Access Structures for Moving Points

Mario A. Nascimento *Jefferson R. O. Silva*
Yannis Theodoridis

Relatório Técnico IC–98-34

Outubro de 1998

Access Structures for Moving Points

Mario A. Nascimento [§] Jefferson R. O. Silva [¶] Yannis Theodoridis [‖]

Abstract

Several applications require management of data which is spatially dynamic, e.g., tracking of battle ships or moving cells in a blood sample. The capability of handling the temporal aspect, i.e., the history of such type of data, is also important. This paper presents and evaluates three temporal extensions of the R-tree, the 3D R-tree, the 2+3 R-tree and the HR-tree, which are capable of indexing spatiotemporal data. Our experiments have shown that the while the HR-tree was the larger structure, its query processing cost was over 50% smaller than the ones yielded by the 3D R-tree and the 2+3 R-tree. Also compared to the (non-practical) approach of storing one R-tree for each of the spatial database states it offered the same query processing cost, saving around one third of storage space.

1 Introduction

The primary goal of a spatiotemporal database is the accurate modeling of the real world; that is a dynamic world, which involves objects whose position, shape and size change over time [T+98]. Real life examples that need to handle spatiotemporal data include storage and manipulation of ship and plane trajectories, fire or hurricane front monitor and weather forecast. Geographical information systems are also a source for spatiotemporal data. For instance, in the agricultural domain, land owners as well as government agencies must keep track of crop situation over time to take decision as such to where use pesticides and/or provide financing to farmers. To take another example, consider a series of snapshots taken from a satellite. This type of data, possibly after some treatment, is clearly spatiotemporal data. As yet another example domain, consider the problem of video (or multimedia in general) database management. Objects that appear in each frame can be considered two-dimensional moving objects, upon which one may want to keep track over time or exploit relationships among them. Summarizing, "database application must capture the time-varying nature of the phenomena they model" [Z+97, Ch. 5]. Spatial phenomena, hence spatial databases, are no exception, therefore spatiotemporal databases should florish as current technology makes it more feasible to obtain and manage such type of data ([Cor98] is a good example of that trend).

[§]Institute of Computing, State University of Campinas, Brazil. mario@cnptia.embrapa.br

[¶]Institute of Computing, State University of Campinas, Brazil. 972147@dcc.unicamp.br

[‖]Dept. of Electrical and Computer Engineering, National Technical University of Athens, Greece. theodor@dbnet.ece.ntua.gr

Among the many research issues related to spatiotemporal data, e.g., query languages and management of uncertainty [W+98], we focus on the issue of optimizing access structures, hence speeding up query processing. Despite the fact that there is much work done on the area of access structures for temporal [TK96, ST97] and spatial data [GG98], not much has been done regarding spatiotemporal data. This paper deals with this very point.

In [T+98], a set of seven criteria were proposed to characterize spatiotemporal data and access structures:

1. *Data types supported*: whether it supports points and/or regions;
2. *Temporal support*: whether the supported temporal dimension is that of valid time, transaction time or both;
3. *Database mobility*: whether the changes in cardinality or the spatial position of the data items, or both, can change over time;
4. *Data loading*: whether the data set evolution is known a priori or not, whether only updates concerning the current state can be made or whether any state can be updated;
5. *Object representation*: which abstraction (e.g., MBRs) is used to represent the spatial objects.
6. *Temporal treatment*: whether it support special actions such as packing or purging (vacuuming) spatial data as time evolves.
7. *Query support*: whether it is able to process not only spatial and temporal queries, but also queries which are spatiotemporal in nature.

After the directions above, and for the purposes of this paper, we assume spatiotemporal data specified as follows:

- The data set consists of two-dimensional points, which are moving within the unit square;
- For each point its versions' timestamps grow monotonically following a transaction time pattern, and
- The cardinality of the data set remains fixed as time evolves.
- Updates are allowed only in the current state of the database;

Hence, according to the terminology in [T+98], this paper considers databases of the point/transaction-time/evolving/chronological class.

Regarding the indexing structures, no packing or purging of data is assumed. Finally they must provide support to process at least two types of queries: (1) containment queries with respect to a time point; and (2) containment queries with respect to a time interval.

By containment query we mean one where given a MBR all points lying inside such MBR should be retrieved.

For simplicity, throughout the paper we assume the two-dimensional space, although extending the presented arguments for higher dimension is not problematic. Also, instead of a new access structure, we investigate how to extend a very well known one, namely the R-tree [Gut84, SRF87, B$^+$90, KF94].

The remainder of the paper is organized as follows. In Section 2 we detail the access structures which we will compare. Next, in Section 3, the methodology we used to generate spatiotemporal data is discussed. Section 4 presents and discusses the experiments we perform regarding space requirements and query performance. Finally, the paper is closed with a summary of our findings and directions for future research.

2 Spatiotemporal Access Structures

We are aware of only five access structures that consider both spatial and temporal attributes of objects, namely MR-trees and RT-trees [XHL90], 3D R-trees [TVS96], and, very recently, HR-trees [NS98] and Overlapping Linear Quadtrees [TVM98].

In the RT-tree the temporal information is kept inside the R-tree nodes. This is in addition to the traditional content of the R-tree nodes. On the other hand searching in the RT-tree is only guided by the spatial data, hence temporal information plays a secondary role. As such queries based solely on the temporal domain cannot be processed efficiently, as they would require a complete scan of the database. No actual performance analysis was reported in [XHL90].

The 3D R-trees, as originally proposed in [TVS96], use standard R-trees to index multimedia data. The scenario investigated is that of images and sound in a multimedia authoring environment. In such a scenario it is reasonable to admit that the temporal and spatial bounds of the indexed objects are known a priori. Aware of that fact the authors proposed two approaches, called the *simple* and the *unified* scheme. In the former one, a two-dimensional R-tree indexes the spatial component of the data set, and a one-dimensional R-tree indexes the temporal component. Query processing is performed using both trees and performing the needed operations between the two returned answer sets. The latter approach uses a single three-dimensional R-tree and treats time as another spatial dimension. The authors conclude that the advantage of using one or the other approach is a matter of trade-off based on how much often purely spatial or temporal queries are posed relatively to spatiotemporal ones.

The Overlapping Linear Quadtrees, the MR-trees and the HR-trees are all based on the concept of overlapping trees [MK90]. The basic idea is that, given two trees where the younger one is an evolution[1] of the older one, the second one is represented incrementally. As such only the modified branches are actually stored, the branches that do not change are simply re-used (we discuss such an idea in more details shortly). The Overlapping Linear Quadtrees, as the name implies are based on Quadtrees [Sam90] and as such are not constrained to index only MBRs. The MR-trees and the HR-trees are very similar in

[1] By evolution we mean a further version based on some changes upon the same data set.

nature and we comment on the HR-tree in more details shortly. Indeed, next we discuss the three access structures we investigate in the remainder of this paper.

2.1 3D R-tree

The structure we discuss here is based on the 3D R-tree proposed in [TVS96]. The most straightforward way to index spatiotemporal data is to consider time to be another axis, along with the traditional spatial ones. Using this rationale, an object which lies initially at (x_0, y_0) during time $[t_i, t_j)$ and then lies at (x_1, y_1) during $[t_j, t_k)$ can be modeled by two line segments in the three-dimensional space, namely the lines: $\overline{[(x_0, y_0, t_i), (x_0, y_0, t_j))}$ and $\overline{[(x_1, y_1, t_j), (x_1, y_1, t_k))}$, which can be indexed using a three-dimensional R-tree.

This idea works fine if the end time of all such lines are known. For instance consider in the above example that the object moves from its initial position to the new one but is to remain there until some time not known *a priori*. All we know is that it lies in its new position until *now*, or *until changed*, no further knowledge can be assumed. The very problem of what *now* or *until changed* means is complex enough by itself (refer to [C+97] for a thorough discussion on the topic). To make things simpler we assume that *now* (or *until changed*) is a time point sufficiently far in the future, about which there is no further knowledge.

What matters to our discussion is that standard spatial access structures are not well suited to handle such type of "open" lines. In fact, one cannot avoid them. It is reasonable to assume that once the position of an spatial object is known, it is unknown when (and if) it is going to move. As such all current knowledge would yield such open lines, which would render known spatial access structures, e.g., R-trees, of little use. Recently, [B+98] investigated that problem and proposed appropriate extensions to R-trees.

One special case where one could overcome such an issue is when all movements are known *a priori*. This would cause only "closed" lines to be input, and thus the above problem would not exist. In the comparisons we make later in the paper using this structure, which we simply refer to as 3D R-tree, we shall make such an assumption. One feature that may favor such an approach is that any R-tree derivative could be used.

2.2 2+3 R-tree

One possible way to resolve the above issue is to use two R-trees, one for two-dimensional points, and another one for three-dimensional lines (hence the name 2+3 R-tree). A similar idea has been proposed in [KTF95] in the context of bitemporal databases. In that paper bitemporal ranges with open transaction time ranges were kept under one R-tree (called front R-tree) as a line segment. Whenever a open transaction time range were closed it would become a closed rectangle, which was to be indexed under another R-tree (called back R-tree), after removing the previously associated line segment from the front R-tree. In the 2+3 R-tree whenever the end time of an object's position is unknown it is indexed under a two-dimensional R-tree, keeping the start time of its position along with its id. Note that the original R-tree (or any of its derivatives) keep only the object's id (or a pointer to the actual data record) and its MBR in the leaf nodes. The two-dimensional R-tree used

in this approach is thus minimaly modified.

Once the end time of an "open" object's current state (i.e., position) is known, we are able to construct its three-dimensional line as explained above, insert it into the three-dimensional R-tree and delete the existing entry from the two-dimensional R-tree.

Using the example above: from time t_i until the time point immediately before[2] t_j the object is indexed under the two-dimensional R-tree. At time t_j, it moves, as such, (1) the point (x_0, y_0) is deleted from the two-dimensional R-tree, (2) the line $\overline{[(x_0, y_0, t_i), (x_0, y_0, t_j))}$ is input into the three-dimensional R-tree, and, finally, (3) the point (x_1, y_1) is input into the two-dimensional R-tree. Keep in mind that the start time of a point position is also part of the information held along with the remainder of its data.

It is important to note that now, both trees may need to be searched, depending on the time point with respect to which the queries are posed. Similar to the case of the 3D R-tree any of the proposed R-tree derivatives could be used, provided that the leaf nodes of the two-dimensional one is minimally modified as discussed above.

A final remark should be done. The 2+3 R-tree is the dynamic version of the 3D R-tree. That is to say that the two-dimensional R-tree serves the single purpose of holding the current (i.e., open) intervals. Should one know all movements *a priori* the two-dimensional R-tree would not be used at all, hence the 2+3 R-tree would be reduced to the 3D R-tree presented earlier.

2.3 HR-tree

The two approaches above have drawbacks. The first suffers from the fact that it cannot handle open-ended lines. The second, while able to overcome that problem, must search two distinct R-trees for a variety of queries. In this section we present the HR-tree [NS98], which is designed to index spatiotemporal data as classified earlier.

Consider again the example in Section 2.1. At time t_i one could obtain the current state (snapshot) of the indexed points, build and keep the corresponing two-dimensional R-tree, repeating this procedure for t_j and t_k. Obviously, it is not practical to keep the R-trees corresponding to all actual previous states of the underlying R-tree. On the other hand it is reasonable to expect that some (perhaps the vast majority) of the indexed points do not move at every timestamp. Consequently R-trees may have some (or many) nodes identical to the previous version. The HR-tree explores this, by keeping all previous states (snapshots) of the two-dimensional R-tree only *logically*.

As an illustration consider the two consecutive (with respect to their timestamps) R-trees in Figure 1, which can be represented in a more compact manner as shown in Figure 2. Though it is just a simple example, it is easy to see that much space could be saved if one could re-use the nodes that did not change from a given state to the next one. Note that with the addition of an array **A** one can easily access the R-tree he/she desires. Perhaps most important, is that once the root node of the desired R-tree for a given timestamp is obtained, query processing cost is the *same* as if all R-trees where kept physically.

Notice however that it is desirable to keep the number of newly created branches as low as possible. For that reason some R-tree variants are not suitable to serve as HR-

[2]We assume, without loss of generality, that the time domain is isomorphic to the rationals.

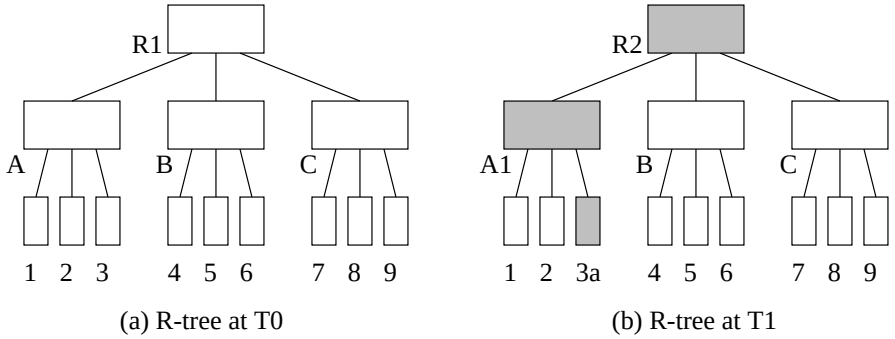

Figure 1: Maintaining, physically, all R-trees.

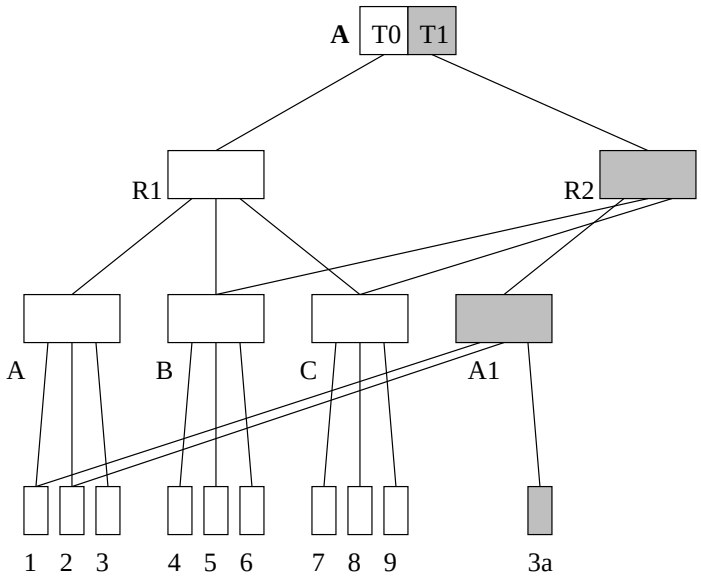

Figure 2: Physical view of the HR-tree logically equivalent to Figure 1.

tree's framework, notably, the R^+-tree [SRF87] and the R^*-tree [B^+90]. In the former the MBRs are "clipped" and one single MBR may appear in several internal nodes, therefore increasing the number of branches to be created in each incremental R-tree. Likewise, the R^*-tree, avoids node splitting by forcing entries re-insertion, which is likely to affect several branches, hence enlarging the HR-tree.

Among the other alternatives, we have found the Hilbert R-tree [KF94] to be very suitable for our purposes and use it as HR-tree's baseline. From now on, unless explicitly mentioned otherwise, we use the term R-tree(s) to refer to the Hilbert R-tree(s) as originally defined.

Since the original HR-tree proposal [NS98] used Guttman's (quadratic) R-tree to show its feasibility, its algorithms need be slightly modified (for the sake of completeness the HR-tree's algorithms are detailed in the Appendix.) The overall idea is that upon insertion of a new point version (i.e., its new location) a new branch has to be created with all of its

nodes are timestamped with the current timestamp, and then the new point is inserted in this branch's leaf node. All nodes which were not modified at all from last timestamp are simply re-used. For instance, Figure 2 representss the resulting HR-tree when a new data point is inserted and which were to fall within the leaf node 3 of the first R-tree in Figure 1. A similar idea is used to delete a point from the HR-tree. As commented above, querying the HR-tree is a simple matter of retrieving the appropriate logicall R-tree root through the array **A**.

3 Data Generation

From the seven issues presented in the introduction of the paper, it is quite clear that the first four are mostly related to the data sets while the last three are more related to spatiotemporal access structures.

With respect to the first four items, we characterize our data as follows:

- The data type being indexed is that of *points*.
- The supported temporal dimension is that of *transaction time*, the new position of any given point always has a begin-time greater than the begin-time of the previous position;
- The data points are *evolving*, i.e., the number of moving points are kept fixed and
- Only the *current state* (snapshot) can be updated.

The access structures in turn are categorized as:

- Able to index points or regions, the last ones would be abstracted by MBRs;
- No purging or vacuuming of data is assumed and
- Queries based on spatial, temporal or spatiotemporal predicates can be processed, although not all with the same efficieny in all structures.

For the data generation itself, we have build GSTD, a spatiotemporal data generator where the user can tune several parameters to obtain data sets which fulfills his/her needs. The two main parameters are the initial data distribution, and the amount of time a point is going to remain at the same location. For the initial data distribution we have experimented both the case of uniform and gaussian distributions.

All data is initially generated assuming a two-dimensional unit square. As time evolves the space is assumed to be toroidal, thus we do not have to worry about a point leaving the initial unit square. We also assume that all but one of the involved parameters (namely, distance covered and time interval between two successive states of an object) follow a gaussian distribution. The one exception is the direction of movement, in our study each point moves to any direction with the same probability. Recall that the distance it moves, though,

follows a gaussian distribution. Further details about the way we generated spatiotemporal data can be obtained in [TN98].

To illustrate how data is generated and how it evolves, Figure 3 shows an initial data set using the gaussian distribution, and four "snapshots" taken after 25, 50, 75 and 100 timestamps. It is easy to note that after 100 timestamps the initial gaussian distribution became closer a uniform one. This feature will allow us to investigate how dependent of the initial spatial data distribution the access structures are. Naturally, a sample of data points which were initially randomly distributed, remains randomly distributed as time evolves.

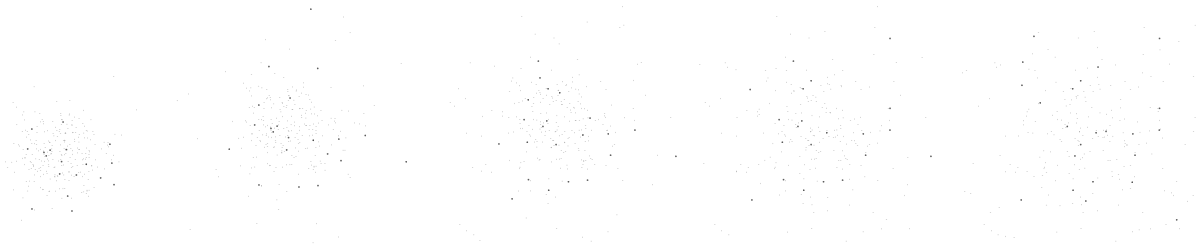

Figure 3: Temporal evolution of sample data points in the unit square.

4 Performance Analysis

As pointed out above our main concern is to investigate the performance yielded by the 3D R-tree, the 2+3 R-tree and the HR-tree when indexing moving points. Recall that in order to use the 3D R-tree one must know the whole history in advance. While some applications, such as digital battle field, may use *previously recorded snapshots*, their *online versions* always involve the *now* parameter. To overcome that problem, one may use the 2+3 R-tree approach discussed in Section 2.2. In any case, we decided to include it as a yardstick. Both the 3D and 2+3 R-trees are Hilbert R-trees in nature (recall that the HR-tree also uses the Hilbert R-tree as its basis).

Our experiments were performed on a Pentium II 300 Mhz PC running LINUX with 64 Mbytes of core memory. The disk pages, i.e., tree nodes, use 1,024 bytes and all programs were coded using GNU's GCC. The cardinality of the data set ranged between 25,000 and 100,000 points. All objects timestamps fell within the unit time interval [0, 1) with a granularity of 0.01, i.e., 100 distinct and equally spaced timestamps could be identified. Two initial distributions were investigated: the uniform and the gaussian one (see Figure 3). The queries are also uniformly and gaussianly distributed. The difference in the generation of the queries is that they are not points, but MBRs. Recall that we are interested in queries of the type "which are the points contained in a given region at (or during) a time point (interval)". For the case of containment queries with respect to a time point, the queries are those shown in Figure 4. The same two-dimensional queries are three-dimensionalized by "adding" a third temporal axis with length following a uniform distribution (to ensure some controlability when obtaining data results) to make containment queries with respect to a time interval. For each time point or interval we ran 100 queries and note the averages

obtained.

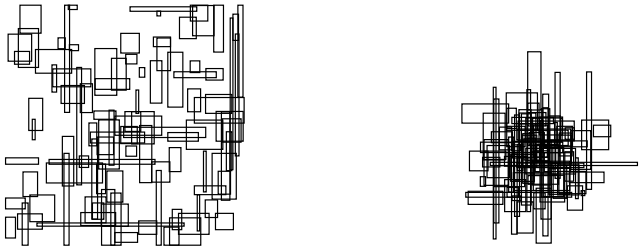

Figure 4: Query MBRs with centers randomly (left) and gaussianly (right) distributed.

Next we show the obtained results regarding storage requirements, and query processing cost (measured in terms of disk I/O).

4.1 Storage Requirements

Figures 5 and 6 show how large the structures are as a function of the number of indexed points. The reported sizes are the total number of disk pages consumed after 100 timestamps evolutions. We have not observed any unexpected correlation between page size and indices sizes.

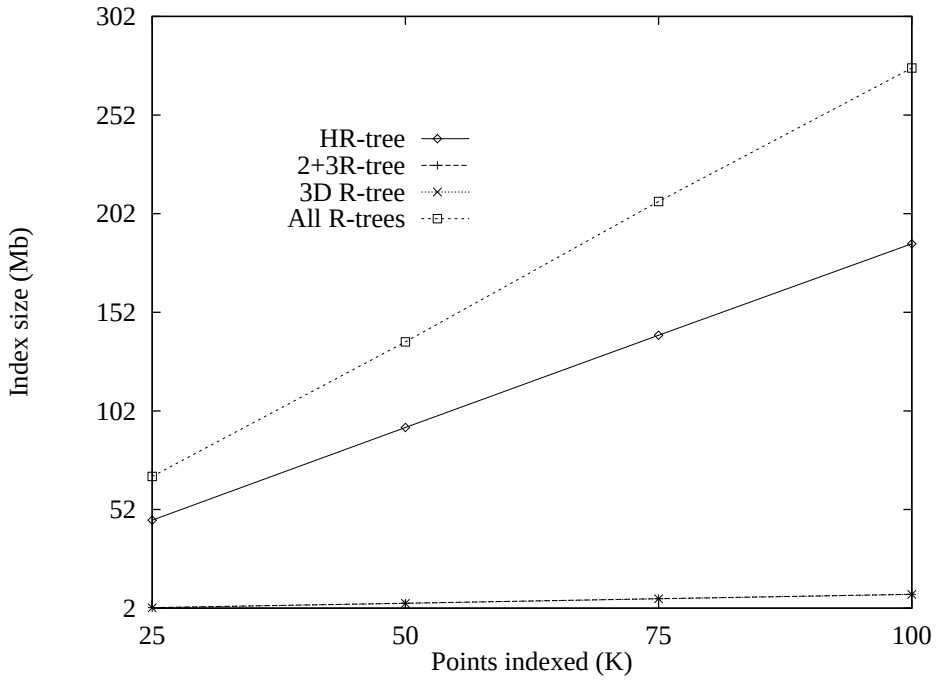

Figure 5: Indices' sizes – random initial distribution.

The obtained results show that, as expected, the HR-tree approach saves space when compared to the extreme case of physically storing all (100) R-trees. In fact, the average

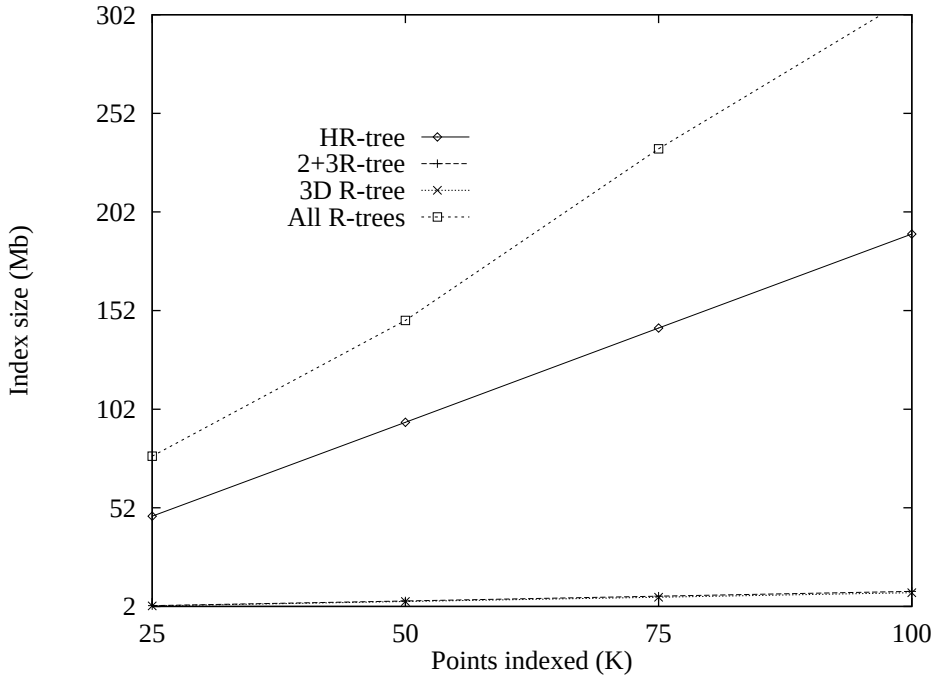

Figure 6: Indices' sizes – gaussian initial distribution.

savings amounted to about 33%. We also noted that the qualitative behavior of either structure does not seem to depend upon the initial distribution of the spatial data. Quantitatively, however, the gaussian distribution demands more storage space.

More interesting however, is the fact that the 3D R-tree and the 2+3 R-tree use roughly the same storage and are much smaller than the HR-tree. This is due to the fact that several internal nodes of the "virtual" R-trees are duplicated in the HR-tree. Indeed, the more dynamic the data points, the smaller the savings in the HR-tree with respect to storing all R-trees separately. However, as we shall see shortly, the HR-tree retains the same query processing cost as if all virtual R-trees were stored, which is considerably smaller than the query processing cost yielded by both 3D R-tree and 2+3 R-tree.

4.2 Index Building Cost

One interesting aspect to consider is the cost (in terms of disk I/Os) need to construct the indices. Figure 7 shows such an information for the data set initially distributed randomly. The curves shapes observed for the initial gaussian distribution was quite similar. As expected the 2+3 R-tree is the one which takes more time (i.e., I/Os) to be built. This is due to the fact that whenever a point moves, there is an insertion (of the new version) and a deletion (of the previous position) on the two-dimensional R-tree, and one insertion of a line (the previous position history) on the three dimensional R-tree. The HR-tree also requires more I/Os as at least one complete branch is updated from one timestamp to another one (assuming at least one point moved). The 3D R-tree on the other hand is the

most economical alternative, given that if all point movements are known *a priori*, only one three-dimensional line is inserted per point movement and no deletions occur.

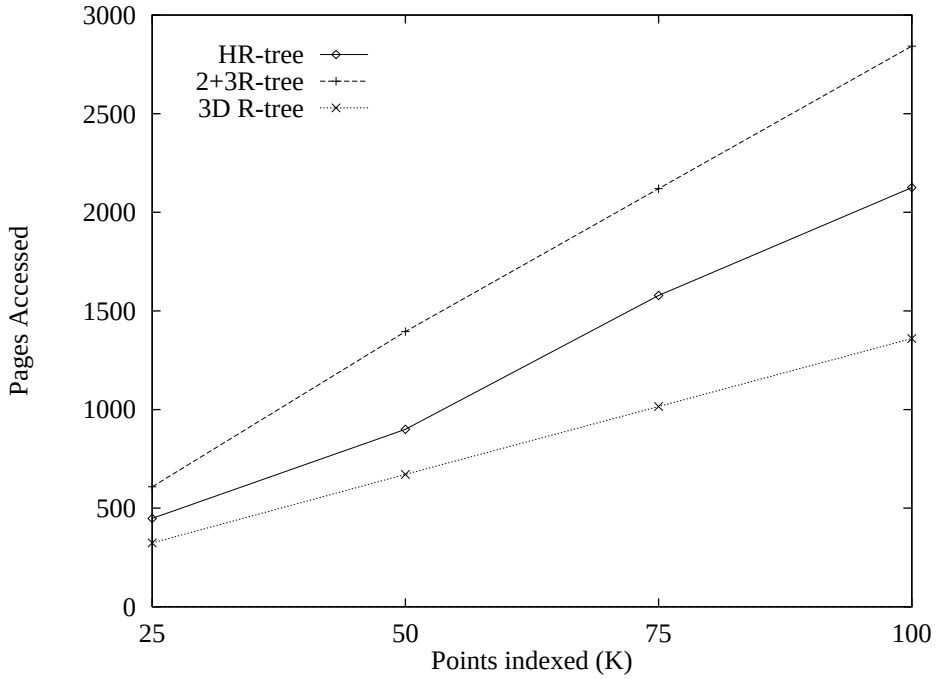

Figure 7: Indexing building cost, initial random distribution.

4.3 Query Processing Cost

There are two cases to consider, one when the query is posed with respect to a specific time point and another when a time interval is considered. We consider each one in turn. Starting with the case of a fixed reference time point query, Figures 8 through 11, illustrate the obtained results when the structures were indexing 50,000 and 100,000 data points. Although not shown in this paper, the obtained figures when indexing 25,000 and 75,000 data points just confirm the qualitative trend suggested by the Figures shown next.

The shapes of the obtained curves are very similar because the same sets of (randomly or gaussianly distributed – Figure 4) queries were used for all the corresponding data sets. Some conclusions we reached for this case are the following.

Our conjecture that the HR-tree would require much less query processing cost was shown to be true in all cases. The HR-tree is well suited to address spatiotemporal queries with the temporal part of the query window being a point. In such a case the tree that corresponds to that timestamp (e.g., the R-tree pointed by **A**[T1] in Figure 2) is obtained and the query processing is exactly the same one of a range query in a single standard R-tree. On the other hand, for both the 3D and 2+3 R-tree the whole structure is involved in the query processing, thus increasing significantly the query processing cost.

The three structures show an interesting behavior on their performance as time passes.

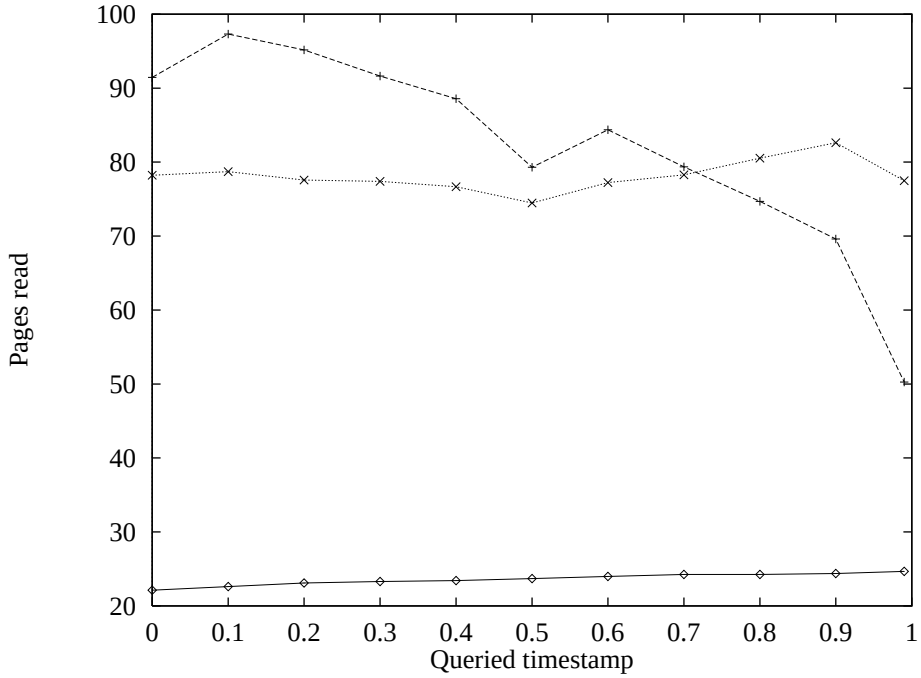

Figure 8: Query processing cost, indexing 50k points, initial random distribution.

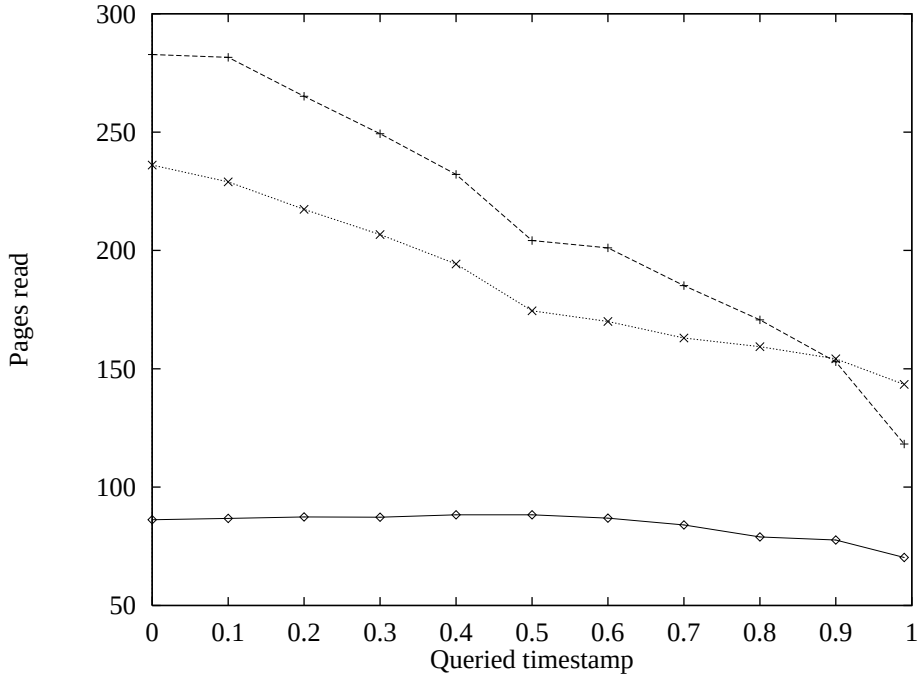

Figure 9: Query processing cost, indexing 50k points, initial gaussian distribution.

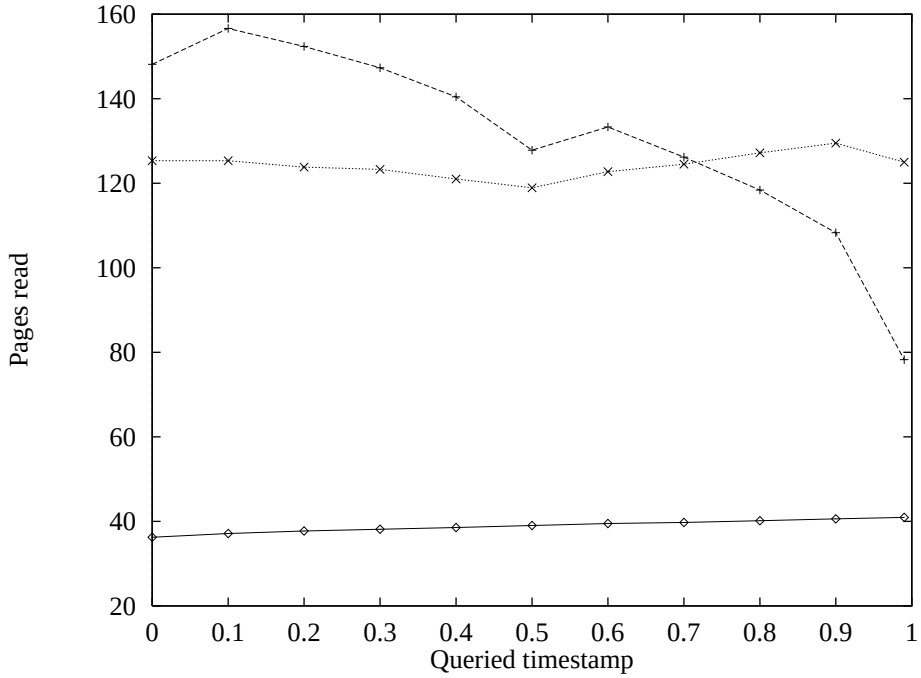

Figure 10: Query processing cost, indexing 100k points, initial random distribution.

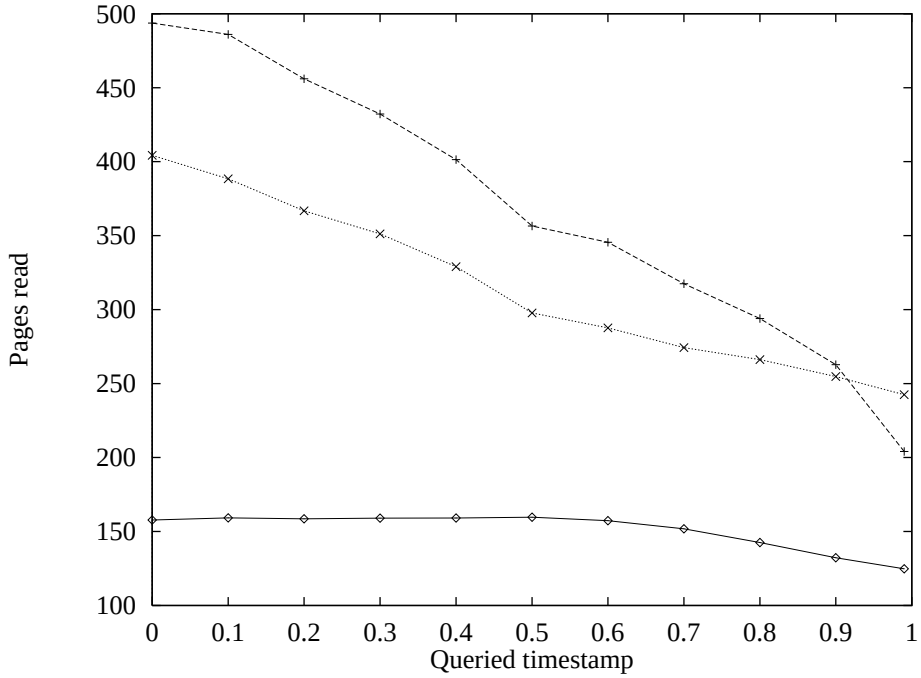

Figure 11: Query processing cost, indexing 100k points, initial gaussian distribution.

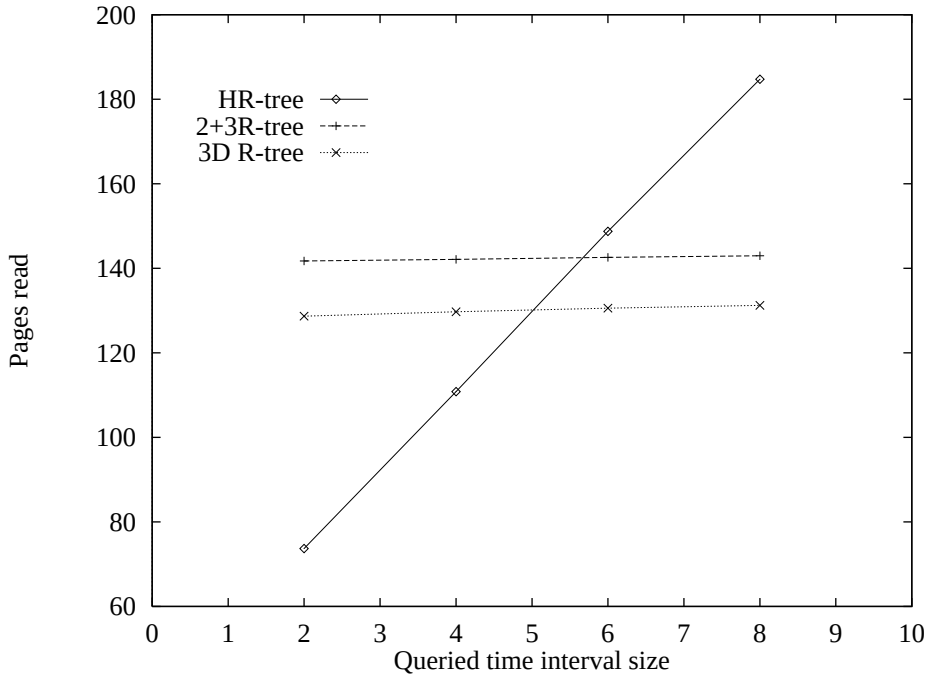

Figure 12: Query processing cost, indexing 100k points, initial random distribution.

The query processing cost of the 3D R-tree does not change considerably when the data is initially distributed in a random fashion, unlike the case when it initially follows a gaussian distribution. The reason for this observation resides in the fact that as time evolves the (focused) gaussian distribution becomes more "uniform" as already illustrated in Figure 3. As such the degree of overlap between the R-tree nodes decreases and hence the query processing cost. A similar behavior stands for the 2+3 R-tree although with a slight modification. The dual nature of the structure causes the query processing cost to be lower as queries address more "recent" timestamps. In those cases the two-dimensional tree is much more involved than its three-dimensional partner, which is more expensive to search. On the other hand, the HR-tree does not behave as mentioned above, i.e., its query processing cost does not change with the growing "uniformity" of the data set. This means that the HR-tree performance mainly depends on the initial data distribution rather than its change over time. This is not an unexpected result given that the HR-tree re-uses as much as possible from the previous (logical) R-tree, hence also inheriting its performance characteristics. Next we investigate the case where the queries are posed with respect to a time interval, instead of a point as done above.

Figures 12 and 13 illustrate the query processing cost of the structures indexing 100,000 points, again following initial random and gaussian distributions respectively when varying the average length of the queried time interval. Such a length is measured in terms of number of consecutive timestamps. That is to say that we measured the structure's performance when the time was, in average, up to 8 timestamps, or up 8% of the total (unit) time range. Note that a time interval of size 1 reduces to querying a specific time point, which

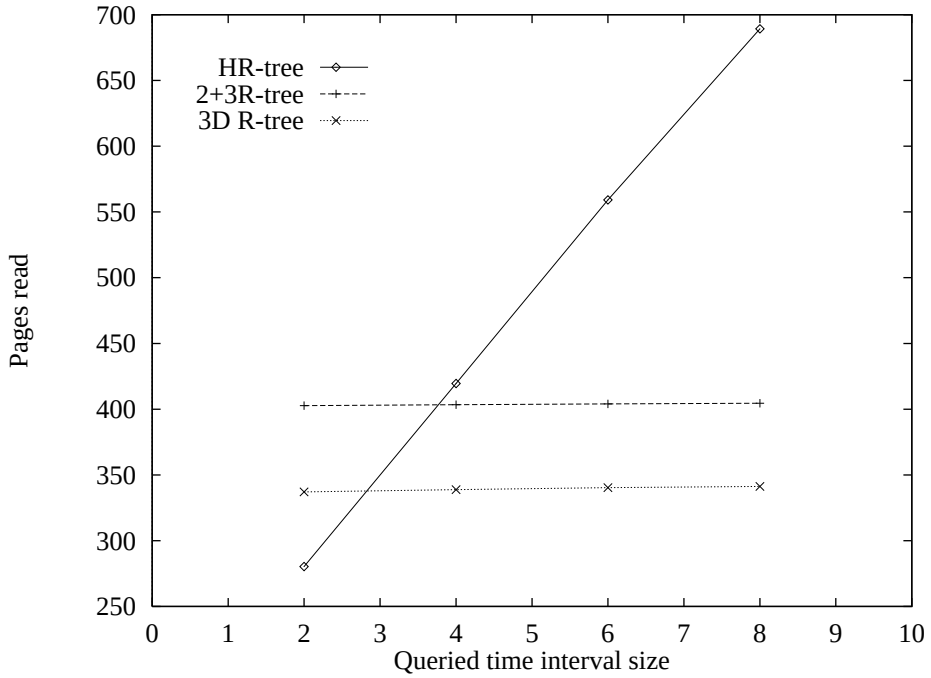

Figure 13: Query processing cost, indexing 100k points, initial gaussian distribution.

we investigated above. Large intervals were not investigated as the curves shown already presented a clear trend. The results observed are very interesting. When querying a time interval in the HR-tree one is forced to traverse several logical R-tree as if all of them were individually stored. In other words, the HR-tree is well suited to search a time point while for time intervals is has to traverse as many R-trees as many time points are covered by that interval. Again we verified that the relative HR-tree advantage is larger when indexing points which were spatially distributed randomly. Note however, that this advantage holds true only for very short time intervals, typically spanning an average of 1 to 5 (3) timestamps when indexing the random (gaussian) data set.

We believe that this shortcoming could be minimized with the use of a carefully designed cache policy. We plan to investigate such an issue in our future work.

5 Summary and Future Research

In this paper we raised the issue that despite the fact that applications dealing with spatiotemporal data are gaining strength, not much has been done regarding implementation and/or extensions of appropriate database management systems. Towards that goal our contribution was to propose and investigate access structures for spatiotemporal data.

To the best of our knowledge it is the first performance study for spatiotemporal access structures, unlike the areas of temporal [ST97] and spatial [GG98] indexing methods where an extended experimentation and comparison is found in the literature. In the particular

field that we discuss in this paper, it was only [TVS96] that compared alternative schemes based on R-trees with respect to analytical cost models proposed in [TS96].

We investigated three R-tree based structures: the 3D R-tree, the 2+3 R-tree and the HR-tree. We have discovered, after several experiments that:

- The less dynamic the data set, the higher the storage savings yielded by the HR-tree when compared to the ideal (from the query processing perspective) but impractical (in terms of disk storage demand) solution of having all logical R-trees physically stored;
- The 3D and 2+3 R-trees tend to be much smaller than the HR-tree;
- Due to the re-use factor, the HR-tree's performance is more dependent on the initial spatial distribution of the data set rather than its change over time;
- When querying a specific time point the HR-tree offers a much better query processing time than the 3D and 2+3 R-trees. In fact, it offers the same performance as if all logical R-trees were physically stored;
- 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.

Considering that with current technology storage is much less of a problem than time to obtain data, we consider the HR-tree a good candidate access structure for spatiotemporal data when most queries are posed with respect to a time point or a very short time range.

The present paper has not dealt specifically with how the above structures behave with respect to movement, direction and speed. For instance, suppose that instead of spreading in the space, all points move coordinately, how would each access structure support this ? Also a few points may move much faster than the others (or vice-versa), is that a feature that will affect the structures' performance ? These and several other questions are currently being investigated. An issue which needs further research has to do with the abstraction of the data set. In this paper we used data points, but it is easy to foresee that several application domains may require the management of spatial regions. For the particular case of MBRs the structures proposed in this paper could be used. We conjecture that their strengths and limitations would remain, but further research is needed to confirm such belief.

As a further step, each structure's performance is planned to be analytically explored, in correspondence with the R-tree analysis for selection and join queries that appears in [TSS98]. Both directions, extensive experimentation and analytical work, converge on building a spatiotemporal benchmarking environment consisting of real and synthetic data sets and access structures for evaluation purposes.

Acknowledgements

Mario A. Nascimento was partially supported by CNPq (process number 300208/97-9 and PRONEX's project SAI) and is also with CNPTIA - Embrapa (mario@cnptia.embrapa.br).

Jefferson R. O. Silva was supported by FAPESP (process number 97/11205-8). Yannis Theodoridis was supported by the EC funded TMR project "CHOROCHRONOS: A Research Network for Spatiotemporal Database Systems", contract number ERBFMRX-CT96-0056. We also thank Timos Sellis for fruitful discussions and comments on earlier drafts of the paper.

Appendix

For the sake of completeness, this section shows the algorithms needed to insert a data point at a particular time t_k, this will create a new branch in the HR-tree; the counterpart to remove a data point is also shown. We assume the reader is familiar with the Hilbert R-tree [KF94] algorithms and re-use them as much as possible. It is important to stress that we assume a transaction timestamp, that is, if the current timestamp reflect the state of the R-tree at time t_i, all updates must bear a timestamp greater than t_i. Moreover, we assume that updates can be made in batch, i.e., more than one update can have a same timestamp.

Table 1: Notation used in the algorithms.

`HR`	a pointer to the HR-tree
`A`	the HR-tree's array of time points
`R`	root node of a R-tree
`On`	MBR inserted in the HR-tree
`Oo`	MBR removed from the HR-tree
`F`	an entry in a R-tree node
`p`	pointer associated to `F`
`N.Nt`	node N's timestamp
`L,LP,N,NR`	pointers to R-tree nodes
`S`	a set of tree nodes
`SW`	a MBR representing a search window
`now`	current point in time
`pt`	most recent entry in `A`
`t`	an indexed (time) entry in `A`
`it,et`	start and end of a time range

The `AdjustTree` algorithm is the original Hilbert R-tree AdjustTree Algorithm [KF94]. The Hilbert R-tree HandleOverflow algorithm should be changed to be used here. When a node is touched by the algorithm and its timestamp is less than *now* (the current point in time), then it has to be duplicated. The notation used is presented in Table 1.

```
Algorithm HR-Insert(On, HR)
1. { create a new state in the HR-tree }
```

```
   if A[pt] < now
      then create a new entry in A indexing now;
2. { create a root NR to insert On }
   invoke CreateBranch(On, HR) to create a new logical R-tree rooted at NR.
   The new logical R-tree contains a leaf node L in which to place On;
3. { insert On in L }
   if L has room for another entry
      then insert On in L;
      else invoke HandleOverflow(L, On). A new leaf may be created if split
           was done.
4. { propagate split upwards }
   form a set S that contains L, its cooperating siblings and the new leaf
      node, if any.
   invoke AdjustTree(S).
5. { grow tree taller }
   if node split propagation caused a root split
      then create a new root NR whose children are the nodes resulting from
           the split;
   adjust the entry in A to point to the new root NR;

Algorithm HR-CreateBranch(On, HR)
1. { initialize }
   set N to be the root pointed to by A[pt] in HR;
   if N.Nt < now
      then create a new node L;
           copy all entries of N into L;
           set L.Nt = now;
           set NR = L;
      else set NR = N;
           set L = N;
2. { leaf check }
   if N is a leaf
      then return NR and L;
3. { choose subtree }
   let F be the entry with the minimum hilbert value grater than the hilbert
       value of On
4. { create a new node of the new branch and
     descending the tree }
   set LP = L;
   set N to be the node pointed to by F.p;
   if N.Nt < now
      then create a new node L;
           copy all entries of N into L;
           set L.Nt = now;
```

```
            adjust the pointer F.p in LP to point to L;
       else set L = N;
5. { Loop until a leaf is reached }
   repeat from step 2;
```

```
Algorithm HR-Delete(Oo, HR)
1. { find the leaf node containing Oo}
   set R to be the root pointed to by A[pt];
   perform an exact match search to find the leaf node L that contain Oo.
   duplicate the branch that contains L if necessary.
   set the timestamp of the nodes duplicated as Now;
   if L cannot be found
      then stop;
2. {create a new state in the HR-tree if necessary }
  if the root was duplicated
      create a new entry in A with time value equal to now;
      Remove Oo from L;
4. if L underflows
      then borrow some entries from s cooperating siblings.
           duplicate the siblings if their timestamp < now
   if all the siblings are ready to underflow
      then merge s+1 to s nodes.
           duplicate them if their timestamps < now
   adjust the resulting nodes.
5. { adjust tree }
   form a set S that contains L and its cooperating siblings (if underflow
      has occurred).
   invoke AdjustTree(S).
6. { shorten tree }
   if the root node has only one child after the tree has been adjusted
      then make the child the new root R;
   adjust A[now] to point to R;
```

Using the algorithms above it is easy to implement the movement of a given point from its current location to a new one at a given timestamp. It suffices to remove the point from its current location and the insert it in its new position. The corresponding algorithm follows:

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

As for queries we are currently interested in two types of containment queries: those that return all points which are contained within a given MBR with respect to (1) one

specific point in time, and (2) a time interval (e.g., $[t_i, t_j]$). The algorithms for them, in the above order, are presented next:

```
Algorithm HR-SearchPoint(S, t, HR)
1. { find the appropriate root R }
   if t = now
      then set R to be the node pointed to by A[pt];
      else set R to be the node pointed to by A[t];
2. { find the MBRs which overlap S }
   invoke original Hilbert Search algorithm
     passing R;
```

```
Algorithm HR-SearchRange(S, it, et, HR)
1. { find the appropriate root R at time it (initial time) }
   if it = now
      then set R to be the node pointed to by A[pt];
      else set R to be the node pointed to by A[it];
2. { find the MBRs which overlap S }
   set t to be it
   while (t <= et)
     invoke original Hilbert Search algorithm passing R;
     set t to be the next time in A[.];
     set R to be the node pointed to by A[t];
```

References

[B+90] N. Beckmann et al. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of the 1990 ACM SIGMOD Intl. Conf. on Management of Data*, pages 322–331, June 1990.

[B+98] R. Bliujute et al. R-tree based indexing of now-relative bitemporal data. Technical Report 25, TimeCenter, 1998. To appear in Proc. of the 24th Very Large Databases Conf.

[C+97] J. Clifford et al. On the semantics of "NOW" in temporal databases. *ACM Transaction on Database Systems*, 22(2):171–214, June 1997.

[Cor98] Informix Corp. Developing datablade modules for informix dynamic server with universal data option. White paper, 1998.

[GG98] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–57, June 1984.

[KF94] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. of the 20th Very Large Databases Conf.*, pages 500–509, September 1994.

[KTF95] A. Kumar, V.J. Tsotras, and C. Faloutsos. Access methods for bi-temporal databases. In *Proc. of the International Workshop on Temporal Databases*, pages 235–254, September 1995.

[MK90] Y. Manolopoulos and G. Kapetanakis. Overlapping B^+-trees for temporal data. In *Proc. of the 5th Jerusalem Conf. on Information Technology*, pages 491–498, August 1990.

[NS98] M.A. Nascimento and J.R.O. Silva. Towards historical R-trees. In *Proc. of the 1998 ACM Symposium on Applied Computing*, pages 235 – 240, February 1998.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Strucutures.* Addison-Wesley, Reading, USA, 1990.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multidimensional objects. In *Proc. of the 13th Very Large Databases Conf.*, pages 507–518, September 1987.

[ST97] B. Salzberg and V.J. Tsotras. A comparison of access methods for time evolving data. Technical Report 18, TimeCenter, 1997. To appear in ACM Computing Surveys.

[T+98] Y. Theodoridis et al. Specifications for efficient indexing in spatiotemporal databases. In *Proc. of the 10th IEEE Intl. Conf. on Scientific and Statistical Database Management*, pages 123 – 132, July 1998.

[TK96] V.J. Tsotras and A. Kumar. Temporal database bibliography update. *ACM SIGMOD Record*, 25(1):41–51, March 1996.

[TN98] Y. Theodoridis and M.A. Nascimento. On the generation of spatiotemporal data. Technical report, TimeCenter, 1998. In preparation.

[TS96] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 161 – 171, June 1996.

[TSS98] Y. Theodoridis, E. Stefanakis, and T. Sellis. Efficient cost models for spatial queries using R-trees. Technical Report 03, Chorochronos, 1998. To appear in *IEEE Transactions on Knowledge and Data Engineering.*

[TVM98] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees: a spatio-temporal access method. In *Proc. of the 6th ACM Intl. Workshop on Geographical Information Systems*, November 1998. To appear.

[TVS96] Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-temporal indexing for large multimedia applications. In *Proc. of the 3rd IEEE Conf. on Multimedia Computing and Systems*, pages 441 – 448, June 1996.

[W+98] O. Wolfson et al. Moving objects databases: Issues and solutions. In *Proc. of the 10th IEEE Intl. Conf. on Scientific and Statistical Database Management*, pages 111 – 122, July 1998.

[XHL90] X. Xu, J. Han, and W. Lu. RT-tree: An improved R-tree index structure for spatiotemporal databases. In *Proc. of the 4th Intl. Symposium on Spatial Data Handling*, pages 1040 – 1049, 1990.

[Z+97] C. Zaniolo et al., editors. *Advanced Databases Systems*. Morgan Kauffman, San Francisco, USA, 1997.