Processing of Spatiotemporal Joins Using Indexes

GERALDO ZIMBRÃO
JANO MOREIRA DE SOUZA
Computer Science Department, Graduate School of Engineering
Federal University of Rio de Janeiro
PO Box 68511, ZIP code: 21945-970 Rio de Janeiro - Brazil,
VICTOR TEIXEIRA DE ALMEIDA
Praktische Informatik IV, Fernuniversität Hagen, D-58084 Hagen, Germany

Email: {zimbrao,jano}@cos.ufrj.br, valmeida@fernuni-hagen.de

Abstract: *It's a well-known fact that the new GIS applications need to keep track of temporal information. Among other operations, a spatiotemporal DBMS should efficiently answer the spatiotemporal join. The best-known spatial index structure, the R-Tree (and its variants), does not preserve the MBRs' evolution. New indexing structures were proposed in the literature allowing the retrieval of present and past states of data, and most of them are R-Tree based. This paper presents an evaluation of spatiotemporal join algorithms using these new structures, particularly a partially persistent R-Tree called Temporal R-Tree and the 2+3D R-Tree. Starting from spatial join algorithms, we present algorithms for processing spatiotemporal joins over time instants and intervals on both spatiotemporal data structures. Finally, we implement and test these new algorithms with a couple of generated spatiotemporal data sets. Our experiments show that our algorithms´ performance is good even in extreme cases, showing its good scalability – especially for the TR-Tree. In addition, with minor adaptations, the main ideas of our algorithms can be used for evaluating joins using partially persistent structures.*

Keywords: Spatiotemporal Access Methods, Spatiotemporal Join

1 Introduction

One of the main tasks to be supported by a Spatiotemporal Database Management System (STDBMS) is the efficient indexing and retrieval of spatiotemporal objects. According to [16], there are two main types of spatiotemporal queries: selections and joins. This paper presents a first study on the spatiotemporal join processing using partially persistent data structures based on R-Trees like the Temporal R-Tree (TR-Tree) [19] and the Partially Persistent R-Tree (PPR-Tree) [6]. A refined algorithm to perform spatiotemporal joins specifically for partially persistent structures is presented and a performance evaluation is done using a number of join queries on generated datasets. Moreover, the main ideas presented in this paper may be used to design join algorithms for partially persistent data structures either for spatiotemporal or just temporal data. The algorithms for the spatiotemporal joins were implemented and tested in an experimental environment and then compared to some other spatiotemporal access methods.

As stated in [16], a STDBMS must (i) offer appropriate data types and a query language to support (static or moving) spatial data, (ii) provide efficient indexing and retrieval methods, and (iii) exploit cost models for specialized spatiotemporal operations for query processing and optimization purposes. Relevant applications that a STDBMS must support include Temporal Geographical Information Systems (TGIS), Mobile Applications, Multimedia Systems, Statistical and Scientific Databases.

A fundamental and very costly operation in databases is the join operation. A spatiotemporal join is an operation that is used to combine spatiotemporal objects of two sets according to some spatiotemporal property. The spatiotemporal intersection join is of great interest, which intends to find all pairs of objects that have a common area (overlap) during their lifespan. Also, the query can be bounded by a specific region and/or by a specific time interval. As stated in [16], the spatiotemporal intersection join is a crucial operation in spatiotemporal databases.

This paper is organized as follows. Section 2 defines the problem and the terms used in this paper. Section 3 presents a brief overview of related works. Section 4 discusses the spatiotemporal join in more detail and presents our algorithms. Section 5 contains the data set

and the join queries descriptions as well as the experimental results. Finally, section six concludes the paper and gives an outlook to future work.

2 Defining the Problem

A Spatiotemporal Access Method (STAM) is an index that organizes spatiotemporal data by its spatial key (the minimum bounding rectangle (MBR)), and its temporal key (the lifespan interval). The prime goal of the STAM is to efficiently handle query processing. The broader the set of queries supported, the more applicable and useful the access method becomes. A set of fundamental query types is ([16]):

- *Selection queries*: are queries of the form "find all objects that have lied within a specific area (or at a specific point), during a specific time interval (or at a specific time instant)".
- *Join queries: are queries of the form "find all pairs of objects that have lied spatially close (i.e., within distance d), during a specific time interval (or at a specific time instant)".*

In the context of spatial databases, selection queries are also called window queries, and join queries are also called spatial joins [3]. We will adopt an analogous naming convention. Moreover, for performance reasons, it is a good idea to define a more powerful join operator: the primitive join operation can combine a selection window. Thus, our join query definition will be: "find all pairs of objects that have lied spatially close (i.e., within distance d) and within a specific area (or at a specific point), during a specific time interval (or at a specific time instant)". We call this a window spatiotemporal join. These kinds of queries are expected to be one of the most common addressed by STDBMS users. Table 1 shows the queries that a STAM should efficiently address. The time interval queries are termed time range queries.

	Window query	**Window spatial join**
Time Instant	***Time Instant Window Query***: find all objects that intersect a given window at a given time.	***Time Instant Spatiotemporal Join:*** find all pairs of intersecting objects at a given time**.**
Time Range	***Time Range Window Query***: find all objects which intersect a given window during a given time interval.	***Time Range Spatiotemporal Join***: find all pairs of intersecting objects during a given time interval.

Table 1 - Queries that a STAM should address.

3 Related Work

In this section, we will summarize some work done in related areas. We will show the main ideas and principles of current work on spatiotemporal indexing.

3.1 Spatiotemporal Access Methods

According to [16], at the time of that publication, only four spatiotemporal indexing methods had appeared in the literature: 3D R-trees [17], MR-trees and RT-trees [18], and HR-trees [11]. These approaches have the following characteristics:

- 3D R-trees treat time as another dimension using a "state-of-the-art" spatial indexing method, namely the R-tree [4], [2].
- RT-trees couple time intervals with spatial ranges in each node of the tree structure by adopting ideas from R-trees and TSB-trees [9].
- MR-trees [XLH90] and HR-trees [11] use overlapping techniques in R-trees (Hilbert [5] for the latter one) to represent successive states of the database [10].

To this list, we should add three other access methods: RST-Trees [12], PPR-Trees [6] and Temporal R-Trees [20], which will be discussed in details in section 3.

The RST-Tree proposed by [12], tries to solve the current time now in the context of bitemporal spatiotemporal databases. The problem is solved letting the MBRs grow with time,

taking a stair-like aspect and an area close to a triangle, reducing considerably the search area, consequently the nodes visited in a query. As stated there, before this work, the data structures were only able to handle data related to the current time now for transaction time using partially persistent techniques, which was made possible also for valid time.
The TR-Tree is a specialization of MVB-Tree to handle spatial data, i.e., it makes some modifications on the structure and the algorithms of the MVB-Tree to use a spatial data structure, the R-Tree [4], and then extends the R-Tree algorithms using the R*-Tree [2] techniques. A complete description of the TR-Tree algorithms as well as a performance evaluation may be found in [19].

The PPR-Tree, proposed in [6], is very similar to the TR-Tree. It is an effort to extend the techniques presented in [1] for the MVB-Tree to spatial data using R-Trees. This data structure is similar to the Bitemporal R-Tree [7, 8]. The main differences between them is that the Bitemporal R-Tree is used for scalar data and have the valid time dimension, but this valid time dimension may be changed by another data dimension and then the structures would be equal. Their work does not perform entry reinsert on node overflow; five merge heuristics are proposed to handle this situation.

Also, there are a number of new proposals for indexing moving points or objects. We will not mention them here since they are outside of the scope of this work. Here, we assume that geometry cannot change continuously, but only in discrete steps, and hence we are not talking about moving objects. According to [16], this case is of great interest for spatiotemporal databases.

3.2 The Temporal R-Tree

In this section we describe the TR-Tree structure in terms of modifications in the original R-Tree, as it was first presented in [4]. The complete algorithms' description and performance evaluation of the TR-Tree can be found in [20].

The TR-Tree index structure is very similar to the R-Tree with some modifications to handle temporal information and structural changes of the tree. Each entry in the TR-Tree will have a MBR, an identifier and a pair of timestamps of birth and death. The lifespan of an MBR will be the right-open interval of birth and death, represented by [birth, death). The special timestamp "*" is used to indicate now, that is, the world current time. The TR-Tree's last time is the time when the last update has occurred. The leaf nodes will hold entries in the form <MBR, tuple-identifier, birth-time, death-time>. Non-leaf nodes contain entries in 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. Also, each node will have two more attributes, birth-time and death-time, indicating its lifespan.

Whenever a new MBR of a polygon is added to the database at time ti its lifetime interval is set to $[t_i,*)$. A MBR remains live until it is deleted or updated. A real world deletion at time t_i is implemented in the database as a logical deletion, by changing the death-time attribute of the MBR entry from "*" to t_i. Updating a MBR at time t_i 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 live at time t_i if birth-time $\leq t_i <$ death-time, that is, $t_i \in$ [birth-time, death-time).

Likewise, when the TR-Tree creates a new node at time ti its lifetime interval is set to $[t_i,*)$. A corresponding entry is set to $[t_i,*)$ in the parent of this node. When the MBR of a node is updated at time t_k its corresponding entry death-time is set to t_k, and a new entry is inserted with the new MBR. Thus, many different entries may exist for a node, but at each time t only one entry is live. If a node should be removed from the tree, due to structural changes like node overflow or underflow, it is logically deleted by setting the death-time attribute of both the node and its corresponding entry. A node is said to be alive at time t_i if

birth ≤ ti < death, that is, $t_i \in$ [birth, death). A non-live node is a killed node, and can't be modified anymore – it is a read-only node.

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 structure indexing the root nodes of the TR-Tree and its related lifespan. We will call this the root array structure.

Time and space efficiency of the R-Tree is based on the properties proposed by Guttman, specially the assumption R1. This property ensures that the height of the R-Tree with N tuple identifiers 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≥2. Those properties should be modified as follows (changes are bold faced):

R1. Every node contains at least m live entries and at most M entries, unless it is the root;

R2. For each entry in a node, MBR is the smallest rectangle that spatially contains the rectangles in the children nodes (for non-leaf nodes) or the data object (for leaf nodes);

R3. The root node has at least two live children nodes, unless it is a leaf;

R4. All leaves of the same root node appear on the same level.

We call the property R1 the weak version condition. Note that a live entry means that it is alive during node lifespan. Now we can state how structural changes are triggered in the TR-Tree:

T1. A *node overflow* occurs as the result of an insertion of an entry into an already full node.

T2. A weak version underflow occurs when the number of live entries in a node becomes less than m (two for root nodes).

T3. 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, in most cases the live node created by the version-split will 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 new property R5 that must be satisfied after a structural change, and call this set of properties the strong version condition.

R5. The number of live entries in a 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, ε≥0, to be defined more precisely in the following.

After a version-split, if the node violates the strong version condition, then the R-Tree-like structural change takes place: overflow leads to a R-Tree standard node split and underflow leads to a real node deletion and all its entries are reinserted. This is the only case that a physical deletion can occur. One should note that this kind of node deletion is the result of a single insert or update operation, so the nodes deleted are intermediate states of the TR-Tree that don't need to be stored.

Since m=M/k, we assure that after a structural change on a node 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 [1], but with no merge restrictions of a B-Tree. So, $k \geq 1/\alpha+(1+1/\alpha)\times\varepsilon-1/m$, where α is the rate of minimum node utilization for the original R-Tree (at most 0.5). In our tests, we have stated α to be 0.5,

k to be 3 and ε to be 0.3. Thus, for a node capacity of 90 entries, the TR-Tree parameters would be as in Table 2.

Weak Version Conditions	Max. entries/node	90
	Min. live entries/node	30
Strong Version Conditions	Max. entries/node	81
	Min. live entries/node	39

Table 2 - Typical TR-Tree parameters

The algorithms of the TR-Tree were designed to allow a block of operations to be done before a new version is created. In this way someone should explicitly do the creation of new versions of the tree. It is an improvement appropriate for a number of cases like massive updates, intermediate steps produced by the application of referential integrity constraints, and other operations that generate invalid or useless data that the system does not need to store, like entries' re-insertion of deleted nodes. Finally, the main ideas used here to make the R-Tree a partially persistent data structure were the same used in the Bitemporal R-Tree [7, 8] that originated from [1].

3.3 Comparisons Between STAMs

The main differences we can state between the TR-Tree and the PPR-Tree reside in the treatment of node underflow and in the implementation of the R*-Tree algorithms. The authors of the PPR-Tree, the same ones of the Bitemporal R-Tree, did not propose a solution for entry re-insertion in nodes with less than m entries. It is known that the idea of entry re-insertion generates better-organized structures than those coming from node merges due to the lack of a complete ordering of the space in more than one dimension. The choice of the neighbor node to be joined to the node with less than m entries is extremely difficult and not always optimal. In [7, 8] five attempts are proposed to finding the better neighbor on which to insert the entries of the node in underflow. We expect, for these reasons, that the structure of the TR-Tree present itself far more organized than that of the PPR-Tree, and with this, that the search and join algorithms achieve better performance.

Finally, we may divide the spatiotemporal structures into 2 classes according to a characteristic extremely relevant to the execution of the joins (and of the queries also) and that is: if the structure considered duplicates or not the entries. From the structures mentioned until now we can state that the 2+3D R-Tree and the RT-Tree do not duplicate the entries, where, on the other hand, the TR-Tree, the HR-Tree and the MR-Tree duplicate them.

In the HR-Tree and in the MR-Tree, entry duplication occurs every time a node must be updated: the entire node is copied and updated, and a new entry is created in its father, repeating the process up to the root node. With this, at every insertion or removal, the path from the root to the leaf where an update occurred is recorded. Thus from this fact, we note the low adequacy of these trees in the treatment of data sets where the update rate varies from medium to large.

On the other hand, the TR-Tree duplicates nodes/entries only when there's a structural update, in other words, an overflow or an underflow. This method results in much less data duplication or redundancy.

4 The Spatiotemporal Join

In this section we shall present the algorithms for the execution of spatiotemporal joins using as an index structure the 2+3D R-tree and the TR-Tree. We shall assume the preexistence of indexes on both sets of data to be used in the join. We chose these two structures for the following reasons:

- In [19], it is evident that the TR-Tree is the most promising structure to be used as a generic spatiotemporal index, capable of carrying out well both queries over a time

instant, as over a time interval, for on average in a set of queries, it was the one with the best performance.

- Only the HR-Tree and the 2+3D R-Tree were able to surpass the TR-Tree's performance in the time instant and time interval queries, respectively. The other structures evaluated did not present satisfactory performance.
- The HR-Tree is not appropriate to index data with a medium to large update rate where it creates much duplication, resulting in very large indices. In fact, its performance in queries involving intervals degrades rapidly as the time interval increases.
- Finally, the PPR-Tree, besides being structurally very similar to the TR-Tree, is based on the R-Tree, where the TR-Tree is actually based on the R*-Tree, which in its turn has proven better performance over the simple R-Tree.

4.1 The Performance of the Spatial Join over the MBRs through R-Trees

Being the R-Tree a recursive structure, the algorithm that uses it to perform a spatial join is also recursive. Thus, given 2 trees represented by the rectangles associated to the nodes of the highest level (the root node), we shall investigate the children of each one of these nodes, in pairs as in a Cartesian product. We shall perform joins in the sub-trees represented by the pairs of child nodes whose rectangles intersect. In each sub-tree join, we shall repeat the process recursively until we reach the leaves, where we shall then obtain the pairs of MBRs that form the response set for the MBR join. Figure 1 shows this process. Note that the information on the dimension of the rectangle is not relevant anywhere in the process: the same algorithm will work for any number of dimensions, just as the R-Tree will. In [3] there's a detailed study on the performing of spatial joins using R*-Trees. In this work, we performed the optimizations pointed out in [3] up to the SJ3 algorithm, which is the join using a plane-sweep algorithm to compare a node with another node. The following algorithms obtain small gains, because they deal basically with optimizations that affect the disk's cache performance, where it only involves the exchange in the order of comparison of the nodes. We decided not to implement these improvements because the LRU cache used presented satisfactory results, in a way that these optimizations would result in gains with little significance in the two trees.

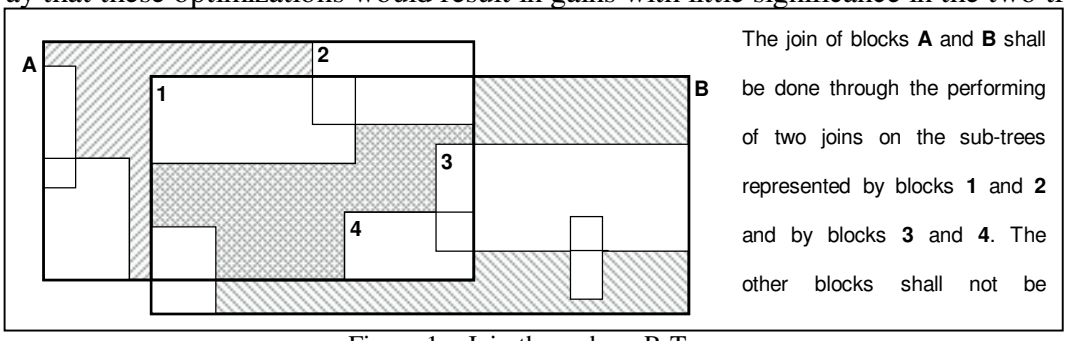

Figure 1 – Join through an R-Tree

One of the advantages of this algorithm is that each pair of MBRs shall be listed only once in the set of candidates, for each MBR can only be inserted in one node. The great disadvantage of the algorithm results from the overlaying of the rectangles corresponding to the nodes of the R-Tree. In order to investigate the MBRs that are in a certain area where there is an overlay of two or more nodes, we shall have to explore more than one sub-tree. When we attempt to perform the join, the problem will become worse, i.e., with quadratic complexity: if m rectangles of a tree intersect a point i and if n rectangles of another tree also intersect this same point i, then it will be necessary to perform m x n joins in the sub-trees.

4.2 The Spatiotemporal Join in the 2+3D R-Tree

In its essence, the structure of each one of the two trees that compose the 2+DR-Tree is identical to the original R-Tree: only one of them has two dimensions (the one that stores the live objects in current time); the other one has three dimensions. To perform a join over two

2+3D R-Trees is then similar to performing four joins in R-Trees: one join over a pair of trees with two dimensions, another over a pair of trees of 3 dimensions, and finally two joins over trees with two and three dimensions. Only the last case holds a particularity, which makes it different from the original algorithm: the performance of a join over trees with different dimensions. Nevertheless, it is important to remind the reader that the number of dimensions is different only in what concerns the structural algorithms of the tree, or, the insertion and the splitting of the nodes. Implicitly we have stored in the two-dimensional tree the information of the third dimension. This dimension corresponds exactly to the time of validity: the entries in this tree have a field with their time of creation and it is known that these are all alive. Therefore, the missing dimension shall always be represented by [a,*), where a is the time of creation of the entry (in the case of the nodes, time of creation of the oldest entry contained in the node). In this way, we are leveled with the case of performing joins in three-dimensional MBR trees. It is also important to observe that the structure of the 2+3D R-Tree doesn't duplicate entries, in a way that, although we have to perform four complete joins, it is not necessary to eliminate the pairs of duplicated responses.

4.3 The Spatiotemporal Join on the TR-Tree

As depicted previously, the TR-Tree is a structure that duplicates entries, in other words, a single node or spatial object can have more than one entry in the tree. This shall bring us the following problem: if we perform the spatiotemporal join in a way analogous to the spatial join in R*-Trees, we shall report pairs of duplicated entries. The problem is a little more serious, for we should remember that a duplicated entry for one node will mean that this node will be visited twice, which implies that all of its entries be reported two times. Figure 2 illustrates this problem.

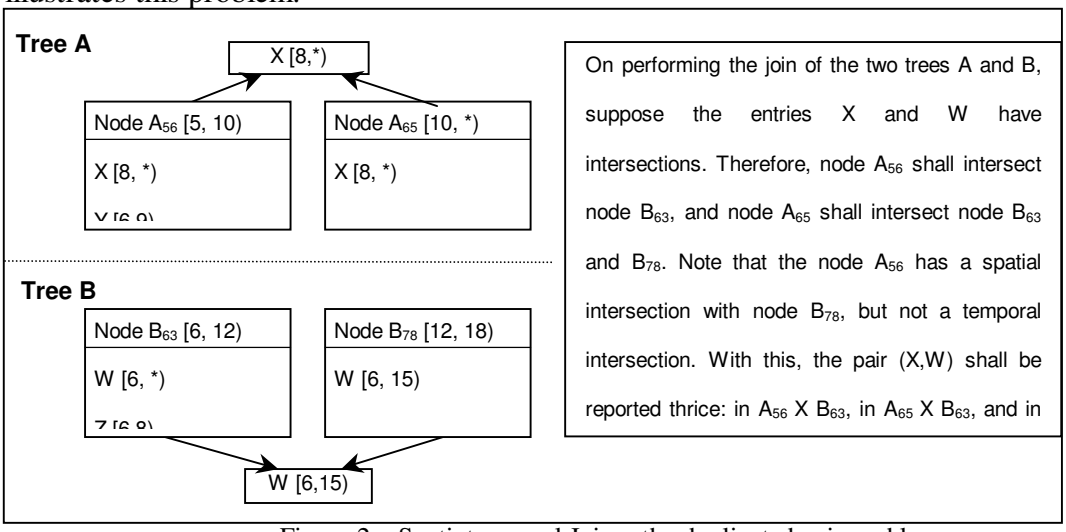

Figure 2 – Spatiotemporal Join – the duplicated pair problem

Worse even, if they are reported in another node for which there is also a duplicated entry, they will be reported two times also in this other node, and so forth in each level of the tree up until the root. Clearly this is not a good approach.

4.4 Avoiding Duplicates

Hence it is necessary to elaborate a strategy that reports each MBR pair only once. More even, for efficiency's sake, it would be useful if each node pair (intermediate MBRs) be considered only once during a join. To achieve this, we should first stress a couple of TR-Tree properties. We shall consider entries, generically, not only the entries to spatial objects, but also the entries to nodes of the TR-Tree. The demonstrations, here constrained by the limited space, are only outlined, but it is easily verifiable that all of them can be proven formally. Take the following assumption: let E be an entry with a life-span [C,M), contained in the

nodes X_1, X_2, ... X_n of the tree A, and let the intervals $[c_1,m_1)$, $[c_2,m_2)$, ... $[c_n,m_n)$ be the respective lifespan of each node. Obviously each $c_i < m_i$ for $1 \leq i \leq n$. We have:

P1. $m_i = c_{i+1}$, for $1 \leq i < n$.

Reason: Entries are duplicated only through the operation of version split, or, the structural update suffered by a node. In this case, all of the entries are inserted in the node created necessarily in the current time, and the node that suffered the structural update is marked as dead. No entry that was duplicated can be inserted in a node that was not created in current time, in some time different from the time of death of the node that contained the entry. In this way, along the lifespan of a certain data object or node, in each instant, one and only one node possesses the entry E for the referred object or node. We shall call X successor to E the node which contains the entry E after X's death. Note that the dead entries in X do not have successors, for they will not be copied when X dies.

P2. $C \in [c_1,m_1)$.

Reason: on creation, an entry should be inserted in some node that has already been created and that is alive.

P3. $M \in [c_n,m_n)$, or E is alive and X_n is, too.

Reason: if the entry E died in the time M, then it will no longer be copied when the node that contains it dies. Moreover, dead entries are not copied. Thus, if M died, it died after insertion in X_n and before the death of X_n. If M did not die, then Xn didn't either, because, in this case, E should've been copied to another node created at the time of X_n's death.

P4. Only one entry for E, the one contained in X_n, could be marked with the time of death of the spatial object or node.

Reason: comes directly from P3.

From P4 emerges a good indication for a query algorithm, pointed out in [19]. It consists in rewriting the original query "find all the MBRs which have overlaps on the R rectangle" into "find all the MBRs which have overlaps on the R rectangle and that, either they are alive at the end of the query or were killed during this interval". This small modification suggests an algorithm that consists in walking through the tree reporting the entries that intercept the query rectangle, but on the dead nodes, listing only the dead entries, for the live entries will be listed later in the successors of this node for each one of the live entries. For the nodes that are alive by the end of the query interval (are really alive or died in an instant after the end of the query interval), we should report all of its entries because we shall not investigate these node's successors for any entry, because these will have creation time outside the query interval. We can easily demonstrate that this algorithm works fine by using induction at the height of the tree. This algorithm was really implemented in [19], tested and brought good results.

Following this idea, we can establish an algorithm with a similar strategy to the join and that doesn't report duplicated entry pairs. Let us suppose that the algorithm investigates each pair of nodes that have a spatial and temporal intersection only once. Undoubtedly, as we've seen in figure 2, there will be pairs that will appear more than once. We should then establish criteria to decide if a certain pair should or not be listed. A good approach would be to list the objects on the last time of its insertion, for as we have seen, although the tree duplicates entries, only one of them, the one associated to the node that doesn't have successors, can mark its death. The interval join transformation, analogous to the one done for the search, would be: "find all the pairs of MBRs with overlays that have the last moment of its intersection during the query interval, or that they are alive in the final time of this interval". We shall note that the time instant spatiotemporal join for the time instant t should use a time interval spatiotemporal join with time interval [t,t+1), and all assumptions made for the time interval spatiotemporal join are valid also to the time instant spatiotemporal join.

Using the query above we can then establish criteria to decide if two pairs of candidate entries should or should not be reported when found. We shall call entry candidate pairs the

pair of entries that have a spatial and temporal intersection, and every time we mention if a node or entry is dead, we refer to its state at the final instant of the query interval:

1) If the nodes where the entry candidate pair are situated are live nodes at the end of the query interval, we understand that these nodes don't have successors (or their successors shall not be visited because they were created after the end of the query interval.), therefore all of the entry candidate pairs should be reported at this moment, for there won't be a new chance to report them.
2) If one of the nodes is dead and the other is alive at the final time of the query, we should report only the candidate pairs in which the entries that belong to the dead node are also dead, for the entry candidate pairs that contain the live entries of the dead node shall appear again in the successors of this node for each one of the entries, and, certainly, each one of these successors will have an intersection with the live node.
3) If both are dead, things get a little more complicated: the candidate pairs, in which we have both of the live entries, certainly should not be reported, for the same shall appear again as a candidate pair in the respective successors. Analogously, the candidate pairs that have only dead entries should also be reported because this is the last time that they will appear in the tree (they don't have successors). But, in the entry pair where there is a dead and a live entry, there is the following difficulty: let A be the node that contains the live entry, and B the node that contains the dead entry. If A died before B, the successor of A shall be compared to B, and in it we shall again find the same entry candidate pair. Note that this problem will not emerge if both nodes died at the same time, for none of the successors shall be compared with the other node. Thus, we should not list the entry pairs in which one is alive and the other is dead if the node that contained the live entry died before the node that contained the dead entry.

The algorithms that perform the time interval spatiotemporal join are listed below:

```
function pairList(eA, eB: Entry; nodeA, nodeB: Node; finalInt: Time): boolean;
begin
  if nodeA.alive(finalInt) and nodeB.alive(finalInt) then
    return true;
  else if (nodeA.alive(finalInt) and nodeB.dead(finalInt) and
           eB.dead(finalInt)) or
          (nodeA.dead(finalInt) and nodeB.alive(finalInt) and eA.dead(finalInt))
  then return true;
  else // both of the nodes are dead.
    if nodeA.timedeath = nodeB.timedeath and
       (eA.isDead(finalInt) or eB.isDead(finalInt)) then
      return true;
    else if (nodeA.timedeath < nodeB.timedeath and eA.isDead(finalInt)) or
            (nodeA.timedeath > nodeB.timedeath and eB.isDead(finalInt)) then
      return true;
    else
      return false;
end
```

Figure 3: Algorithm to decide if a pair of candidate entries should or not be listed.

Note that the function above should be called during the plane-sweep of a pair of nodes for each entry (in our algorithm, we call pairList in the internalLoop procedure). Some function call optimizations can be made, like, for example, if both the nodes are alive we should report all of the candidate pairs, but they would result in a less didactic code. Just to position the reader, we shall show the plane-sweep code of non-optimized nodes (in our tests, we used an optimized version). In [3] this algorithm is called sortedIntersectionTest. The variable "exchange" indicates if the order of the pairs should be exchanged (so that it is not necessary to write two almost identical functions).

```
proc sortedIntersectionTest( NodeA, NodeB: Node; rBusca: Rectangle; IntSearch: Interval;
var output: ListEntries )
begin
  output := EmptyList; i := 1; j := 1;
  listA := list of Entries of NodeA that have a spatial intersection with rSearch
           and a temporal with IntSearch, ordered on the lowest coordinate x and
           on the lowest coordinate y;
  listB := list of Entries of NodeB which have a spatial intersection with rSearch
           and a temporal with IntSearch, ordered on the lowest coordinate x and
           on the lowest coordinate y;
  while i ≤ listA.size and j ≤ listB.size do
    if listA[ i ].min.x < listB[ j ].min.x then begin
      internalLoop(listA[i], j, listB, IntSearch.final,
                   false, NodeA, NodeB, output);
      i := i + 1;
    end
    else begin
      internalLoop(listB[j], i, listA, IntSearch.final,
                   true, NodeB, NodeA, output);
      j := j + 1;
    end;
end
```

Figure 4: Algorithm to perform the plane-sweep.

Because of space constraints, we shall not show here the algorithm that synchronously goes through TR-Trees, because it is analogous to the one used to go through an R*-Tree presented in [3], and is the same one also used for the trees of the 2+3D R-Tree. It is important to emphasize that, at the level of the array of root nodes, we shall perform a join for each pair of root nodes that possesses a spatial and temporal intersection. There is no need for improvements at this level, for at each moment, each R-Tree of the TR-Tree owns one and

```
proc internalLoop( test: Entry; notMarked: integer; list: ListEntries;
                   finalInt: Time; exchange: boolean; node_1, node_2: Node;var output:
ListEntries )
begin
  k := notMarked;
  while k ≤ list.size and list[ k ].min.x < test.max.x do
  begin
    if test.min.y ≤ list[ k ].max.y and test.max.y ≥ list[ k ].min.y and
       test.interval ∩ list[ k ].interval ≠ ∅ and
       listPair( test, list[ k ], node_1, node_2, finalInt ) then
      if exchange = false then
        output.insert( Pair( test, list[ k ] ) )
     else
        output.insert( Pair( list[ k ], test ) );
    k := k + 1;
  end
end
```

only one root node, which guarantees that each root node pair be investigated only once.

Figure 5: Procedure to perform the join on two nodes of the tree.

A formal proof of the correctness of the proposed algorithm is necessary. Let A and B be two nodes with a pair of candidate entries E and F. It means that E is inside the node A and F is inside the node B and $E \cap F \neq \emptyset$, i.e., the E and F MBRs' intersect. We have to demonstrate that the nodes A and B are compared together once and only once by the time interval spatiotemporal join algorithm proposed in this paper.

First, we will show that the algorithm will report all the pairs of candidate entries. The algorithm traverses all the root nodes of both TR-Trees in a synchronized way, and then, for each pair of trees, a depth search is performed. It is clear then that the result is complete, but it is not clear that this result is minimal, i.e., that it does not contain duplicates.

An induction in the level of the nodes A and B in the tree will be used to show that the result is minimal. It is not necessary that the nodes A and B have the same level, but this will be assumed to simplify the demonstration. A proof without this assumption would be very similar.

The initial step of the induction occurs when level=0, i.e., we are on the pair of root nodes of the TR-Tree. In this level of the structure, there is no overlap between the lifetimes of the root nodes in the same structure. Because of this, the initial loop of the algorithm never compares two pair of nodes more than once. It can be seen in Figure 6.

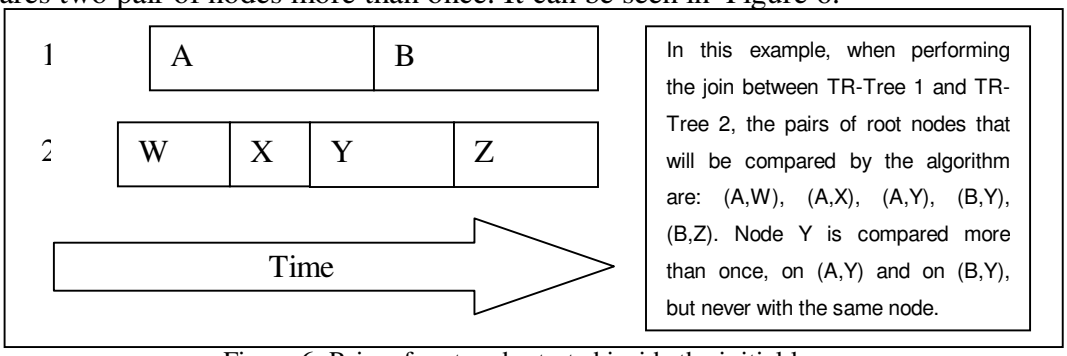

Figure 6: Pairs of root nodes tested inside the initial loop.

Suppose that the induction hypothesis is true for level n on both TR-Trees, i.e., the pairs of nodes in level n of both TR-Trees will be visited once and only once by the time interval spatiotemporal join algorithm. It must be shown that this hypothesis is also true for the nodes in the level n+1. Suppose that A and B are nodes in level n of the TR-Trees 1 and 2. Let E and F be a pair of candidate entries inside the nodes A and B, respectively.

The algorithm goes down on the structure by the three conditions cited below and summarized here to make the demonstration clear:

Condition 1: If A and B are live nodes at the end of the query time interval, then all pairs of candidate entries must be reported.

Condition 2: If one of the nodes is dead and the other is alive at the end of the query time interval, then only the pairs of candidate entries where the entries inside the dead node are also dead must be reported.

Condition 3: If both nodes A and B are dead, the pairs of live candidate entries must not be reported, the pairs of dead candidate entries must be reported, and in the case of the pairs of candidate entries where one entry is alive and the other is dead, the pair will not be reported if the node that contains the live entry died before the node that contains the dead entry.

This demonstration will be divided into three distinct cases:

1. If both of the nodes are alive at the end of the query time interval. In this case, the algorithm visits the pair of candidate entries E and F and then goes down on the structure reaching the nodes X and Y appointed by E and F respectively, or reporting E and F in case the nodes A and B are leaves. The nodes A and B do not have successors, once they are alive. The algorithm would visit the nodes X and Y only if it went down by E and F inside some antecessor of A and B. There are three possibilities:

   1.1. The pair of nodes is an antecessor of A and B. All antecessors of A and B are dead, so, only the condition 3 can be satisfied. The entries E and F are alive inside these nodes, otherwise they would not be copied to A and B. For dead nodes and live entries, condition 3 is not satisfied. In this case, the algorithm would not report the entries E and F.

   1.2. The pair of nodes contains an antecessor of node A and node B. All antecessors of node A are dead. There is a dead node (antecessor of A) and a live node (B). Only condition 2 can be satisfied, but the entry E is alive in all antecessors of node A, what invalidates condition 2.

   1.3. The pair of nodes contains an antecessor of node B and node A. This is the same case shown before in 1.2.

   In this way, if both of the nodes A and B are alive, the algorithm goes down by the entries E and F only on these nodes, because they do not have successors and for the antecessors the conditions to go down on the structure are not satisfied.

2. If both nodes A and B are dead at the end of the query time interval, three distinct cases may occur:

2.1. Both entries E and F are alive. These entries will not be reported because any of the three conditions is satisfied. One should note that these entries would be reported on the successors of A and B.

2.2. Both of the entries E and F are dead. These entries will be reported by the algorithm through condition 3. These entries will not be present on the successors of the nodes A and B because they area alive. So they may be related only on the antecessors of the nodes A and B where they are alive. Three cases may occur:

2.2.1. The pair of nodes contains both antecessors of nodes A and B. In this case, the antecessors are dead and the entries E and F are alive. None of the conditions is satisfied for this case.

2.2.2. The pair of nodes contains an antecessor of the node A (A') and node B. All the antecessors of node A are dead and entry E inside them is alive. The other node, node B, is dead and entry F is also dead. Condition 3 of the algorithm will report the entries E and F only if the node A' died before the node B. If it occurs, nodes A and B would not be compared by the algorithm, because they would not have temporal intersection, and consequently, E and F would not have temporal intersection either. But it is assumed on the graphical of the proof that E and F is a pair of candidate entries. This statement is true only in the case that E and F are both dead. Figure 7 shows graphically this problem.

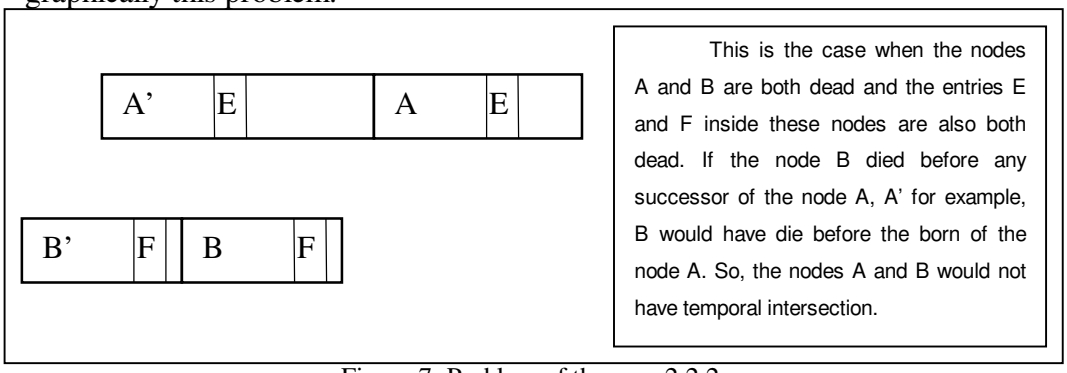

Figure 7: Problem of the case 2.2.2.

2.2.3. The pair of nodes contains an antecessor of node B and of node A. This is the same case shown below in 2.2.2.

2.3. One of the entries is dead and the other is alive. Suppose, without loss of generality, that the entry that is alive is the entry E inside node A. Node B consequently contains the dead entry F. Condition 3 is satisfied if node A died after node B. So, it is not necessary to check any successors of node A. It is not necessary to check any successors of node B because the entry F inside of node B is dead and is not present on any successor. Three cases may occur:

2.3.1. The nodes are both antecessors of A and B. In this case, both of the nodes are dead and the entries E and F are alive. It has been shown before that no condition may be satisfied for this case.

2.3.2. The pair of nodes contains an antecessor of node A (A') and node B. In this case, the antecessor of node A, node A', is dead and entry E inside it is alive. The nodes being tested, A' and B, are both dead, one (A') containing a live entry (E) and other (B) containing a dead entry (F). This case has the same properties of case 0, so, the entries E and F will not be reported.

2.3.3. The pair of nodes contains an antecessor of node B (B') and of node A. As what happens in the last case, it has the same properties of case 1.1.2.

3. One of the nodes is alive and the other dead at the end of the query time interval. Suppose, without loss of generality, that node A is alive and that node B is dead. The algorithm will report only the dead entries in node B. These entries will not appear on any successor of node B, and only antecessors of A and B must be checked. Three cases may occur:

3.1. The pair of nodes contains both antecessors of nodes A and B. In this case, both of the nodes are dead and both of the entries E and F are alive. It has been shown before that no condition may be satisfied for this case.

3.2. The pair of nodes contains one antecessor of node B and node A. In this case, the antecessor of node B is dead and entry F is alive, which invalidates all conditions.

3.3. The pair of nodes contains one antecessor of node A and node B. All antecessors of node A are dead and the entry E inside them is alive. If entry F is alive in node B, condition 3 cannot become true. Otherwise, condition 3 may be satisfied only if the node antecessor of A which contains the live entry E died after the node B that contains the live entry F. This case is impossible as it was on case 2.2.2. and can be better viewed in Figure 4.

Having fulfilled all the possibilities, the statement that the induction hypothesis is true for level n+1 can be inferred. This statement, together with the initial step of the induction finalizes the proof of the minimalism of the time interval spatiotemporal join algorithm.

4.5 Temporal Joins using the MVB-Tree

Notice that the proposed algorithm can be adapted to work on the MVB-Tree and solve the same problem of duplicated entries, since it doesn't use any spatial property in order to not list the duplicated pairs, and the MVB-Tree has the same temporal properties listed as P1, P2, P3 e P4. In fact, in a technical report presented by [21], a similar approach is used, under the name of BCO, although it uses conditions that are different from those presented here. In the future we shall investigate the similarities between the two algorithms because both were developed to solve a similar problem, although in different domains.

4.6 Spatiotemporal Joins using the MV3R-Tree

This work can be extended to propose an efficient algorithm to perform spatiotemporal joins using the MV3R-Tree [14], once this STAM is a mix of a 3D R-Tree and a structure similar to the TR-Tree. This work can also be applied to choosing when to use the 3D R-Tree or the TR-Tree when performing the spatiotemporal join. The time instant selection query under the MV3R-Tree is performed using the TR-Tree and the time interval selection query is performed using the 3D R-Tree.

5 Experimental Results

5.1 Data Set Generation

For our performance evaluation tests, we generated random data sets using the GSTD (Generate SpatioTemporal Data) software proposed in [15]. There are four kinds of data sets according to the number of versions of 250, 500, 750 and 1000. Each data set has 100.000 rectangle objects initially inserted and the same number of modifications into the existing objects, resulting on 200.000 insertions and 300.000 operations (insertions plus deletions). For each number of versions, we have generated two pairs of data sets varying the seed for the random numbers, leading to 8 pairs of data sets. We used 4 Kbytes pages and an independent LRU disk cache for each structure with 97 pages of 4 Kbytes each, plus a cache with the height of the tree (in our tests, 3 at most) to keep the path from the root node to the leaves. The machine used was an Intel Pentium III 700MHz with 256 Mb of main memory, running Linux Conectiva Server 6.0 and gcc 2.95 compiler. We have tried to eliminate the influence of the Linux Operating System disk cache in our tests. The total CPU time was calculated using the real measured computational time minus the real measured disk accesses time. This measured disk accesses time was calculated using the number of disk accesses multiplied by a mean time of one access (in our tests this was about five milliseconds).

Table 3 shows the space utilization of the nodes for the two structures in question with 4 kbytes pages.

	TR-Tree	**2+3D R-Tree**	
		2D R-Tree	3D R-Tree
Min. Entries / Node	64	85	72
Max. Entries / Node	128	170	145

Table 3 – Space utilization of the nodes with 8 kbytes page.

We measured the index construction time as well as the size in bytes of the index files. These results are shown in the charts of Figure 8. The numbers shown in these charts are the mean one for each of the five data sets with the same characteristics.

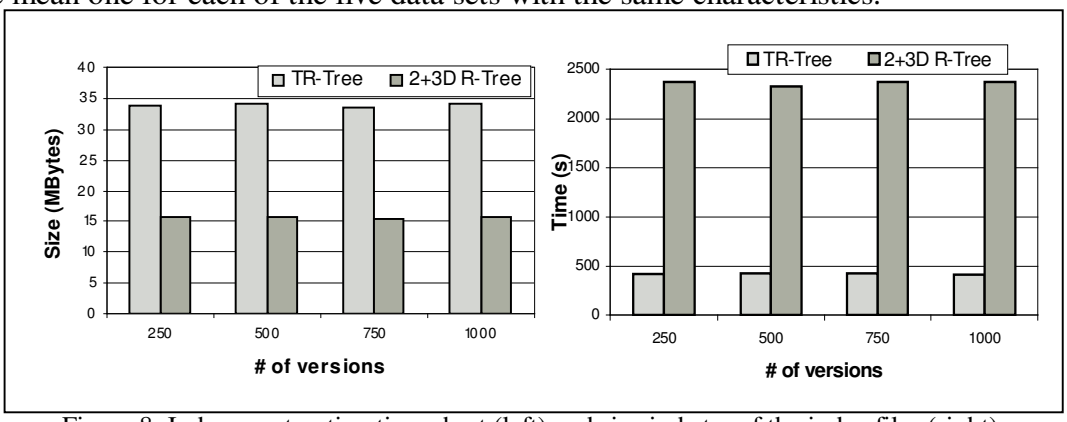

Figure 8: Index construction time chart (left) and size in bytes of the index files (right).

We can state, with the help of these figures, that the index size and the index creation time of both of the indexes almost do not suffer from the influence of the number of versions in the structure. However the index size of the TR-Tree is almost double the 2+3D R-Tree, its creation is about five times faster.

5.2 Evaluated Joins

For each data set we executed 50 time instant spatiotemporal joins and 50 time interval spatiotemporal joins. The spatial selection window and the time interval or time instant were chosen randomly.

Figure 9 shows the resulting sum of the total time for all time instant spatiotemporal queries for each data set.

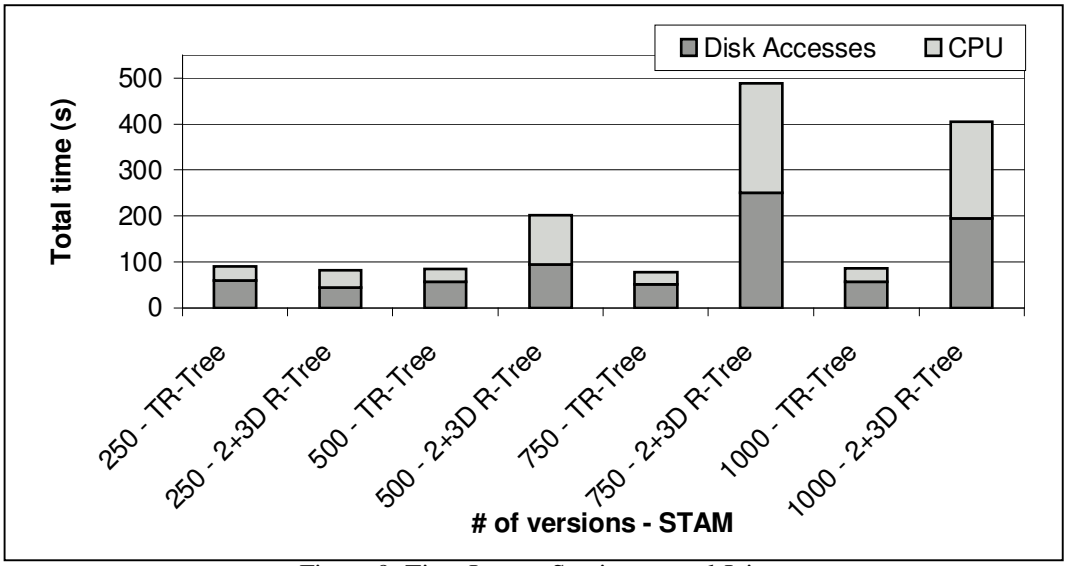

Figure 9: Time Instant Spatiotemporal Join

We can note in this figure that the TR-Tree index structure has a much better performance than the 2+3D R-Tree for data sets with a large number of versions. We can also note that the

algorithm stated in this paper to perform the time instant spatiotemporal join in the TR-Tree has good scalability, once the total time spent is almost the same for all kinds of data sets. On the other hand, the algorithm, proposed for the 2+3D R-Tree, does not have this good characteristic.

Figure 10 shows the resulting sum of the total time for all time interval spatiotemporal queries for each data set.

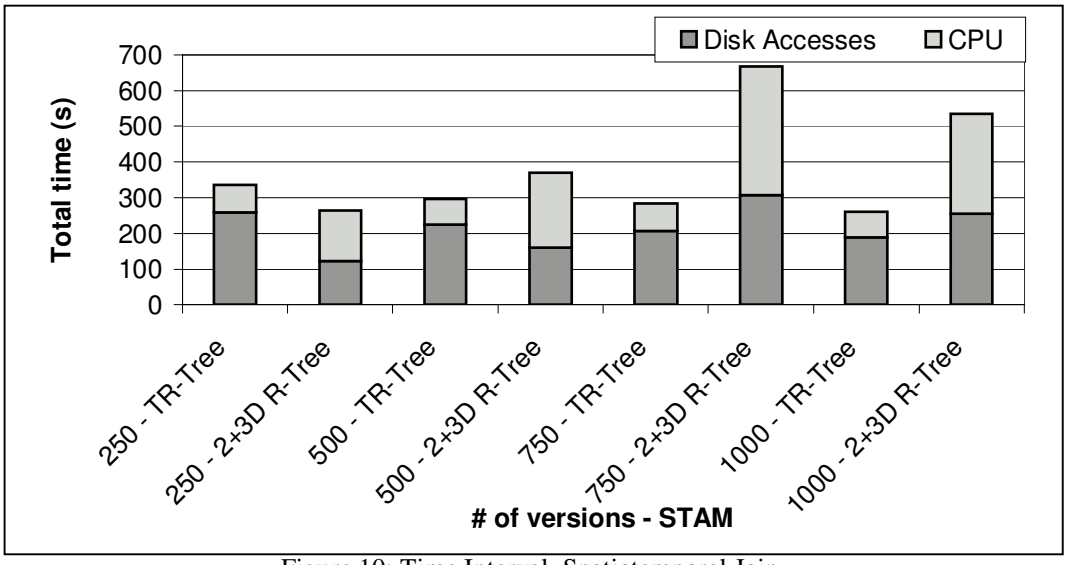

Figure 10: Time Interval Spatiotemporal Join.

We can note again the good behavior of the join algorithm proposed for the TR-Tree against the increase of the number of versions. For the time interval spatiotemporal join, the difference between the two STAMs compared here is not that big as it was for the time instant spatiotemporal join.

Another good observation here is that the CPU time is not appalling, and in some cases, especially for the 2+3D R-Tree, it is half of the total query time.

6 Conclusion

After performing all these queries, using different datasets, we can conclude that our algorithm, evaluated in the TR-Tree, has better performance than the 2+3D R-Tree using an R*-tree-like join approach. In addition, the TR-Tree's index construction was five times faster than the 2+3D R-Tree, although the TR-Tree size was two times the 2+3D R-Tree size.

This is a preliminary work on performance evaluation of spatiotemporal joins. Much more work has to be done comparing other STAMs proposed in the literature recently, like MVR-Trees [14] for example.

As ongoing work, we are investigating the use of TR-Trees in real applications, some of them with some kind of temporal or spatial dominance. Additionally, we are working on a parallel version of our algorithms – both the TR-Tree index updates and query evaluation.

7 References

1. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer: "An Asymptotically Optimal Multiversion B-tree", The VLDB Journal, vol.5(4), pp. 264-275, December 1996.

2. N. Beckmann, H. P. Kriegel, R. Schneider, et al. 1990. "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles". In: Proceedings of the ACM SIGMOD Intl. Conf on Management of Data, Atlantic City, NJ, 1990.

3. T. Brinkhoff, H. P. Kriegel, and B. Seeger: "Efficient processing of Spatial Joins Using R-Trees". In Proceedings of the 1993 ACM-SIGMOD Conference, Washington, DC, USA,May 1993.

4. 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.

5. I. Kamel and C. Faloutsos, “Hilbert R-Tree: An improved R-Tree using fractals”, In Proceedings of the 20th Very Large Databases Conference, pages 500-509, September 1994.

6. G. Kollios, D. Gunopulos, V.J. Tsotras, A. Delis, M. Hadjieleftheriou: “Indexing Animated Objects Using Spatiotemporal Access Methods”, To appear at IEEE Transactions on Knowledge and Data Engineering

7. A. Kumar, V. J. Tsotras, and C. Faloutsos: “Access Methods for Bitemporal Databases”. In Recent Advances in Temporal Databases, J. Clifford, and A. Tuzhilin (eds), pp. 235--254, Springer Verlag, 1995.

8. A. Kumar, V. J. Tsotras, and C. Faloutsos: “Designing Access Methods for Bitemporal Databases”. TR3764, Dept. of Comp. Sci. University of Maryland, 1997.

9. D. Lomet, B. Saltzberg, “Access Methods for Multiversion Data”, Proceedings of ACM SIGMOD Conference, 1989.

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

11. M. A. Nascimento, J. R. O. Silva, “Towards Historical R-trees”, Proceedings of ACM Symposium on Applied Computing (ACM-SAC), 1998.

12 Saltenis S. and C. S. Jensen, “R-Tree Based Indexing of General Spatio-Temporal Data”', TimeCenter Technical Report TR-45, December 1999, and Chorochronos Technical Report CH-99-18, December 1999.

13. 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.

14. Tao, Y., Papadias, D., “MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries”. Proceedings of Very Large Data Bases Conference (VLDB), Rome, Sept 11-14, 2001.

15. Theodoridis, Y., Silva, J.R.O., and Nascimento, M.A. On the Generation of Spatiotemporal Datasets. 6th Intl. Symposium on Spatial Databases (SSD'99), Hong Kong, China, July 1999.

16. Yannis Theodoridis, Timos Sellis, Apostolos N. Papadopoulos, and Yannis Manolopoulos: “Specifications for Efficient Indexing in Spatiotemporal Databases”, Proceedings of SSDBM’98, Capri, Italy, July 1-3 1998.

17. Y. Theodoridis, M. Vazirgiannis, and T. Sellis: “Spatio-Temporal Indexing for Large Multimedia Applications” Proceedings of the 3rd IEEE Conference on Multimedia Computing and Systems (ICMCS), 1996.

18. X. Xu, J. Han, W. Lu: “RT-tree: An Improved R-tree Index Structure for Spatiotemporal Databases”, Proceedings of the 4th International Symposium on Spatial Data Handling (SDH), 1990.

19. Zimbrão, G., Almeida, V. T, Souza J. M, “Efficient Processing of Spatiotemporal Queries in Temporal Geographical Information Systems”, 6th International Conference on Information Systems, Analysis and Synthesis ISAS 2000.

20. G. Zimbrão, V. T. Almeida, and J. M. Souza: “The Temporal R-Tree”. Technical Report ES492/99, COPPE/Federal University of Rio de Janeiro, Brazil, March 1999.

21. D. Zhang, V. Tsotras and B. Seeger, “A Comparison of Indexed Temporal Joins”, TimeCenter Technical Report TR-50, 2000.