R-Tree Based Indexing of Now-Relative Bitemporal Data

Rasa Bliujūtė Christian S. Jensen Simonas Šaltenis Giedrius Slivinskas
Department of Computer Science
Aalborg University, Denmark
{rasa, csj, simas, giedrius}@cs.auc.dk

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, leading to bitemporal data. Only little work, based mostly on R-trees, has addressed the indexing of bitemporal 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 bounding regions may be triangular as well as rectangular. New heuristics for the algorithms that govern the index structure are provided. As a result, dead space and overlap, now also functions of time, are reduced. Performance studies indicate that the best extended index is typically 3–5 times faster than the 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 and transaction time 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. 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, and temporal support is finding its way into the SQL standard [Sno96]. 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. Even less research has been done on creating indices that support data with both valid and transaction time, so-called bitemporal data.

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—spatial indices can be adapted for indexing bitemporal data. Several existing proposals [KTF95] [KTF97] are based on the R*-tree [Bec90].

The 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 progressing current time. We term such data *now-relative*. It occurs naturally and frequently. Consider an example where we want to record new employees in a company's database. The time 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 progressing 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, its 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

Proceedings of the 24th VLDB Conference
New York, USA, 1998

valid-time interval is now-relative.

The paper describes how to support now-relative bitemporal data in R-tree [Gut84] 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 and motivates the need for a new bitemporal index. The structures of the proposed R*-tree extensions are given in Section 4, and Section 5 presents algorithms for the insert, delete, and search operations. Section 6 presents performance studies. The final section concludes and points to research directions.

2 Background

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, tuples each have a number of non-temporal attributes and four time attributes: `VTbegin` and `VTend`—the times when the tuple's information became and ceased to be true in the modeled reality; `TTbegin` and `TTend`—the times when the tuple became and ceased to be current in the database.

A tuple is now-relative if its information is valid until the current time or if the tuple is part of the current database state. This is represented by the use of variables, which denote the current time, for the time attributes `VTend` and `TTend` [Cli97]. The variable UC (denoting 'until changed') is used for `TTend`, and the variable NOW is used for `VTend`. Table 1 exemplifies the 4TS format. The time granularity is a month, and the current time (CT) is 9/97.

Table 1: The EmpDep Relation

	Emp	Dep	TTbegin	TTend	VTbegin	VTend
(1)	John	Adv	4/97	UC	3/97	5/97
(2)	Tom	Mgm	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)	Ann	Mgm	5/97	UC	3/97	NOW

Tuple (1) records that the information "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. 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).

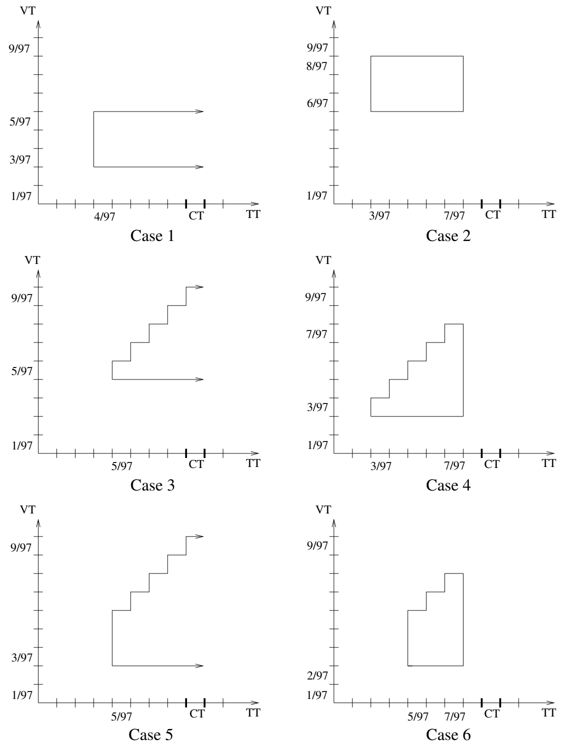

Figure 1: Bitemporal Regions

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 1 illustrate the *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 the transaction-time and the valid-time direction as time passes (Tuple (3), Case 3). Information can be recorded in the database after it becomes true in the modeled reality. In this situation, also having

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

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 information 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 end, which would extend to the current time, would initially be smaller than the valid-time start, 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 1), see Figure 2 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 2: Possible Combinations of Time Attributes

We have set the context for using spatial indices for indexing bitemporal data. The next section discusses the existing indices for bitemporal data that are based on spatial indices.

3 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 1) 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 3.

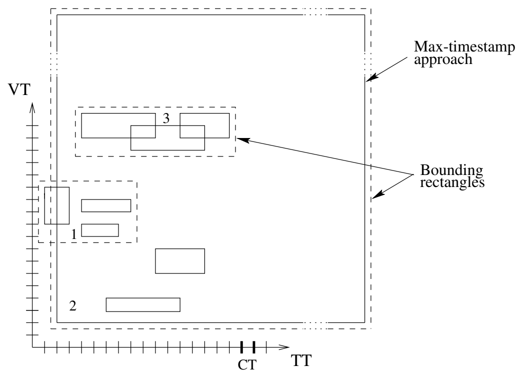

Figure 3: Indexing Growing Bitemporal Regions Using Maximum Timestamp Values

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 rectangles, while the *back* R-tree indexes all static rectangles. Observing that all growing rectangles are in the front tree and that they all end at the (progressing) current time, Kumar et al. show that storing only the transaction-time begin values with fixed valid-time intervals in the front tree is adequate to support now-relative transaction time. The 2-R approach contends well with now-relative transaction time, but both trees often have to be searched in a single query, resulting in more disk accesses and diminishing the advantages of the decreased overlap. The problem of representing now-relative valid time also remains open.

It is 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 3 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

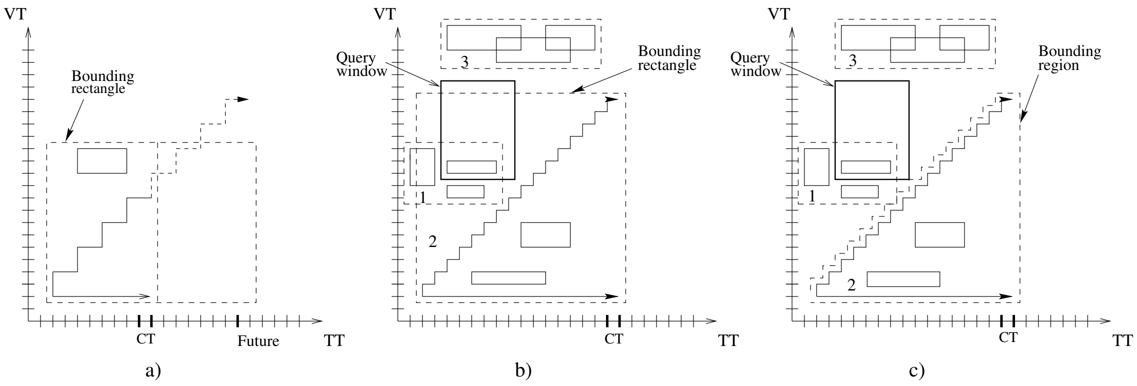

Figure 4: Graphical Representation of a) a "Hidden" Growing Stair-shape, b) a Minimum Bounding Rectangle of Node 2, and c) a Minimum Bounding Region of Node 2

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.

The 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 of 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 these indices. In this section, we extend the static structure of the R*-tree by introducing variables NOW and UC in index nodes.

4.1 Recording Exact Geometries 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 to record minimum bounding rectangles, that 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 3 and 4(b).

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

A non-leaf node entry contains four timestamps, a flag `Hidden`, and a pointer to a child node. Here, the 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 directions in an entry of a non-leaf node. A sample tree, corresponding to Figure 4(b), is given in Figure 5(a).

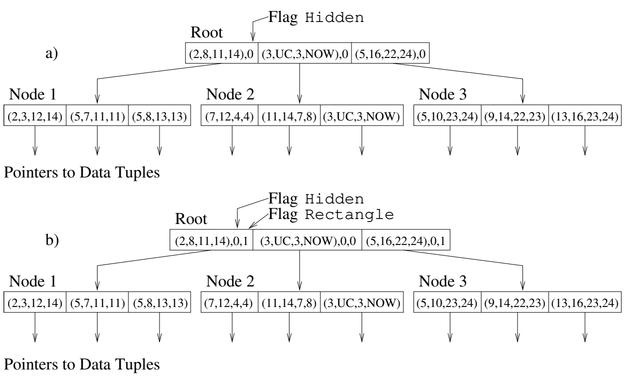

Figure 5: Extended Versions of the R*-tree

A small growing stair-shape may be placed together with other regions in a larger bounding rectangle having 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 4(a). The flag `Hidden` is used to handle such stair-shapes.

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.

The tree structure just described 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 4(b). The search extends to nodes 1 and 2 since their minimum bounding rectangles overlap with the query window, but 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 allow 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 4(c), we see the benefit when the same regions as in Figure 4(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 $y = x$, we do not want it to be a rectangle. Figure 5(b) shows the extended tree with the `Rectangle` flag (cf. Figure 4(c)). 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 performance and the relevant tree properties (dead space and overlap) of the more general regions in non-leaf nodes.

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

5 Index Algorithms

Section 5.1 covers the basic index algorithms and lower-level algorithms. Sections 5.2–5.4 describe in depth the insertion algorithm by covering the original R*-tree algorithm and its improvements including a time parameterization. Section 5.5 briefly describes algorithms for the intermediate GR-tree and the maximum-timestamp approaches.

5.1 Search, Deletion, and Insertion

Search, deletion, and insertion are the main operations on the tree.

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 in the index-node entries.

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.

The R*-tree insertion algorithm first invokes the ChooseSubtree algorithm to find an appropriate node in which to place a 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[2] 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.

These algorithms employ lower-level algorithms that determine whether a pair of regions overlap and whether one region contains another region; and 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.

While the original R*-tree search, deletion, and insertion algorithms are suitable for the GR-tree, new lower-level algorithms, capable of manipulating bitemporal regions (recall Figure 1) encoded using flags and timestamp variables, must be provided. Flags and timestamp variables require special treatment in these algorithms.

The original R*-tree insertion algorithm can be employed for the GR-tree, but 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 be inefficient for bitemporal regions.

5.2 The Original R*-Tree Insertion Algorithm

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 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, parts a)–c) in Figure 6 show that the minimum bounding region of node 2 requires the smallest overlap-area enlargement when inserting a new entry. Ties are resolved by choosing the node whose minimum bounding region requires the *least*

[2]This algorithm implements forced reinsertion, introduced in [Bec90].

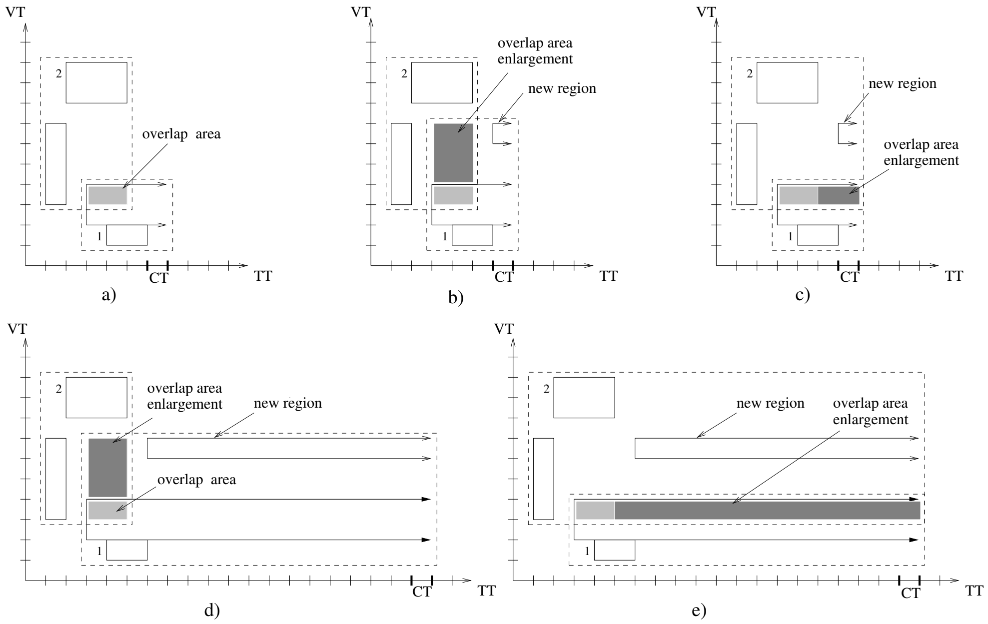

Figure 6: Overlap Between Two Minimum Bounding Regions (a) Before Insertion of a New Entry, (b) After Insertion of a New Entry into Node 1, and (c) After Insertion of a New Entry into Node 2, (d) Case (b) After a Period of Time, (e) Case (c) After a Period of Time

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.[3]

The R*-tree Split 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 overlap between the resulting bounding rectangles (*overlap-value* of the distribution) should be as small as possible.
3. The sum of the areas of the resulting bounding rectangles (*area-value* of the distribution) should be as small as possible.

The subset of all possible distributions to investigate is selected as follows. Along each of the two axes, entries of the overfull node are sorted according to their bottom and top values, i.e., according to `VTbegin` and `VTend` values for the valid-time axis and according to `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 sorting 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.

[3] 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 Original R*-Tree Split Algorithm

RS1 For each axis: (1) sort the rectangles by their lower then by their upper value and determine all distributions as described above; (2) compute S, the sum of margin-values of all the distributions for the axis.
RS2 Let the axis with the minimum S be the split axis.
RS3 Along the split axis, choose the distribution with the minimum overlap-value. Resolve ties by choosing the distribution with the minimum area-value.

The original R*-tree RemoveTop algorithm sorts the entries of the overfull node by the distances of their centers from the center of the minimum bounding rectangle of the overfull node and chooses to remove and reinsert the p percent of the entries with the largest distances. Experiments show that $p = 30\%$ yields the best performance [Bec90].

5.3 Parameterization

The original R*-tree ChooseSubtree, Split, and RemoveTop 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 by appropriately dividing entries of a node into two nodes.

In the GR-tree, the indexed regions may be functions of time. This implies that quantities such as overlap and dead space are also functions of time. This leads to the introduction of a *time parameter* in the 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 in some node may yield the smallest overlap-area enlargement at the current time, but this enlargement may not remain the smallest as time passes, because a growing entry in a node forces the node's 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 may be the best after some time. For example, parts a)-c) of Figure 6 suggest to include the new entry in node 2, while parts d) and e) illustrate that it is probably better to insert the new entry in node 1, because the overlap of the two minimum bounding regions then remains constant as time passes.

The parameterized ChooseSubtree, Split, and RemoveTop algorithms do not differ substantially from the corresponding original R*-tree algorithms. They invoke new lower-level algorithms (Section 5.1) for performing operations such as intersection and overlap as of the time specified by the time parameter.

The time parameter should improve the capability of the insertion algorithm to handle bitemporal regions. The performance study in Section 6.2 considers which specific time parameter values to use invoking the algorithms.

5.4 Improved Algorithms

Beyond the parameterization, other options exist for improving the ChooseSubtree, Split, and RemoveTop algorithms. Special attention can be paid to the different types of bitemporal regions that these algorithms have to contend with. From the point of view of the algorithms, there are four different types of bitemporal regions.

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

We say that an entry is of type *t* if the region represented by that entry is of type *t*; in the same way, a node is of type *t* if its bounding region is of type *t*.

Having in mind the importance of small overlap and dead space in the tree, it is natural to prioritize the first type of nodes as the best and the fourth type of nodes as the worst. Based on this prioritization of node types, a general heuristic that could govern the ChooseSubtree, Split, and RemoveTop algorithms is to group entries into nodes so as to achieve the best types of the nodes possible, keeping the number of bad nodes in the tree as low as possible. This implies that entries of the same type should be grouped together. For example, wanting to achieve the lowest possible number of nodes bounded with growing stair-shapes, we have to group growing stair-shapes together into nodes bounded by growing stair-shapes; distributing them among several nodes is bad because, e.g., inserting a single growing stair-shape into a static node means that this node must be bounded by a growing stair-shape.

5.4.1 The Choose Subtree Algorithm

The original R*-tree ChooseSubtree algorithm considers the overlap and area enlargements when choosing the node in which to insert a new entry. We designed a slightly modified version of this algorithm that uses an additional heuristic, taking into account the type of the new entry and the types of the nodes where the entry can be inserted. The modified ChooseSubtree algorithm selects the group of nodes of the same type where it is the best to insert the new entry. Then, it passes that group of nodes to the original R*-tree ChooseSubtree algorithm[4], which makes its decision according to overlap and area enlargement.

More specifically, the modified ChooseSubtree algorithm first tries to select a group of nodes of the same type such that, when the new entry is inserted in any node of that group, the type of that node will remain the same. If several groups of different type nodes satisfy this condition, the group with nodes of the best type is chosen.

If no groups at all qualify, the new entry will make the type of the chosen node worse. In this case, the algorithm chooses a group of nodes of the same type such that, when the entry is inserted into any node of that group, the type of that node will be worsened the least. If there are several such groups, the algorithm chooses the group of nodes of the worst type.

In the performance studies, both the original R*-tree ChooseSubtree algorithm and the modified ChooseSubtree algorithm, termed the *additional-heuristics ChooseSubtree* algorithm, are tested.

5.4.2 Split Policies

Two approaches to improving the Split algorithm can be taken. First, similarly to the ChooseSubtree algorithm, the Split algorithm could explicitly contend with the different types of entries, trying to achieve good types of resulting nodes. Second, the original R*-tree Split algorithm could be modified so that it investigates additional distributions, but uses the same set of heuristics. We investigate each approach in turn.

Following the heuristic formulated in the beginning of Section 5.4, the *additional-heuristics Split algorithm* tries to split entries of an overfull node into two nodes so that each node may be bounded by a region of the best type possible. At the same time, it tries not to distribute entries of the same type into two different nodes.

Each of the two nodes produced by the split can be bounded by a region of the four types mentioned. There are ten possible pairs of types of the resulting two bounding regions. We prioritize these pairs according to their goodness (see Figure 7). A pair of bounding regions x_1 and x_2 is considered better than a pair of bounding regions y_1 and y_2 if:
$(type(x_1) \neq type(y_1) \vee type(x_2) \neq type(y_2)) \wedge$
$((type(x_1) \leq type(y_1) \wedge type(x_2) \leq type(y_2)) \vee$

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

$(type(x_1) < \max(type(y_1), type(y_2)) \wedge type(x_2) < \max(type(y_1), type(y_2))))$.

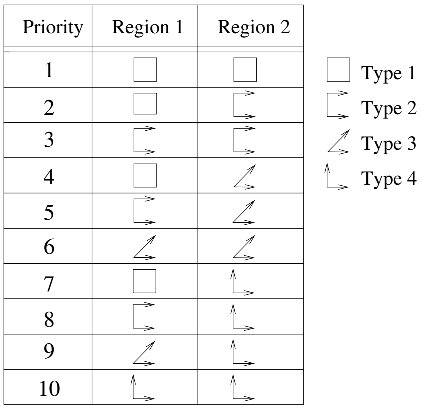

Figure 7: Pairs of Bounding-Region Types

The Additional-Heuristics Split Algorithm

AHS1 From the ten pairs of types of bounding regions, select the pair (t_1, t_2) such that: (a) it is possible to achieve this pair of bounding-region types when dividing the entries of the overfull node into two nodes and (b) no other pair with a higher priority can be achieved. Let the node to be bounded with a region of type t_1 be N_1, and let the node to be bounded with a region of type t_2 be N_2. Let $\mathcal{S}$ contain all entries of the overfull node and let $t_1 \leq t_2$.
AHS2 Move to N_2 all entries from $\mathcal{S}$ that cannot be put into N_1 because of the type of its bounding region. Move to N_1 all entries from $\mathcal{S}$ that cannot be put into N_2 because of the type of its bounding region.
AHS3 Let $\mathcal{S}_t$ denote all entries from $\mathcal{S}$ of type t. For t = 1 to 4, if there are no entries of type t in N_2 and $|\mathcal{S}_t| + |N_1| \leq M - m + 1$, move $\mathcal{S}_t$ into N_1, else if there are no entries of type t in N_1 and $|\mathcal{S}_t| + |N_2| \leq M - m + 1$, move $\mathcal{S}_t$ into N_2.
AHS4 If $|\mathcal{S}| = 0$, stop.
AHS5 If $|N_1| = 0 \wedge |N_2| = 0$, invoke the additional-sorts Split algorithm (to be described shortly) and stop.
AHS6 If $|N_1| = 0$, pick a "seed" entry e from $\mathcal{S}$ for Guttman's quadratic *Distribute* algorithm [Gut84] such that its inclusion intoN_2 would enlarge that node's minimum bounding region the most. Put e into N_1. Goto **AHS8**.
AHS7 If $|N_2| = 0$, pick a seed entry e from $\mathcal{S}$ and put it into N_2.
AHS8 Apply Guttman's quadratic *Distribute* algorithm and stop.

Note that the above algorithm uses a time-parameterized version of Guttman's quadratic *Distribute* algorithm.

We now consider the second approach to improving the original R*-tree split algorithm. The original 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-shapes.

The R*-tree Split algorithm considers distributions of entries based on the four sortings (recall Section 5.2). More distributions may be considered by introducing additional sortings, and this may be advantageous because the new sortings could implicitly address the differences between rectangles and stair-shapes.

The *additional-sorts Split algorithm* given below first calls the original R*-tree Split algorithm and then investigates additional distributions based on two more sortings. In the first sorting, entries are sorted by their `VTend - TTbegin` value, which expresses how far the upper-left corner of the region is from the axis $y = x$. `VTend` is set to the appropriate fixed value if the region encoded by the entry is a growing rectangle and its `VTend` is NOW. 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 sorting, 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 in Section 5.2.
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.

5.4.3 The RemoveTop Algorithm

In addition to the original R*-tree RemoveTop algorithm, alternative RemoveTop algorithms are possible.

First, a RemoveTop algorithm can be induced from the Split algorithm employed, which can be explained as follows. The RemoveTop algorithm is expected to identify in some way the "worst" entries of an overfull node for reinsertion. RemoveTop is thus similar to the Split algorithm in that it has to divide entries of an overfull node into two groups. The difference from the Split algorithm is that these groups must have predefined numbers of entries.

Second, a RemoveTop algorithm of quadratic complexity can be employed. It removes entries that, when removed, shrink the area of the minimum bounding region of the node the most. After removing one entry, this algorithm scans the remaining entries to find the next entry to remove.

5.5 Algorithms for the Intermediate GR-Tree and Maximum-Timestamp-Approach-Based Indices

We have presented the algorithms for the GR-tree. The intermediate GR-tree is simpler: although it records 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 lower-level 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 it therefore gets only rectangles as its

Table 2: Workload Parameters

Parameter	Description	Values used	"Average" value
SS	percentage of stair-shaped regions in the index	0, 20, 40, 60, 80, 100	60
Ins	probability of insertion	50, 60, 70, 80, 90, 100	70
Dev	deviation of `VTbegin`, when the mean is the insertion time	1000, 5000, 10000, 25000, 50000	5000
VL	maximum valid-time interval length	50, 100, 500, 1000, 3000, 5000	500
QmaxI	maximum valid- and transaction-time intervals given in a query	1, 100, 300, 500, 1000, 3000	300

arguments (the original R*-tree intersection algorithm can be used), and the algorithm for computing the minimum bounding region of a node produces only rectangles as its results.

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

In the next section, we do performance studies for the GR-tree, the intermediate GR-tree, and both the R*-tree and the 2-R index using the maximum-timestamp approach. The latter two indices 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 Section 3).

6 Performance

This section reports on a series of experiments aimed at exploring the performance and other characteristics of the indices that support now-relative bitemporal data. Section 6.1 discusses data and query generation. Section 6.2 presents a study aimed at choosing a good time-parameter value and the best combination of the ChooseSubtree, Split, and RemoveTop algorithms for the GR-tree and the intermediate GR-tree. This sets the stage for a comparison, in Section 6.3, of the performance of the two tuned GR-trees, the R*-tree (1-R) and the 2-R index.

6.1 Data and Query Generation

The four indices were implemented using the Generalized Search Tree Package, GiST [HNP95]. The numbers of I/O operations are measured using simulation. The page size is set to 1024 bytes, and one tree node occupies one page. Thus, one node read or write corresponds to one page access (one I/O operation). A buffer of size 100 pages is allocated for each index (for the 2-R index, two buffers of 50 pages are allocated). We include a buffer because Leutenegger and Lopez [LL98] showed that omitting a buffer may lead to quantitatively and qualitatively incorrect conclusions. The root is always kept in the buffer; for the other nodes, the least-recently-used page replacement policy is employed. If a node is changed during an insertion or a deletion, its page is changed in the buffer and is marked as a "dirty" page. Dirty pages are written to disk at the end of the operation or when they have to be removed from the buffer.

To fairly compare search and update performance of 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. One update operation occurs at each point in the life-time. First, we perform 4000 insertions in a sequence. Then, insertions occur with probability *Ins* and deletions occur with probability $1 - Ins$.

When inserting regions, we use several parameters. We let the valid-time begin of a bitemporal region 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*. The valid-time interval length is uniformly distributed between 0 and *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*.

We intermix queries with update operations in the workload, with the aim of measuring search performance throughout the entire index life-time. We use bitemporal range queries (25% of all queries), point queries (25%), and transaction timeslice queries (50%). Besides, 65% of all queries have their transaction-time end equal to the current time. Parameter *QmaxI* denotes the maximum valid-time range for bitemporal range queries and timeslice queries, and the maximum transaction-time range for bitemporal range 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 2. Each of the experiments usually uses different values of one of the parameters and average values of the other parameters.

In the experiments, we compute for each used workload the average number of I/Os performed by the update and search operations present in that workload.

6.2 Tuning the Indices

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 5.3). The next step is to consider which specific time-parameter values to use.

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, new regions are inserted, and existing ones are 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 for varying workloads, allowing us in turn to identify an overall good time-parameter value.

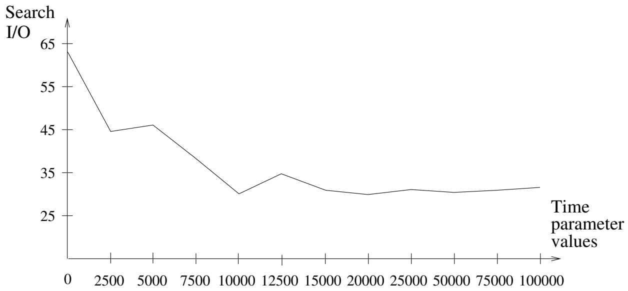

Figure 8: Search I/O Cost for Trees Constructed Using Different Time-Parameter Values and Using a Workload with Average Workload Parameters, but *SS* = 20

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 8) 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.

Another set of experiments was carried out to select the best combination of Split, RemoveTop, and ChooseSubtree algorithms. The original R*-tree, additional-heuristics, and additional-sorts Split algorithms; the original R*-tree, split-like, and quadratic RemoveTop algorithms; and the original R*-tree and additional-heuristics ChooseSubtree algorithms were considered. The three Split algorithms were combined with the three RemoveTop algorithms and with the two ChooseSubtree algorithms. Thus, eighteen combinations of algorithms were investigated in total. The GR-tree was tested using four sets of workloads with varying values of *SS*, *Dev*, *VL*, and *Ins*.

The results of the experiments are shown in Figure 9, where the average number of disk accesses during a search operation is plotted for the GR-trees constructed using different combinations of Split, RemoveTop, and ChooseSubtree algorithms. To see the overall gain in search performance achieved by usage of new algorithms and the time parameter, we also show the results for the GR-trees constructed using the original R*-tree algorithms with the time-parameter value 0.

The combination of the additional-heuristics Split algorithm, the quadratic RemoveTop algorithm, and the additional-heuristics ChooseSubtree algorithm show the best performance. The results also suggest that the additional-heuristics Split algorithm can be substituted by the additional-sorts Split algorithm without sacrificing the search performance. This can be explained by the observation that using the additional-heuristics ChooseSubtree algorithm, after some initial period of tree construction, most nodes of the tree become homogeneous, holding entries of the same type. The additional-heuristics Split algorithm invokes the additional-sorts Split algorithm in such cases (cf. Section 5.4.2).

Similar experiments were performed for the intermediate GR-tree. The best results were achieved using the same combination of the Split, RemoveTop, and ChooseSubtree algorithms and time parameter value 10,000.

6.3 Comparison of the Four Indices

Section 6.2 dealt with the tuning of the GR-tree. In this section, we compare search and update performance of both tuned GR-trees and the two maximum-timestamp-approach based indices, 1-R and 2-R. We use two sets of workloads: the first with varying *SS* values, and the second with varying *QmaxI* values. Figure 10 presents the search and update performance of the trees constructed using both sets of workloads.

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. 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 queries often lead to search in both of them.

The performance of the indices is influenced by the dead space and overlap. When the percentage of growing stair-shapes gets bigger, the overlap in both GR-trees increases more significantly as time passes, thereby decreasing the performance. With a growing percentage of stair-shapes, dead space also increases in the intermediate GR-tree. This does not apply for the GR-tree because it employs strict insertion and splitting policies and uses minimum bounding stair-shapes. In 1-R and 2-R, dead space and overlap are excessive because they depend on the maximum timestamp value, which must be very big in order to exceed any fixed time value used throughout the existence of an index.

In summary, our studies indicate that both GR-trees outperform the maximum-timestamp-approach-based indices by a significant margin (the only exception is the good update performance in the 2-R index). If we consider only the GR-trees, we can observe that in most cases, using minimum bounding regions instead of minimum bounding rectangles improves index performance.

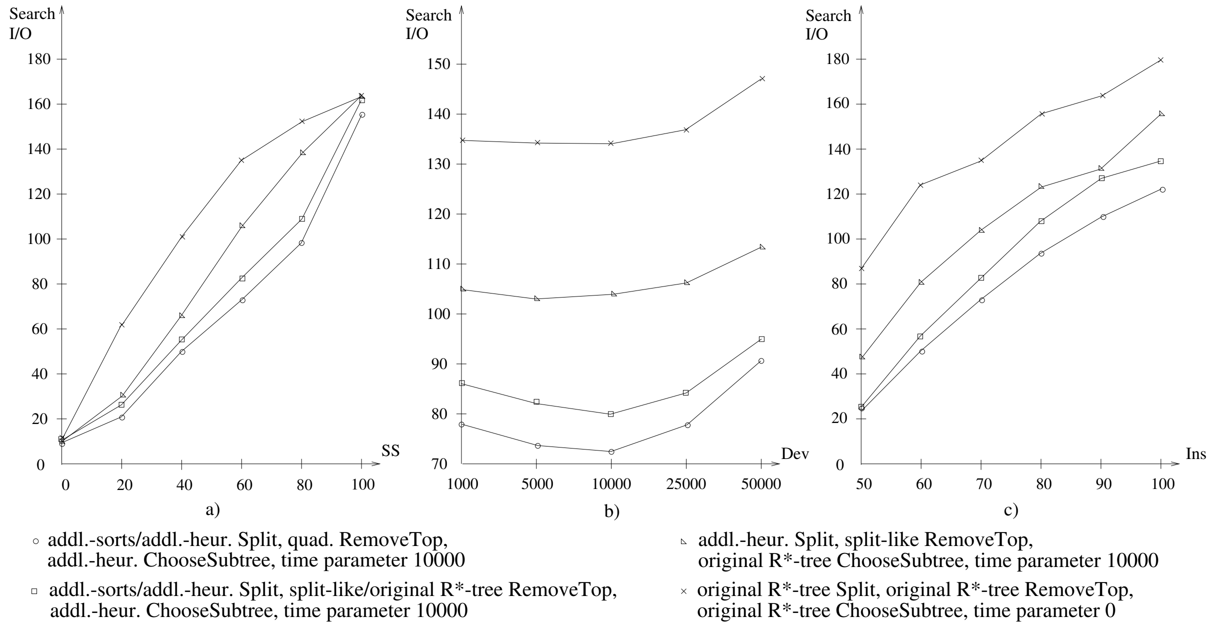

Figure 9: Search I/O Cost for Trees Constructed Using Different Combinations of RemoveTop, Split, and ChooseSubtree Algorithms and Using Workloads with Average Workload Parameters, but (a) Varying *SS*, (b) Varying *Dev*, and (c) Varying *Ins*

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 are accommodated by 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 indexing now-relative data. 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 of the R*-tree, the GR-tree, for general bitemporal data. 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 data. 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.

A new suite of index algorithms was developed to support the new index structure. 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. One new ChooseSubtree, two 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 best combination of the proposed algorithms, with a time parameter of 10000, yields an index that significantly outperforms the index with the original R*-tree algorithms. The GR-tree outperforms the indices using the straightforward approach 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 noticeable improvement.

Currently, we are working on improving the space utilization of the GR-tree. The sequential nature of transaction time can be addressed when performing splits. It is also 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 index, thus avoiding the inefficient maximum-timestamp solution. Finally, the theoretical analysis of R-trees is still lightly researched, making analytical studies of GR-trees a desirable direction. The time parameter and its influence on performance introduces new challenges to the analytical studies of R-trees.

Acknowledgements

This research was supported in part by grants from the European Commission, the Danish Technical Research Council, the Danish Centre for IT Research, and the Nykredit Corporation.

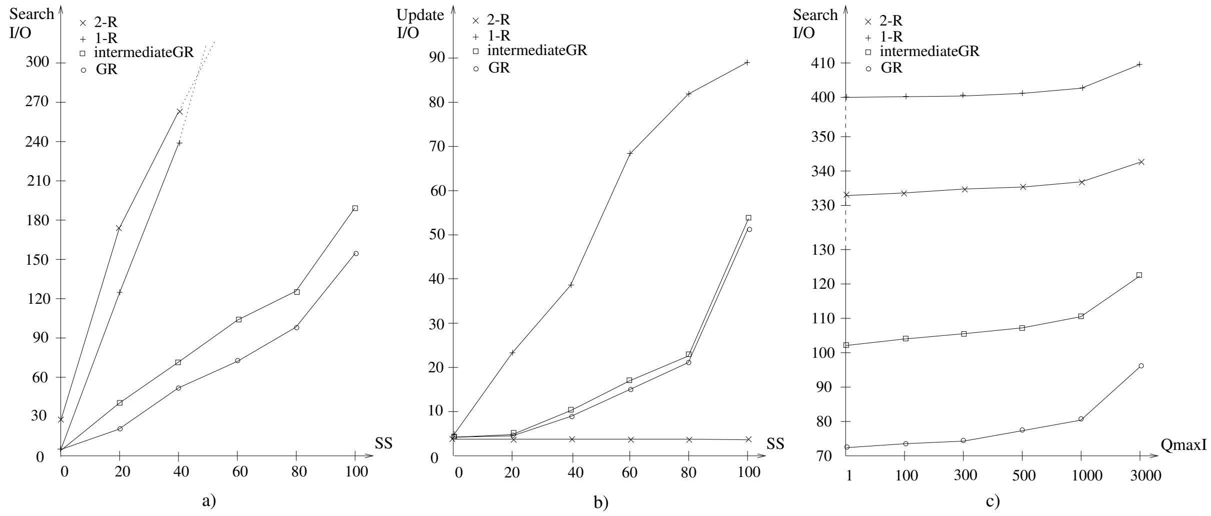

Figure 10: Search I/O Cost for Different Trees Constructed Using Workloads with Average Workload Parameters, but (a) Varying *SS* and (c) Varying *QmaxI*; (b) Update I/O Cost for Different Trees Constructed Using Workloads with Average Workload Parameters, but Varying *SS*

References

[Bec90] N. Beckmann et al. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *ACM SIGMOD'90*, pp. 322–331.

[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. *ACM SIGMOD'84*, pp. 47–57.

[HNP95] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. *VLDB'95*, pp. 562–573.

[Jag90] H. V. Jagadish. Linear Clustering of Objects with Multiple Attributes. *ACM SIGMOD'90*, pp. 332–342.

[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. *VLDB'94*, pp. 500–509.

[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, Dept. of Comp. Sci. University of Maryland (1997).

[LL98] S. T. Leutenegger and M. A. Lopez. The Effect of Buffering on the Performance of R-Trees. *ICDE'98*, pp. 164–171.

[NDE96] M. A. Nascimento, M. H. Dunham, and R. Elmasri. M-IVTT: An Index for Bitemporal Databases. *DEXA'96*, pp. 779–790.

[SA85] R. T. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. *ACM SIGMOD'85*, pp. 236–246.

[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 JTC1/SC21/WG3 DBL MAD-146r2 (1996).

[Sno96] R. T. Snodgrass et al. Adding Transaction Time to SQL/Temporal. ANSI X3H2-96-502r2, ISO/IEC JTC1/SC21/WG3 DBL MCI-147r2 (1996).

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-Tree: A Dynamic Index for Multi-Dimensional Objects. *VLDB'87*, pp. 507–518.

[SRF97] T. Sellis, N. Roussopoulos, and C. Faloutsos. Multidimensional Access Methods: Trees Have Grown Everywhere. *VLDB'97*, pp. 13–14.

[ST97] B. Salzberg and V. J. Tsotras. A Comparison of Access Methods for Temporal Data. TimeCenter TR-18 (1997). To appear in ACM Computing Surveys.