The Temporal R-Tree

Geraldo Zimbrão
Jano Moreira de Souza
Victor Teixeira de Almeida

ES-492/99
Programa de Engenharia de Sistemas e Computação

The Temporal R-Tree

Geraldo Zimbrão
Jano Moreira de Souza
Victor Teixeira de Almeida

Computer Science Department
Graduate School of Engineering
Federal University of Rio de Janeiro
PO Box 68511, ZIP code: 21945-970
Rio de Janeiro - Brazil,
zimbrao@cos.ufrj.br
jano@cos.ufrj.br
valmeida@cos.ufrj.br

Abstract

It's a well-known fact that the new GIS applications need to keep track of temporal information. However, the most known spatial index structure, the R-Tree and its variants, does not preserve the MBRs evolution. A first but inefficient approach is to add one dimension to data space in order to store time. In this work, we propose an alternative approach: a new index structure called Temporal R-Tree that deals with spatiotemporal data. The Temporal R-Tree, or TR-Tree, allows the retrieving of present and past states of data. There is little data duplication, which is guaranteed by the block copying mechanism. The retrieving time is comparable to the original R-Tree.

1. Introduction

This paper describes a method to make the R-Tree family of data structures partially persistent. According to [BROD94], a partially persistent data structure is a data structure in which old versions are remembered and can always be searched. However only the latest version of the data structure can be modified. An interesting application of a partially persistent R-Tree is indexing of a Temporal Geographic Information System.

A typical Geographic Information System, GIS for short, models geographic objects as individual points, lines or polygons. According to [LANG92], in the development field of GIS there is a compelling need to describe spatial change over time; that is, to design a temporal GIS. A Temporal Geographic Information System is a GIS with additional functionality, capable to model and handle temporal information. A Temporal GIS models geographic objects as individual points, lines or polygons that have a specific lifetime. In such systems, each update to data creates a new version. The system never deletes an object: it just records its death time. Thus, the amount of data is ever growing. The need for indexing TGIS is obvious: the bigger the database the more explicit the performance gains from using

sophisticated indexing techniques. However, this demands new techniques for indexing both spatial and temporal data, a field where few research have been done.

R-Trees [GUTT84] and its variants are the most popular structures for indexing spatial data due its simplicity and efficiency. So, R-Trees are suitable candidates to serve as basis for a new spatiotemporal indexing structure. In this paper, we describe the Temporal R-Tree, TR-Tree for short, which keeps old versions of the data. The TR-Tree is an enhanced R-Tree able to index both spatial and temporal information. The algorithms presented here are inspired by an idea presented in [OHLE94] and [BECK96], and thus the TR-Tree is our spatial version of the Multiversion B^+-Tree.

We assume that the TR-Tree will store *transaction time* for its Minimum Bounding Rectangle (MBR). In the context of temporal databases, the transaction time of a database fact is the time when the fact is current in the database and may be retrieved [JENS94]. Transaction times are consistent with the serialization order of the transactions. Transaction-time values cannot be later than the current transaction time, usually known as *now*. In addition, as it is impossible to change the past, transaction times cannot be changed. Thus, partially persistent data structures seem to be a good choice for indexing transaction time databases.

Each entry in the TR-Tree will have a MBR and a pair of timestamps *birth* and *death*. The lifespan of an MBR will be the right-open interval *birth* and *death*, represented by [*birth*, *death*). The special timestamp "*" is used to indicate *now*, that is, the current time. When a new MBR is added in the database at time *t* its lifetime interval is set to [*t*, *). The MBR remains current (or live) until the time at which it is deleted or updated. A real world deletion at time t_2 is implemented in the database as a *logical* deletion, by changing the *death* attribute of the entry from "*" to t_2. Updating the MBR at time t_3 is implemented by the logical deletion of the corresponding entry and the insertion of a new entry with the new MBR. A MBR is said to be current (or live) at time *i* if $birth \leq i < death$. Moreover, for every time *i*, $i < now$.

The TR-Tree efficiently stores a collection of R-Trees, each one covering a non-overlapping sequence of intervals representing the entire lifespan of the dataset. By construction, querying the current version of a TR-Tree will have the same asymptotic time costs as querying the plain corresponding R-Tree. Any change in the current R-Tree will lead to a new R-Tree, and both trees will be stored at the TR-Tree. So, at any time *i*, querying the TR-Tree can be compared to querying the plain R-Tree as it was at time *i*. The only overhead in such a query will be the time spent to locate the corresponding R-Tree root in the TR-Tree. Good space utilization will be assured by using a mechanism called *version split* that may be compared to the node copying operation of [DRIS89]. In this work, we focus on managing the TR-Tree, rather than benchmarking it.

This paper is organized as follows. Section two presents a brief overview on related works. Section three describes the TR-Tree structure and its basic ideas. Section four presents the algorithms for searching, inserting and deleting and a detailed explanation of each one. Section five presents our conclusions and future directions for this work.

2. Related Work

A variety of temporal access methods have been proposed in recent years. A comprehensive survey can be found in [SALZ94]. Most of the proposed approaches assume that changes occur in increasing time order; therefore they are appropriate for maintaining the evolution of transaction time. Some of these temporal access methods are, with more or less effort, adaptable to the spatial domain. However, to the best of our knowledge, until now there is only one published paper facing the problem of making a R-Tree able to handle spatiotemporal information [NASC98]. We will discuss that work in the following paragraphs.

A straightforward approach would consider time as another spatial dimension, and then work with 3D bounding boxes rather than 2D rectangles. However, as pointed out in [NASC98], the live MBR would have one of its sides being extended continuously since *now* is always moving forward. In other words, current MBRs would have an open end, while the R-Tree is not designed to handle this. Even if someone fixes this problem, it would imply a large overlap ratio among the nodes of the R-Tree, what degrades its overall performance.

[NASC98] proposed a technique to enhance the R-Tree in such a way that old states were preserved. Their work was inspired by an idea from [BURT90] and generalized by [MANO90] to manage temporally evolving B+-Trees in general. The basic idea behind those techniques was to keep current and past states of the trees by retaining the original tree and only replicating the nodes that were modified. The unchanged nodes were not replicated, but rather were pointed by the new nodes. They named this new structure Historical R-Tree. The great advantage of this structure is its simplicity. The main drawbacks are the inadequate space utilization and the time spent on copying the modified nodes at each update. Also, to answer a query that specify a time range many nodes will be traversed twice, and many nodes will differ only in one entry. Therefore, this structure is not a good choice for indexing spatial data that has a medium to large changing ratio.

In [LANG92] some approaches are proposed that treat time as a spatial dimension and then use a multidimensional access method: R-Tree, grid file, quadtree and BSP trees. Some of these structures were built by using clipping techniques, and in the spatiotemporal data it leads to a explosive number of clipped objects: in her own words, “the magnitude of increased data volume was unexpected”. Other approaches, although did not perform any clipping, did not achieve good results in answering spatial or temporal range

queries. The most promising approach was the R-Tree, although it presents the drawback of many overlapping nodes for current data. As she pointed out, "the results of the R-Tree experiment are more promising than anticipated".

Another work that deals with the indexing of spatiotemporal data, in two-dimensional space, is [CAST98]. The focus of that work is to handle continuously changing data, such as the position of moving objects. Among the proposed strategies, they represent the objects in the R-Tree by their X, Y, and time coordinates. Another proposed strategy is to represent the objects in the R-Tree by using 5-dimensions that store the movement of an object from one point to another. Finally, the third proposed strategy is to apply a transformation technique to represent the object as vertical line segments in multidimensional space. None of the proposed strategies modify the R-Tree structure or its algorithms. Moreover, these strategies are not appropriate to handle areas, that is the aim of our work.

The Multiversion B+-Tree presented in [OHLE94] is a good approach to handle non-spatial data. The main idea of that work is to keep old data and perform what is called a *version split* whenever a node is filled up. Then, only the live data is copied to new nodes avoiding to replicate old data. Our work resembles that one in the context of spatiotemporal data and R-Trees, and we have tried to agree with their naming conventions.

3. TR-Tree Index Structure

In this section we describe the proposed TR-Tree structure in terms of modifications in the original R-Tree, as it was presented in [GUTT84]. The TR-Tree index structure is very similar to the R-Tree with some modifications to handle temporal information. The leaf nodes will hold index record entries of the form < *MBR*, *tuple-identifier*, *birth-time*, *death-time* > and non-leaf nodes contain entries of the form < *MBR*, *child-pointer*, *birth-time*, *death-time* >. The tuple identifier is a surrogate to the polygon representation, and will be omitted in our examples.

Another important modification in the TR-Tree is that it may have more than one root node pointing to the R-Trees, i.e. it will have more than one tree inside of the structure of the TR-Tree. However, at a given time, there will be one and only one root associated to it. As a consequence, the TR-Tree should not be seen as a tree, but as a directed acyclic graph. Also, there is an array or a hash structure indexing the root nodes of the TR-Tree and its related lifespan. We will call this the root index structure.

Time and space efficiency of the R-Tree is based on the six properties proposed by Guttman [GUTT84], especially the assumptions (1) and (3). Those properties ensure that the height of the R-Tree with N index records is at most $|log_m N| - 1$. To maintain this efficiency, the properties of the R-Tree must be generalized to handle the existence of different version entries in a node. Let M be the maximum number of

entries that will fit in one node and let $m = M / k$ be a parameter specifying the minimum number of live entries in a node, where k is a constant, $k \geq 2$. Those six properties should be modified as follows:

(1) Every leaf node contains at least m **live** index records and at most M index records unless it is the root;

(2) For each index record < *MBR*, *tuple-identifier*, *birth-time*, *death-time* > in a leaf node, *MBR* is the smallest rectangle that spatially contains the n-dimensional data object (for example, a polygon) represented by the indicated tuple;

(3) Every non-leaf node contains at least m **live** entries and at most M entries unless it is the root;

(4) For each entry < *MBR*, *child-pointer*, *birth-time*, *death-time* > in a non-leaf node, *MBR* is the smallest rectangle that spatially contains the rectangles in the child node;

(5) The root node has at least two **live** children unless it is a leaf;

(6) All leaves **of the same root node** appear on the same level.

We call the set of (1), (3) and (5) properties the *weak version condition*. Now we can state how structural changes are triggered in the TR-Tree (*index records* of leaf nodes should be handled in the same way that *entries* in a non-leaf node):

- A *node overflow* occurs as the result of an insertion of an entry into an already full node, i.e. a node that contains M entries.
- A *weak version underflow* occurs when the number of current version entries in a node becomes less than *m* (or two for root nodes).
- A node underflow never occurs, since entries are never deleted from nodes but only marked as dead.

A structural change to handle a node overflow or to restore the weak version condition is performed based on the block copy operation, i.e. the node is marked as dead and the current version entries are copied into a new node. We call this operation *version split*.

Considering a node overflow, most of the cases the live node created by the version split may be an almost full block or even a full block. To avoid this case and the similar phenomenon of an almost empty block, we state the following two new properties (7) and (8) that must be satisfied after a structural change, and call this set of properties the *strong version condition*.

(7) The number of live entries in a non-leaf node after a structural change must be in the range from $(1 + \varepsilon) \times m$ to $(k - \varepsilon) \times m$, where ε is a tuning constant, $\varepsilon > 0$, to be defined more precisely in the following.

(8) The number of live index records in a leaf node after a structural change must be in the range from $(1 + \varepsilon) \times m$ to $(k - \varepsilon) \times m$.

In this way, we guarantee that after a structural change on a block at least $\varepsilon \times m + 1$ insertions or deletions of entries can be performed on this node before the next structural change becomes necessary. The choice of ε can be done in the same way as in [OHLE94], but with no merge restrictions. So, $k \geq 1/\alpha + (1 + 1/\alpha) \times \varepsilon - 1/m$, where α is the constant of minimum node utilization for the original R-Tree.

With these modifications on the R-Tree, the TR-Tree is an index structure that can handle time information and still maintain the good time and space efficiency of the original R-Tree. We assume that the related root node of time *t* can be found at *O*(1) access. The search then proceeds as in the original R-Tree. Because, according to the weak version condition, each node of the TR-Tree stores at least *m* entries for version *t*, and therefore it behaves like an R-Tree with *m* as the minimum number of entries.

4. Searching and Updating

4.1 Searching

The search in the TR-Tree receives as arguments a rectangle *S* and a search time *t*. Initially, the tree associated with that time *t* must be found in the root index structure. Once found the corresponding root node R, the search goes similarly to the search in the R-Tree, where it only walks through branches of the tree where time *t* is between the *birth-time* and *death-time* of the nodes.

```
Algorithm Search
Description: Given an TR-Tree, find all index records whose rectangles overlap a search
             rectangle S and their lifespan includes the search time t.

Begin
    Find the root node R of the tree corresponding to time t in the root index structure
    Invoke RootSearch at root node R, the search rectangle S and the search time t
End

Algorithm RootSearch
Description: Given a root node R, find all index records whose rectangles overlap a search
             rectangle S and their lifespan includes the search time t.
Begin
    if R is a leaf node then
        Check all entries E to determine whether
            E.birth-time ≤ t ≤ E.death-time and E.MBR overlaps S. If so, E is a qualifying
            record.
    else
        Check all entries E to determine whether
            E.birth-time ≤ t ≤ E.death-time and E.MBR overlaps S. For all overlapping
            entries, invoke RootSearch recursivelly on the tree whose root node is pointed
            by E.child-pointer
End
```

4.2 Insertion

The insertion in the TR-Tree receives as arguments an index record *E* and a time *t* that must be the same or higher of the current time of the TR-Tree. Initially a search is executed to find the leaf node where the entry will be inserted. This search is made only in the current root node of the TR-Tree. If the entry is already inserted in the TR-Tree, the insertion must be aborted, else the entry is inserted into the given non-leaf node, it triggers all the structural changes necessary if a violation of any of the weak and strong version condition occurs.

```
Algorithm Insert
Description: Insert a new index record E in a TR-Tree at time t.

Begin
    Let R be the current root node of the TR-Tree
    Invoke ChooseNode to select a leaf node L in which to place E
    if L is the current root node of the TR-Tree R then
        invoke InsertRootNode to insert the entry E into the root node R
    else
        invoke InsertNode to insert the entry E into the leaf node L
End

Algorithm ChooseNode
Description: Search for a node L at height h in a tree with root node R in which to place
             a new entry E. This algorithm is similar as Guttman's ChooseLeaf. If the
             height is not given, the algorithm searches for leaf nodes.

Begin
    Let N be the root node R
    while the height of the node N is not equal to h or N is a non-leaf node do
        Let F be the live entry in N whose rectangle F.MBR needs least enlargement to
            include E.MBR. Resolve ties by choosing the entry with the rectangle of
            smallest area.
        Let N to be the child node pointed to by F.child-pointer
    return N
End

Algorithm InsertNode
Description: Insert the entry E in a node A of the TR-Tree.

Begin
    if there is space for one more entry in the node A then
        Enter E into A and invoke AdjustTree on A to adjust the upwards bounding boxes
    else
        Copy current version entries of node A into a new node B (version split of A)
        Let P be the live parent entry of A in the TR-Tree. Kill the entry P.
        Enter E into B. { This may momentarily lead to a block overflow in B, such an
                          overflow is eliminated immediately }
        if strong version underflow condition occurs at node B then
            Let P be the live parent node of A and let E be the entry referring to B to be
                inserted in the parent node of B
            invoke InsertNode to insert the entry E into node P
            invoke CondenseTree passing node B
        else if strong version overflow condition occurs at node B then
            Distribute entries of B evenly among B and BB (key split of node B). Here
                we can use SplitNode of Guttman's R-Tree with one of the three algorithms,
                the exhaustive algorithm, the quadratic-cost algorithm and the linear-cost
                algorithm.
            let P be the live parent node of A
            let E and EE be the entries referring to B and BB to be inserted in the
                parent node of B and BB
            if P is the current root block of the TR-Tree then
                invoke InsertRootNode to insert the entries E and EE into the node P
            else
                invoke InsertNode to insert the entries E and EE into the node P
        else
            invoke AdjustTree on B to adjust the upwards bounding boxes.
End
```

```
Algorithm InsertRootNode
Description: Insert the entry E in a root node R of the TR-Tree.

Begin
    if there is space for one more entry in the root node R then
        Enter E into R
    else
        Kill the entry R into the root index structure
        Create a new root node R with the current version entries of old R root node
        Enter E into R. { This may momentarily lead to a block overflow in R, such an
                          overflow is eliminated immediately }
        if the root has only one child then
            Make this child the new root node R
        else if strong version overflow condition occurs at node R then
            Distribute entries of R evenly among RR and RRR. Here we can use SplitNode of
                Guttman's R-Tree with one of the three algorithms, the exhaustive
                algorithm, the quadratic-cost algorithm and the linear-cost algorithm.
            Clean the root block R
            Let E and EE be the entries referring to R and RR to be inserted in the new
                root node R
            invoke InsertRootNode to insert the entries E and EE into the root node R
        Insert the entry R into the root index structure
End

Algorithm AdjustTree
Description: Ascend from a node L to the root, adjusting covering rectangles.

Begin
    if L is not a root node then
        Let P be the live parent node of L and E the entry of P pointing to L
        Kill the entry E
        Invoke InsertNode to insert the entry E with E.MBR covering all rectangles of L.
            { the algorithm InsertNode will handle this recursively until the root }
End.

Algorithm CondenseTree
Description: Given a node L, kill this node and relocate its entries. Propagate node
             elimination upward as necessary.
Begin
    Let Q an array of set of eliminated nodes indexed by its height on the tree be empty
    Let h be the height of L in the tree
    Insert into Q[h] all of the current version entries in L
    Let P be the live parent node of L and let E be the entry referring to L in P
    Kill the entry E in P
    while P is not the root node of the tree and a weak version underflow occurs in P do
        Let h be the height of P in the tree
        insert into Q[h] all of the current version entries in P
        Let L be the node P
        Let P be the live parent node of L and let E be the entry referring to L in P
        Kill the entry E in P
    Reinsert the set of entries in Q using the algorithm Insert, but inserting the entries
        at height determined by its index on Q
End
```

4.3 Deletion

The deletion in the TR-Tree receives as arguments an index entry *E* to be deleted. Initially, a search is made to find if the entry is contained in the TR-Tree. Like the insertion algorithm, this search is only made through the current root node of the TR-Tree. Once a leaf node is found containing the entry to be deleted, this entry is marked as dead. A weak version underflow should occur and if so, all the other live entries at this leaf node are reinserted in the TR-Tree using the algorithm Insert.

```
Algorithm Delete
Description: Remove index entry E from the TR-Tree.

Begin
    Let R be the current root node of the TR-Tree
    Invoke FindLeaf to locate the leaf node L containing E
    Kill the entry E in L and invoke AdjustTree passing L
    if weak version underflow occurs in L then
        reinsert all current version entries in L with L.birth-time = now using algorithm
           Insert
End

Algorithm FindLeaf
Description: Given a root node R, Find the leaf node containing the index entry E.

Begin
    if R is not a leaf then
        Check each live entry F in R to determine if F.MBR contains E.MBR. For each such
            entry invoke FindLeaf on the tree whose root is pointed by F.child-pointer
            until E is found or all entries have been checked
    else
        Check each live entry to see if it matches E. if E is found return R
End
```

4.4 Example

We will illustrate the ideas of the algorithms by using an example. In this example we will insert all the rectangles of the ***figure 1*** in alphabetical order and then delete some of them. The parameters of the TR-Tree for this example are set up as follows: M=6, m=2 (k=3) and ε=0.5. Hence, the weak version condition assures that each node in the TR-Tree contains at least two and at most six current version entries unless it is a root node. The strong version condition, by its time, assures that after a structural change, new nodes contain at least three and at most five current version entries.

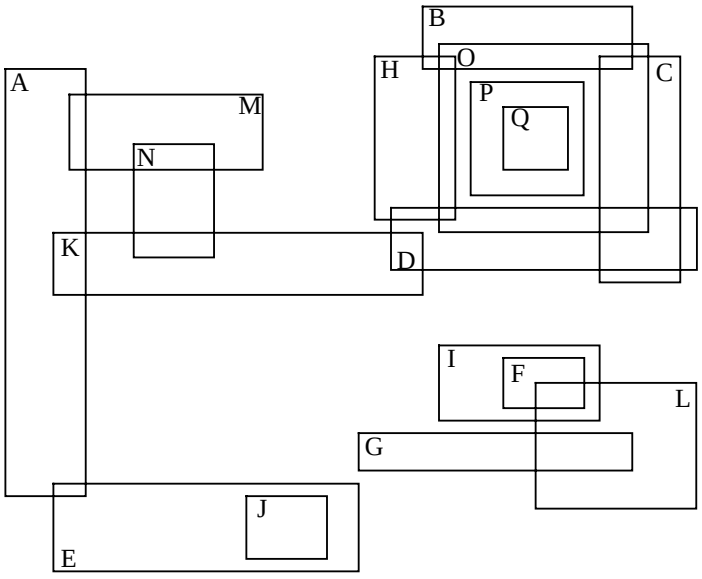

Figure 1 – Rectangles to be inserted

Initially, the TR-Tree is empty. The insertion of the rectangle A creates the first leaf root node R1 which is inserted into the root's array. The next insertions of the rectangles B, C, D, E and F fill the root node R1. This situation is shown in the ***figure 2*** below.

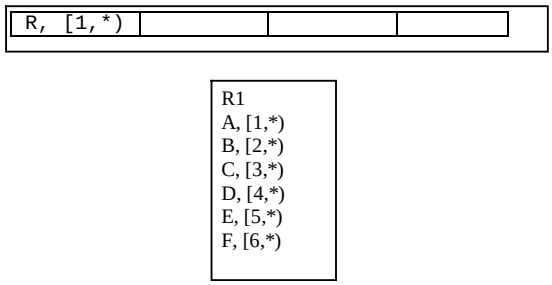

Figure 2 – Situation at time 6

The insertion of the rectangle G triggers a node overflow of the root node R1 in version 7. A version split is performed with this node R1 with all of its current version entries copied into a new leaf root node R2 and this node is marked as dead at time 7. A strong version overflow occurs and the node R2 is divided into two new leaf nodes N1 and N2 making R2 a non-leaf root node having two children N1 and N2. Two rectangles BB1 and BB2 covering all the rectangles in N1 and N2 are created in R2. Here, we illustrate how the current version tree grows. The situation after this structural change is shown in the ***figures 3*** and ***4*** below.

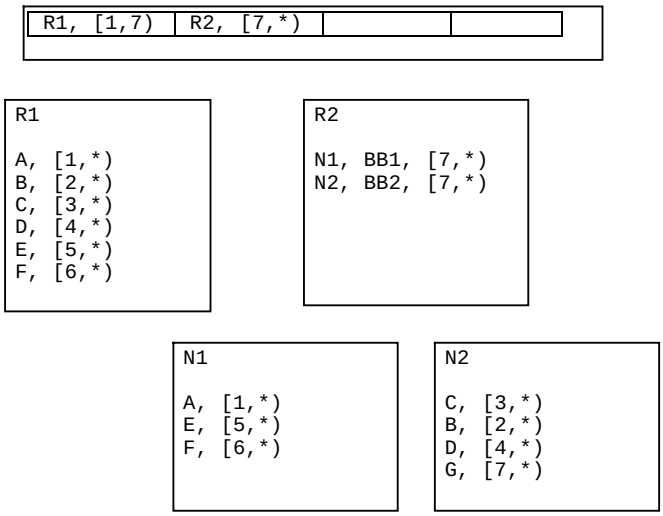

Figure 3 – situation at time 7

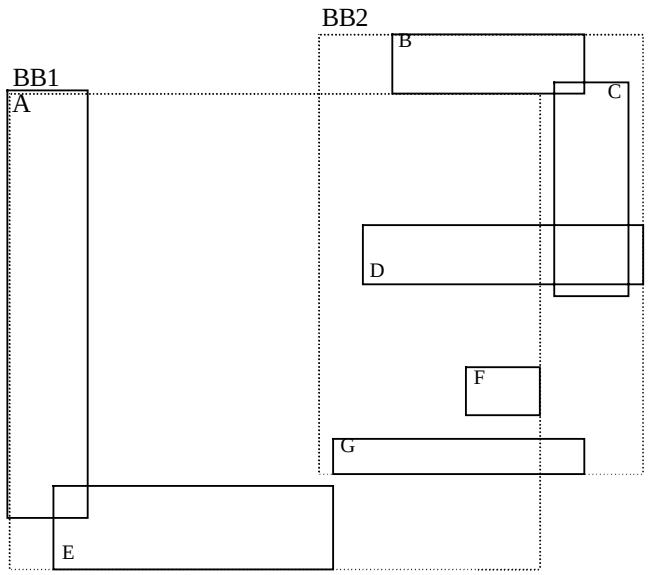

Figure 4 – MBRs at time 7

The insertions of the rectangles H, I, J and K does not alter the structure of the TR-Tree, but the insertion of L in a full node N2 leads to a node split followed by a version split creating the nodes N3 and N4, the node N2 is marked as dead at time 12 at its live parent node R2. The situation at this point is in the ***figure 5*** below:

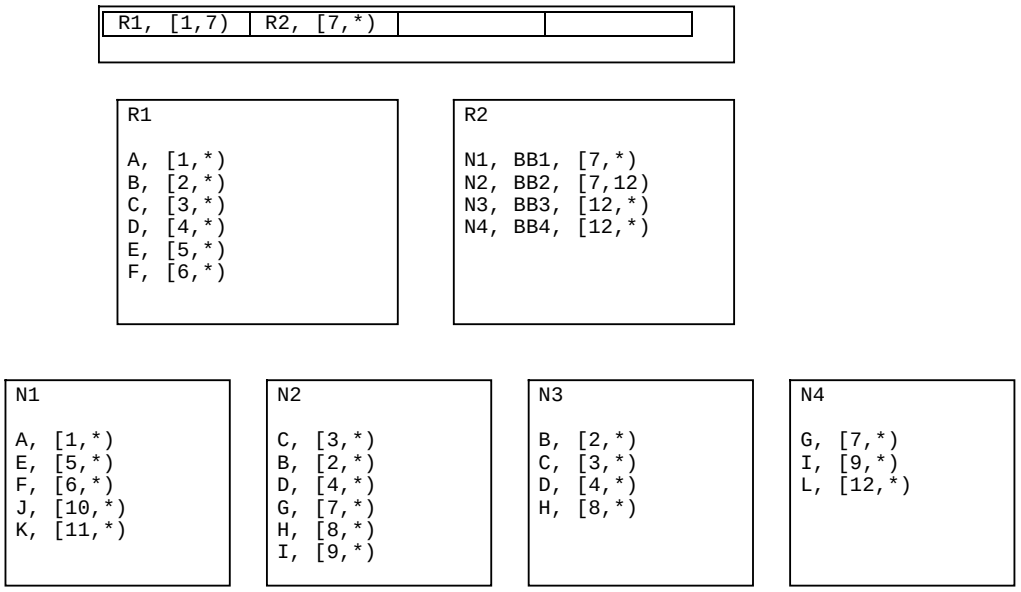

Figure 5 – situation at time 12

The rectangle M is inserted at node N1 which fills its node. After that, the insertion of the rectangle N makes a structural change just like the rectangle L, a node split is followed by a version split and the new leaf nodes N5 and N6 are created. N1 is marked as dead at time 14 into R2. The rectangles O and P are inserted into the node N3 and when we try to insert the rectangle Q, this node is full. A node split is performed followed by a version split and the nodes N6 and N7 are created, but there is no space for them at the root node R2. This insertion triggers a node split in the root node R2 and the number of current version entries does not conflict with the strong version condition. So, a new root node R3 is created and inserted into the root's array and the old root node R2 are marked as dead at time 17. At this time, all the insertions were made and the structure of the TR-Tree is shown in the ***figure 6***:

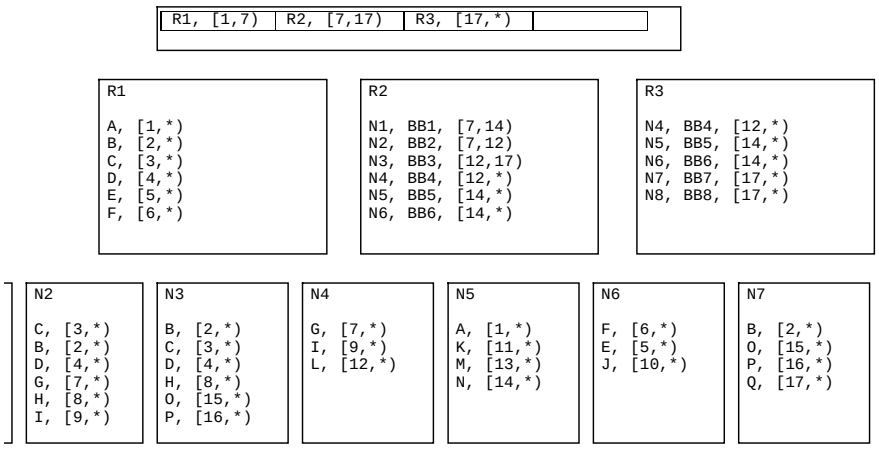

Figure 6 – situation at time 17

The deletion of the rectangle L in the node N4 just mark it as dead at time 18. The bounding box entry in the live parent of N4, the root node R3, must be adjusted. This entry is marked as dead at time 18 too and a new entry is created in the node R3 pointing to the node N4 with a bounding box covering all the rectangles in N4. The root node R3 is now full.

When we delete the rectangle I in the node N4, a weak version underflow occurs. To treat this underflow this node is marked as dead in its live parent node R3 and the current version entries (only the rectangle G) are reinserted in the TR-Tree. The reinsertion of the rectangle G in the node N6 needs an enlargement of its bounding box in its live parent node, the root node R3. Once the root node R3 is full, its modification triggers a node split with the creation of the new current version root node R4. The root's vector is then modified to mirror the new TR-Tree index structure.

At the end of the example the TR-tree will look like ***figure 7***.

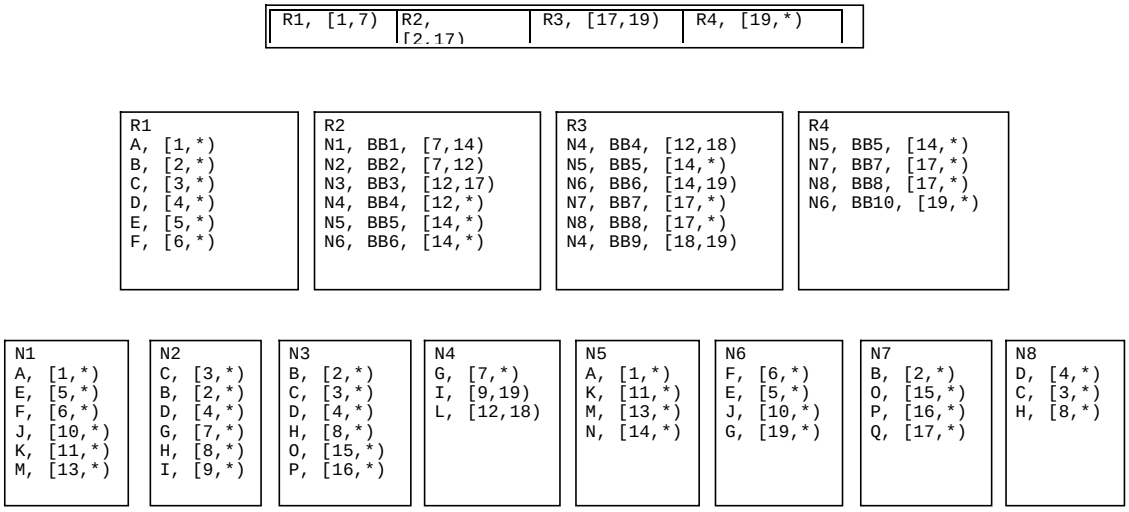

Figure 7 – situation at time 19

5. Conclusions

In this paper, we proposed a new index structure called Temporal R-Tree that deals with spatiotemporal data. The TR-Tree allows the retrieving of present and past states of data. There is little data duplication, which is guaranteed by the split version and block copying mechanisms. The retrieval time remains comparable to the original R-Tree.

Future research is needed in several directions, such as: implementation and benchmarking the TR-Tree, compare the results to the Historical R-Tree and other approaches, and investigating the suitability of the TR-Tree to perform spatiotemporal joins.

BIBLIOGRAPHY

[BECK96] Bruno Becker et al., An Asymptotically Optimal Multiversion B-Tree. VLDB Journal 5(4): 264-275 (1996)

[BROD94] Brodal, G.S.: “Partially Persistent Data Structures of Bounded Degree with Constant Update Time”. In Nordic Journal of Computing, volume 3(3), pages 238-255, 1996.

[BURT90] 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.

[CAST98] Scott Casteel, Romit Mukherjee, & Dong Xie, “Querying Spatio-Temporal Database Using R-Tree Index”, Technical Report CS411, University of Illinois at Urbana-Champaign 1998.

[DRIS89] Driscoll, J.R., Sarnak, N., Sleator, D.D., and Tarjan, R.E. “Making data structures persistent”. Journal of Comp. and System Science. 38:86-124, 1989.

[GUTT84] A. Guttman: “R-Trees: A Dynamic Index Structure for Spatial Searching”. In Proceedings of the ACM SIGMOD Intl. Conf on Management of Data, Boston, MA, 1984.

[JENS94] 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.

[LANG92] Langran, G.: “Time in Geographical Information Systems”. Taylor & Francis, London, 1992.

[MANO90] 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.

[NASC98] Nascimento, M.A, and Silva, J.R., Towards Historical R-trees, proceedings of ACM SAC'98, p. 235-240. Atlanta, USA, Feb/98.

[OHLE94] Thomas Bernhard George Ohler, “On the integration of Non-Geometric Aspects into Access Structures for GIS”, PhD dissertation submitted to the Swiss Federal Institute of Technology, 1994.

[SALZ94] 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).