An Incremental Batch-Oriented Index for Bitemporal Databases

Jefferson R. O. Silva
Telecom R&D Center (CPqD)
Brazil
jeff@cpqd.com.br

Mario A. Nascimento
Dept of Computing Science
Univ. of Alberta, Canada
mn@cs.ualberta.ca

Abstract

Bitemporal databases record not only the history of tuples in temporal tables, but also record the history of the databases themselves. We address the problem of indexing such bitemporal databases by investigating the use of an incremental indexing structure, the HR-tree, which was originally aimed at spatiotemporal databases. The HR-tree's most attractive feature is that it can process queries as if all previous database snapshots were indexed physically, however, all such states are indexed only logically. In our experiments we have found that the HR-tree is much more efficient (up to 80% faster) than previously proposed approaches based on two coordinated R-trees when processing queries based on a single transaction time point and valid time being either point or intervals. As for size, the HR-tree was found to be better suited for workloads where the number of updates per transaction timestamp is reasonably large (over one thousand updates in our studies), otherwise it is prone to require large storage space.

Keywords: *temporal databases, bitemporal indexing, access structures, R-trees.*

1. Introduction

Temporal databases have been the object of quite some research regarding many aspects and much has been published in the field [18]. It has been long recognized that two dimensions of time need be supported by a database management system (DBMS) in order to enhance the temporal modeling capabilities of a database. These two time dimensions are the *valid time* (the time range when the fact is true in the modeled world) and *transaction time* (the time range when the fact is logically stored in the database). A third dimension, user-defined time, is also needed for modeling purposes, but need not be supported by the DBMS. A *Bitemporal Database* (2TDB) is one which supports both valid time and transaction time [8]. In such a case one is allowed to ask queries based on different (past and possibly future) states of the database and/or tuples. Hence, 2TDBs allow corrections/predictions to tuples while also maintaining the history of the database itself.

Our goal in this paper is to investigate the efficiency of an index structure originally designed for spatiotemporal databases, the HR-tree [13, 14], when indexing 2TBDs. In spatiotemporal databases, one must index not only the spatial attributes of objects, which we refer to as *spacestamp*, but also their evolutions as time progresses. The current spacestamp is valid until it is changed. We also assume it is stored until the current point in time, denoted by *now*. As such, the transaction and valid time of the spacestamps are *now-relative* [5]. However, for the purpose of this paper, we assume that the valid-time of those spacestamps are *not now-relative*, i.e., their end time is known in advance.

As an example of a bitemporal scenario (with a spatiotemporal flavor) that fits the above requirements, i.e., now-relative transaction timestamps and not now-relative valid timestamps, consider satellite imagery. Each image about a certain area has a well-defined valid time, and each such image is stored until the next one is obtained. Note that even in the case when a new image is not obtained in due time, the valid time of the current one is likely to expire due to the very nature of moving objects (e.g., ships, planes, hurricanes) down in Earth. That is, every image has a pre-defined valid time, and a transaction time which is now-relative. In addition, older images are unlikely to be physically discarded, instead they should be only logically deleted by setting their transaction timestamps accordingly.

The rest of this paper is organized in the following manner. Section 2 reviews previously proposed approaches. Section 3 presents briefly the HR-tree structure. In Section 4 we discuss how we generated data sets for evaluating the HR-tree and also present the results obtained. Section 5 concludes the paper with a summary of the paper's findings.

2. Related Work

While a reasonable number of papers have been published on the issue of indexing either valid time or transaction time databases, only a handful have addressed the problem of indexing 2TDBs [15]. In what follows we review some of the approaches proposed recently, most based on R-trees [7]: the M-IVTT [12]; the BIT, BRT, and 2R-tree [10]; the GR-tree [5] and the 4R-tree [4].

The M-IVTT (Multiple Incremental Valid Time Trees) is a two level hierarchical index based on B^+-trees. In the upper level, one tree indexes the transaction time of events. Underneath this transaction time tree (TTT) there is a forest of Valid Time Trees (VTTs.) Each VTT is pointed to by an entry in the TTT and indexes the valid timestamps of all records existing at one point in (transaction) time. Due to potentially large demand for space, only some VTTs are kept full, along with sufficient information (patches) on how to reconstruct any of the other ones.

The BIT (Bitemporal Interval Tree), the BRT (Bitemporal R-tree) and the 2R-tree structures index closed (i.e., not now-relative) valid time ranges and now-relative transaction time. The BIT and BRT follows the partial-persistent methodology. In a partially persistent structure only the newest version of an object can be modified, whereas in an ephemeral structure old versions of an object are discarded when an update occurs. The authors modify the Interval Tree [6], which is an ephemeral memory based structure with good worst-case performance into the BIT, which is disk based, partially persistent and well paginated. The BRT makes an R^*-tree [2] a partially persistent following the approach of the MVB [1] and MVAS [19]. Like that structure, the BRT is a directed acyclic graph of pages. The structure is then formed by several logical R-trees, representing the evolution of objects in the transaction time sense.

The 2R-tree uses two R-trees (named *front* and *back* R-trees) to index bitemporal data. The bitemporal domain is mapped to a two-dimensional space (valid time $\times$ transaction time) as follows. An object with an unknown transaction end time is stored in the front R-tree as a line. Recall that in this approach valid time ranges are bounded and, naturally, the transaction start time is always known. Once this object is updated, it is removed from the front R-tree and inserted into the back R-tree as a rectangle. Figure 1(a) shows an example of this approach. The front R-tree indexes two objects, inserted at (transaction) time T and T' and which are still current in the database, i.e., bear an open transaction end time. The back R-tree, on the other hand, indexes two other objects which were current in the database during [T, T'] and [0, T'] respectively.

Note, however, the valid time interval can be transformed into a point in a three-dimensional space (valid start time $\times$ valid end time $\times$ transaction time). Likewise, the rectangles formerly in the back R-tree are now transformed in three-dimensional segments. The advantage of using such an approach is that the amount of overlap among the indexed objects is diminished, hence the underlying R-trees can offer a better performance. Figure 1(b) illustrate the three-dimensional case of Figure 1(a).

The GR-tree and 4R-tree index both now-relative valid and now-relative transaction time. The GR-tree extends the R^*-tree [2] to store both static tuples (with closed valid and transaction time ranges) and growing objects (with either valid or transaction end time unknown). The variables `UC` and `NOW` are introduced, meaning a open (or yet unknown) transaction end time and a open (or yet unknown) valid end time, respectively. In this new tree, the indexed objects in its nodes can be either a growing rectangle or a growing stair-shape object, in addition to the standard MBRs supported by the R^*-tree. The R^*-tree algorithms need to be modified to handle the new variables. By storing such growing objects, the dead space among objects in the GR-tree is decreased when compared to using the R^*-tree and hence it becomes much more efficient.

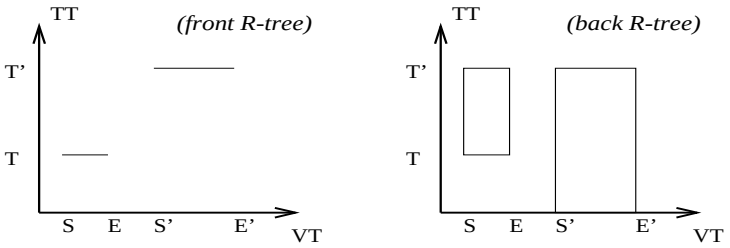
To further reduce dead space, the 4R-tree maps a growing rectangle into a closed line and a growing stair-shape object to a point. Using such a transformation the proposed approach is able to use "off-the-shelf" R-trees (which is the main goal of the proposal). Indeed, eliminating the UC and NOW variables introduced in the GR-tree, the objects are now indexed in four R^*-trees, depending on whether their valid and transaction end time are open or not. As objects are updated they may move between such R^*-trees like in the 2R-tree approach. In fact, in the case of bitemporal data with no now-relative valid time ranges the 4R-tree reduces to the 2R-tree.

Even though we have not discussed, in all approaches above incoming queries need be slightly modified. Further details can be found in the original papers.

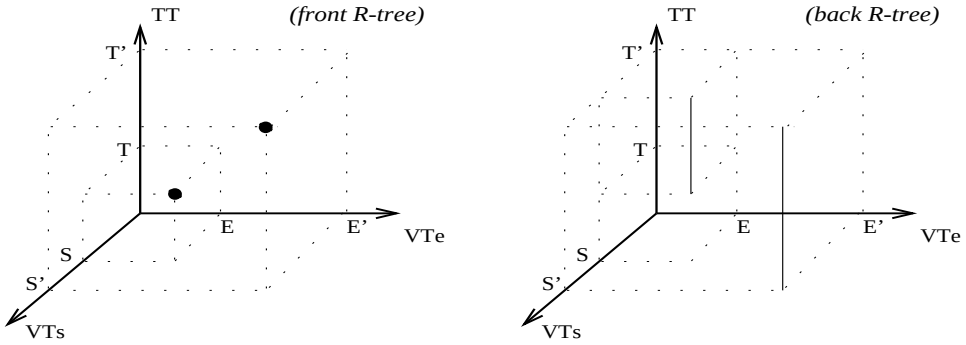
3. The HR-tree for 2TBDs

As most other proposed access structures for 2TDBs the HR-tree is also based on R-tree. The HR-tree was originally designed as a spatiotemporal indexing structure, as such, let us use a spatiotemporal scenario as a motivation. Consider an object O which lies at spacestamp S_0 during time $[t_0, t_1)$ and then lies at spacestamp S_1 during $[t_1, t_2)$ and so on and so forth. In a 2TDB perspective this is equivalent to say that object O had a given state valid during (closed) interval S_0 and known from t_0 until (exclusively) t_1, and another state valid during (closed) interval S_1 and known from time t_1 until (exclusively) t_2.

These characterize different states (snapshots) of the spatiotemporal database. One trivial way to index such states



(a) The interval based 2R approach (2Ri)



(b) The point based 2R approach (2Rp)

Figure 1. The two variants of the 2R-tree approach.

would be to build an R-tree for each of them. Although this is obviously not a practical solution, it is reasonable to assume that sibling R-trees may have some (potentially many) identical nodes. The HR-tree explores this by keeping all previous states of the two-dimensional R-tree *only logically* instead of physically. This is achieved by allowing consecutive instances of R-tree to overlap, i.e., to share nodes. This idea was also proposed in [11] but for B^+-trees in the context of (single dimension) temporal databases. As an illustration consider the two consecutive (with respect to their timestamps) R-trees in Figure 2(a) and (b), which can be represented in a much more compact manner as the HR-tree shown Figure 2(c). In this example all objects (at the leaf node level) but object 3 are current in the database as of time T1 and as such have their transaction end time open (i.e, equal to *now*). Object 3 on the other hand has its transaction time equal to [T0, T1), meaning that a query posed at transaction time T1 traverses the *logical* R-tree rooted at R2 and does not "see" object 3, as one would expect.

Although it is just a simple example, it is easy to see that much space could be saved by re-using the nodes that did not change from a given state to the next one. Note that with the addition of a simple structure **A** (an array in the figure, but which could be a B^+-tree if warranted) the root node of the desired R-tree, current for a given timestamp, can be obtained quickly, and thus the query processing cost is essentially the same as if all R-trees where kept physically. This becomes useful in the case of transaction time point queries. However, should one query a transaction time range, then several logical R-trees would need to be searched, which could be costly. Details about the HR-tree structure as well as its companion algorithms can be found in [13, 14].

It is rather simple to use the HR-tree to index 2TDBs with not now-relative valid time and now-relative transaction time. Bounded valid time ranges can be considered degenerate two-dimensional MBRs, which the R-tree can handle well, and thus also the HR-tree. The overall idea is to: (1) index the initial set of tuples under an HR-tree; and (2) as tuples are updated new branches under the HR-tree are created.

Note that it is highly desirable to keep the number of newly created branches in the HR-tree 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 [2]. That structure avoids node splitting by forcing re-insertion, which is likely to affect several branches, hence "swelling" the HR-tree. A similar reasoning also applies to the R^+-tree [16] which may duplicate entries. Among the other alternatives, we have found the Hilbert R-tree[1] [9] to be suitable for our purposes and we use it as HR-tree's baseline. It does not yield duplication, aims at avoiding node splits and has been widely reported as quite efficient.

As the BRT, BIT, 2R-tree, GR-tree and 4R-tree the HR-

[1]HR-tree should not be confused with a shorthand for Hilbert R-tree. In fact, HR-tree stands for Historical R-tree.

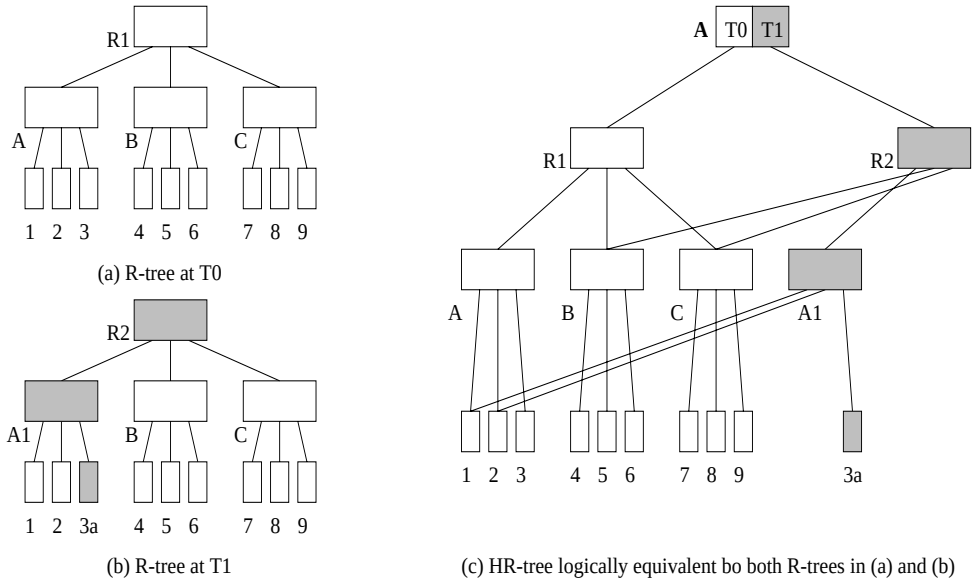


Figure 2. A single HR-tree logically equivalent to multiple R-trees.

tree is based on R-trees. Unlike the GR-tree and 4R-tree, and like the BRT and BIT, the HR-tree was not designed to handle now-relative valid time. Differently than the 2R-tree and 4R-tree, the HR-tree maintains only one single structure. A unique feature of the HR-tree is that it is able to query each indexed database state (logical R-tree) as if it was stored individually. This provides very good query processing time, with minimal overhead, unlike all other structures. There is, however, the price of a potentially large overhead in space. We investigate these issues in the next section.

4. Experimental Analysis

We now present some of the results obtained when investigating the HR-tree's performance. As the papers proposing the BRT, BIT, GR-tree and 4R-tree did, we will also use the 2R-tree approach (described in Section 2), as a reference against which we compare our proposal. In fact, we use both approaches, the interval based (which we refer to as 2Ri) and the point based (which we refer to as 2Rp). All R-trees used in the experiments, including the one used as a basis for the HR-tree, are Hilbert R-trees implemented as described in [9].

As usual in this area we focus on three main issues: update cost, query processing cost and storage requirements (the first two are measured in terms of disk I/Os). Before discussing the figures obtained let us sketch how the data sets used were generated. Some of the following criteria have been inspired by [10, 5].

We mainly deal with data sets with closed valid time ranges, that is, the initial and end valid time are known. An exception to this is Section 4.4. All the data sets suffer 100,000 updates (insertions or deletions). Each indexing structure is initially populated with 5,000 insertions, which is followed by 95,000 insertions/deletions. Three differents groups of data sets were generated, each one having different insertions/deletions ratios, namely: 60/40, 75/25 and 90/10. From now on we refer to these groups as the 60/40, 75/25 and 90/10 (data) files, respectively. Finally, each data group has four files, varying the number of updates per transaction timestamp, we experimented this number being 100, 500, 1,000 and 5,000. This reflects how better (or worse) a given structure handle different sizes of batch updates per transaction timestamp. Notice that this implies in data files having from 1000 to 20 transaction timestamps.

Without loss of generality all time values are real numbers between 0 and 1. This is due to the implementation of the Hilbert R-tree (thus the HR-tree) we currently have and is not a limitation of the structures presented. The average length of the valid time ranges is 0.05 (i.e., 5% of the maximum timespan) and it was generated using an exponential distribution.

4.1. Update Cost

The first issue investigated was the cost for updating the indexing structures. Figure 3(a) presents the average number of disk pages accessed per update in all three structures for the 60/40 data files. The HR-tree has the lowest average I/Os per update, followed by the 2Ri and 2Rp. All structures benefit (though not considerably) from having a larger batch of updates per transaction timestamp. The HR-tree outper-

forms both 2R-tree based approach because at each transaction timestamp the logical R-trees updated in the HR-tree is smaller than the R-tree updated by both 2R approaches. In the HR-tree just one logical R-tree is "visible" per transaction timestamp, whereas in both 2R approaches all updates regardless of their transaction timestamp are "visible" under the same structure. Therefore the HR-tree can be updated more efficiently. As for the two other data files (75/25 and 90/10) we noticed that as the insertions/deletions rate increases, the HR-tree requires nearly the same number os I/Os whereas both variations of the 2R-tree improve their performance. For instance, Figure 3(b) shows the obtained figures for the 90/10 data file. This suggests that the 2R-tree based approaches seem to handle more efficiently a relatively larger number of insertion than deletions.

In general, the lower the insertions/deletions ratio, the better the HR-tree's relative performance. This was indeed verified in the remaining of the experiments, and can be explained as follows. The higher the number of deletions per transaction timestamp the higher the likelihood that nodes already modified in those transaction timestamps (by the deletions) can be re-used. Hence, new nodes need not be created, enhancing update time. When there is a much larger number of insertions (relative to the number of deletions) there is a higher probability that new nodes need be created, hence consuming disk I/Os.

4.2. Query Processing Cost

To query the data indexed, we have performed transaction time point and valid time point/range intersection queries, denoted respectively as */point/point queries (i.e., querying for records with no restriction on non-temporal keys valid at a given time point at a certain transaction time point) and */range/point queries (i.e., queries have no restriction on non-temporal keys and are concerned with records valid during a range at a given point in transaction time), after simplifying the notation introduced in [17]. In both cases, the transaction time is randomly chosen from one of the indexed transaction timestamps. For the */point/point case the valid time is a random time point within [0, 1). For */range/point queries, the valid start time is randomly chosen and the queried time length has an exponential distribution with average equal to 5% of the maximum timestamp. Each query file created has 250 queries and the average figures are the ones reported. For the time being we do not consider the tuples' keys in the queries.

As noted earlier the HR-tree may not yield good performance when querying transaction time ranges, i.e., queries of the type: */point/range and */range/range. Indeed, this was verified in [14] in the context of spatiotemporal databases. As such, we do not to deal with such type of queries in this paper. Nevertheless, we plan to tackle such a shortcoming in our future research.

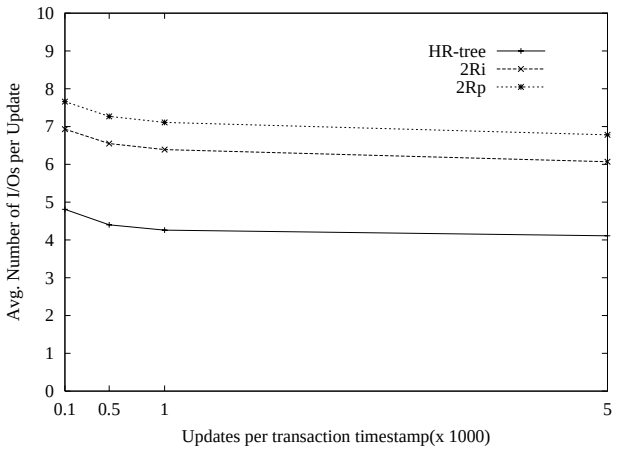
Figure 4(a) presents the average number of disk pages accessed per query in */point/point queries for the 60/40 data files. The HR-tree has the best query performance, requiring about 68% less disk access than the 2Rp, which (as expected) was the best of the 2R based implementations. As discussed before, for the 75/25 and 90/10 data files, the HR-tree's loses some of its relative advantage. Nevertheless, it still offers the best query performance, being about 50% and 33% faster than the 2Rp, in each case respectively. It was interesting to note that the 2Rp is also more robust than the 2Ri with respect to the update ratio. Indeed, Figure 4(b) shows that for the 90/10 data file, the gap between the 2Rp and 2Ri is larger than in the case of the 60/40 data file (Figure 4(a)).

Figure 5(a) depicts the results for */range/point queries, using the 60/40 data file and query ranges with average length being 5% of the maximum timespan. Consistently, the HR-tree yielded the best performance, being about 77% faster than the 2Rp which again outperforms the 2Ri. The HR-tree remains the best struture when using the 75/25 and 90/10 data files, being at least 50% faster than the 2Rp in all cases investigated.

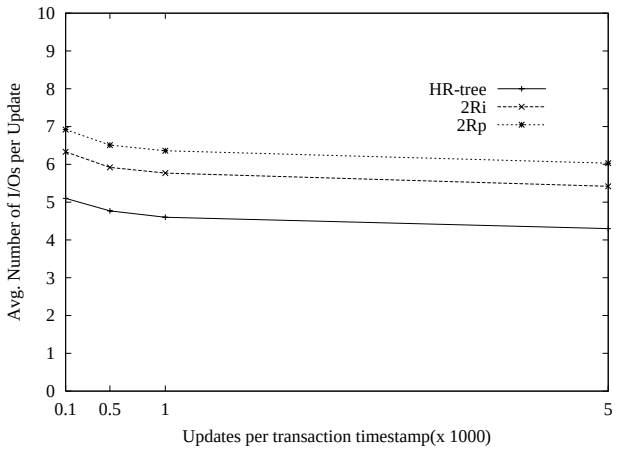
We have also experimented smaller and large query ranges, with average length of 1% and 10% of the maximum timespan. For the 1% case, the performance was virtually the same as the one obtained in the */point/point case (Figure 4(a)). When using larger queries (10% of the maximum timespan) we noticed that the relative advantage of the HR-tree becomes even more noticeable, as shown in Figure 5(b). In fact, it becomes over 80% faster than the 2Rp and the 2R-p's curve becomes closer to the 2Ri's. While in Figure 5(a) the 2Rp is about 33% faster than the 2Ri, when querying larger ranges (Figure 5(b)) this advantages falls to around 15%. When querying points (Figure 4(a) or short valid time ranges (the 1% case) the 2Rp was over 40% faster than the 2Ri. This shows that the relative gain obtained by indexing points instead of ranges (thus diminishing the degree of overlap) may be lost when querying large ranges. This behavior was observed when using the 75/25 and 90/10 data sets as well, where again the HR-tree, which is always the faster index, loses its relative advantage, although not drastically.

4.3. Storage Requirements

Figure 6 shows the size of the indexes created, for the 60/40 data file. The size of the 2Ri and 2Rp structures do not increase with the number of updates per transaction timestamp increases since the number of objects indexed in those structures does not increase. On the other hand, the larger the number of updates per transaction timestamp the lower the size of HR-tree. As argued earlier, this

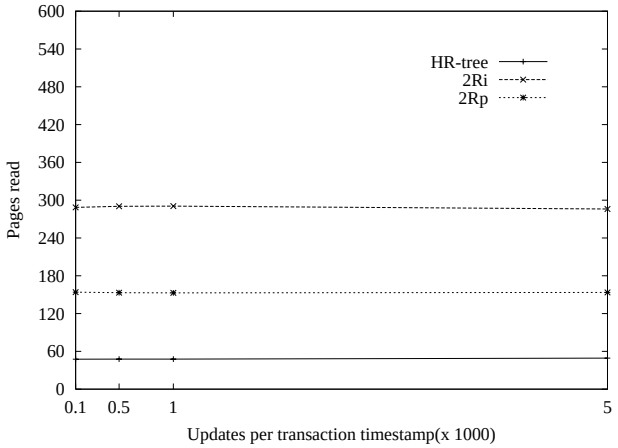


(a) Cost of updates, 60/40 data file.

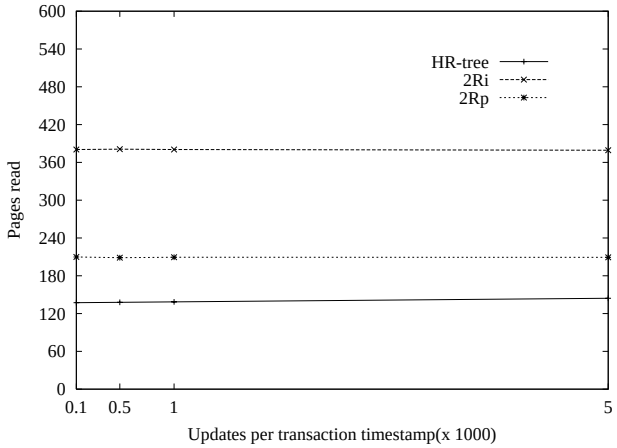


(b) Cost of updates, 90/10 data file.

Figure 3. Performance for updates

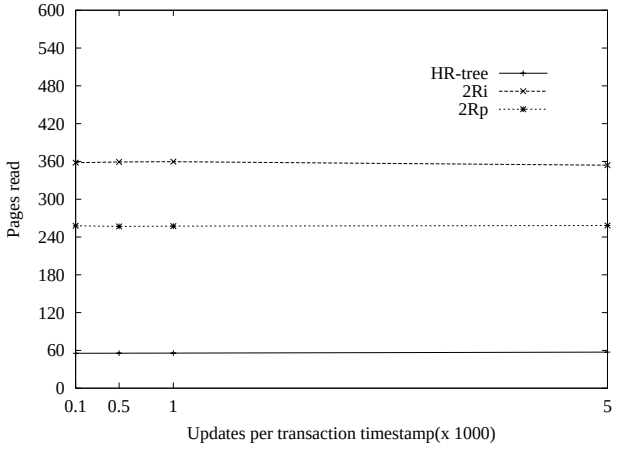


(a) 60/40 data file.

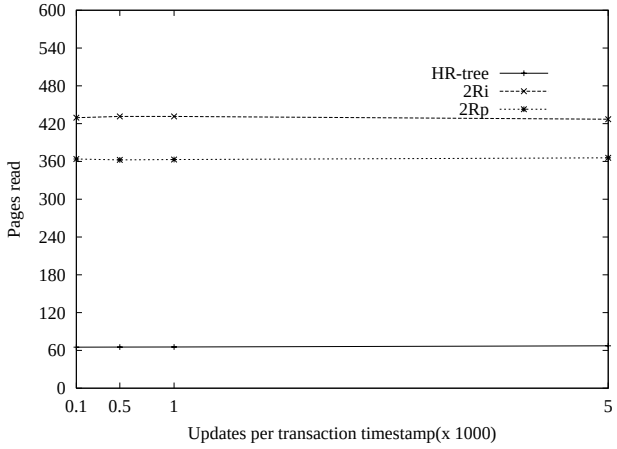


(b) 90/100 data file.

Figure 4. Cost of */point/point queries



(a) ranges limited to 5% of maximum timespan.



(b) ranges limited to 10% of maximum timespan.

Figure 5. Cost of */range/point queries

happens because less tree branches are replicated per transaction timestamp. This leads us to claim that the HR-tree is not suitable for scenarios with a low number of updates per transaction timestamp, even though the query performance does not change nearly as much as the size, indeed the HR-tree is consistently the fastest index. The 2Ri implementation yields an index sligthly smaller than the 2Rp one. In the 2Ri, objects in the two dimensional space are stored, whereas the 2Rp stores three dimensional objects. This yields more objects per disk page (tree node) in the 2Ri, therefore a better tree node utilization. For the 75/25 and 90/10 data files the results were qualitatively similar. Quantitatively, however, the HR-tree's curve shifts up faster with the increase in the insertion/deletion rate for the reasons discussed before.

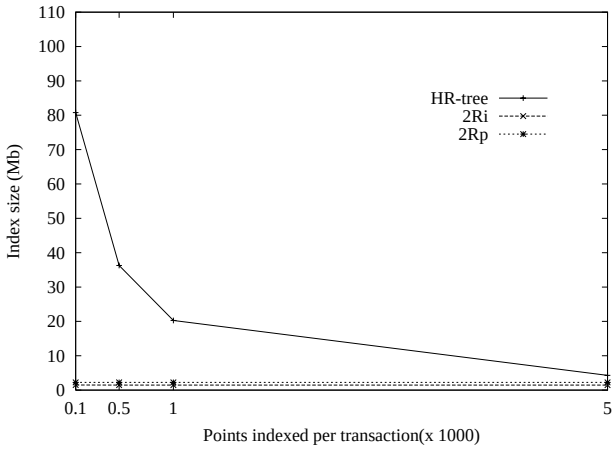


Figure 6. Indices sizes.

4.4. Indexing now-relative valid time

Thus far we have assumed only closed valid time ranges and now-relative transaction time. Even though we do not consider this a strong limitation, for the sake of completeness, we investigate next the issue of indexing *now-relative valid time*, i.e., valid time ranges with valid end time equal to *now*.

Given that updating such valid end time continuously is unfeasible, one possible approach is to assume such valid end time to be a point sufficiently far in the future. In the test cases ran we assume such time point to be 1, i.e., the maximum timestamp used in our experiments.

While on the one hand this is a simple to implement approach, on the other hand it causes a large degree of overlap among the R-tree nodes, which is known to affect its performance negatively. Nonetheless, we investigated how the indexing structures behave when indexing different ratios of open-ended valid time ranges, denoted by PNow. For brevity we only discuss the results obtained for the 60/40 data file, using 5,000 updates per transaction timestamp.

Figure 7 shows the results for */range/point queries, using medium length range queries. The results obtained for */point/point queries were qualitatively similar and are not shown. The HR-tree is not nearly as much afected by the fact of having objects with open valid end time, as are the 2R approaches, which is a somewhat surprising, but nevertheless very interesting, result. This happens because even though the degree of overlap between the R-tree nodes does increase for all structures, the HR-tree suffers much less as such a drawback is reflected only within each transaction timestamp separately. In other words, within each tree in the 2R approaches, the overlap is literally "accumulated". Therefore the larger the number of ranges with maximum valid end time the worse the performance.

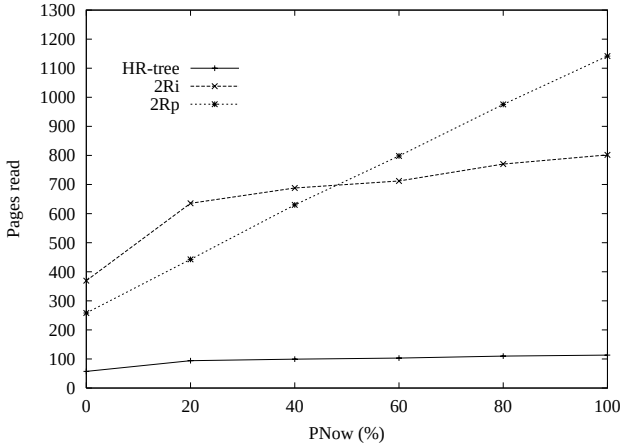


Figure 7. Cost of */range/point queries for medium ranges varying the ratio of ranges with open valid end time.

Due to the way the queries are mapped, the 2Rp's performance deteriorates much faster than the 2Ri's. The closer to the maximal timestamp is the valid end time, the bigger the chance that this point is be retrieved by a query. It is also interesting to note that 2Ri's curve is much sharper at the beginning (low values of PNow) but it is surpassed by 2Rp's curve at about PNow = 50. That gives a useful criteria for one interested in using the 2R approach with now-relative valid time.

5. Conclusions

Due to some similarity between the spatiotemporal and bitemporal indexing problem we have investigated the use of a spatiotemporal index structure, the HR-tree, for indexing 2TDBs. Using the 2R-tree [10] as a reference, we fo-

cused mainly on indexing bounded (i.e., not now-relative) valid time ranges and now-relative transaction time. Performance was measured using point and range queries on valid time. The structures' query performance and sizes were investigate with respect to: number of updates per transaction timestamp; insertions/deletions ratio; and size of the queried valid time ranges. Even though we do not deal with the indexing and querying of non-temporal data, e.g., keys, either the HR-tree or the 2R-trees (as well as all other R-tree based structures) could be used for such a task.

Overall, the HR-tree has a good update performance. For all data files and queries investigated, the HR-tree yielded consistently the best search performance, requiring, most of the time, less than half of the I/Os required by the best 2R-tree approach. In addition, the lower the ratio insertion/deletion of objects the better it is for the HR-tree. In addition, the HR-tree appears to be sound with respect to now-relative (i.e., open) valid time ranges.

Size-wise, the HR-tree is very dependent on the update rate. The more updates per transaction timestamp, the smaller the resulting tree. Several applications may present such high update rate characteristic, e.g., in the banking/financial domain. With current technology one could have updates bearing very fine transaction timestamps, say milliseconds. However, it is hardly reasonable to consider that queries will be posed using such a fine granularity. As such all transactions happening within, say a minute, could bear the same transaction timestamp and be inserted into the index in a batch mode. We believe that this rationale would also apply to other application domains. Another possibility, requiring some small further processing at update time, would be to collect all incoming transactions along with their original (fine grain) timestamps in a buffer and index all the updates in that buffer at pre-specified time intervals. In such a case, if the user poses a query with respect to a lower level timestamp than the one actually indexed (say milliseconds instead of minutes) then false-hits would likely need to be filtered out of the query's answer. Notice that such filtering would avoid access to actual data records (i.e., useless I/Os). The user could then experiment with differents time granularities in order to decrease false-hits at the possible expense of obtaining a larger (but nevertheless fast) index.

As the BTR was also compared to the 2R approach, an indirect comparison between the HR-tree and BRT performance presented in [10] seems to indicate that the structures may have comparable search performance for the queries investigated in this paper. We should make clear though that the BRT experiments assume one single update per transaction timestamp. Such an update rate would render the HR-tree unfeasibly large. We only conjecture that the HR-tree may be comparable to the BRT in terms of query processing time. Even though the GR-tree and 4R-tree were also compared against the 2R-trees a direct comparison between those and the HR-tree cannot be easily made as those structures assume now-relative valid time, which is not the case of the HR-tree.

Future research should focus on making the HR-tree more robust in terms of the space overhead; investigating whether the overlapping approach could be used with other range indexing structures (e.g., [3]); investigating different cache policies for bitemporal data and comparing the HR-tree to other biutemporal access structures.

Acknowledgments

This work was performed while both authors were with the State University of Campinas, Brazil. Jefferson R. O. Silva was supported by FAPESP (Process 97/11205-8). Mario A. Nascimento was partially supported by CNPq (Process 300208/97-9) and Pronex/FINEP project "SAI: Advanced Information Systems" (Process 76.97.1022.00). The authors thank S. Saltenis and C. S. Jensen for their constructive comments.

References

[1] B. Becker et al. On optimal multiversion access structures. In *Proc. of Symposium on Large Spatial Databases*, pages 123–141, June 1993.

[2] 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.

[3] G. Blankenagel and R. H. Güting. External segment trees. *Algorithmica*, 12(6):490–532, 1994.

[4] R. Bliujūtė et al. Light-weight indexing of general bitemporal data. Technical Report 30, TimeCenter, 1998.

[5] R. Bliujūtė et al. R-tree based indexing of now-relative bitemporal data. In *Proc. of the 24th Intl. Conf. on Very Large Databases*, pages 345–356, August 1998.

[6] H. Edelsbrunner. A new approach to rectangle intersections, part i xxxxxx ii. *Int. Journal of Computer Mathematics*, 13:209–229, March 1983.

[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] C. Jensen et al. A consensus glossary of temporal database concepts. *ACM SIGMOD Record*, 23(1):52–64, 1994. (Updated version available at `http:www.cs.auc.dk/~csj/Glossary`).

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

[10] A. Kumar, V. Tsotras, and C. Faloutsos. Designing access methods for bi-temporal databases. *IEEE Trans. on Knowledge and Data Engineering*, 10(1):1–20, 1998.

[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. Nascimento, M. Dunham, and R. Elmasri. M-IVTT: An index for bitemporal databases. In *Proc. of the 7th Intl. Conf. on Databases and Expert Systems Applications*, pages 779–790, September 1996.

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

[14] M. Nascimento, J. Silva, and Y. Theodoridis. Evaluation of access structures for discretely moving points. In *Proc. of the Intl. Workshop on Spatiotemporal Database Management*, pages 171–188, September 1999.

[15] B. Salzberg and V. Tsotras. A comparison of access methods for time evolving data. *ACM Comp. Surveys*, 31(2):158–221, 1999.

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

[17] V. Tsotras, C. Jensen, and R. Snodgrass. An extensible notation for spatiotemporal index queries. *ACM SIGMOD Record*, 27(1):47–53, 1998.

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

[19] P. Varman and R. Verma. An efficient multiversion access structure. *IEEE Trans. on Knowledge and Data Engineering*, 9(3):391–409, 1997.