An Incremental Index for Bitemporal Databases

Jefferson R. O. Silva and Mario A. Nascimento

November 13, 1998

TR-38

Title	An Incremental Index for Bitemporal Databases
	Copyright © 1998 Jefferson R. O. Silva and Mario A. Nascimento . All rights reserved.
Author(s)	Jefferson R. O. Silva and Mario A. Nascimento
Publication History	November 1998. A TIMECENTER Technical Report.

Abstract

Bitemporal databases record not only the history of tuples in temporal tables, but also record the history of the databases themselves. Indexing structures, which are a critical issue in traditional databases, became even more critical for bitemporal databases. We address this problem by investigating an incremental indexing structure based on R-trees, called the HR-tree, which was originally aimed at spatiotemporal databases. We have found that the HR-tree is much more efficient (up to 80% faster) than previously proposed approaches based on two R-trees when processing transaction time point based queries. As for size, the HR-tree was found to be better suited for medium to large sized batch updates, otherwise it is prone to be quite large.

1 Introduction

Temporal databases have been the object of quite some research regarding many aspects and much has been published in the field [TK96]. It has been recognized that two dimensions of time need to 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 [J$^+$94]. A *Bitemporal Database* (2TDB) is thus one which supports both valid time and transaction time. In such a case one is allowed to ask queries based on different 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 short paper is to investigate the efficiency of an index structure originally designed for spatiotemporal databases, the HR-tree, [NS98, NST98], for indexing 2TBDs. In spatiotemporal databases, one must index not only the spatial extents of objects, which we refer to as *spacestamp*[1], but also their evolutions as time progresses. The current spacestamp is the one stored until it is changed. Hence we assume it is stored until the current point in time, denote by *now*. As such the transaction time of the spacestamps are *now-relative* [B$^+$98b]. On the other hand we assume that the valid-time of those spacestamps are *not now-relative*, i.e., they are known in advance.

As an example of a spatiotemporal scenario that fits the above requirements 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 where 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.

Note that if we simply replace the notion of spacestamps for the notion of regular tuple attributes we obtain a 2TBD. In other words, instead of assigning valid time to spacestamps we do so for a set of regular tuple's attributes. This similarity is the motivation we use to investigate the HR-tree's performance when indexing 2TDBs.

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 we obtained. Section 5 concludes the paper with a summary and offers directions for further work.

2 Related Work

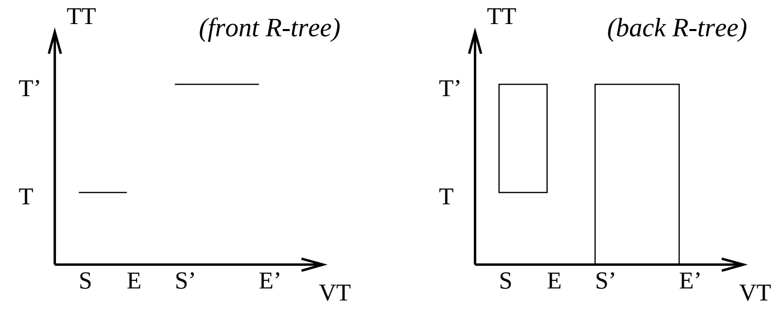
While a reasonable number of papers have been published on the issue of indexing either valid time or transaction time databases, only a handfull have addressed the problem of indexing 2TDBs [ST97]. In what follows we review some of the approaches proposed recently, most based on R-trees [Gut84]: the M-IVTT [NDE96], the Bitemporal Interval Tree, Bitemporal R-Tree, the 2R-tree [KTF98], the GR-tree [B$^+$98b] and the 4R-tree [B$^+$98a].

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, having its leaves pointing to valid time trees. Underneath this transaction time tree there is a forest of Valid Time Trees (VTTs.) Each VTT indexes the valid time of all records existing at that 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.

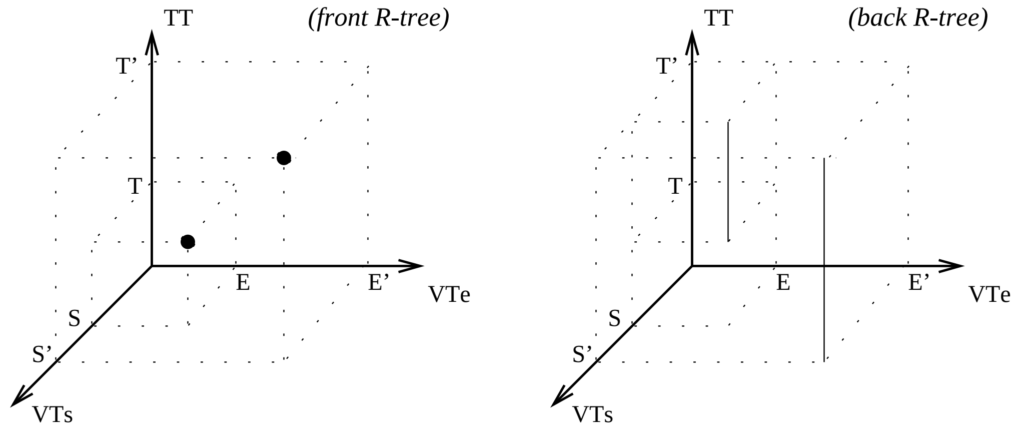
[1]A spacestamp can be either a N-dimensional MBR (Minimum Bounding Rectangles), a point or any other appropriate abstraction.

The Bitemporal Interval Tree (BIT), the Bitemporal R-tree (BRT) and 2R-tree structures index closed valid time ranges and now-relative transaction time objects. 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 [Ede83], 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 [B$^+$90] partially persistent, following the approach of the MVB [B$^+$93] and MVAS [VV97]. Like those structures, 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 the 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 is 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.



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



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

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

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. Figure 1(b) shows how the temporal data in Figure 1(a) would be indexed using such transformation. The advantage of using such a point based approach is that the amount of overlap among the indexed objects is diminished, hence the underlying R-trees can offer a better performance.

The GR-tree and 4R-tree index both now-relative valid and transaction time. The GR-tree extends the R*-tree [B$^+$90] to store both static tuples (with closed valid and transaction time ranges) and growing objects (with either valid or transaction end time unknown). 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. 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 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, the objects are 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, it is interesting to note that in the case of bitemporal data with no now-relative valid time the 4R-tree reduces to the 2R-tree.

Even though we have not discussed, in all approaches above incoming queries should be modified accordingly, further details can be found in the original papers.

3 The HR-tree for 2TBDs

As most other proposals the HR-tree is also based on R-tree. The HR-trees were 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. These "movements" characterize different states (snapshots) of the spatiotemporal database. One trivial way to index such states 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 [MK90] 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.

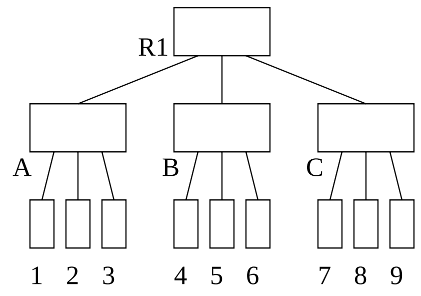
Although it is just a simple example, it is easy to see that much space could be saved by re-using the nodes that did not change from a given state to the next one. Note that with the addition of a simple structure **A** (an array in the figure, but which could be a B-tree if warranted) the root node of the desired R-tree, current for a given timestamp, can be obtained quickly, and thus the query processing cost is the *same* as if all R-trees where kept physically. This becomes handy 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 [NS98, NST98].

As argued earlier in the paper it is rather simple to use the HR-tree to index 2TDBs with 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. Given that, 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.

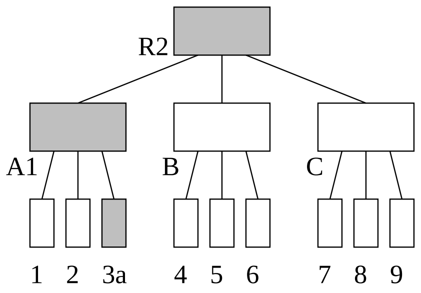
Note that it is highly desirable to keep the number of newly created branches in the HR-tree as low as possible. For that reason some R-tree variants are not suitable to serve as HR-tree's framework, notably, the R$^+$-tree [SRF87] and the R*-tree [B$^+$90]. In the former the MBRs are "clipped" and one 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 Among the other alternatives, we have found the Hilbert R-tree[2] [KF94] to be suitable for our purposes and use it as HR-tree's baseline. It does not yield duplication, avoids re-insertion and is indeed reported to be quite efficient.

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

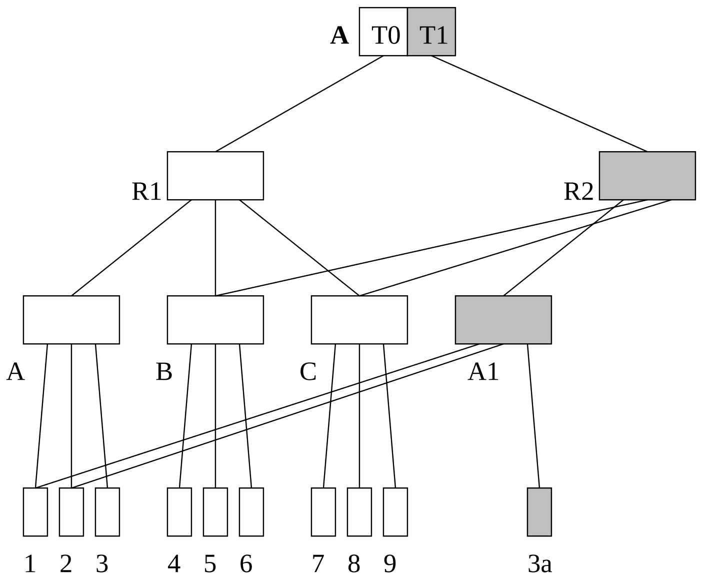
[2]Note that HR-tree should not be confused with a shorthand for Hilbert R-tree, in fact, HR-tree stands for Historical R-tree.



(a) R-tree at T0



(b) R-tree at T1



(c) HR-tree logically equivalent bo both R-trees in (a) and (b)

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

good query processing time, with very little, if any, overhead, unlike all other structures. There is, however, the price of a potentially large overhead in space. We investigate this issue, among others, in the next section.

4 Performance 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 use the 2R-tree approach (described in Section 2), as references against which we compare our proposal – in fact, we will 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 [KF94].

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 we used were generated. Some of the following criteria have been inspired by [KTF98, B^+98b].

We have used only data sets with closed valid time ranges, that is, the initial and end valid time are known. All the data sets have 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 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/O 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 outperforms the 2R approach because at each transaction timestamp the logical R-tree updated in the HR-tree is smaller than the R-tree updated by the 2R approach. In the HR-tree just one logical R-tree is "visible" per transaction timestamp, whereas in the 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 slightly more I/Os, while both variations of the the 2R improve their performance. In fact, we also observed that for the 90/10 data file the 2Ri performs nearly as well as the HR-tree.

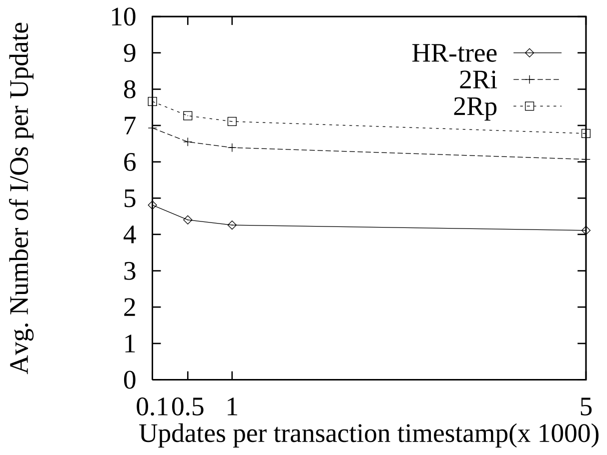


Figure 3: Cost of updates, 60/40 data file

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. This 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 the 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 queries, denoted respectively as */point/point and */range/point queries, after simplifying the notation introduced in [TJS98]. In both cases, the transaction time is randomly chosen from one of the transaction timestamp indexed. For the */point/point case the valid time is a random time point within [0, 1). For */range/point queries, the queried time length has also an exponential distribution with average equal to 0.05. Each query file created has 250 queries and the average figures are the ones reported.

As note 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 [NST98] in the context of spatiotemporal databases. As such, we chose not to deal with such type of queries in this paper.

Figure 4 presents the average number of disk pages accessed per query in 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 was expected to be the best of the 2R based implementations. As discussed above, 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, respectively.

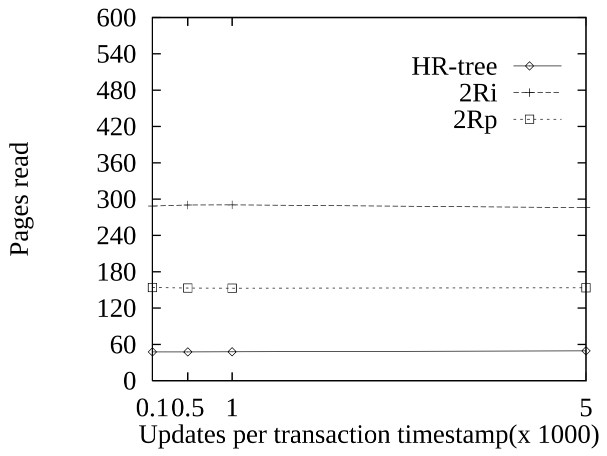


Figure 4: Cost of */point/point queries.

Figure 5 depicts the results for */range/point queries, using the 60/40 date file and both the indexed and query ranges with average lengths of 5% of the maximum timespan. Consistently, the HR-tree yield 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.

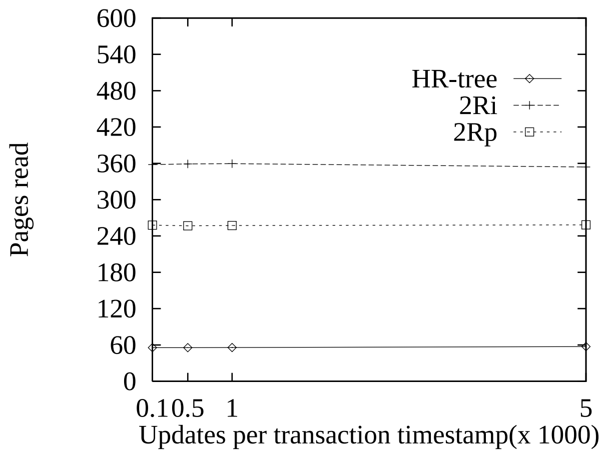


Figure 5: Cost of */range/point queries.

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). When using larger queries we noticed that the relative advantage of the HR-tree becomes even larger. In fact, it becomes over 80% faster than the 2Rp and the 2Rp's curve becomes closer to the 2Ri's. While in Figure 5 the 2Rp is about 33% faster than the 2Ri, when querying larger ranges (twice as large in this case) this advantages falls to around 15%. When querying points (Figure 4 or short valid time ranges (the 1% case) the 2Rp was over 40% faster than the 2Ri. This shows that the gain obtained by indexing points instead of ranges (thus diminishing the degree of overlap) may be lost when querying large regions. 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 as the ratio of insertions/deletions increases.

4.3 Storage Requirements

Figure 6 shows the size of the indexes created, for the 60/40 data file. The 2Ri and 2Rp structures have a linear behavior as the number of updates per transaction timestamp increases since the number of objects indexed, i.e., the number of updates, in the structure 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 is so because less tree branches are duplicated. 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 performance does not change nearly as much, and the HR-tree is consistently the faster index. The 2Ri implementation yields an index sligthly smaller than the 2Rp one. In the 2Ri, objects in the two dimensional space are stored, insted of the 2Rp, which stores three dimensional objects. This implies in more objects per page in the 2Ri against the 2Rp, which means a lower index structure. 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.

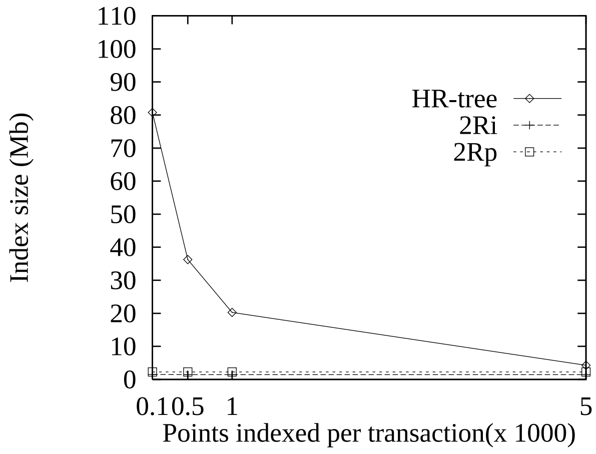


Figure 6: Indices sizes.

5 Conclusions

Due to some similarity between spatiotemporal and bitemporal objects, we have investigated the use of a spatiotemporal index structure, the HR-tree to the 2TDB indexing problem. Using the 2R-tree [KTF98] as a reference, we focused mainly on indexing bounded (i.e., not now-relative) valid time ranges and now-relative transaction time. Performance was measured upon queries of type */point/point and */range/point. 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 and the 2R-trees (as well as all other R-tree based structures) could be used for such a task.

Regarding the above variables we noticed that as the number of updates per transaction timestamp increases, the HR-tree's size diminishes, indicating that HR-tree is better suited for batch updates. Likewise the lower the ratio insertion/deletion of objects the better it is for the HR-tree. Overall, the HR-tree has a good update performance. The only case where it nearly tied with the 2R approach happened when the number of insertions was much bigger than the number of deletions. 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. 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 timestamp in a buffer and index all index the data in the buffer at pre-specified time intervals. For instance, one could have an accuracy of milliseconds for transaction timestamps, but index updates every minute, or so. 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), therefore not being too expensive. The user could then experiment with differents time granularities in order to decrease false-hits at the possible expense of obtaining a larger (but fast nevertheless) index.

As the BTR was also compared to the 2R approach, an indirect comparison between the HR-tree and BRT performance presented in [KTF98] 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 the case of reasonably sized batch updates. 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: (*i*) investigating how the HR-tree performs when indexing now-relative valid time (perhaps after some modification in its structure); (*ii*) investigating the effect of cache structures; (*iii*) designing a benchmarking data set against which previously proposed structures could be evaluated and compared and; (*iv*) investigating whether the overlapping approach could be used with other range indexing structures (e.g., [BG94]).

Acknowledgments

Jefferson R. O. Silva (972147@dcc.unicamp.br) was supported by FAPESP (Process 97/11205-8). Mario A. Nascimento was partially supported by CNPq (Process 300208/97-9) and is also a researcher with Embrapa (mario@cnptia.embrapa.br). This research was developed as part of MCT/PRONEX's project SAI. The authors thank S. Saltenis and C. S. Jensen for their constructive comments.

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$^+$93] B. Becker et al. On optimal multiversion access structures. In *Proc. of Symposium on Large Spatial Databases*, pages 123–141, June 1993.

[B$^+$98a] R. Bliujūtė et al. Light-weight indexing of general bitemporal data. Technical Report 30, TimeCenter, 1998.

[B$^+$98b] 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.

[BG94] G. Blankenagel and R. H. Güting. External segment trees. *Algorithmica*, 12(6):490–532, 1994.

[Ede83] H. Edelsbrunner. A new approach to rectangle intersections, part i xxxxxx ii. *Int. Journal of Computer Mathematics*, 13:209–229, March 1983.

[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.

[J$^+$94] C.S. Jensen et al. A consensus glossary of temporal database concepts. *ACM SIGMOD Record*, 23(1):52–64, 1994.

[KF94] 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.

[KTF98] A. Kumar, V.J. Tsotras, and C. Faloutsos. Designing access methods for bi-temporal databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):1–20, 1998.

[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.

[NDE96] M. A. Nascimento, M. H. 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.

[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.

[NST98] M. A. Nascimento, J. R. O. Silva, and Y. Theodoridis. Access structures for moving points. Technical Report 33, TimeCenter, 1998.

[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.

[TJS98] V.J. Tsotras, C.S. Jensen, and R.T. Snodgrass. An extensible notation for spatiotemporal index queries. *ACM SIGMOD Record*, 27(1):47–53, 1998.

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

[VV97] P.J. Varman and R.M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.