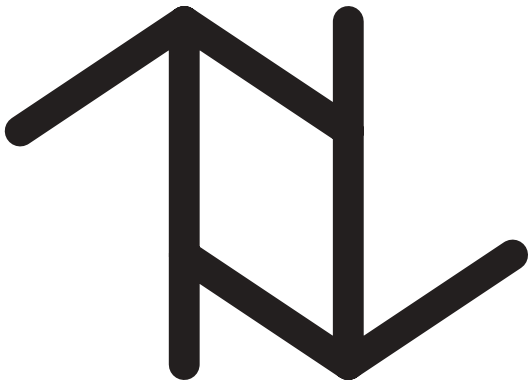

R-tree Based Indexing of Now-Relative Bitemporal Data

Rasa Bliujute, Christian S. Jensen, Simonas Saltenis, and Giedrius Slivinskas

March 12, 1998

TR-25

A TIMECENTER Technical Report

Title	R-tree Based Indexing of Now-Relative Bitemporal Data
Author(s)	Rasa Bliujute, Christian S. Jensen, Simonas Saltenis, and Giedrius Slivinskas
Publication History	March 1998. A TIMECENTER Technical Report.

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector)
Michael H. Böhlen
Renato Busatto
Heidi Gregersen
Dieter Pfoser
Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector)
Anindya Datta
Sudha Ram

Individual participants
Curtis E. Dyreson, James Cook University, Australia
Kwang W. Nam, Chungbuk National University, Korea
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, University of South Florida, USA
Andreas Steiner, ETH Zurich, Switzerland
Vassilis Tsotras, University of California, Riverside, USA
Jef Wijsen, Vrije Universiteit Brussel, Belgium

For additional information, see The TIMECENTER Homepage:
URL: `<http://www.cs.auc.dk/general/DBS/tdb/TimeCenter/>`

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

The databases of a wide range of applications, e.g., in data warehousing, store multiple states of time-evolving data. These databases contain a substantial part of *now*-relative data: data that became valid at some past time and remains valid until the current time. More specifically, two temporal aspects of data are frequently of interest, namely *valid time*, when data is true, and *transaction time*, when data is current in the database. The latter aspect is essential in all applications where accountability or trace-ability are required. When both aspects are captured, data is termed bitemporal.

A number of indices have been devised for the efficient support of operations on time-varying data with one time dimension, but only little work, based mostly on R-trees, has addressed the indexing of two- or higher-dimensional temporal data. No indices exist that contend well with now-relative data, which leads to temporal data regions that are continuous functions of time. The paper proposes two extended R*-trees that permit the indexing of data regions that grow continuously over time, by also letting the internal bounding regions grow. Internal regions may be triangular as well as rectangular, and new heuristics for the algorithms that govern the index structure are provided. As a result, dead space and overlap, now also continuous functions of time, are reduced. Performance studies indicate that the best extended index is typically 3–5 times faster than existing R-tree based indices.

1 Introduction

Data stored in a database has two fundamental temporal aspects—valid time and transaction time [SA85] [JS96]. The valid time of a database fact is the time when the fact is true in the modeled reality, while the fact's transaction time is the time during which it is current in the database. Valid time is meaningful and necessary for a wide range of applications, and transaction time is particularly useful in applications where traceability or accountability are important. Applications dealing with temporal data would benefit from temporal support being built into the DBMS. In response to this, several dozen temporal data models and query languages have been proposed, a few with prototype implementations, and temporal support is finding its way into the SQL standard [SNO96]. In contrast, much less research has addressed the design of efficient temporal-query processing techniques. This paper addresses the need for efficient indexing of temporal data.

Existing research shows that regular indices such as B+-trees are unsuited for temporal data [ST97], and there has recently been proposed a number of indices for temporal data. The majority are for transaction-time data, and only few support valid-time data. Significantly less research has been done on creating indices that support data with both valid and transaction time, so-called bitemporal data.

Some of the existing bitemporal indices follow the *partial persistence* [DRI89] approach and do not treat valid and transaction time symmetrically, but obtain bitemporal support by making a valid-time access structure partially persistent. Such indices do not efficiently support range queries on transaction time. Another approach is to base bitemporal indices on spatial indices, due to the similarities between bitemporal and spatial data: the combined valid and transaction time of a fact can be treated as a region in two-dimensional space. Several existing proposals [KTF95] [KTF97] are based on the R*-tree [BEC90], which is known as the most efficient member of the R-tree family of spatial indices.

Existing bitemporal indices fall short in efficiently supporting data related to the current time, i.e., data for which the end of the valid time or transaction time is not fixed, but tracks the current time and thus continuously extends as time passes. We term such data *now-relative* data. It occurs naturally and frequently in many situations. Consider an example where we want to record new employees in a company's database. When the employees start working (valid-time interval begin) is known, but we frequently do not know when the employees will leave. This is captured by letting the valid-time end extend to the growing current time. The same applies to transaction time. The transaction-time interval begin is the time when we insert

a fact into the database. Since we do not know when the fact will stop being current in the database, the transaction-time end is not fixed, but extends to the current time. Existing indices support efficiently only now-relative transaction-time intervals. None support data where the valid-time interval is now-relative.

The paper describes how to support now-relative bitemporal data in R-tree based indices, and it proposes two extended R*-trees. The new indices permit the indexed data regions to grow as time progresses, by also letting the internal bounding regions grow. Internal regions may be triangular as well as rectangular, and new heuristics for the algorithms that govern the index structure are provided. As a result, dead space and overlap, now functions of time, are reduced. This reduces the number of paths followed during a search, and performance studies indicate that the best extended index is typically 3–5 times faster than existing R-tree based indices.

The presentation is structured as follows. In Section 2, we briefly describe important concepts and explain how the time associated with bitemporal data may be described using two-dimensional regions. Section 3 surveys the existing work related to the indexing of temporal data, including temporally adapted spatial indices and motivates the need for a new bitemporal index. The structures of the proposed R*-tree extensions are given in Section 4. Section 5 presents algorithms for the insert, delete, and search operations. Section 6 presents performance studies. The final section offers conclusions and possibilities for future work.

2 Background

We first explain how valid and transaction time are associated with database facts using TQuel's four-timestamp format [SNO87]. We then show that the time associated with bitemporal data can be viewed as two-dimensional regions, which suggests that bitemporal data may be indexed using adapted spatial indices.

The temporal aspects termed valid and transaction time have proven to be of interest in a wide range of database applications. Valid time captures when a fact is true in the modeled reality, and transaction time captures when a fact is current in the database [SA85, JS96]. These two aspects are orthogonal in that each could be independently recorded, and each has specific properties associated with it. The valid time of a fact can be in the past or in the future (allowing to store information about the past and the future) and can be changed freely. In contrast, the transaction time of a fact cannot extend beyond the current time and cannot be changed.

To investigate the indexing of bitemporal data, we need a suitable representation of bitemporal data. TQuel's four-timestamp format [SNO87] (4TS) is the most popular for this purpose. With this format, one database fact is represented in one or more tuples that each have a number of non-temporal attributes and four time attributes:

`VTbegin`—the time when the fact became true in the modeled reality,

`VTend`—the time when the fact ceased to be true in the modeled reality,

`TTbegin`—the time when the fact with its valid time became current in the database, and

`TTend`—the time when the fact with its valid time ceased to be current in the database.

A fact is now-relative if it is valid until the current time or is part of the current database state. This is represented in the 4TS format by the use of variables, which denote the current time, for the time attributes `VTend` and `TTend` [CLI97]. The variable UC is used for `TTend`, and the variable NOW is used for `VTend`. The relation in Figure 1 exemplifies this representation of bitemporal data. The time granularity is a month, and the current time is assumed to be 9/97.

Tuple (1) records that the fact "John works in Advertising" was true from 3/97 to 5/97 and that this was recorded during 4/97 and is still current. Tuple (3) records that "Jane works in Sales" from 5/97 until the the current time, that we recorded this belief on 5/97, and that this remains part of the current database state.

Specific constraints apply to insertions, deletions, and modifications of tuples in a bitemporal database.

	Employee	Department	TTbegin	TTend	VTbegin	VTend
(1)	John	Advertising	4/97	UC	3/97	5/97
(2)	Tom	Management	3/97	7/97	6/97	8/97
(3)	Jane	Sales	5/97	UC	5/97	NOW
(4)	Julie	Sales	3/97	7/97	3/97	NOW
(5)	Julie	Sales	8/97	UC	3/97	7/97
(6)	Michelle	Management	5/97	UC	3/97	NOW

Figure 1: A Fragment of EmpDep Relation

When inserting a new tuple, the constraints `VTbegin` $\leq$ `VTend` and `VTbegin` $\leq$ 'current time' if `VTend` is equal to NOW apply to valid time; and the constraints `TTbegin` = 'current time' and `TTend` = UC apply to transaction time. Any *current* database tuple can be deleted or modified. Deleting a tuple, the `TTend` value UC is changed to the fixed value 'current time'-1[1], making the tuple not current anymore (e.g., Tuple (2)); tuples are not physically deleted. A modification is modeled as a deletion followed by an insertion (e.g., an update led to Tuple 4 and Tuple 5).

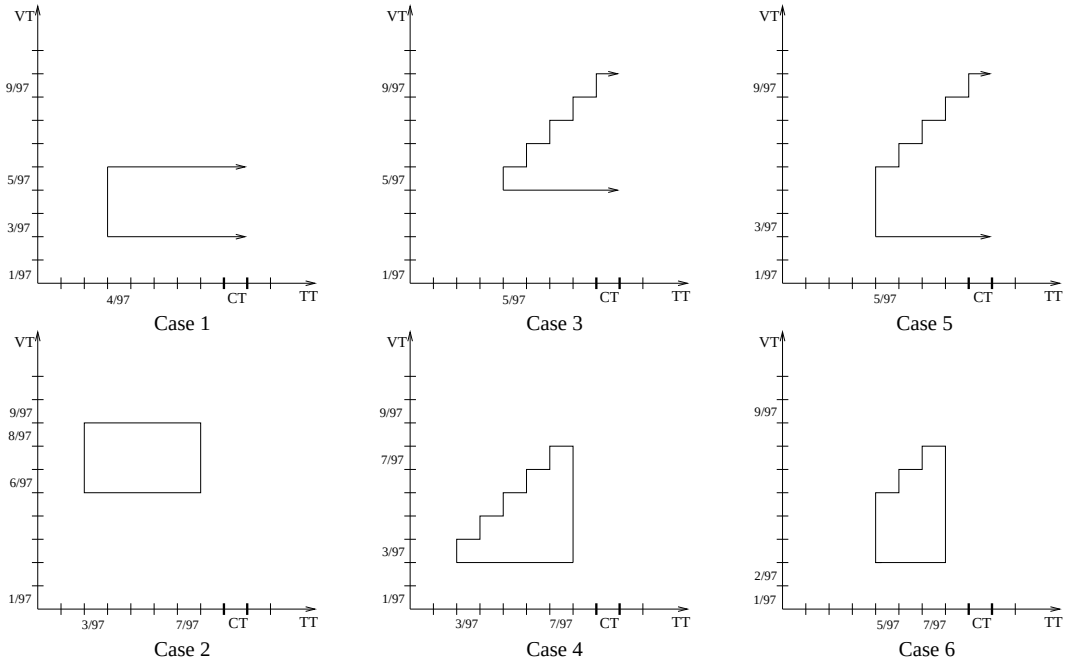

Figure 2: Bitemporal Regions

The temporal aspect of a tuple can be represented graphically by a two-dimensional ("bitemporal") region in the space spanned by valid and transaction time [JS96]. Cases 1–5 in Figure 2 illustrate *bitemporal regions* of Tuples (1–4) and (6), respectively.

A now-relative transaction-time interval yields a rectangle that "grows" in the transaction time direction as time passes (Tuple (1), Case 1). Having both transaction- and valid-time intervals being now-relative yields a stair-shaped region growing in both transaction time and valid time as time passes (Tuple (3), Case 3). Facts can be recorded in the database after they become true in the modeled reality. In this situation, also having both the transaction- and valid-time intervals being now-relative yields a stair-shape with a high first step (Tuple (6), Case 5).

It is also possible to record a fact in the database before it becomes true in the modeled reality. In this case, the valid-time end must be a ground value (Tuple (2), Case 2); otherwise, the valid-time start, which

[1]We use closed intervals and let [`TTbegin`, `TTend`] denote the interval that includes `TTbegin` and `TTend`.

would extend to the current time, would initially be larger than the valid-time end, violating the second insertion constraint. If, at some time, a tuple stops being current, the bitemporal region stops growing (Tuples (2), (4); Cases 2, 4, 6).

Stated generally, we obtain six combinations of time attributes for which the bitemporal regions are qualitatively different (Figure 2), see Figure 3 where 'tt1', 'tt2', 'vt1', and 'vt2' denote ground values that satisfy the constraints given above.

	TTbegin	TTend	VTbegin	VTend	
Case 1	tt1	UC	vt1	vt2	
Case 2	tt1	tt2	vt1	vt2	
Case 3	tt1	UC	vt1	NOW	(tt1=vt1)
Case 4	tt1	tt2	vt1	NOW	(tt1=vt1)
Case 5	tt1	UC	vt1	NOW	(tt1>vt1)
Case 6	tt1	tt2	vt1	NOW	(tt1>vt1)

Figure 3: Possible Combinations of Time Attributes

We have set the context for using spatial indices for indexing bitemporal data. There already exist some indices for bitemporal data that are based on the spatial index, but they do not accommodate now-relative valid-time intervals well; the next section discusses the existing indices.

3 Overview of the Existing Bitemporal Indices

A wealth of indices for temporal data exist; references [BER97, ST97] provide comprehensive surveys. We focus on the indexing of bitemporal data. In one approach, a bitemporal index is obtained by making a valid-time index partially persistent [DRI89]. The Bitemporal Interval Tree [KTF95] represents this approach. Another approach is to view bitemporal data as a special case of spatial data (recall Figure 2) and to adapt spatial indices to bitemporal data. This is the approach we adopt in this paper.

Many indices have been developed for spatial data [SAM90]. One of the most robust indices for spatial data with extent (i.e., non-point data) is the R-tree [GUT84] in its different variants—e.g., the R$^+$-tree [SRF87], the R*-tree [BEC90], and the Hilbert R-tree [KF94]. All variants of the R-tree try to minimize the overlap between the minimum bounding rectangles of the nodes at each level of the tree and to minimize the dead space in the bounding rectangle of each node (dead space is the space in the minimum bounding rectangle not occupied by any enclosed rectangle). Minimizing overlap reduces the I/O-incurring branching of search into several subtrees. Minimizing dead space reduces the probability that queries unnecessarily access disk pages, eventually finding no qualifying data.

The R*-tree is promising for indexing of bitemporal data, but it is not directly applicable because it accommodates only static rectangles. We have to also contend with growing rectangles and static and growing stair-shapes. The straightforward approach to accommodating growing bitemporal regions is to represent them using static rectangles that extend to the maximum possible transaction- and valid-time values. As a consequence, the minimum bounding rectangles in internal tree nodes also extend to the maximum values, resulting in excessive dead space and overlap; see Figure 4.

Kumar et al. [KTF95, KTF97] propose a new approach to handling now-relative transaction time, but do not address now-relative valid time. In their approach (the 2-R approach), they use two R-trees. The *front* R-tree indexes all growing (i.e., current) rectangles, while the *back* R-tree indexes all static (i.e., logically deleted) rectangles. Observing that all growing rectangles are in the front tree and that they all end at the (growing) current time, Kumar et al. show that storing only the non-growing transaction-time begin values in the front tree is adequate to support now-relative transaction time. The 2-R approach contends well with now-relative transaction time, but it suffers from the penalty that both trees often have to be searched in a

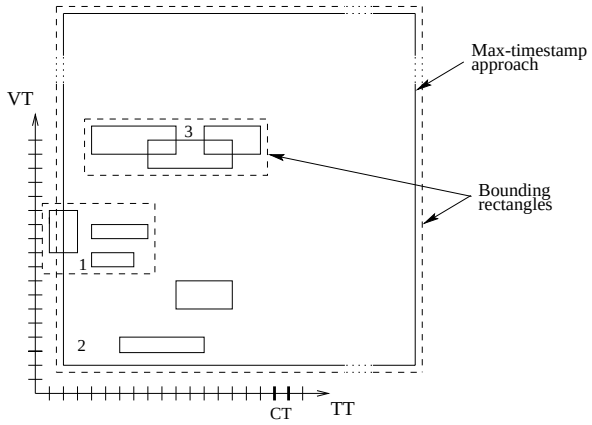

Figure 4: Indexing Growing Bitemporal Regions Using Maximum Timestamp Values

single query, resulting in more disk accesses and diminishing the advantages of the decreased overlap.

It is also possible to combine the spatial index approach and the partial persistence approach. Reference [KTF97] presents the Bitemporal R-Tree, where an R-tree is used to index the valid-time intervals and key values of the data objects, and transaction-time support is achieved by making the structure partially persistent. However, the Bitemporal R-Tree does not accommodate now-relative valid-time intervals, and, like all structures based on partial persistence, it introduces some space overhead. Experiments that do not consider now-relative valid time [KTF97] indicate that this tree has very good query performance.

The straightforward approach to accommodating now-relative valid-time intervals that was exemplified in Figure 4 does not seem promising. With this approach, many queries with valid-time interval above the current time will access the resulting very large rectangles that have valid-time end values bigger than any valid time specified in queries and valid-time begin values smaller than or equal to the current time. Yet, none of these accesses will contribute to the answer of the query because the actual bitemporal data regions represented by these rectangles have valid-time end values equal to the current time, and the valid time specified in the query is greater than the current time. This straightforward approach, which we call the *maximum-timestamp approach*, does not utilize the knowledge of the actual shapes of bitemporal regions. To achieve the best performance, a bitemporal index should utilize this knowledge.

In subsequent sections, we present an extension to the existing spatial index that efficiently handles bitemporal data with both fixed and now-relative valid- and transaction-time intervals.

4 Structure

Having identified shortcomings in the existing bitemporal indices, the next step is to show how these shortcomings may be eliminated by extending the R*-tree. In this section, we present the static structure of the extended R*-tree in two steps. In Section 4.1, we introduce variables NOW and UC in index nodes. This enables the index to record the exact geometry of bitemporal regions in leaf nodes and to record minimum bounding rectangles in non-leaf nodes. In Section 4.2, the use of minimum bounding regions instead of minimum bounding rectangles in non-leaf nodes is suggested.

4.1 Recording the Exact Geometry in Leaf Nodes

By using variables NOW and UC at all tree levels, it becomes possible to record the exact geometry of the bitemporal regions (Section 2) in leaf nodes and minimum bounding rectangles, which grow when the

regions inside them grow, in non-leaf nodes. In comparison with the maximum-timestamp approach, dead space and overlap is much reduced; compare node 2 in Figures 4 and 5(b).

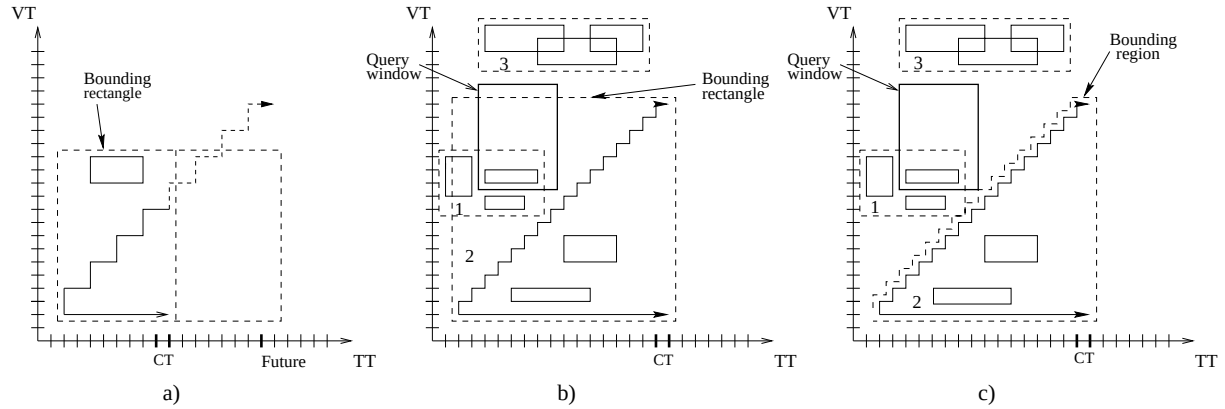

Figure 5: Graphical Representation of a) a "Hidden" Growing Stair-shape, b) a Minimum Bounding Rectangle of Node 2, and c) a Minimum Bounding Region of Node 2

With this extension, the content of tree nodes does not differ significantly from that of the original R-tree. A leaf-node entry contains (1) 4 timestamps encoding a bitemporal region and (2) a pointer to the actual bitemporal data stored in the database. The possible combinations of the 4 timestamps are shown in Figure 3, and they encode the bitemporal regions in Figure 2.

A non-leaf node entry contains (1) 4 timestamps, (2) a flag `Hidden`, and (3) a pointer to a child node. Here, timestamps represent a minimum bounding rectangle that encloses all child-node entries. Note that timestamps (tt1, UC, vt1, NOW) represent a stair-shape in a leaf-node entry, but represent a rectangle growing in both transaction and valid time in an entry of a non-leaf node. A sample tree is given in Figure 6(a).

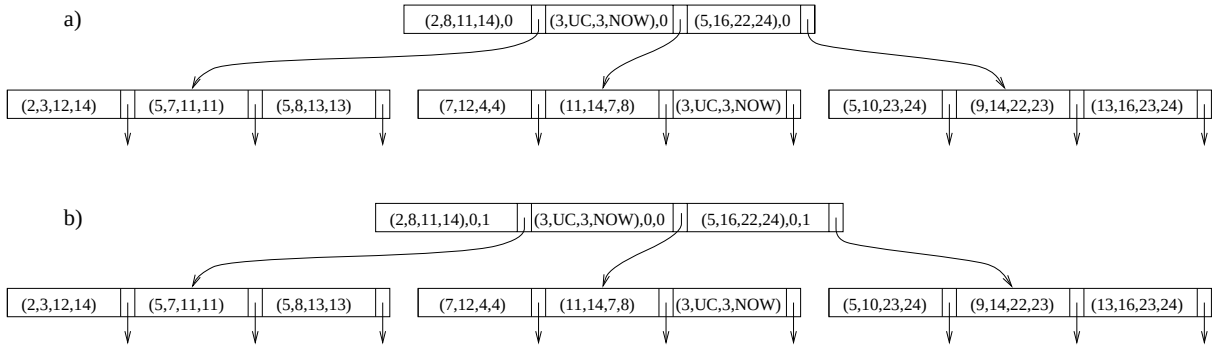

Figure 6: Extended Versions of the R*-tree

The flag `Hidden` is necessary to handle special cases. A small growing stair-shape may be placed together with other regions in a larger bounding rectangle with a fixed valid-time end (that is bigger than the current time). One day, the stair-shape will outgrow its bounding rectangle, making this rectangle invalid, see Figure 5(a). The flag `Hidden` is used to handle such cases. In Section 5, we explain precisely how the tree algorithms manipulate this flag.

Considering properties of the tree, let M denote the maximum number of entries that fit in a node, and let m denote the minimum number of entries that must be in any non-root node. We then have that $m \leq M/2$. In addition, the tree is balanced. We assume that one tree node is stored in one disk page; thus, the retrieval of a tree node requires one I/O operation.

The new tree structure reduces dead space and overlap, but further improvement is possible. Assume

that we want to find all regions that overlap with the query window given in Figure 5(b). The search extends to nodes 1 and 2 since their minimum bounding rectangles overlap with the query window. However, no regions qualify for the answer in node 2.

4.2 Using Minimum Bounding Regions

We take one step further and lift the restriction that the minimum bounding regions in non-leaf nodes be rectangles, and we instead allow entries in non-leaf nodes to also encode all the kinds of bitemporal regions introduced in Section 2. In some cases, it may be reasonable to group stair-shapes together in one node and bound them with a stair-shape instead of a rectangle. Consider the example from the previous section: in Figure 5(c), we see the benefit when the same regions as in Figure 5(b) are bounded with a stair-shape instead of a rectangle. Performing the same search, we now have to access only node 1.

In order to indicate whether the 4 timestamps in an entry of a non-leaf node encode a minimum bounding rectangle or a minimum bounding stair-shape, we introduce a flag, `Rectangle`, in entries of non-leaf nodes. This is needed to separate the situations where we want a `VTend` value of NOW and a `TTend` value of UC to denote a growing stair shape versus a growing rectangle. If a minimum bounding region does not enclose any regions that go above the line $x = y$, we do not want it to be a rectangle. Figure 6(b) shows the extended tree with the `Rectangle` flag. The tree with this node structure, we term the *GR-tree*.

To summarize, we have extended the R*-tree in two steps. We have done this in order to be able to do performance experiments on both the GR-tree and the *intermediate version* of the GR-tree (with minimum bounding rectangles in non-leaf nodes), to see the effect on the search speed and the relevant tree properties (dead space, overlap, and pagination) of the more general regions in non-leaf nodes.

Since the GR-tree node entries not only encode static rectangles, but also encode, e.g., growing stair-shapes, the original tree algorithms must be reconsidered.

5 Index Algorithms

The algorithms to accompany the new index structure with bitemporal regions are based on the R*-tree algorithms, which must be modified because they were designed only for static rectangles. In Section 5.1, before covering the basic index algorithms, we present underlying algorithms that are used in all operations on the index. Section 5.2 describes the algorithm for choosing the node for a new entry and, the rest of Section 5 studies the modified Split and RemoveTop algorithms.

5.1 Search, Deletion, and Insertion

Search, deletion, and insertion are the main operations on the tree. These utilize a new suite of underlying algorithms capable of manipulating the new bitemporal regions (e.g., recall Figure 2), encoded using flags and timestamp variables.

When an entry in a tree node is accessed, the following algorithm is used to adjust the `VTend` value of the encoded region.

```
IF flag Hidden is set AND VTend is fixed and less than the current time
THEN set VTend to NOW
```

The shape of the region encoded by the entry can be determined from the `Rectangle` flag (if the entry is in a non-leaf node) or from the variable NOW (if the entry is in a leaf node). The variable UC indicates whether the region is static or growing. We utilize this knowledge in the algorithms described below, but the actual values of UC and NOW, i.e., the top-right corner of the region, are also needed. These values depend on the current time and can be found in the following way:

```
IF TTend is equal to UC THEN set TTend to the current time
IF VTend is equal to NOW THEN set VTend to TTend
```

The set of underlying algorithms includes algorithms that compute whether a pair of regions overlap and whether one region contains another region, algorithms that compute the area and margin of a region, the distance between the centers of minimum bounding rectangles of two regions, the intersection of a pair of regions, and the minimum bounding region of a node. These algorithms manipulate the flags and timestamp variables in the tree nodes. With these underlying algorithms in place, we can turn to the algorithms for search, insertion, and deletion.

The R*-tree algorithm for search [BEC90] scans the tree, evaluating the predicate given in the query (e.g., equality, overlap) on the query window and the regions encoded by the index-node entries. This algorithm is well suited also for the GR-tree—it just has to be based on the new underlying algorithms.

Deletion in the R*-tree is done in the following way: if a node from which an entry is deleted gets underfull, all other entries from that node are deleted and are re-inserted into the tree at the same level. Thus, the insertion algorithm is responsible for maintaining a good structure of the tree. In the sequel, we adapt this scheme to the GR-tree.

The R*-tree insertion algorithm [BEC90] first invokes the ChooseSubtree algorithm to find an appropriate node in which to place the new entry. If the selected node already contains M entries, the OverflowTreatment algorithm is invoked. If, during the insertion of the new entry, this is the first call of OverflowTreatment at the given level of the tree, the RemoveTop algorithm is invoked; otherwise the Split algorithm is invoked. The RemoveTop algorithm removes p entries from a node and reinserts them. In the worst case, all these entries are reinserted into the same node or they overflow some other node. In these cases, OverflowTreatment is called again, and this time it invokes the Split algorithm. The split of a node can result in overflow of the parent node. If this happens OverflowTreatment is called for the parent node. Experiments show that $p = 30\%$ yields the best performance [BEC90].

Since the R*-tree was designed for static rectangles, the criteria according to which (1) a relevant node is selected (ChooseSubtree), (2) p entries for removal are selected (RemoveTop), and (3) the entries of the overfull node are split into two nodes (Split) are likely to not be efficient for bitemporal regions, which can be growing and of different shapes.

5.2 Choosing the Node to Place a New Entry

The ChooseSubtree, RemoveTop, and Split algorithms try to ensure that the tree structure is as good as possible at the current time, be it by selecting an appropriate node or appropriately dividing entries of a node into two nodes.

The ChooseSubtree algorithm places a new entry in the tree. It starts at the root node and traverses the tree. At each visited node, the algorithm places a new entry in the subtree where the placement of the entry leads to the least enlargement of the overlap between all the bounding regions of the subtrees of the node.

To determine the overlap enlargement when placing an entry in a subtree, the overlap between the subtree's minimum bounding region, not including the new entry, and the minimum bounding regions of all the other subtrees is determined. Then the overlap, when the minimum bounding region of the subtree is extended with the new entry, is determined, and the overlap enlargement resulting from the placement of the entry is found. The subtree, or node, where including the new entry yields the *least overlap-area enlargement* is selected. For example, in Figure 7, the minimum bounding region of node 2 requires the smallest overlap-area enlargement when accommodating the new entry. Ties are resolved by choosing the node whose minimum bounding region requires the *least area enlargement* when including the new entry, and further ties are resolved by choosing the node whose minimum bounding region has the *smallest*

area with the new entry enclosed.[2] The Split and RemoveTop algorithms solve similar problems and are discussed in the subsequent sections.

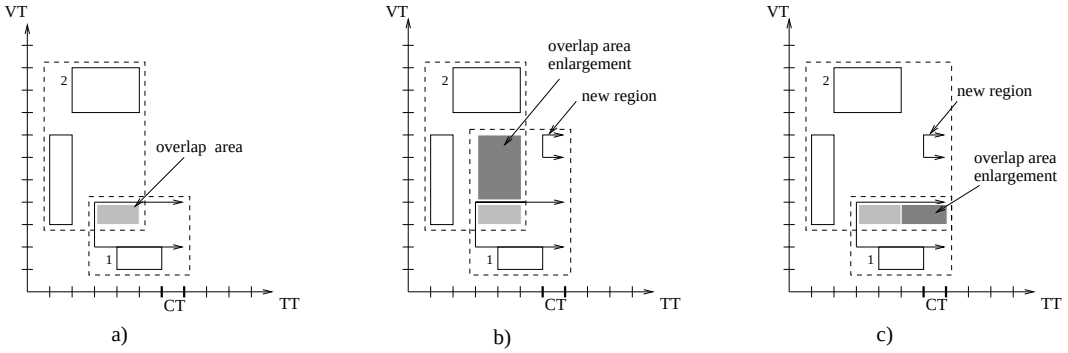

Figure 7: Overlap Between Two Minimum Bounding Regions (a) Before Insertion of a New Entry, (b) After Insertion of a New Entry into Node 1, and (c) After Insertion of a New Entry into Node 2

In the GR-tree, the indexed regions may be constant or functions of time, and the regions of internal nodes are capable of tracking the indexed regions, making the internal regions also functions of time. This implies that quantities such as overlap and dead space are functions of time. This leads us to introduce a *time parameter* in the tree algorithms.

For example, in the case of the ChooseSubtree algorithm, the time parameter allows us to compute the overlap-area enlargement not only as of the current time, but also as of some later time. A growing entry placed under some node may yield the smallest overlap-area enlargement at the current time, but this enlargement may not remain the smallest when time passes, because the growing entry in a node forces its minimum bounding region to also grow. It may be better to place the entry in a node that is not the best at the current moment, but it may be the best after some time. For example, Figure 7 shows that it is better to include the new entry in node 2, while Figure 8 illustrates that it is better to insert the new entry in node 1, because the overlap of the two minimum bounding regions then remains constant as time passes.

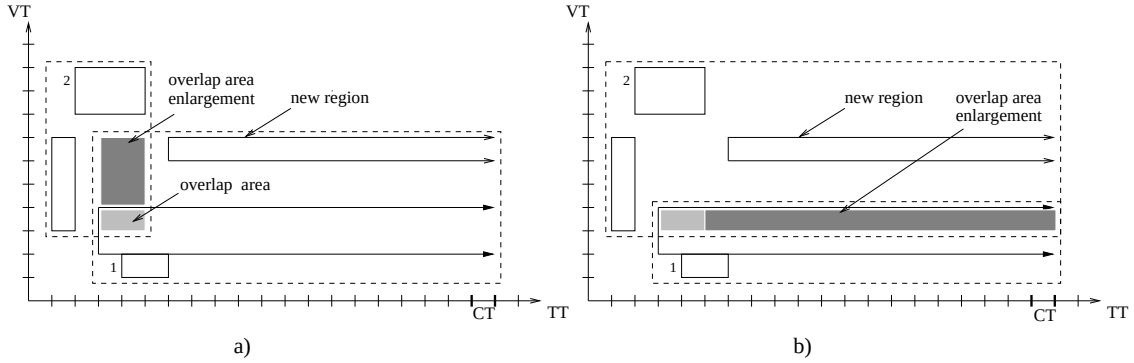

Figure 8: Overlap Between Two Minimum Bounding Regions After Some Time: a) Node 1 with a New Entry; b) Node 2 with a New Entry

The extended ChooseSubtree, RemoveTop, and Split algorithms that use the time parameter do not differ substantially from the corresponding original R*-tree algorithms. They use the underlying algorithms introduced in Section 5.1 for performing operations such as intersection and overlap as of the time specified by the time parameter.

[2]The algorithm differs slightly for leaf and non-leaf nodes. For non-leaf nodes, overlap area enlargement is not considered—only area enlargement and area are considered.

The underlying algorithms ensure that the Split, RemoveTop, and ChooseSubtree algorithms work correctly, and the introduction of the time parameter should improve their capability to handle bitemporal regions. The performance study in Section 6.2 considers which specific time parameter values to use in invocations of the algorithms. Still, the algorithms do not fully exploit the knowledge of the specific shapes of bitemporal regions.

5.3 Types of Bitemporal Regions in an Overfull Node

While we have not found any ways of improving the ChooseSubtree algorithm beyond generalizing it to use the time parameter, several options exist for improvement of the Split and RemoveTop algorithms, both of which are called on an overfull node.

It is useful to examine all the cases that algorithms must address in terms of types of bitemporal regions in the overfull node. From the point of view of the Split and RemoveTop algorithms, there are five such different types of regions.

1. Static rectangles.
2. Static stair-shapes.
3. Rectangles growing in one direction (with variable UC).
4. Rectangles growing in both directions (with variables UC and NOW, and the `Rectangle` flag set)[3]
5. Growing stair-shapes (with variables UC and NOW, and the `Rectangle` flag not set).

A node may contain different combinations of the different kinds of regions, ranging from the case where the node contains only static rectangles to the case where the node contains a mix of all the five types of regions. This yields a total of $C_5^1 + C_5^2 + C_5^3 + C_5^4 + C_5^5 = 31$ cases that the Split and RemoveTop algorithms have to contend with.

The Split algorithm must also observe the restriction that no node should hold less than m entries. It may be advantageous to devise split policies that force regions of the same type to be put in the same node instead of distributing them between two nodes. But if there is more than $M - m + 1$ regions of the same type in an overfull node, such a split is not possible. More generally, if the node contains n types of regions ($1 < n \leq 5$), none or one of these types can exceed the $M - m + 1$ limit, giving us $n + 1$ different cases. Thus, instead of 31 cases, we have $5 + C_5^2 \cdot (2 + 1) + C_5^3 \cdot (3 + 1) + C_5^4 \cdot (4 + 1) + C_5^5 \cdot (5 + 1) = 5 + 10 \cdot 3 + 10 \cdot 4 + 5 \cdot 5 + 1 \cdot 6 = 106$ cases. The Split and RemoveTop algorithms should explicitly or implicitly contend with all these cases.

5.4 The Original R*-tree Split Algorithm

As a precursor to presenting the Split algorithms for the GR-tree, we review the original R*-tree Split algorithm [BEC90]. This algorithm investigates a subset of all the possible distributions of entries into two nodes and finds the best distribution according to three heuristics.

1. The sum of the margins of the resulting bounding rectangles (*margin-value* of the distribution) should be as small as possible.
2. The sum of the areas of the resulting bounding rectangles (*area-value* of the distribution) should be as small as possible.
3. The overlap between the resulting bounding rectangles (*overlap-value* of the distribution) should be as small as possible.

[3] These regions can only exist in non-leaf nodes.

The subset of all possible distributions to investigate is selected in the following mode. Along each of the two axes, entries of the overfull node are sorted according to their bottom and top values, i.e., according to the `VTbegin` and `VTend` values for the valid-time axis and according to the `TTbegin` and `TTend` values for the transaction-time axis. Then, for each of the four sortings, the algorithm investigates $M - 2m + 2$ distributions. The i-th distribution is generated by assigning the first $m - 1 + i$ entries of the sort to the first node and the rest to the other. The R*-tree Split algorithm is divided into two steps. Using the first heuristic above, one axis is selected. Then, the last two heuristics are used considering only the distributions along this axis.

The Original R*-tree Split Algorithm

RS1 For each axis: (1) sort the entries by the lower then by the upper value of their rectangles and determine all distributions as described above; (2) compute S, the sum of margin-values of all the distributions for the axis.

RS2 Choose the axis with the minimum S as the split axis.

RS3 Along the chosen split axis, choose the distribution with the minimum overlap-value. Resolve ties by choosing the distribution with the minimum area-value.

Returning back to the cases that the Split algorithm must contend with, the R*-tree Split algorithm could be used without changes for the two trivial cases where all entries are static rectangles or all entries are static stair-shaped regions. To handle the rest of the cases, two approaches could be taken. First, the original R*-tree Split algorithm could be modified so that it investigates additional distributions, but uses the same set of heuristics. Second, additional heuristics could be introduced that target explicitly the growing and stair-shaped entries. Subsequent sections pursue both approaches.

5.5 The Additional-Sorts Split Algorithm

The R*-tree Split algorithm considers distributions of entries based on four sorts. More distributions may be considered by introducing additional sorts, and this may be advantageous because the new sorts could implicitly address the differences between rectangles and stair-shapes.

The additional-sorts Split algorithm first calls the original R*-tree Split algorithm[4] and then investigates additional distributions based on two more sorts. In the first sort, entries are sorted by their `VTend - TTbegin` value, where `VTend` is set to the appropriate fixed value if it was NOW. Value `VTend - TTbegin` expresses how far the upper-left corner of the region is from the axis $y = x$. For stair shapes, the value 0 is used instead of `VTend - TTbegin`, because the stairs of stair-shaped regions always lie on the axis $y = x$. In the second sort, the lower-right corners of the regions are used, i.e., entries are sorted by `VTbegin - TTend`. The algorithm is sketched next.

The Additional-Sorts Split Algorithm

ASS1 Invoke the original R*-tree Split algorithm.

ASS2 If all entries are static rectangles or all entries are static stair-shapes, exit.

ASS3 Sort the entries by (`VTend - TTbegin`) and by (`VTbegin - TTend`). Determine all distributions as described above.

ASS4 Among the distributions generated in ASS3 and the one chosen in ASS1, select the one with the minimum overlap-value. Resolve ties by choosing the distribution with the minimum area-value.

The additional-sorts Split algorithm relies fully on the time parameter to separate growing entries from static ones and on the investigation of the additional distributions to achieve an assignment of stair-shaped bounding region to one node, if it is possible. Another approach is to be more explicit.

[4] When using the original R* -tree algorithm for the GR-tree, we assume that it uses the new underlying algorithms.

5.6 The Additional-Heuristics Split Algorithm

The idea behind the additional-heuristics Split algorithm is to more explicitly address stair-shaped and growing regions. Three new heuristics may be used when there is at least one growing region in an overfull node (otherwise, we believe that the additional-sorts Split algorithm can be effectively used).

1. Distribute entries so that only one node has to be bounded with a growing region.
2. Distribute entries so that none or only one node has to be bounded with a region growing in both directions.
3. Distribute entries so that none or only one node has to be bounded with a rectangle growing in both directions.

The heuristics are inter-related: the first one is the most restrictive and implies the other two, and the second implies the third. Thus, if heuristic (1) is satisfied (all growing regions are placed in one node), heuristics (2) and (3) are satisfied. On the other hand, if heuristic (1) fails (the number of growing regions is bigger than $M + 1 - m$), the other two heuristics can still be satisfied. The next split algorithm uses these heuristics.

The Additional-Heuristics Split Algorithm

AHS1 If all entries are static, invoke the additional-sorts Split algorithm and exit.

AHS2 Attempt to satisfy heuristic (1) (try to put all growing regions into one node).

AHS3 If heuristic (1) fails, attempt to satisfy heuristic (2) (try to put all regions growing in both directions into one node).

AHS4 If heuristics (2) fails, attempt to satisfy heuristic (3) (try to put all rectangles with their upper-left corners above the axis $y = x$ into one node).

AHS5 If all heuristics failed, invoke the additional-sorts Split algorithm and exit; otherwise (we have one node with growing entries), apply Guttman's quadratic *Distribute* algorithm [GUT84] to distribute the remaining entries. Use as the "seed" entry for the empty node the entry from the remaining entries such that its inclusion in the first node would enlarge that nodes minimum bounding region the most.

Note that we use the time-parameterized version of Guttman's *Distribute* algorithm. This means that the last heuristic applied is "Minimum bounding regions with as small as possible area after *t* time units."

In the algorithm above, we have tried to apply the most restrictive heuristics first, but it is possible, and may be advantageous, to give priority instead to a less restrictive heuristic. To exemplify, let us analyze the situation where we have to split an overfull node and where the number of growing entries is small enough that they fit in one node. Let us also suppose that there are no minimum bounding rectangles growing in both directions among those growing entries, but that there are some growing rectangles with their upper-left corners above the axis $y = x$ and some growing stair-shapes. Thus, the growing node resulting from the algorithm above will be growing in both directions and will be rectangular. Instead, we could choose to let both split-generated nodes be growing, but with none of them having minimum bounding rectangle growing in both directions. This can be achieved by placing the rectangular regions above $y = x$ in one node and the stair-shaped growing regions in the other. In this case, we have prioritized heuristic (3) over heuristic (1).

This exemplifies that changing the priority of the heuristics in the additional-heuristics Split algorithm can produce different results. Other possible variants of the additional-heuristics Split algorithm may be obtained by omitting one or two of the heuristics from the algorithm. In the performance experiments we study a total of nine variants of the algorithm.

5.7 The RemoveTop Algorithm

We present three variants of the RemoveTop algorithm; their performance characteristics are studied later in the paper. The RemoveTop algorithm is expected to in some way identify the "worst" entries of an overfull node for reinsertion. RemoveTop is thus similar to the Split algorithm in the way that it has to separate entries of an overfull node into two groups. The difference from the Split algorithm is that these groups must have predefined numbers of entries. This means that a slight modification of the Split algorithm can serve well as the RemoveTop algorithm.

The second RemoveTop algorithm that we include is the original R*-tree RemoveTop algorithm, which sorts the entries by the distances of their centers from the center of the minimum bounding rectangle of the overfull node and chooses to remove the p entries with the largest distances. Adapting this algorithm to the GR-tree, we use the centers of the bounding rectangles of the entries.

The third RemoveTop algorithm that we consider removes entries that, when removed, shrink the area of the minimum bounding region of the node the most. We could employ an algorithm of quadratic complexity which, after removing each entry, scans all the rest to find the next one to remove.

5.8 Algorithms for the Intermediate GR-Tree and Maximum-Timestamp Approaches

We have presented the algorithms for the GR-tree. The intermediate GR-tree is simpler: although it employs the same general bitemporal regions in its leaf nodes as does the GR-tree, it uses only static and growing rectangles in its non-leaf nodes. This means that the new underlying algorithms for overlap, area, containment, margin, and distance computations must be used (see Section 5.1) for leaf nodes. But other algorithms, for example, algorithms to compute the intersection of a pair of regions and to compute the minimum bounding region of a node, are simpler than those for the GR-tree. The intersection algorithm is invoked for non-leaf nodes only, and therefore gets only rectangles as an input (the original R*-tree intersection algorithm can be used), and the algorithm computing the minimum bounding region of a node produces only rectangles as results.

As for the GR-tree, we will use our proposed Split and RemoveTop algorithms along with the original R*-tree ones for the intermediate GR-tree in order to choose the best pair of Split and RemoveTop algorithms for the tree.

In the next section, we do performance studies for the GR-tree, the intermediate GR-tree, and the maximum-timestamp approach using one R*-tree and two R*-trees. Both maximum-timestamp approaches use the original R*-tree algorithms. The only additional computation needed is to check whether found leaf-node entries actually qualify for the answer of a query (recall the example in Section 3).

6 Performance

In this section, we perform a series of experiments aimed at exploring the performance and other characteristics of the bitemporal indices that support now-relative bitemporal data. The section is divided into several parts, each conducting a separate mini-study.

First, Section 6.1 discusses data and query generation. Section 6.2 presents a study aimed at choosing the optimal time-parameter value and the best pairs of Split and RemoveTop algorithms for the GR-tree and the intermediate GR-tree. This sets the stage for a comparison, in Section 6.4, of the performance of the two tuned GR-trees and the straightforward maximum-timestamp approach using one R*-tree (1-R) and two R*-trees (2-R). In addition, we measure pagination, dead space, and overlap in all four indices. Section 6.3 precisely defines these properties.

6.1 Data and Query Generation

The four indices were implemented using the Generalized Search Tree Package, GiST [HNP95]. To fairly compare search and update performance on the indices, the same data has to be inserted into the trees and the same queries have to be run on them.

We use so-called *workloads* to simulate the construction and usage of an index for a certain period, termed the index life-time. In our experiments, a workload typically contains 60,000 update operations. An update operation is either an insertion or a (logical) deletion, and these may insert or (logically) delete a bitemporal region into or from a tree, respectively. One update operation occurs at each point in the life-time. First, we perform 4000 inserts in a sequence. After that, insertions occur with probability 0.58 and deletions occur with probability 0.42.[5]

When inserting regions, we use several parameters. We choose the valid-time begin of a bitemporal region to be strongly bounded to the insertion time of a region. Specifically, it is normally distributed with a mean equal to the insertion time and with some deviation, *Dev*, that specifies how densely regions are spread around the $y = x$ axis. If *Dev* is very big, regions are scattered throughout the valid-time universe. The valid-time interval length is uniformly distributed between 0 and the maximum valid-time interval length, *VL*. Alternatively, the valid-time end can be NOW, i.e., regions can be stair-shapes. The percentage of stair-shaped regions to be inserted in an index is denoted as *SS*.

A workload also contains queries intermixed with the update operations. We perform bitemporal range queries, i.e., queries with specified valid- and transaction-time intervals, where the maximum interval length is *maxQI*. (If *maxQI* is 0, we get bitemporal point queries.) We use *overlap* as the query predicate, meaning that data regions that overlap with the given query window qualify for the result.

The data and query generation parameters described above are termed *workload parameters* and are defined in Table 1. They are given different values in different experiments, and specific values are given together with each concrete experiment.

Parameter	Description	Used values	Used as average value
SS	percentage of stair-shaped regions in the index	0, 20, 40, 60, 80, 100	60
Dev	deviation of `VTbegin`, when the mean is the insertion time	10, 30, 70, 120	30
VL	maximum valid-time interval length	50, 100, 300, 600	100
maxQI	maximum valid- and transaction-time interval given in query	0, 5, 100, 500, 3000	100

Table 1: Workload Parameters

Since the probability of insertion is bigger than the probability of deletion, long-lived regions dominate in the constructed tree. In the insertion, only growing regions are inserted, thus, the number of growing regions in the tree increases as time passes, and the probability for each of them to be (logically) deleted, i.e., to stop grow, decreases. Thus, the average transaction-time interval length of a region is relatively large. We feel that this is a realistic setup.

We intermix queries with update operations in the workload with the aim to measure search performance throughout the entire index life-time. In the experiments, we compute for each used workload the average I/O cost of update and search operations present in that workload.

[5] We also experimented with other values of insertion and deletion probabilities, as well as with different numbers of operations per time point. Since these experiments did not give different results, their description is omitted in this paper.

6.2 Selection of Time-Parameter Value, Split, and RemoveTop Algorithms

We have already seen that the properties, e.g., overlap, that govern the heuristics used in the ChooseSubtree, Split, and RemoveTop algorithms are time-dependent, and this led to the parameterization of these algorithms by time (cf. Section 5.2). The next step is to consider what specific time-parameter values to use during invocations of these algorithms.

If the time-parameter value is set to *t*, the algorithms aim to achieve a tree that is at its best as of *t* time units after the current time. But only the data present in the index at the current time is considered. In practice, the tree is queried and new regions are inserted and existing ones deleted all the time. So the objective is to find a time-parameter value that yields the best average search performance throughout the entire index life-time.

We have carried out extensive studies of the GR-tree with the goals of understanding how different time-parameter values affect the performance of the tree at different times and for varying workloads and of identifying an overall good time-parameter value.

The results of the experiments show, that there is no single time-parameter value that works best in all cases. However, when the time parameter is set to 0, the search I/O cost of the resulting tree is always the biggest. This is especially visible when the percentage of stair-shaped regions (*SS*) in the tree is low (Figure 9) or the regions are strongly bounded to their insertion time (a low *Dev* value). In the remaining studies, we have chosen a time-parameter value of 10,000 for the GR-tree and the intermediate GR-tree because values around 10,000 consistently showed good average case performance.

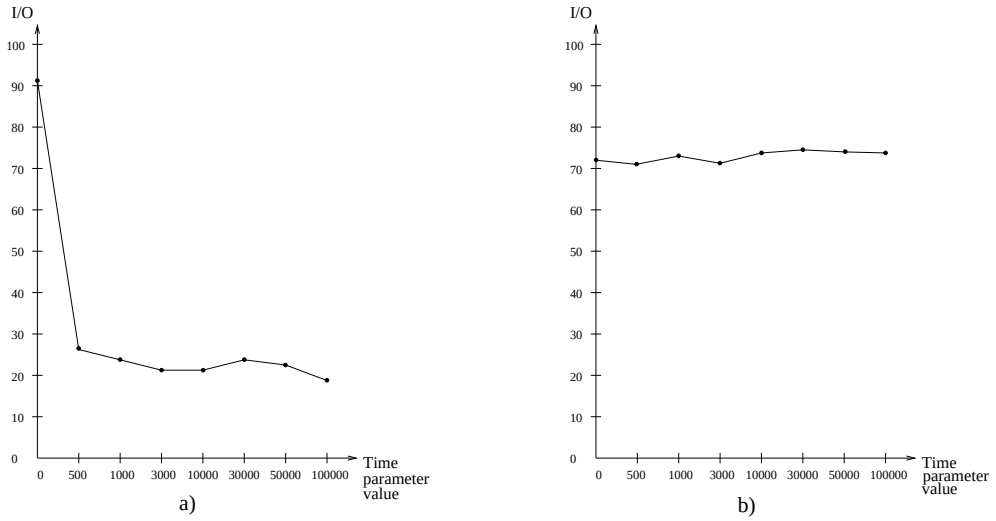

Figure 9: Search I/O Cost for Trees Constructed Using Different Time-Parameter Values and Using Workload with *Dev* = 30, *VL* = 100, *MaxQI* = 5, *SS* = 20 (a), 80 (b)

Another set of experiments was carried out to select the best Split algorithm for the GR-tree. The time parameter value was fixed at 10,000, and the original R*-tree RemoveTop algorithm was used. Then different split algorithms were used during insertion.

A total of twelve split algorithms were investigated: nine flavors of the additional-heuristics algorithms, two flavors of the additional-sorts algorithm, and the original R*-tree Split algorithm. The GR-tree was tested using four sets of workloads with varying values of *SS*, *Dev*, *VL*, and *maxQI*.

Based on these experiments, three split algorithms were chosen for further investigation: the original R*-tree Split algorithm, the additional-sorts Split algorithm (see Section 5.5), and the additional-heuristics Split algorithm that prioritizes heuristic (3) (see Section 5.6). Each of the selected split algorithms was combined with two new RemoveTop algorithms: the split-like algorithm and the quadratic algorithm (the performance

using the original R*-tree RemoveTop algorithm was already measured in the first set of experiments, where the twelve split algorithms were investigated). The experiments were performed using the same sets of workloads. Figure 10 shows the search performance on the trees constructed using the five most effective combinations of Split and RemoveTop algorithms.

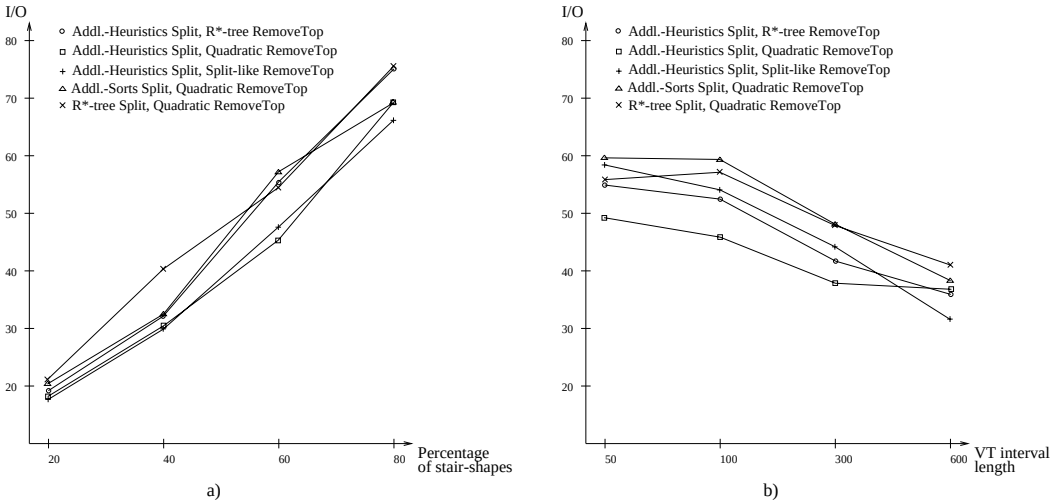

Figure 10: Search I/O Cost for Trees Constructed Using Different Combinations of RemoveTop and Split Algorithms and Using Workloads with: a) varying *SS*, *Dev* = 30, *VL* = 100, and *maxQI* = 100; b) varying *VL*, *SS* = 60, *Dev* = 30, and *maxQI* = 100

The results of the experiments suggest that the combination of the additional-heuristics Split algorithm and the quadratic RemoveTop algorithm should be chosen for use in the GR-tree insertion algorithm. (The combination of the additional-heuristics Split algorithm and the split-like RemoveTop algorithm is not substantially worse.)

To see the overall gain in search performance achieved by the usage of new Split and RemoveTop algorithms and the time parameter, we ran the same workloads using the original R*-tree Split and RemoveTop algorithms and using time parameter values of 0 and 10,000. Figure 11 shows the substantial gains in search performance achieved by using the new Split and RemoveTop algorithms and a time-parameter value 10,000. Figure 11(a) confirms the importance of the time parameter when *SS* is low.

Similar experiments were performed for the intermediate GR-tree. The best results were achieved using the combination of the additional-heuristics Split algorithm and the split-like RemoveTop algorithm.

6.3 Tree Properties

Along with computing the update and search performance for the two tuned GR-trees and the two maximum-timestamp approaches, we will measure dead space, overlap, and pagination in the four trees. These three properties define the "goodness" and intuitively reflect the applicability of a particular tree. In this section, we define each of these three properties precisely.

The dead space in node N, $DS(N)$, is the difference between the area of a minimum bounding region of node N and the area of the union of all the entries in node N. The dead space at level L can be computed from the dead space of each node at that level. The formula for the dead space in node N is given next.

$$DS(N) \quad = \quad \textit{area}(MBR(N)) - \textit{area}(e_1 \cup e_2 \cup \cdots \cup e_{entries(N)}) \tag{1}$$

The entries of node N are denoted e_1, e_2, $\cdots$, $e_{entries(N)}$, where $entries(N)$ is a function returning the number of entries in node N. Function $MBR(N)$ returns the minimum bounding region of node N.

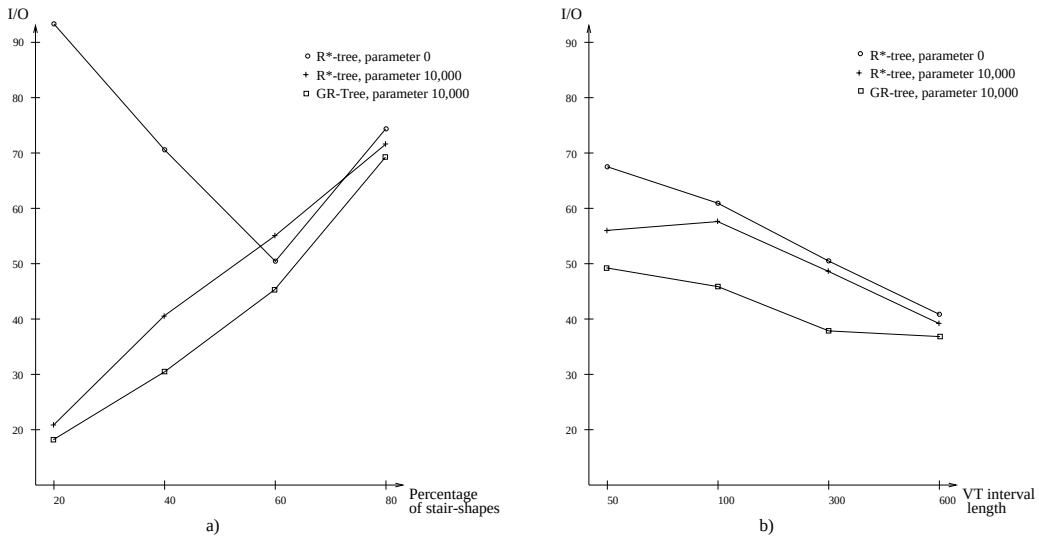

Figure 11: Search I/O Cost for Trees Constructed Using Different Combinations of RemoveTop and Split Algorithms and Different Time Parameter Values and Using Workloads with: a) varying *SS*, *Dev* = 30, *VL* = 100, and *maxQI* = 100; b) varying *VL*, *SS* = 60, *Dev* = 30, and *maxQI* = 100

We define overlap in node N, $OVL(N)$, as the difference of the sum of all areas of node N entries and the area of the union of all node N entries. The overlap at level L is computed from the overlaps of all nodes at level L. We give a formula for the overlap in node N.

$$OVL(N) \quad = \quad (\sum_{e \in N} \textit{area}(e)) - \textit{area}(e_1 \cup e_2 \cup \cdots \cup e_{entries(N)}) \tag{2}$$

We experimented with 1024K size pages. Each tree node is stored in one page, and the maximum number of entries that can fit in one page (*PageCapacity*) is equal to 50. The pagination of tree T is defined as follows.

$$P(T) \quad = \quad \frac{\sum_{N \in T} entries(N)}{\textit{PageCapacity} \cdot \textit{nodes}(T)} \cdot 100\% \tag{3}$$

Function *nodes*(T) returns the number of nodes in tree T. All trees in our experiments have three levels. A three-level tree can store up to $1 + 50 + 50^2 = 2,551$ node, i.e., up to $2,551 \cdot 50 = 127,550$ entries.

It is reasonable to compute dead space of each non-root level. (The root does not have a minimum bounding region and, thus, does not have dead space.) We define the dead space of the tree as the sum of dead spaces at each *non-root* level. In the same manner, the overlap of the tree is defined as the sum of overlaps at each *non-leaf* level. The overlap in the leaf nodes depends on the actual data and does not negatively affect the tree. It should be noticed that dead space and overlap are more harmful at the higher levels of the tree than at the lower levels.

We give the measured numbers of dead space, overlap, and pagination for the GR-trees in the next section.

6.4 Comparison of the Four Bitemporal Indices

Section 6.2 dealt with the tuning of the GR-tree. It was determined that the best GR-tree is achieved using the additional-heuristics Split algorithm that prioritizes heuristic (3), together with the quadratic RemoveTop and a time parameter value of 10,000. For the intermediate GR-tree with minimum bounding rectangles in non-leaf nodes, the split-like RemoveTop algorithm appeared to work better than the quadratic one.

In this section, we compare the tree properties, as well as update and search performance of both tuned GR-trees and the two maximum-timestamp approaches, 1-R and 2-R. We use two sets of workloads: the first with different *SS* values, and the second with different *maxQI* values. Average values are used for the other workload parameters. Figure 12 presents the update and search performance for the trees constructed using both sets of workloads.

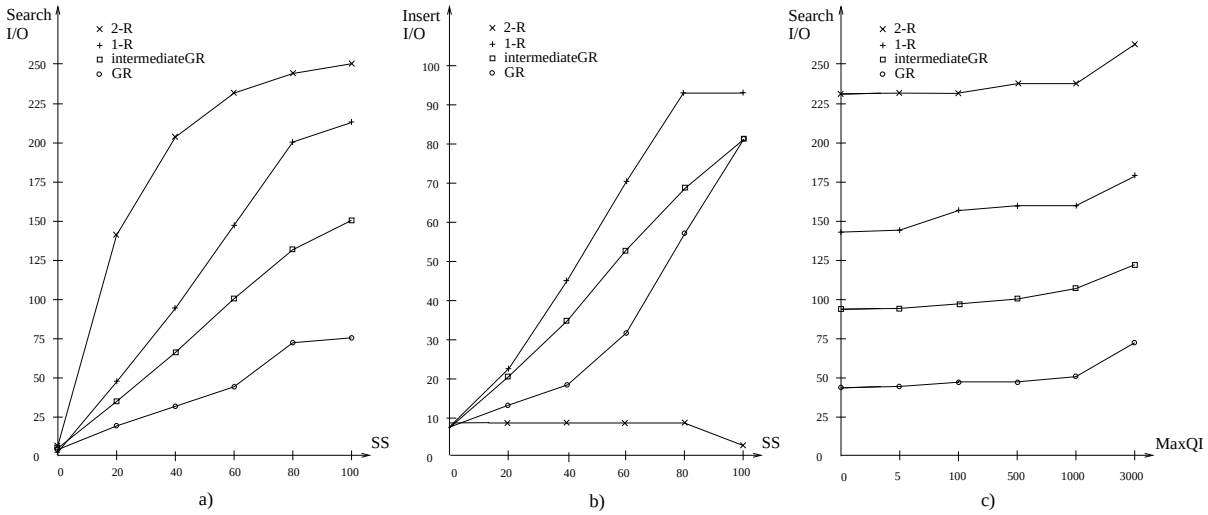

Figure 12: a) Search I/O Cost for Different Trees Using Workloads with Varying *SS*, *Dev* = 30, *VL* = 100, and *maxQI* = 100; b) Update I/O Cost for Different Trees Using Workloads with Varying *SS*, *Dev* = 30, *VL* = 100, and *maxQI* = 100; c) Search I/O Cost for Different Trees Using Workloads with *SS* = 60, *Dev* = 30, *VL* = 100, and Varying *maxQI*

Considering search I/O cost, the GR-tree outperforms both the 1-R and the 2-R trees and the intermediate GR-tree. The update I/O cost is the lowest in the 2-R index because there are two trees instead of one. If a region is given, it is easy to identify whether it should be inserted into the front tree or into the back tree. The front and back trees taken separately are smaller than the trees of the other indices. At the same time, two trees negatively affect the search performance because look-ups have to be performed in both.

Pagination is similar in all trees and varies between 70% and 75%, although the 2-R back tree is an exception with a pagination of around 55%.

Dead space and overlap for the two GR-trees are reported in Figure 13. The dead space and overlap increase when the percentage of stair-shapes (*SS*) increases, thereby decreasing the search performance. Note the special case when *SS* = 100, i.e., all regions are stair-shapes. Since the final GR-tree uses minimum bounding regions in non-leaf nodes, dead space becomes very small. Nevertheless, the performance does not get better because the overlap still increases. Dead space and overlap numbers are not given for 1-R and 2-R; they are excessive because they depend on the maximum timestamp selected, which must be very big because it must exceed any possible fixed valid-time end value used throughout the existence of an index.

In summary, we have seen that both GR-trees outperform the maximum-timestamp approaches by a substantial factor (the only exception is the good update performance in the 2-R tree). If we consider only the GR-trees, we can observe that in most cases, using minimum bounding regions instead of minimum bounding rectangles improves search performance by at least a factor of 2 and update performance by up to a factor of 2. Dead space and overlap values increase as the number of stair-shaped regions grow (an exception is dead space in the GR-tree when all regions are stair-shapes), but they are always lower in the GR-tree than in the intermediate GR-tree. We also found that dead space is particularly big in the leaf nodes of the intermediate GR-tree: typically 40% of each node is empty.

Dead Space

SS	The intermediate GR-tree	The GR-tree
0	$0.38 \cdot 10^9$	$0.38 \cdot 10^9$
20	$67.2 \cdot 10^9$	$2.56 \cdot 10^9$
40	$133 \cdot 10^9$	$2.74 \cdot 10^9$
60	$209 \cdot 10^9$	$4.84 \cdot 10^9$
80	$285 \cdot 10^9$	$15.7 \cdot 10^9$
100	$329 \cdot 10^9$	$0.18 \cdot 10^9$

Overlap

SS	The intermediate GR-tree	The GR-tree
0	$1.99 \cdot 10^9$	$1.99 \cdot 10^9$
20	$133 \cdot 10^9$	$66.5 \cdot 10^9$
40	$268 \cdot 10^9$	$124 \cdot 10^9$
60	$420 \cdot 10^9$	$188 \cdot 10^9$
80	$564 \cdot 10^9$	$288 \cdot 10^9$
100	$661 \cdot 10^9$	$334 \cdot 10^9$

Figure 13: Dead Space and Overlap in the GR-trees.

7 Conclusions

Because regular indices such as the B^+-tree are unsuited for indexing temporal data, a number of indices for temporal data have been proposed. None of these support now-relative valid-time intervals, which exist in almost all temporal data models and are natural and meaningful for many kinds of applications. For bitemporal indices based on R-trees, the maximum-timestamp approach is a straightforward solution to the problem. But with this approach, facts with now-relative valid-time intervals are represented using very large rectangles, and the resulting search performance is poor due to excessive dead space in the index nodes and overlap between nodes.

We proposed an extension to the R^*-tree, the GR-tree, which accommodates bitemporal data represented using TQuel's four-timestamp format. Now-relative valid and transaction-time intervals are supported using variables NOW for valid time and UC for transaction time. Index leaf nodes capture the exact geometry of the bitemporal regions of the database facts. Bitemporal regions can be static or growing, rectangles or stair-shapes. We explored two versions of the GR-tree: one using minimum bounding *rectangles* in non-leaf nodes, and one using minimum bounding *regions* in non-leaf nodes.

Because the new index structure not only accommodates rectangles, but also stair-shapes, the existing R^*-tree algorithms are not directly applicable—they have to be extended. We initially defined new underlying algorithms that are used in search, insert, and delete operations on the index. Because dead space and overlap in the GR-trees are functions of time and because the index algorithms utilize these, a time parameter was added to the index algorithms. Also, eleven new Split and two new RemoveTop algorithms that take into account the specific properties of the bitemporal regions were introduced.

The performance studies show that the use of the various time-parameter values and new Split and RemoveTop algorithms does not give significant performance improvements when each is used in isolation. However, using the combination of a time parameter value of 10,000, the additional-heuristics Split algorithm, and the quadratic RemoveTop algorithm yields significant search and update performance improvements compared with the use of the original insertion algorithm (where the time parameter value is 0 and the original R^*-tree Split and RemoveTop algorithms are used). The GR-tree outperforms the straightforward approaches by at least a factor of 3. We also experienced that using minimum bounding regions instead of merely rectangles in non-leaf nodes of the GR-tree yields a significant improvement.

It is possible to elaborate on the paper's idea and introduce more general shapes than stair-shapes in non-leaf nodes. This would require more complex computations and more storage space, but might reduce dead space and overlap enough to further improve the overall search and update performance. It also appears to be possible to integrate the handling of now-relative data into other existing bitemporal indices, such as the 2-R tree, thus avoiding the inefficient maximum timestamp solution. Finally, the theoretical analysis of R-trees is still lightly researched, and analytical studies of GR-trees remains an open topic. The time parameter and its influence on performance introduces new challenges to the analytical studies of R-trees.

Acknowledgements

A project completed during Spring 1997 provided the outset for the studies reported in this paper. Results from that project indicated that it was possible to design an ”R-tree” for now-relative bitemporal data. We thank S. Andersen and P. S. Larsen for their efforts. This research was supported in part by the Danish Technical Research Council through grant 9700780 and by the CHOROCHRONOS project, funded by the European Commission DG XII Science, Research and Development, as a Networks Activity of the Training and Mobility of Researchers Programme, contract no. FMRX-CT96-0056.

References

[BEC90] N. Beckmann et al. The R*-tree: An Efficient and Robust Access Methods for Points and Rectangles. *Proceedings of ACM SIGMOD*, pp. 322–331 (1990).

[BER97] E. Bertino et al. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers (1997).

[CLI97] J. Clifford et al. On the Semantics of “NOW” in Databases. *ACM TODS*, 22(2):171–214 (1997).

[DRI89] J. R. Driscoll et al. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1):86–124 (1989).

[GUT84] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. *Proceedings of ACM SIGMOD*, pp. 47–57 (1984).

[HNP95] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. *Proceedings of VLDB*, pp. 562–573 (1995).

[JAG90] H. V. Jagadish. Linear Clustering of Objects with Multiple Attributes. *Proceedings of ACM SIGMOD*, pp. 332–342 (1990).

[JEN93] C. S. Jensen et al. A Consensus Test Suite of Temporal Database Queries. TR R-93–2034, Department of Mathematics and Computer Science, Aalborg University (1993).

[JEN94] C. S. Jensen et al. A Consensus Glossary of Temporal Database Concepts. *ACM SIGMOD Record*, 23(1):52–64 (1994).

[JS96] C. S. Jensen and R. Snodgrass. Semantics of Time-Varying Information. *Information Systems*, 21(4):311–352 (1996).

[KF94] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. *Proceedings of VLDB*, pp. 500–509 (1994).

[KTF95] A. Kumar, V. J. Tsotras, and C. Faloutsos. Access Methods for Bitemporal Databases. In *Recent Advances in Temporal Databases*, J. Clifford, and A. Tuzhilin (eds), pp. 235–254, Springer-Verlag (1995).

[KTF97] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. TR-3764, Department of Computer Science, University of Maryland (1997).

[NDE96] M. A. Nascimento, M. H. Dunham, and R. Elmasri. M-IVTT: An Index for Bitemporal Databases. *Proceedings of DEXA*, pp. 779–790 (1996).

[SA85] R. T. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. *Proceedings of ACM SIGMOD*, pp. 236–246 (1985).

[SAM90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley (1990).

[SNO86] R. T. Snodgrass. Temporal Databases. *IEEE Computer*, 19(9):35–42 (1986).

[SNO87] R. T. Snodgrass. The temporal query language TQuel. *ACM TODS*, 12(2):247–298 (1987).

[SNO95] R. T. Snodgrass et al. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers (1995).

[SNO96] R. T. Snodgrass et al. Adding Valid Time to SQL/Temporal. ANSI X3H2-96-501r2, ISO/IEC JTC 1/SC 21/WG 3 DBL-MAD-146r2 (1996).

[SRF87] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. *Proceedings of VLDB*, pp. 507–518 (1987).

[ST97] B. Salzberg and V. J. Tsotras. *A Comparison of Access Methods for Temporal Data*. TimeCenter TR-18 (1997).