Efficient Processing of Spatiotemporal Queries in Temporal Geographical Information Systems

Geraldo Zimbrão
Jano Moreira de Souza
Renata Chaomey Wo
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, jano, valmeida, wo} @cos.ufrj.br

Abstract

It's a well-known fact that new GIS applications need to keep track of temporal information. However, the most widely used spatial index, the R-Tree and its variants, do not preserve the evolution of bounding boxes. Some new indexing structures were proposed in the literature that allows the retrieving of present and past states of data. In this paper we evaluate the most popular indexes structure to perform spatiotemporal queries. Spatiotemporal queries can be grouped in two classes: queries involving time instants and queries involving time intervals. We point out which are the best indexes for each class. In this paper we also propose a new data structure termed the Temporal R-Tree, which combines several features of Becker et al. Multi-Version B-Tree with the spatial indexing characteristics of the R-Tree. We have shown that the Temporal R-Tree is the only structure suitable for efficiently supporting both classes of queries - the other structures proposed in the literature are fine for just one of these classes. A number of comparative queries on several data sets were performed to show the scalability of the Temporal R-Tree. Our experiments show that its performance and space utilization are very good even on worst cases, showing that the Temporal R-Tree is by now the best choice for indexing spatiotemporal data.

Keywords: Spatiotemporal access methods, index.

1. Introduction

The main task to be supported by a Spatiotemporal Database Management System (STDBMS) is the efficient storage and retrieval of spatiotemporal (i.e., time evolving spatial) objects. This paper presents a new data structure, the Temporal R-Tree, and evaluates its use to index spatiotemporal data in the context of STDBMS. The Temporal R-Tree is an emerging Spatiotemporal Access Method, STAM for short, related to the Bitemporal R-Tree [KTF95, KTF97] and the Multiversion B+-Tree [BGO+96]. The Temporal R-Tree can efficiently support window queries, both on current and past states of data. Also, window queries that span across many states of data can be efficiently performed using the Temporal R-Tree index.

Spatiotemporal Database Management Systems, as stated in [TSPM98], should offer appropriate data types and query language to support spatial data that evolves along the time, provide efficient indexing and retrieval methods, and exploit cost models of spatiotemporal operations for optimization purposes. Relevant applications that a STDMS should support include Geographic Information Systems (GIS), Transaction / Valid Time Databases, Multimedia Systems, as well as Scientific and Statistical Databases, such as databases for medical, weather, environmental, astrophysics etc. A STDBMS is a system that keeps track of old states of data. In such systems, each update to data creates a new version of it. The system never deletes an object: it just records its death time. Thus, the amount of data is ever growing. The need for indexing spatiotemporal databases is obvious: the bigger the database the more explicit the performance gains from using sophisticated indexing techniques. However, this demands new techniques for simultaneously indexing spatial and temporal data.

A crucial query that a plain Spatial DBMS should answer is the *selection query*, or *window query*: "find all objects that are inside a specific rectangle". According to [TSPM98], the window query is also a basic query that a STDBMS should answer, but now taking into account the spatiotemporal nature of data. In essence there are two classes of window query on spatiotemporal data: a window query on a single version of data (present or past versions) that represents a time instant, and a window query on an set of versions that represents a time interval. The first query can be stated as "find all objects that were inside the rectangle (x1, y1, x2, y2) at time *t*". The second one can be stated as "find all objects that were inside the rectangle (x1, y1, x2, y2) during interval time (t1, t2)".

There are a number of other structures suitable for spatiotemporal indexes. The most promising are: 2+3D R-Tree [TVS96], RT-Tree [XHL90] and HR-Tree [NS98]. According to [NST98] and [NTS99], the HR-Tree is the best structure to perform window queries on time instants (or small intervals of time), and the 2+3D R-Tree is a competitive approach to traverse large intervals of time. Their experiments focus on discretely moving points (i.e., points standing at a specific location for a time period and then moving "instantaneously", and so on and so forth).

The Temporal R-Tree, or TR-Tree, is a partially persistent R-Tree [G84]. According to [B94], 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. R-Trees 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 STAM. The TR-Tree is an enhanced R-Tree able to index both spatial and temporal

information. It is correct to say that the TR-Tree is the R-Tree counterpart of the Multiversion B+-Tree [BGO+96]. There is a number of applications for the TR-Tree. An interesting one is providing index mechanisms for spatiotemporal databases.

We assume that the TR-Tree will store *transaction time* for its MBR. In the context of temporal databases, the transaction time of a database fact is the time when the fact became current in the database and may be retrieved [JCG+94]. According to [TSPM98], adding support for transaction time to spatial databases meets the requirements of many applications. 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.

In this work, we performed more then 1,400 queries on several data sets (real and derived) using the TR-Tree and the other spatiotemporal access methods above mentioned, focusing on discretely moving points. For evaluating the *time instant window query*, the HR-Tree has the best performance, although its space utilization is the worst one when there are many versions of the data. In this case, the TR-Tree has a comparable performance and a much better space utilization. For evaluating the *time interval window query*, the 2+3R-Tree is the best one in many data sets, and has the best space utilization. But our experiments show that the TR-Tree performance is comparable to the winner structures in both classes of queries (in many cases, the TR-Tree outperforms them). Also, in terms of index building time and space utilization, the TR-Tree outperformed the other structures in most data sets.

Finally, in a mixed set of queries, that is, 50% of each class of query, the TR-Tree outperforms all other structures. So, the TR-Tree is the best candidate structure for indexing data in spatiotemporal databases when both classes of query are needed, even when there are no pre-built indexes.

This paper is organized as follows. Section two states the problem. Section three presents a brief overview on related works. Section four discusses the TR-Tree structure and the basic algorithms for inserting, "deleting", updating and searching. Section five presents and comments the performance evaluation of TR-Tree and the other structures in terms of time spent on index building, query execution and disk utilization. Section six concludes the paper and gives an outlook to an ongoing work: the spatiotemporal join query.

2. Defining the Problem

In the context of R-Trees the search operation is usually called the window query, and returns all MBRs that overlaps a given rectangle *R*. The TR-Tree counterpart operation searches for all MBRs that overlaps *R* at a given time *t*. The point query is a special case of a window query where the rectangle *R* degenerates to a point. The query window can be temporal extended to allows retrieving of all MBRs that overlaps a rectangle *during* a given interval time $[t_1,t_2)$. Such a query is called the time interval window query, and according to [L92] it is one of the most useful query that a Temporal GIS should answer. Also, [L92] points that both kinds of query are very important for many applications.

The search operation in R-Tree is I/O bound, since the algorithm itself is very trivial. A good performance measurement is the ratio between entries in the answer set and accessed nodes. The optimum performance is achieved when this value is close to the maximum node entries. One should note that only in one case the optimum result can be found: when we have only one node and it is completely full. In all other cases, intermediate nodes (root or non leaf) will be accessed, and since the data is only in leaf nodes, the ratio will be smaller then the maximum node entries. Also, most of the nodes will have less then the maximum node entries. The good performance of the R-Tree is assured by the restriction that a node can't have less then a fixed number of entries (called the minimum node entries).

In the structures proposed in the literature, we can identify an important characteristic: the replication of data. The HR-Tree and our approach, the TR-Tree, replicates entries, and the RT-Tree and the 2+3D R-Tree don't. Replication of data is useful when performing time instant window queries, since old entries and future entries will not be stored together, so each visited node will have most of the entries live in the time of the query – leading to similar results to the original R-Tree, as pointed out in [NTS99]. But in the time interval window query, replication decreases the performance since the same MBR will be listed twice, so the non-replicating structures should present a better performance. As a matter of fact, [NTS99] and our tests corroborate this, although in many cases the TR-Tree performance is close or outperforms the other trees.

3. Related Work

In this section, we will summarize some work done in related areas; the other index structures and their approaches to answer the window query on spatiotemporal data.

The Multiversion B+-Tree presented in [O94] is a good approach to handle temporal but non-spatial data. The main idea of that work is to keep the deleted data in the node 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 the replication of old data. The resulting structure is a condensed collection of overlapped B+-Trees that keeps all the past states of the index plus the current one. Our work resembles that one in the context of spatiotemporal data and R-Trees, and we have tried to agree with its naming conventions. Although many of its performance assumptions for B-Trees does not hold for R-Trees, our experiments achieve good results in terms of disk space, I/O operations and time spent.

The Bitemporal R-Tree proposed in [KTF95] and [KTF97], in the context of bitemporal databases (valid and transaction time databases), is an implementation of a partially persistent R-Tree very close to our work. The Bitemporal R-Tree is used to index a non-spatial key, its valid time and its transaction time. The transaction time is handled by applying a partially persistent approach to a R-Tree that indexes key values and valid time intervals. Valid time intervals are also mapped to 2D points. Once these assumptions are stated, there is a number of design choices that were made that prevents the Bitemporal R-Tree from indexing spatiotemporal data directly – many adjustments must be done in the MVBT versioning mechanism.

At the time of the publication of [TSPM98], only four spatiotemporal indexing methods have appeared in the literature: 3D R-trees [TVS96], MR-trees and RT-trees [XHL90], and HR-trees [NS98]. 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 [G84, BKS+90],

- MR-trees and HR-trees use overlapped R-trees to represent successive states of the database [MK90], and
- RT-trees couple time intervals with spatial ranges in each node of the tree structure by adopting ideas from R-trees and TSB-trees [LS89].

[NST98] is a pioneer work on comparing index structures for spatiotemporal data. In that work, they conclude that HR-tree is the best choice to index spatiotemporal data when most of the queries are time instant queries, that is, queries that require access to only one version of data set. The HR-Tree and two other approaches, the 3D R-Tree [TVS96] and the 2+3 R-Tree proposed there.

According to [TSPM98], the HR-tree structure is not a good choice for indexing spatial data that has a medium to large changing ratio. Also, it is not well suited to support time interval queries. In fact, one of the [NST98] conclusions is that: "If instead of a time point a time interval is queried, the HR-tree loses its advantage rather quickly with the increase in the length of the queried time interval". The HR-tree break off is about five timestamps – after that its performance is very poor.

To this list, we should add two other access methods: Overlapping Linear Quad trees [TVM98] and RST-Trees [SJLL00]. The Overlapping Linear Quad tree is a structure suitable for storing consecutive raster images according to transaction time (a database of evolving images). This structure saves considerable space without sacrificing time performance in accessing every single image. Moreover it can be used for answering efficiently window queries for a number of consecutive images (spatiotemporal queries). The RST-Tree provides support for both valid time and transaction time (bitemporal) spatial data, thus it is a more generic approach. It combines some ideas from R*-Trees [BKS+90] and GR-Trees [BJSS98]. The kind of data that RST-Trees indexes is different from the data indexed by the other structures mentioned before: it has two time dimensions, the valid time and the transaction time, plus the spatial extension. Also, since it is a very recent work we had no time to analyze it and include it in our comparisons.

Also, there is 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 the position of a spatial object and its shape cannot change continuously, but only in discrete steps, and hence we are not talking about moving objects. According to [TSPM98], this case is of great interest for the spatiotemporal databases.

4. The TR-Tree

In this section we will present the TR-Tree algorithms for single time searching, inserting and deleting. All of these operations are the TR-Tree counterpart of the R-Tree basic operations. We will describe them in terms of modifications to the original R-Tree, as it was first presented in [G84]. In addition, the TR-Tree has a specific operation called time interval search. Also, as a result of our implementation choices, we will present a little adjustment for the TR-Tree algorithms that allows many insertion and deletions at the same time. Some procedures were omitted due two reasons: simplicity or similarity with the correspondent R-Tree procedure, or to save space. A more detailed version of the algorithms presented here can be found at [ZAS99].

4.1 Basic Structure

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 *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 world current time. The TR-Tree *last time* is the time when the last update has occurred. The leaf nodes will hold 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. Also, each node will have two more attributes, *birth-time* and *death-time*, indicating its lifespan. Each MBR for a child node should be the MBR enclosing all the MBRs in the entries of the child node during the [*birth-time*, *death-time*) interval pointed in the entry – if the entry *death-time* is set to "*" then we use the parent node *death-time*, and so on.

Whenever a new MBR of a polygon is added to the database at time t_i its lifetime interval is set to [t_i,*). A entry 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 entry from "*" to t_i. Updating a MBR of a polygon 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. An entry 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*). Moreover, for every time t_i, $t_i <$ *now* holds.

Likewise, when the TR-Tree creates a new node at time t_i 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 ***grows*** at time t_k(e.g., when inserting a new entry), the last entry pointing to this node is updated with the new MBR. When the MBR of a node ***shrinks***, nothing is done. Thus, many different entries may exists for a node, but at each time *t* only one entry is *live*. Also, the MBR of the last entry holds the MBRs of other entries for the same node. Finally, each live entry reflects the MBR of the node at the *death* time of the node where the entry is (the parent node of the entry node, and so on).

If a node should be removed from the tree, due to structural changes like node overflow or underflow, its logically deleted by setting the *death-time* attribute of both the node and its corresponding entry. A node is said to be *live* at time t_i if *birth* $\leq t_i <$ *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. In the literature, this kind of structure is also called an *overlapping* structure.

By construction, querying a TR-Tree at any time has the same time and disk costs as querying the corresponding standard R-Tree. Any change in the current R-Tree leads to a new R-Tree, and both trees will be efficiently stored at the TR-Tree. So, at any time t_i, querying the TR-Tree can be compared to querying the plain R-Tree as it was at time t_i. The only overhead in such a query will be the time spent to locate the corresponding R-Tree root in the TR-Tree array of roots. This array can be efficiently indexed by the *death-time* of the roots since each one is created in chronological order. Anyway, in our experiments, for real data sets of 128,971 elements and creating a new version at each insert/delete operation the maximum number of roots was 20, so they could be kept in main memory.

Good space utilization is assured by using a mechanism called *version-split*, as one can see in Figure 1.

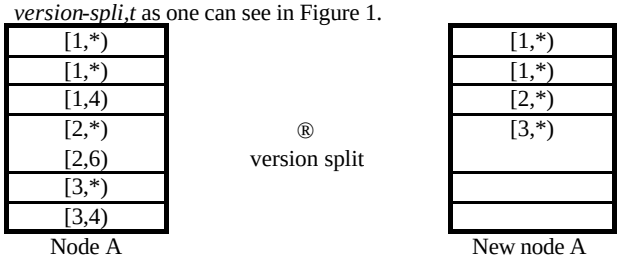

Figure 1 - Version split: only live entries are copyied.

Time and space efficiency of the R-Tree is based on the properties proposed by Guttman, specially the assumption R1. This property ensure 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, $k \geq 2$. Those properties should be modified as follows (changes are bold faced):

R1. Every node contains at least m **live** entries at any time of its lifespan, that is, any time $t \in$ [*birth-time*, *death-time*), 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) during the lifespan of the node and the child node or data object;

R3. The root node has at least two live entries for children nodes at any time of its lifespan, 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 *live* 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 also call this operation *version-split*. Considering a node overflow, most of the 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, $\varepsilon > 0$, to be defined more precisely in the following.

After a version-split, if the node violates the strong version condition, then 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 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 [BGO+96], 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 1.

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

Table 1 - 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 the creation of new versions of the tree should be explicitly done by someone. It is an improvement appropriate for a number of cases like massive update, intermediate steps produced by the application of referential integrity constraints, and other operations that generates invalid or useless data that the system does not need to store, like reinsertion of entries of deleted nodes. Finally, the main ideas used here to made the R-Tree a partially persistent data structure were derived from [BGO+96], and were adjusted to provide support for spatial data.

For a detailed version of the TR-Tree algorithms, one should see [ZAZ99].

5. Performance Evaluation

In this section, we present and comment the results of evaluating the proposed algorithms. The test machine was a Pentium II 350 Mhz PC with 64 Mb of main memory running Linux. In the time interval queries, the size of the indexes is related to the number of I/O operations, so we also show the size of the structures. The TR-Tree was implemented using the algorithms showed above. To perform the comparisons, the HR-tree index structure was implemented as stated in [NST98].

Also, we have implemented the 2+3D R-Tree as stated in [TVS96] and an improved RT-Tree as stated in [XHL90] – improved in the sense that we have implemented it using the R*-Tree algorithms for splitting and forced reinsertion. All the structures uses nodes of 4k and a LRU cache that accommodates 29 nodes. Table 2 shows the min and max node entries for each structure. One should note that the TR-Tree entry requires more space than the other structures entries, so it presents less entries per node. Also, the numbers for the 2+3D R-Tree represents the 2D R-Tree and the 3D R-Tree substructures. Finally, the numbers for the TR-Tree reflects the *weak* and *strong* conditions.

Structure	Min entries per node	Max entries per node
2+3D R-Tree	84/72	169/145
HR-Tree	84	169
RT-Tree	72	145
TR-Tree	41/53	113/125

Table 2 - Structures parameters

5.1 Data Sets

Our base data set is the same that was used in [BKS93], and consists of 128,971 MBRs for line objects in an area of California representing rivers and railway tracks. To simulate a temporal dimension, we have artificially deleted some of the MBRs in this data set to create another data sets. To perform the tests, we try to simulate the environment of a hypothetical application where there is a number of live objects that is maintained relatively constant by balanced updates. A large group is initially inserted and then the updates are grouped in small sets of deletions and insertions

We generate 9 datasets. There were 128.971 insertions and 103.176 deletions for each data set. Operations were bulked in order to produce datasets with different number of versions.

Data set	#Versions	Data set	#Versions
DS20_1	1	DS20_100	100
DS20_5	6	DS20_1k	1002
DS20_10	10	DS20_10k	10318
DS20_15	16	DS20_100k	103178
DS20_20	20	DS20_1_1	206359

Table 3 - Data set description

5.2 Queries Description

We have created two kinds of interval queries: time dominating queries and spatial dominating queries. The time interval of the queries includes 20, 40, 60, 80 and 100% of the total lifespan of each data set that the query was performed. Also, the rectangle window of queries comprises 1, 10, 50 and 100% of the total space.

So, we have a total of 40 different queries to be performed in 10 datasets, for each of the 4 structures, leading to the total of 1600 queries. But it was infeasible the use of HR-Tree to index the data sets with more then 1000 versions, so for some datasets we have only 3 index. The total number of queries was about 1400.

We have summarized the results of the comparisons of queries in two big classes: the instant queries and the interval queries. Also, we have measured the index building performance for each data set.

5.3 Results

Index Building: Figure 3 shows the size of each index. The TR-Tree index is competitive, although it is not the best. Moreover, its size is not affected by the number of versions in the data set. For the last datases, the HR-Tree size was over 1 Gigabyte (the DS20_1k HR-Tree was 600Mb).

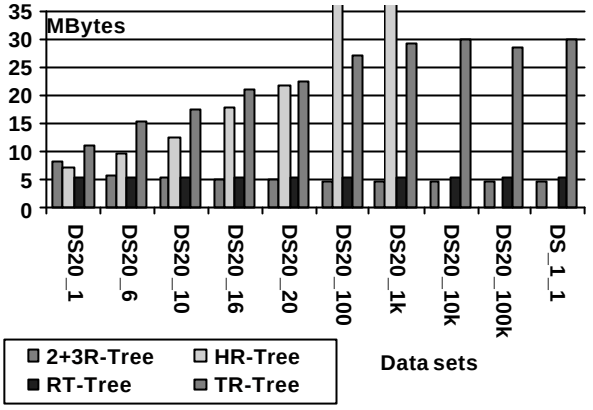

Figure 3 - Index size

Time Instant Window Queries: We have computed the total time for all instant queries. Figure 4 summarizes the results. Our results are very close to the HR-Tree, and many times it outperforms the HR-Tree.

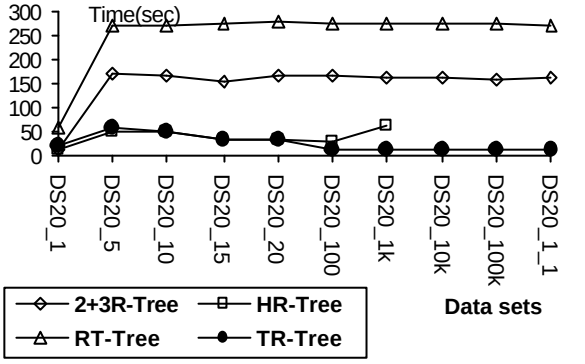

Figure 4 – Seconds spent on Time Instant Queries

Time Interval Window Queries: Figure 5 summarizes the results. As one can see, although the TR-Tree is not the best, it is competitive. Moreover, the number of versions in the data set does not affect its performance.

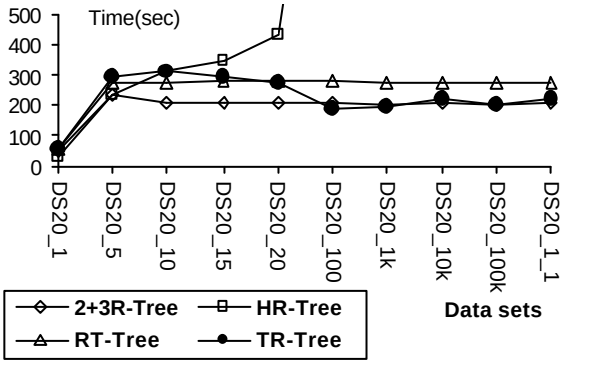

Figure 5 – Seconds spent on Time Interval Queries

Mixed Queries: We have computed the total time for a mix of 50% of time instant and 50% of interval queries. Figure 6 summarizes the results. In this scenario, the TR-Tree outperforms all other structures. So, it is the best choice when both kinds of queries are needed.

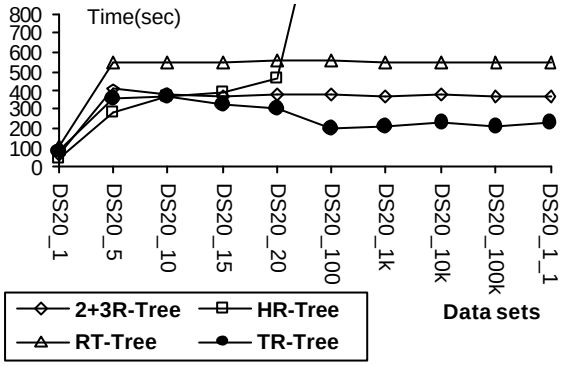

Figure 6 – Seconds spent on Mixed Queries

6. Conclusion

In this paper, we have proposed, implemented and evaluated a new index structure called Temporal R-Tree that deals with the basic spatiotemporal data: point, lines, regions and transaction time. 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. Just small overheads are required for the insert and delete operations when compared to R-Tree standard operations. The performance of the window query is the same that the original R-Tree.

In addition, it is possible to search in past states of the tree and efficiently answer the time range window query, a typical operation for spatiotemporal databases. The TR-Tree is the first implemented spatiotemporal index able to efficiently answer time instant and time interval query. Other approaches can efficiently answer only one of these kinds of query. Also, our experiments show that the TR-Tree performance is comparable to the winner structures in both classes of queries (in many cases, the TR-Tree outperforms them). This makes the TR-Tree a competitive approach for spatiotemporal indexing in transaction time spatiotemporal databases.

Finally, in a mixed set of queries, that is, 50% of each class of query, the TR-Tree outperforms all other structures. So, the TR-Tree is the best candidate structure for indexing data in spatiotemporal databases when both classes of query are needed, even when there are no pre-built indexes.

Future work includes extending the TR-Tree to support Transaction and valid time, and perform spatiotemporal joins.

References

[B94] G. S. Brodal: "Partially Persistent Data Structures of Bounded Degree with Constant Update Time". In Nordic Journal of Computing, volume 3(3), pages 238-255, 1996.

[BGO+96] 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.

[BJSS98] R. Bliujute, C. S. Jensen, S. Saltenis, G. Slivinskas: "R-Tree Based Indexing of Now-Relative Bitemporal Data". Proceedings of 24rd International Conference on Very Large Data Bases, New York City, New York, USA, August 24-27, 1998.

[BKS+90] 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.

[BKS93] 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.

[BKSS94] T. Brinkhoff, H. P. Kriegel, R. Schneider, and B. Seeger: "Multi-step Processing of Spatial Joins". In Proceedings of the 1994 ACM-SIGMOD Conference, Minneapolis, USA, May 1994.

[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.

[G84] 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.

[JCG+94] C. Jensen, J. Clifford, S. Gadia, A. Segev and R. Snodgrass: "A consensus glossary of temporal database concepts". ACM SIGMOD Record 23, 1 (Jan 1994), 52--64.

[KTF95] 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.

[KTF97] A. Kumar, V. J. Tsotras, and C. Faloutsos: "Designing Access Methods for Bitemporal Databases". TR3764, Dept. of Comp. Sci. University of Maryland, 1997.

[L92] Langran, G.: "Time in Geographical Information Systems". Taylor & Francis, London, 1992.

[LS89] D. Lomet, B. Saltzberg, "Access Methods for Multiversion Data", Proceedings of ACM SIGMOD Conference, 1989.

[MK90] 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.

[NS98] M. A. Nascimento, J. R. O. Silva, "Towards Historical R-trees", Proceedings of ACM Symposium on Applied Computing (ACM-SAC), 1998.

[NST98] M. A. Nascimento, J. R. O. Silva and Y. Theodoridis: "Access Structures for Moving Points". Time Center Technical Report TR-33, September/98.

[NTS99] M A. Nascimento, J. R. O. Silva and Y. Theodoridis: "Evaluation of Access Structures for Discretely Moving Points". Spatio-Temporal Database Management 1999: 171-188.

[O94] T. B. G. Ohler, "On the integration of Non-Geometric Aspects into Access Structures for GIS", PhD dissertation submitted to the Swiss Federal Institute of Technology, 1994.

[SJLL00] S. Saltenis, C. S. Jensen, S. T. Leutenegger, M. A. Lopez: Indexing the Positions of Continuously Moving Objects. To be published in SIGMOD Conference 2000.

[TSPM98] 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.

[TVS96] 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.

[XHL90] 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.

[ZAS99] 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. www.cos.ufrj.br/~valmeida/zaz99.pdf