Evaluation of Access Structures for Discretely Moving Points

Mario A. Nascimento[1], Jefferson R. O. Silva[2], and Yannis Theodoridis[3]

[1] Institute of Computing, State Univ. of Campinas, Brazil
nascimento@computer.org
[2] Telecommunications R&D Center, Brazil
jeff@cpqd.com.br
[3] Computer Technology Institute, Greece
yannis.theodoridis@cti.gr

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 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). We explore several parameters, e.g., initial spatial distribution, spatial query area and temporal query length. We found out that the HR-tree usually outperforms the other candidates, in terms of query processing cost, specially when querying time points and small time intervals. However, the main side effect of the HR-tree is its storage requirement, which is much larger than that of the other approaches. To reduce that, we explore a batch oriented updating approach, at the cost of some overhead during query processing time. To our knowledge, this study constitutes the first extensive, though not exhaustive, experimental comparison of access structures for spatiotemporal data.

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 [19]. 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 [5]. As 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. Indeed, a "database application must capture the time-varying nature of the phenomena they model" [26, Ch. 5]. Spatial phenomena are no exception to this observation. Therefore, spatiotemporal databases should florish

as current technology makes it more feasible to obtain and manage such type of data ([4] is a good example of that trend).

Among the many research issues related to spatiotemporal data, e.g., query languages and management of uncertainty [24], we focus on the issue of optimizing access structures, hence speeding up query processing. Despite the fact that there is much work done on surveying and evaluating access structures for temporal [15] and spatial data [6], not much has been done regarding spatiotemporal data. This paper deals with this very point.

In particular, we focus on benchmarking access structures that maintain the whole "history" of each moving object and which are able to answer queries of the type "which objects were located within a specific area at a specific time instance (or during a specific time interval)". This class of access structures currently includes a very limited number of proposals; to our knowledge, there exist only five, namely MR-trees and RT-trees [25], 3D R-trees [22], and more recently, HR-trees [13] and Overlapping Linear Quadtrees [23], based on the popular R-trees [7, 17, 1, 8] and Quadtrees [16].

There also exist a small number of proposals aiming at supporting queries that deal with the future, i.e., "which objects will (certainly or probably) be located within a specific area after (or within) a certain time". This class of access structures usually store current location and some extra information (such as speed and direction) to make safe predictions for the future locations of objects [9, 24].

Although we admit that both applications are of equal importance, in this paper we only consider the former class. Indeed, in many real life examples, it is discrete locations rather than continuous motion that is detected by, e.g., a global positioning system (GPS) and stored in a database. Thus point trajectories are actually "simulated" either by storing discrete locations and valid time or by linearly interpolating consecutive points (the so-called projection versus interpolation-based approaches in [20]) and this simulation could be enriched by adding tolerance due to imprecision [14]. In the rest of the paper, we evaluate the first approach, i.e., discuss indexing of discretely moving points.

In [19], a set of seven criteria were proposed to characterize spatiotemporal data and access structures in the class of interest:

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 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 - Minimum Bounding Rectangles) is used to represent the spatial objects.
6. *Temporal treatment*: whether it supports special actions such as packing or purging (vacuuming) spatial data as time evolves.

7. *Query support*: whether it is able to process not only pure 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 in a *discrete* manner within the unit square;
- Updates are allowed only in the current state of the database;
- The timestamp of each point's version grows monotonically following a transaction time pattern, and
- The cardinality of the data set remains fixed as time evolves.

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 reference MBR, all points lying inside such MBR should be retrieved.

Hence, according to the terminology in [19], this paper considers databases of the point/transaction-time/evolving/chronological class. For simplicity, throughout the paper we assume the two-dimensional space, although extending the presented arguments for higher dimension is not problematic.

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 used to generate spatiotemporal data is discussed. Section 4 presents and discusses the experiments we perform regarding space requirements, update and query performance. Finally, the paper is closed with a summary of our findings and directions for future research.

2 Spatiotemporal Access Structures

As mentioned earlier, we are aware of only five access structures that consider both spatial and temporal attributes of objects, namely MR-trees and RT-trees [25], 3D R-trees [22], HR-trees [13] and Overlapping Linear Quadtrees [23].

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 [25].

The 3D R-trees, as originally proposed in [22], 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 beforehand. 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 an one-dimensional R-tree indexes the temporal component. Query processing is performed using both trees and performing the necessary 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 [11]. The basic idea is that, given two trees where the younger one is an evolution 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. The Overlapping Linear Quadtrees, as the name implies are based on Quadtrees [16] and as such are not constrained to index only MBRs. The MR-trees and the HR-trees are very similar in 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 [22]. 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_i, y_i) during time $[t_i, t_j)$ and at (x_j, y_j) during $[t_j, t_k)$ can be modeled by two line segments in the three-dimensional space, namely the lines: $\overline{[(x_i, y_i, t_i), (x_i, y_i, t_j))}$ and $\overline{[(x_j, y_j, t_j), (x_j, y_j, t_k))}$, which can be indexed using a three-dimensional R-tree.

This idea works fine if the end time of all such lines is 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 beforehand. 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 handling *now* or *until changed* is complex enough by itself (refer to [3] 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, [2] investigated that problem in the context of temporal databases and proposed appropriate extensions to R*-trees. Another approach was presented in [18] where a Quadtree based index is periodically reconstruted using the current information about the objects' location. The objects' motion equations are also maintained associated to the

index. Between index reconstructions, future objects positions are inferred using the indexed data and the motion equations.

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). The two-dimensional points would represent the current spatial information about the data points, whereas the three-dimensional lines would the represent (piecewise) the historical information. A similar idea has been proposed in [10] in the context of bitemporal databases. In that paper bitemporal ranges with open transaction time ranges are were kept under one R-tree (called front R-tree) as a line segment. Whenever an open transaction time range is closed it becomes a closed rectangle, which is 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, while 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 minimally modified.

Once the end time of an "open" object's current state (i.e., position) is known, we proceed with (a) constructing its three-dimensional line as explained above, (b) inserting it into the three-dimensional R-tree and (c) deleting the existing entry from the two-dimensional R-tree.

Using the example from the previous section: from time t_i until the time point immediately before[1] 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 both trees may need to be searched, depending on the time point with respect to which the queries are posed. 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 object movements

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

a priori the two-dimensional R-tree would not be used at all, hence the 2+3 R-tree reduces 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 review the HR-tree [13], 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 corresponding 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 change their positions 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 Figures 1(a) and (b), which can be represented in a more compact manner as shown in Figure 1(c). Note that with the addition of an array **A** one can easily access the R-tree he/she desires. In fact, 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.

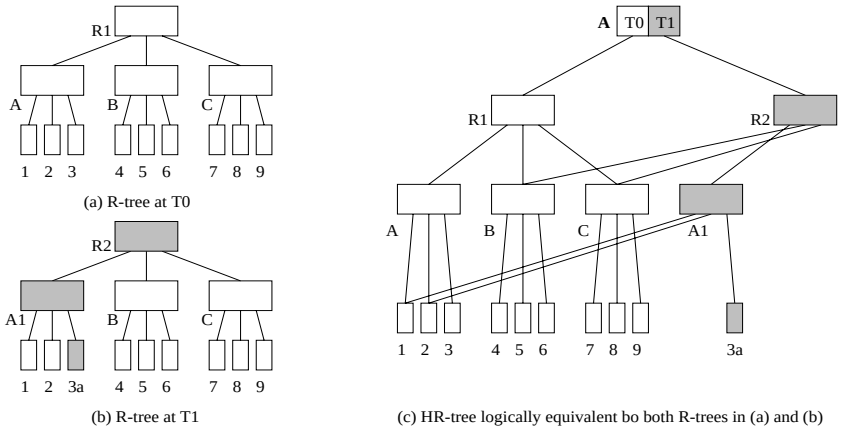


Fig. 1. Example of the HR-tree approach.

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-tree's framework, notably, the R^+-tree [17] and the R^*-tree [1]. 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 [8] 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. The interested reader can find all HR-tree's algorithms detailed in [12].

3 Data Generation

For the data generation itself, we have used GSTD[2] [20], a spatiotemporal data generator where the user can tune several parameters to obtain data sets which fulfill his/her needs. Some of them are: the initial data distribution, the amount of time a point is going to rest at the same location (*interval*), the distance it will move (*shift*) and where it is going to move to in the space (*direction*).

All data is initially generated assuming a two-dimensional unit square. Any point moving out of such a square has its coordinates adjusted to fit in the workspace. We also assume that the interval follows a gaussian distribution and the shift and direction follow the uniform one.

To illustrate how data is generated and how it evolves, Figure 2 (3) shows an initial data set using the gaussian (skewed) distribution, and four "snapshots" taken after 25, 50, 75 and 100 timestamps. It is easy to note that after 100 timestamps the initial distribution becomes very close to a uniform one. This feature will allow us to investigate how dependent of the initial spatial data distribution the access structures are. Naturally, data points which were initially randomly distributed, remain randomly distributed as time evolves.

Fig. 2. Evolution of gaussianly distributed data points.

[2] `http://www.dblab.ece.ntua.gr/~theodor/GSTD` (with a mirror site available at `http://www.dcc.unicamp.br/~mario/GSTD`).

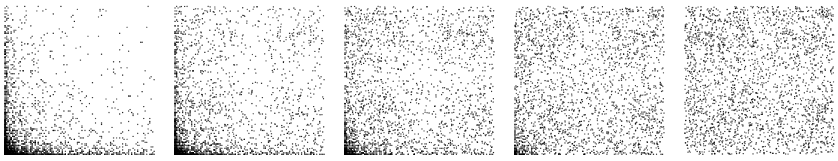


Fig. 3. Evolution of skewedly distributed data points.

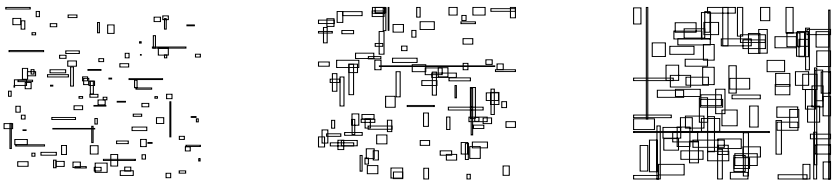
4 Performance Comparison

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*, others involve the *now* parameter. To overcome that problem, one may use the 2+3 R-tree approach as discussed in Section 2.2. In any case, we decided to include the 3D R-tree as a yardstick. Both the 3D and 2+3 R-trees are Hilbert R-trees in nature.

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 C++ compiler. No caching is explored for the investigated structures. Although it is certain that caching would improve the structures' performance we believe that different cache policies would be better suited for each of the structures. Therefore, instead of taking the risk of using a policy which could benefit of the structures in particular, we decided to leave this issue to be investigated thoroughly in the future.

Although this paper presents quantitative results using only a data set cardinality of 100,000 data points, we have also experimented with other sizes of data sets (25K, 50K and 75K) and reached to similar conclusions [12]. 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. Three initial spatial distributions were investigated: uniform, gaussian and skewed. The GSTD parameters were set in such a way that the points could move randomly in the workspace and therefore the final distributions tend to the uniform one (see Figures 2 and 3). The queries were uniformly distributed, and three different area sizes were used, denoted respectively as small, medium and large MBRs (Figure 4).

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)". The same set of two-dimensional queries are three-dimensionalized by "adding" a third temporal axis to make containment queries with respect to a time interval. For each time point or interval we ran 100 queries and report the averages. Next we show the obtained results regarding storage requirements, index building cost and query processing cost (measured in terms of disk I/O). Query processing



(a) Small MBRs (b) Medium MBRs (c) Large MBRs

Fig. 4. Query MBRs with centers uniformly distributed.

cost was measured when the database was in a static state, i.e., no updates were processed concurrently to the queries.

4.1 Storage Requirements

Figure 5 shows how large each of the structures are after indexing the initial 100,000 data points, and all their evolutions, i.e., after 100 timestamps, as a function initial spatial distribution. It is clear that the qualitative behavior of either structure does not seem to depend upon the initial distribution of the spatial data.

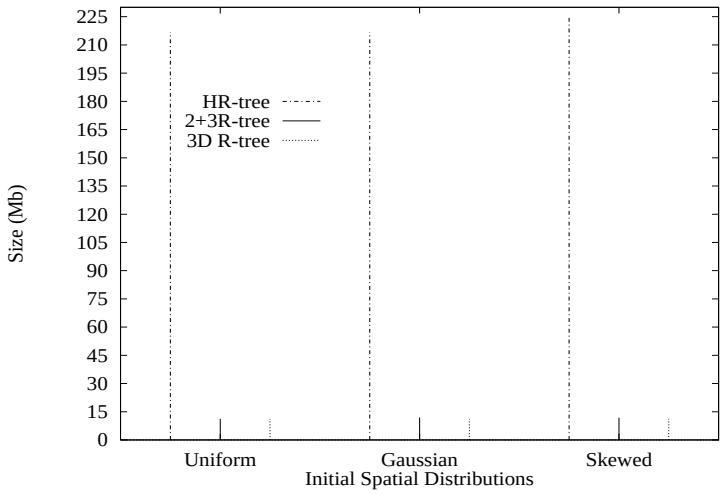


Fig. 5. Indices' sizes.

One result not shown in the figure is that, as expected, the HR-tree approach does save space when compared to the extreme case of physically storing all (100) R-trees. In fact, the average savings amounted to over 33%. On the other hand, the HR-tree is an order of magnitude larger than the 2+3 and 3D R-tree, which

both used roughly the same space. This is due to the fact that several branches of the "virtual" R-trees are duplicated in the HR-tree, as the indexed points are highly dynamic.

In order to alliviate this shortcoming we experimented the following approach. Instead of indexing all movements at the very timestamp when they happen, one could use a batch-oriented approach. The idea is to collect all movements into a buffer and from time to time flush out all of them into the same virtual R-tree. For instance, consider points locations which would originally be indexed under timestamps t, $t+1$ and $t+2$. Using such an approach all such points could be indexed at time $t+2$. The tradeoff in doing this is that one may not be able to query only the time point of interest. In the example just presented querying time point $t+1$ would require retrieving the (virtual) R-tree at time $t+2$, and consequently filter the answer properly. In other words, such batching may yield false-hits. We investigate this trade-off in more details shortly.

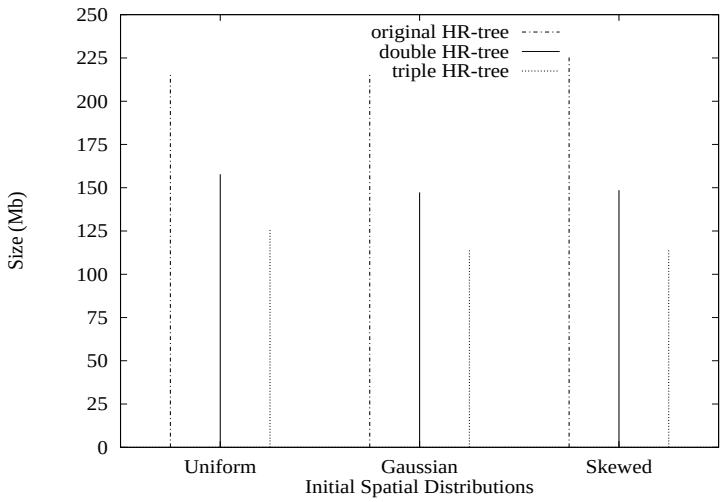


Fig. 6. HR-tree size using a batch oriented approach.

Figure 6 shows the obtained results when all movements were indexed at their exact timestamp (as in Figure 5), and at every two and three timestamps (which we refer to as double and triple approach). The gains were about 30% and 45% respectively. Note that even though the HR-tree became smaller it still remains larger than the other two structures. Fortunately, as we shall see shortly, the imposed overhead due to the false-hits is not likely to be as high the yielded gain (depending on the query size).

4.2 Index Building Cost

One interesting aspect to consider is the cost (in terms of disk I/Os) needed to construct the indices. Figure 7 shows such an information for all three initial distributions after indexing 100,000 data points and their evolutions. As expected the 2+3 R-tree is the one which takes more time (hence, 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 appears as the runner-up 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.

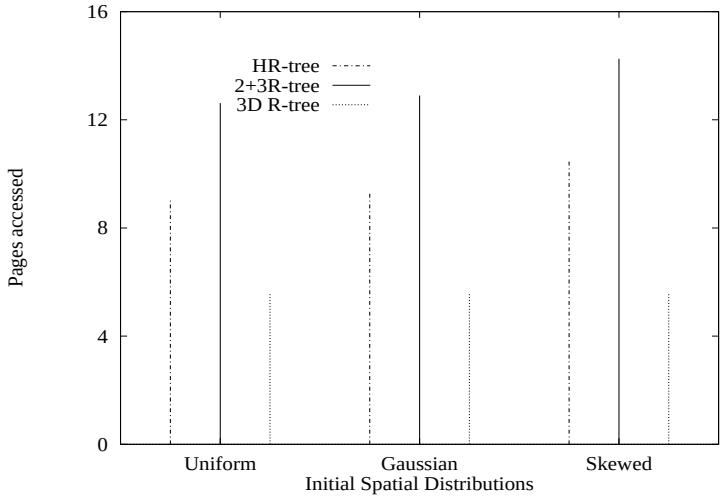


Fig. 7. Indexing building cost.

Unlike in the case for the indices sizes, the batching approach yields only some marginal improvements (when used for the HR-tree) regarding update cost.

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. The U, G and S labels in x-axis of the following figures denote the initial uniform, gaussian and skewed spatial distribution respectively. The results shown are the

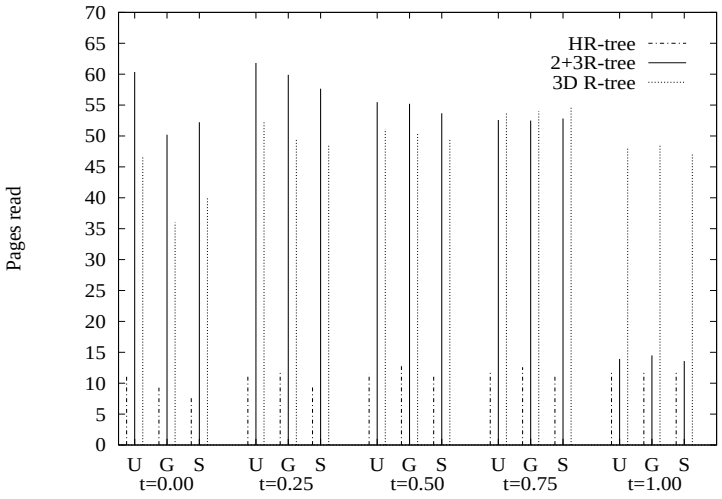


Fig. 8. Point query processing cost, small MBRs.

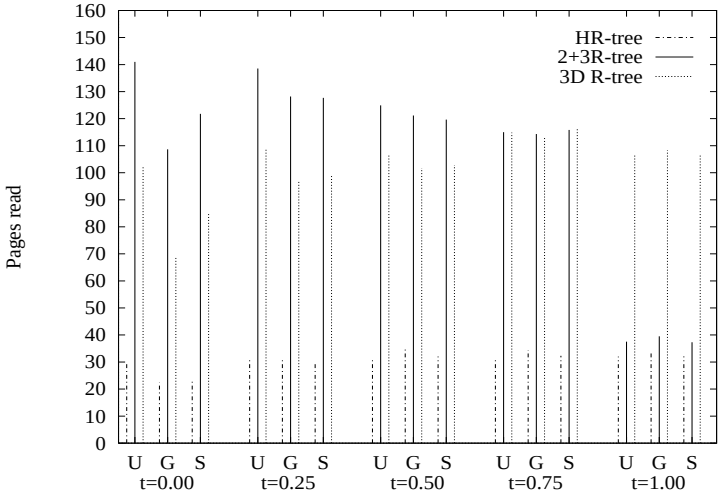


Fig. 9. Point query processing cost, large MBRs.

average number of I/Os obtained when querying time points 0, 0.25, 0.5, 0.75 and 1.

Figures 8 and 9 show the results obtained when querying each structure, at the timestamps described above, and using the small and large query MBRs respectively. The results from the medium MBRs are qualitatively the same, and quantitatively proportional to the query size (with respect the small and large MBRs), thus they are not shown for the sake of brevity.

Our conjecture that the HR-tree would require substantially smaller query processing time was shown to be true in all cases. When the temporal part of the query is a point, the tree that corresponds to that timestamp is obtained and the query processing is exactly the same one of a containment 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. In general, querying the HR-tree was always 3 to 4 times cheaper than querying the 3D R-tree.

One noteworthy result is obtained when querying the 2+3 R-tree at time 1, i.e., that end of the indexed temporal window. At that time point, the three-dimensional of the two R-trees is not traversed because there is no object with an end time equal to 1, thus all the answers come from the 2D R-tree – where there is always a current version for all indexed points. In fact, when querying time point 0.99, i.e., the indexed timestamp immediately before the last one, the 2+3 R-tree would require over 130% more I/Os than necessary when querying time point 1. Therefore the good performance obtained when querying that particular timestamp is by no means typical.

Note that as time evolves the 2+3 R-tree and the 3D R-tree exchange places as the runner-up structure. After some amount of updating, having all current and historical data in a single structure yields a large overlap ratio in the tree MBRs and thus a more complex tree traversal. Finally, it also became clear that the size of the query MBR affects all structures equally.

It is now necessary to measure the trade-off posed when one used the double or triple HR-tree approach as discussed earlier (we noted earlier that such an approach was able to reduce the HR-tree's size). Figure 10 shows the results when querying large query MBRs at time points 0, 0.25, 0.5, 0.75 and 1 (the results were qualitatively the same for small and medium sized queries). The difference in the curves for the original and the double and triple approach represents the number of pages which were read due to the batching, i.e., overhead pages. The average overhead was 17% for small queries, 20% for medium queries and 30% for large queries. It is important to note that despite such overhead the HR-tree still remains the overall faster index structure (compare Figure 9 to Figure 10).

Next we proceed to investigate how the structures perform when querying a time interval instead of a point. Figures 11, and 12 illustrate the query processing cost when varying the average length of the queried time interval. Such a length is measured in number of consecutive timestamps. That is, we measured the structure's performance when the time interval was up to 10% of the total (unit) time window length. Again the results using medium query MBRs,

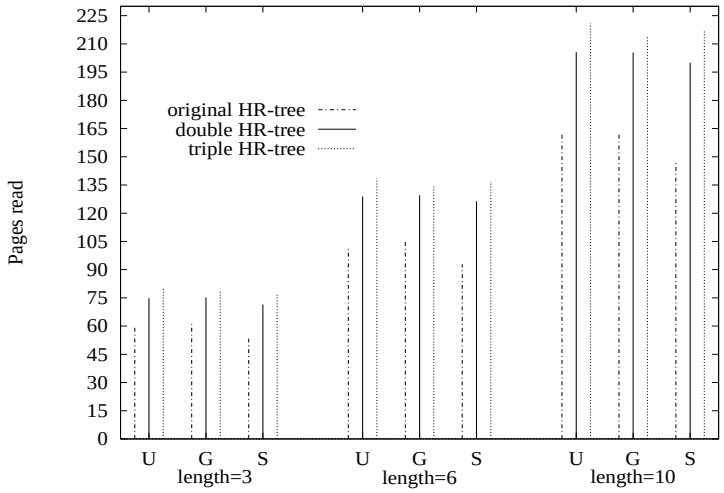


Fig. 10. Point query processing overhead using the batch oriented approach and large MBRs.

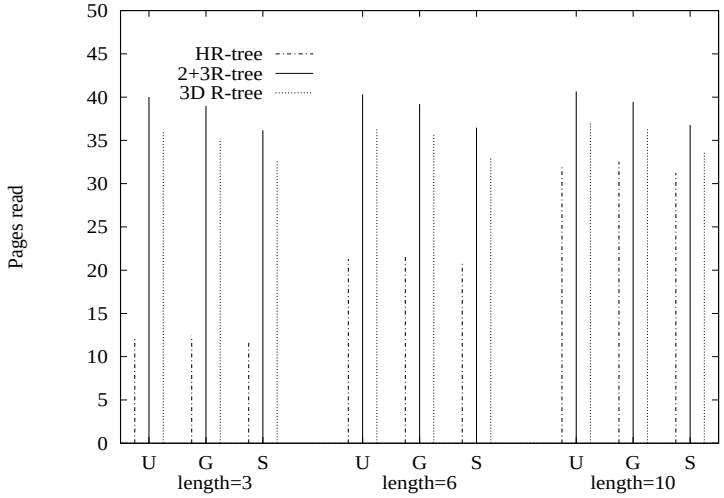


Fig. 11. Interval query processing cost, small MBRs.

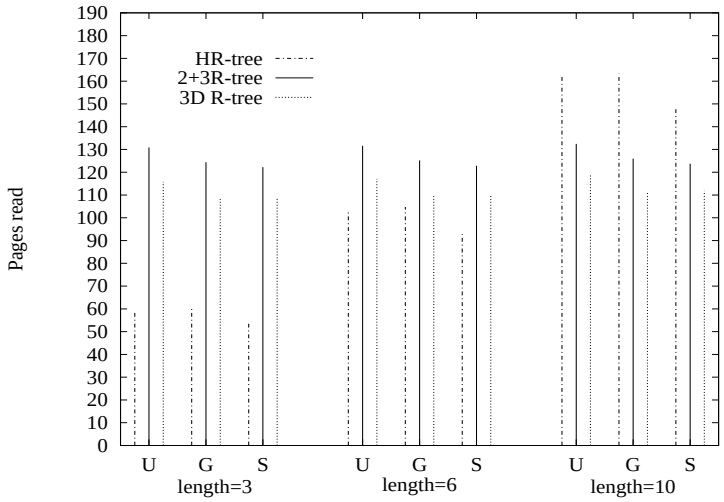


Fig. 12. Interval query processing cost, large MBRs.

were qualitatively the same. Larger intervals were not investigated as the curves shown already presented a clear trend.

While the HR-tree is well suited to search a time point, for time intervals it has to traverse as many logical R-trees as many indexed time points are covered by that interval. As a result the HR-tree loses its relative advantage relatively fast with the increase in the queried interval length. In fact, we observed this is worsened by the increase in the query MBR area.

Figure 13 shows that the overhead imposed by the batching approach is now harmful (comparing to the performance delivered by the other structures). That is, batching updates does not seem worthwhile for the case of querying time intervals.

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 present and investigate access structures for spatiotemporal data. To the best of our knowledge it is the first performance study for spatiotemporal access structures. We implemented three R-tree based structures: the 3D R-tree, the 2+3 R-tree and the HR-tree; and investigated several parameters that affect their performance:

- initial distribution of point data (uniform vs. gaussian vs. skewed)
- temporal extent of query windows (point vs. interval time queries)
- time snapshot of point time queries (from less to more recent)

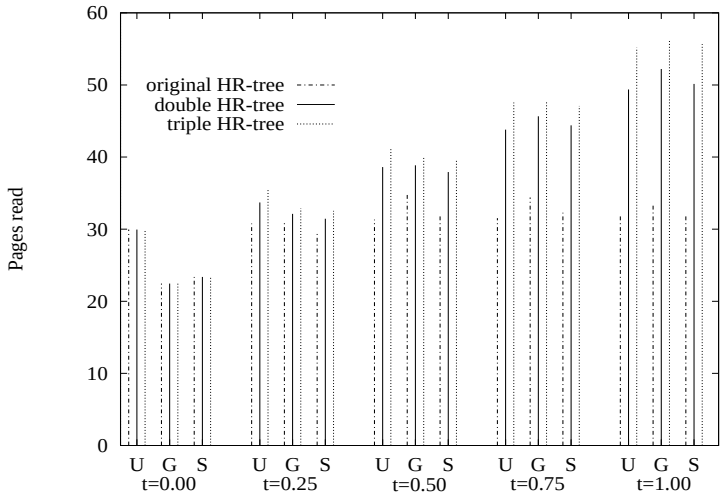


Fig. 13. Interval query processing overhead using the batch oriented approach and large MBRs.

– updating batch sizes (for the HR-tree)

After several experiments (more detailed experiments may be found in [12]), we have discovered 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;
– The use of a batching approach at update time is capable of reducing substantially the HR-tree's size, yielding some overhead at query processing 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. The overhead imposed by the batching approach is acceptable as the HR-tree remains the best performer;
– 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. In such a case the batching approach overhead for the HR-tree is hardly worthwhile.

Considering that with current technology storage is much less of a problem than time to query 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. We also have to note that, unlike HR-tree and

2+3 R-tree, the 3D R-tree is not capable of supporting on-line spatiotemporal applications that involve the *now* (or *until changed*) parameter.

The present paper has not dealt specifically with how the above structures behave with respect to movement direction and speed and the use of caching structures and policies. For instance, suppose that instead of spreading in the space, all points move coordinately towards the same direction, 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.

As future work, 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 [21]. 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 by the Pronex/FINEP funded project "SAI: Advanced Information Systems" (process number 76.97.1022.00), and was also with Embrapa. He is currently with the Univ. of Alberta, Canada. Jefferson R. O. Silva was supported by FAPESP (process number 97/11205-8). Yannis Theodoridis was partially supported by the EC funded TMR project "CHOROCHRONOS: A Research Network for Spatiotemporal Database Systems", contract number ERBFMRX-CT96-0056 while he was with NTU Athens. We thank Timos Sellis for fruitful discussions and comments on earlier drafts of the paper.

References

1. 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.
2. R. Bliujute et al. R-tree based indexing of now-relative bitemporal data. In *Proc. of the 24th Very Large Databases Conf.*, pages 345–356, August 1998.
3. J. Clifford et al. On the semantics of "NOW" in temporal databases. *ACM Transactions on Database Systems*, 22(2):171–214, 1997.
4. Informix Corp. Developing datablade modules for informix dynamic server with universal data option. White paper, 1998.
5. Egenhofer, M.J. and Golledge, R.G. (Eds.). *Spatial and Temporal Reasoning in Geographical Information Systems.* Oxford Univ. Press, New York, USA, 1998.
6. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):123–169, 1998.
7. 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.

8. 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.
9. G. Kollios, D. Gunopulos, and V.J. Tsotras. On indexing mobile objects. In *Proc. of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 261–272, May 1999.
10. 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.
11. 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.
12. M. A. Nascimento, J. R. O. Silva, and Y. Theodoridis. Access structures for moving points. Technical Report 33, TimeCenter, 1998.
13. 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.
14. D. Pfoser and C.S. Jensen. Capturing the uncertainty of moving-objects representation. In *Proc. of the 6th Intl. Symposium on Spatial Databases*, July 1999. To appear.
15. B. Salzberg and V.J. Tsotras. A comparison of access methods for time evolving data. Technical Report 18, TimeCenter, 1997. To appear at *ACM Computing Surveys*.
16. H. Samet. *The Design and Analysis of Spatial Data Strucutures*. Addison-Wesley, Reading, USA, 1990.
17. 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.
18. J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree based dynamic attribute indexing method. *Computer Journal*, 41(3):185–200, 1998.
19. 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.
20. Y. Theodoridis, J.R.O. Silva, and M.A. Nascimento. On the generation of spatiotemporal datasets. In *Proc. of the 6th Intl. Symposium on Spatial Databases*, July 1999. To appear.
21. Y. Theodoridis, E. Stefanakis, and T. Sellis. Efficient cost models for spatial queries using R-trees. Technical Report CH-98-5, Chorochronos, 1998. To appear at *IEEE Transactions on Knowledge and Data Engineering*.
22. 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.
23. 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*, pages 1–7, November 1998.
24. 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.
25. 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.
26. C. Zaniolo et al., editors. *Advanced Databases Systems*. Morgan Kauffman, San Francisco, USA, 1997.