms defined by Guttman [5]. `SplitNode` is used when a new object `On` is inserted into a full node. In this case, all the entries should be divided between two nodes. A small change is needed in the original `SplitNode` Algorithm though, namely the algorithm must set the timestamp of the newly created nodes. The `AdjustTree` ascends from a leaf node to the root, adjusting the covering rectangles and propagating splits as necessary. It is interesting to note that the logical view of the H-R-tree is that of a standard R-tree at some point in time, such as `now`, and as such, node splits are handled as if we were manipulating a standard R-tree.

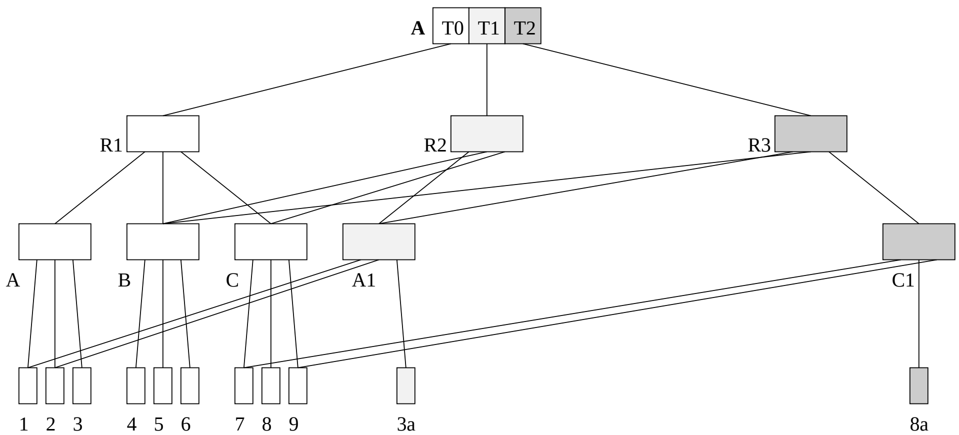
Figure 3 shows an example where from T0 to T1 a new MBR, labelled 7 was inserted. The algorithm decided that it should be inserted in node B, which was already full. Therefore a node split happened, resulting in nodes B1 and B2. As the root R1 was not full, no further splitting is needed. From T1 to T2, no split is needed when MBR 8 is input into node A, thus only a single branch (optimal case) is replicated.

Notice that in the worst case a node split is propagated all the way from leaf to root, resulting in two (i.e., a constant and low number) branches being replicated. Therefore, considering an R-tree uses $O(n/B)$ space[2], the H-R-tree after u updates uses $O(n/B + u \log_B n)$ space. If all previous R-tree states were kept $O(un/B)$ space would be required.

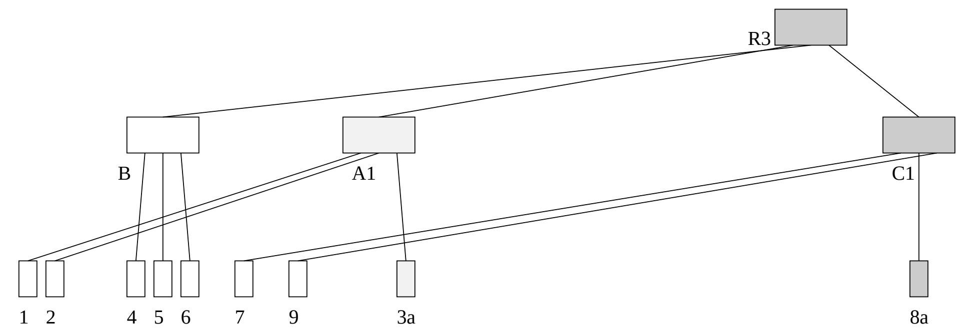
3.2 DELETING (LOGICALLY) AN MBR FROM THE CURRENT R-TREE

The `Delete` algorithm removes an MBR `Oo` from an H-R-tree `OR` at time `A[pt]`. A new entry in `A` is created, indexing `now`. A new logical R-tree is created, which reflects the new state of the R-tree without the MBR `Oo`. After `Oo` is removed, `Delete` invokes a slightly modified version of Guttman's CondenseTree algorithm to eliminate the node if it has too few entries and to relocate its entries. These are re-inserted in the R-tree rooted at `A[now]`. The CondenseTree algorithm

[2] n is the number of indexed MBRs.



(a) Physical view, semantically equivalent to Figure 1, but using the Historical R-tree.



(b) Logical view of the Historical R-tree state at time T3

Figure 2: Physical and logical views of the Historical R-tree.

also propagates node elimination upward as necessary, adjusting covering rectangles. The only two modifications necessary in Guttman's CondenseTree are: (1) in the insertion algorithm used to re-insert the entries removed, the `Insert` algorithm described above must be used instead of the original one, and (2) a node containing entries to be inserted in set Q is only removed if its timestamp is `now`. A node will be removed only if it was duplicated by the `Delete` algorithm, otherwise, it belongs to a previous state of the R-tree, and as such it should not be removed.

```
Algorithm Delete(Oo, OR)
1. { find the leaf node containing Oo}
   set R to be the root pointed to by A[pt];
   set Q, the queue of nodes already traversed,
     to be empty;
   invoke FindLeaf passing R to locate the leaf
     node L containing Oo;
   if L cannot be found
      then stop;
2. { create a new state in the H-R-tree }
   create a new entry in A with time value equal
     to now;
3. { create a new branch without Oo }
   create a new node LP;
   set LP.Nt = now;
   remove the first element of Q and put all of
     its entries in LP;
   set R = LP;
   set A[now] to point to LP;
   while Q is not empty
      create a new node L;
      set L.Nt = now;
      remove the next element N of Q and put all
        of its entries in L;
      adjust the entry in LP pointing to N to point
        to L;
      set LP = L;
   Remove Oo from L;
4. { adjust tree }
   Invoke CondenseTree, passing L;
5. { shorten tree }
   if the root node has only one child after the
   tree has been adjusted
      then make the child the new root R;
   adjust A[now] to point to R;
```

```
Algorithm FindLeaf(R)
1. { search subtrees storing the branch that
     contains Oo }
   if R is not a leaf
   then for each entry F in R
           if F.I overlaps Oo
              then if R is not in Q yet
                      then Put R in Q;
              let R be the root of the subtree
                pointed to by F.p;
              invoke FindLeaf passing R;
        until Oo is found or all entries are checked;
        if Oo was not found and R was inserted in Q
           then remove last R inserted into Q;
   else check each entry to see if it matches Oo;
```

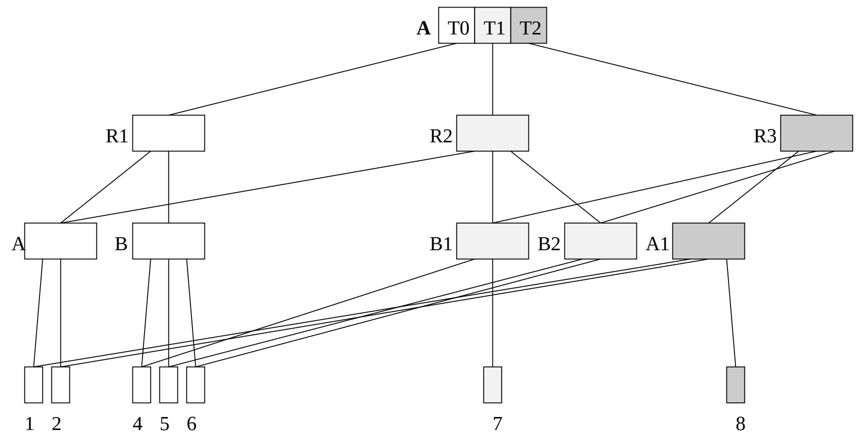


Figure 3: Example of how insertion can affect the H-R-tree.

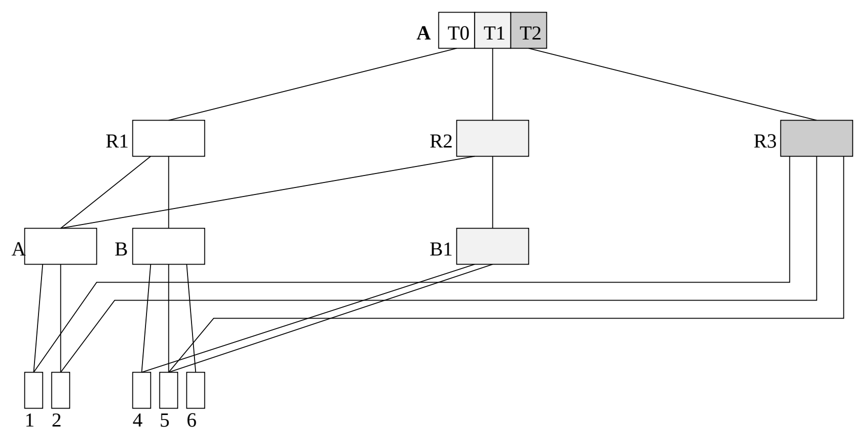


Figure 4: Example of how deletion can affect the H-R-tree.

```
       if Oo is found
          then put R in Q;
       return R;

Algorithm CondenseTree(R)
1. { initialize }
   set N = L;
   set Q, the set of eliminated nodes, to be empty;
2. { find parent entry }
   if N is the root
      then go to 6;
      else let P be the parent of N;
           let F be N´s entry in P;
3. { eliminate under-full node }
   if N has fewer than m entries
      then delete F from P;
           copy the entries of leaf nodes of subtree
             rooted at N to set Q;
           if N.Nt = now
              then remove N;
4. { adjust covering rectangle }
   if N has not been eliminated
      then adjust F.I to tightly contain all entries
        in N;
5. { move up one level in tree }
   set N = P and repeat from 2;
6. { re-insert orphaned entries }
   re-insert all entries of nodes in set Q;
```

Notice that the `CondenseTree` algorithm may re-insert $m-1$ MBRs in the current version of the R-tree. Recall that m is the minimum number of entries occupied in any R-tree node. Therefore, considering the algorithm `Insertion` presented earlied, in the worst case, `Delete` yields $2\times(m-1)$ branches being replicated. It is worthwhile stressing that this is a constant number.

As an illustration, Figure 4 shows an H-R-tree where from time T0 to T1, MBR 6 was deleted. As B1 still remains with its minimum load (assuming $m = 2$), no further modifcation is needed but replicating the branch that lead to MBR 6. From T1 to T2, MBR 4 is deleted and a more interesting situation arises. After deleting MBR 4 B1 falls under its minimum load, and thus all its entries must be re-inserted, freeing that node. Freeing that node leave the root of the current R-tree with only one child, which cannot occur, again per R-tree's definition. A new root (R3) is then created containing the remaining MBRs 1, 2 and 5.

3.3 MODIFYING A SINGLE MBR IN THE CURRENT R-TREE

Modifying an MBR in the current R-tree is rather trivial. All one needs to do is to remove the current version of the

MBR `Oo` and insert the new version `On` using the algorithms `Delete` and `Insert` already discussed.

```
Algorithm Modify(Oo, On, OR)
1. { Delete the current MBR version }
   Invoke Delete passing Oo and OR;
2. { Insert the new MBR version }
   Invoke Insert passing On and OR;
```

We have already noted that another `Modify` algorithm could be devised. The main idea is to forcefully enlarge or shrink the MBRs, higher in the tree, that involve the MBR being modified. This would ensure that only one branch would be replicated per MBR modification. The cost of such alternative is yet to be investigated.

3.4 QUERYING THE CURRENT R-TREE

The `Search` algorithm finds all MBRs in the H-R-tree `OR` that overlap the search windows `S`. The search can be made in some specific past state of the R-tree or in the current state, that is the R-tree pointed to by `A[pt]` The `Search` algorithm first finds the (R-tree) root in the H-R-tree that is pointed to by A[t]. After that, Guttman's original Search algorithm is used to find the MBRs that overlap the search windows `S`.

```
Search(S, t, OR)
1. { find the appropriate root R }
   if t = now
      then set R to be the node pointed to by A[pt];
      else set R to be the node pointed to by A[t];
2. { find the MBRs which overlap S }
   invoke Guttman's original Search algorithm
     passing R;
```

For a discussion on how to query the R-tree we refer the reader to Guttman's original paper [5], as no modifications (at all) to the original algorithm are required.

4 DIRECTIONS FOR FUTURE RESEARCH

We have presented a systematic way to deal with modifications of an R-tree while preserving all of its previous states. While the resulting structure, which we named Historical R-tree (H-R-tree) saves substantial space when compared to saving all previous R-tree states, it does not degenerate query processing time. Future research is needed in several directions, such as:

- Implementation and benchmarking of the H-R-tree, including the generation of spatial-temporal data;
- Using Z-order [11] along with the original Historical B^+-trees [10] to index spatial-temporal data (and consequently compare the result to the H-R-tree);
- Adapting the idea of Persistent B^+-trees [8] to R-trees (and also compare the result to the H-R-tree);
- Investigate how the proposed approach depends on the initial R-tree configuration (recall one could use any of the R-trees, but the R^+-tree, algorithms to generate the initial R-tree) and
- Determine how concurrency control algorithms are impacted in the proposed approach.

REFERENCES

[1] Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conferencee on Management of Data* (June 1990), pp. 322–331.

[2] Burton, F., et al. Implementation of overlapping B-trees for time and space efficient representation of collection of similar files. *The Computer Journal 33*, 3 (1990), 279–280.

[3] Burton, F., Huntbach, M., and Kollias, J. Multiple generation text files using overlapping tree structures. *The Computer Journal 28*, 4 (1985), 414–416.

[4] Gaede, V., and Günther, O. Multidimensional access methods. To appear in ACM Computing Surveys. `http://www.wiwi.hu-berlin.de/~gaede/survey.rev.ps.Z`, 1997.

[5] Guttman, A. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conferencee on Management of Data* (Jun 1984), pp. 47–57.

[6] Jensen, C., Clifford, J., Gadia, S., Segev, A., and Snodgrass, R. A consensus glossary of temporal database concepts. *ACM SIGMOD Record 23*, 1 (Jan 1994), 52–64.

[7] Kamel, I., and Faloutsos, C. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th Very Large Databases Conference* (1994), pp. 500–509.

[8] Lanka, S., and Mays, E. Fully persistent B^+-trees. In *Proceedings of the 1991 ACM SIGMOD International Conferencee on Management of Data* (May 1991), pp. 426–435.

[9] Leutenegger, S., Lopez, M., and Edgington, J. STR: An efficient and simple algorithm for R-tree packing. In *Proceedings of the 13th IEEE International Conference on Data Engineering* (April 1997), pp. 497–506.

[10] Manolopoulos, Y., and Kapetanakis, G. Overlapping B^+-trees for temporal data. In *Proceedings of the 5th Jerusalem Conference on Information Technology* (August 1990), pp. 491–498.

[11] Orestein, J. Spatial query processing in an object-oriented database system. In *Proceedings of the 1986 ACM SIGMOD International Conferencee on Management of Data* (May 1986), pp. 326–336.

[12] Salzberg, B., and Tsotras, V. A comparison of access methods for time evolving data. Tech. Rep. NU-CCS-94-21, College of Computer Science, Northeastern University, Boston, USA, 1994. (To appear in ACM Computing Surveys).

[13] Samet, H. *The Design and Analysis of Spatial Data Strucutures.* Addison-Wesley, Reading, MA, 1990.

[14] Sellis, T., Roussopoulos, N., and Faloutsos, C. The R^+-tree: A dynamic index for multidimensional objects. In *Proceedings of the 13th Very Large Databases Conference* (September 1987), pp. 507–518.

[15] Tansel, A., et al., Eds. *Temporal Databases: Theory, Design and Implementation.* Benjamin/Cummings, Redwood City, USA, 1993.