The VLDB Journal (2001) / Digital Object Identifier (DOI) 10.1007/s007780100058

Indexing of now-relative spatio-bitemporal data

Simonas Šaltenis, Christian S. Jensen

Department of Computer Science, Aalborg University 9220 Aalborg Øst, Denmark; E-mail: {simas,csj}@cs.auc.dk

Edited by T. Sellis. Received: 7 December 2000 / Accepted: 1 September 2001
Published online: 18 December 2001

Abstract. Real-world entities are inherently spatially and temporally referenced, and database applications increasingly exploit databases that record the past, present, and anticipated future locations of entities, e.g., the residences of customers obtained by the geo-coding of addresses. Indices that efficiently support queries on the spatio-temporal extents of such entities are needed. However, past indexing research has progressed in largely separate spatial and temporal streams. Adding time dimensions to spatial indices, as if time were a spatial dimension, neither supports nor exploits the special properties of time. On the other hand, temporal indices are generally not amenable to extension with spatial dimensions. This paper proposes the first efficient and versatile index for a general class of spatio-temporal data: the discretely changing spatial aspect of an object may be a point or may have an extent; both transaction time and valid time are supported, and a generalized notion of the current time, *now*, is accommodated for both temporal dimensions. The index is based on the R*-tree and provides means of prioritizing space versus time, which enables it to adapt to spatially and temporally restrictive queries. Performance experiments are reported that evaluate pertinent aspects of the index.

Keywords: Access method – Bitemporal data – Transaction time – Valid time – Spatio-temporal data – Multidimensional indexing – R-tree

1 Introduction

Society is facing rapid advances in key information technologies. Processors are becoming faster, cheaper, and smaller; new wireless communications-related technologies such as WAP and BlueTooth are being introduced; and positioning technologies such as GPS become increasingly precise and inexpensive. Trends such as these will render the outer reaches of the Internet wireless and most users of the Internet mobile. In addition, these trends will give increased prominence to geo-referenced, or spatial, information and its evolution across time.

Specifically, an increasing number of database applications will manage spatiotemporal aspects of real-world, physical objects. Such objects have positions and extents in space, and these positions and extents may change as time passes. Example spatiotemporal objects range from people and vehicles to land parcels, residences, stores, hotels, and hurricanes [8]. The past, present, and anticipated future positions of such objects are often of interest in population studies, urban planning, marketing, sales, traffic management, vehicle navigation support, land management, and environmental studies. Some of these applications are dependent on the capture of continuous movement, while others are concerned with the discretely changing positions and extents. This paper proposes an efficient indexing technique for the latter kind of application.

Two temporal aspects of data are generally considered fundamental. The *valid time* of a fact is the time(s), past, present, or future, when the fact is true in the modeled reality, while the *transaction time* of a fact is the time(s) when the fact was or is current in the database [11,21]. Data with both valid and transaction time associated is termed bitemporal. Full spatiotemporal support implies considering these two temporal aspects as well as two or three dimensions of space.

In addition, special semantics of time must be supported and, if possible, exploited. Specifically, time intervals associated with objects may be *now-relative*, meaning that their end points track the progressing current time. To illustrate, consider the recording of addresses. The time a person resides at a given address may often extend from a known start time (the valid-time interval begin) to some unknown future time, which is captured by letting the valid-time interval end extend to the progressing current time. The same applies to the transaction time – the time a tuple is inserted into the database is known, but we do not know when the tuple will be deleted. This notion of *now* is peculiar to time and has no counterpart in space.

The previously proposed spatiotemporal indices [17,24] assume only one time dimension, adopting one of two approaches. Either overlapping index structures are used that in-

dex spatial objects at different time instances and save space by sharing the unchanged parts of the indices [18,27,28], or time is added as another dimension to an existing spatial index [25]. In this paper, we present an index, termed the R^{ST}-tree ("SpatioTemporal"), that adopts the fundamental structure of the spatial R^*-tree to index bitemporal-data and adds additional dimensions to support spatiotemporal data. Section 6 considers the utility of previous proposals for the problem addressed in this paper and compares them to the paper's proposal.

Section 2 presents the type of data that can be indexed with the new index. Section 3 solves the problem of choosing the minimum bounding regions to be used in the entries of the index, and algorithms for the tree operations are given in Sect. 4. Section 5 presents performance studies, Sect. 6 considers related proposals, and Sect. 7 concludes the paper.

2 Background

This section presents two example application areas illustrating the spatiotemporal data and queries supported, and it describes the temporal aspects of the data to be indexed.

2.1 Example application areas

Demographic data is used in applications such as advertisement, direct marketing, urban planning, and social studies. It is of great value to record the changing addresses, or locations, of the people in the database, and we will assume that the database records the history of the position (e.g., latitude and longitude) of each person's residence (and possibly place of work). This means that we are faced with 2-D point locations that may change discretely from time to time. With this data available, it is possible to answer a query such as "Who lived close to a chemical plant during the period when the plant leaked toxic materials into the environment?" or "Who moved out of a certain neighborhood during a specific period?"

Cadastral systems exemplify another kind of spatiotemporal application. Here the boundaries of land parcels are recorded together with the history of their change. And for legal reasons, all records must be maintained in an append-only fashion so that even mistaken records are retained. This is accomplished via transaction-time support. For indexing, we bound each land parcel with a minimum bounding rectangle and associate it with valid- and transaction-time intervals. An example query could be "Who owned some part of a specific piece of land sometime during the period from 1975 to 1980, as best known by 1990?" This is a spatiotemporal containment query that constrains all four dimensions: the two spatial dimensions, valid time, and transaction time.

The above examples illustrate the types of data and queries supported by the new index. According to the criteria for classification of spatiotemporal access methods proposed by Theodoridis et al. [24], the R^{ST}-tree supports 2-D points and regions; it is bitemporal; both the cardinality and the positions of the spatial objects may change over time; the index is dynamic; and spatial, temporal, and spatiotemporal containment queries are supported. As mentioned, the index also supports now-relative data, which we proceed to characterize.

Table 1. The demographic relation

	Person	Pos.	$TT^{\vdash}$	$TT^{\dashv}$	$VT^{\vdash}$	$VT^{\dashv}$
(1)	John	Pos1	4/97	UC	3/97	5/97
(2)	Tom	Pos2	3/97	7/97	6/97	8/97
(3)	Jane	Pos3	5/97	UC	5/97	NOW
(4)	Julie	Pos4	3/97	7/97	3/97	NOW
(5)	Julie	Pos4	8/97	UC	3/97	7/97
(6)	Ann	Pos5	5/97	UC	3/97	NOW + 1
(7)	Scott	Pos6	4/97	UC	5/97	NOW − 2

2.2 General bitemporal data

We adopt the standard four-timestamp format [22] for capturing valid and transaction time. With this format, each tuple is timestamped with four time attributes: $VT^{\vdash}$ and $VT^{\dashv}$– the times when the tuple's information became and ceased to be true in the modeled reality; $TT^{\vdash}$ and $TT^{\dashv}$– the times when the tuple became and ceased to be current in the database.

A tuple is now-relative if it is valid until the current time or is part of the current database state. This is captured using variables that denote the current time in the attributes $VT^{\dashv}$ and $TT^{\dashv}$ [6]. Variable UC (denoting 'until changed') is used in $TT^{\dashv}$, and variable NOW is used in $VT^{\dashv}$. Figure 1 shows an example table with now-relative data. The time granularity is a month, and the current time (CT) is 9/97.

Tuple 1 records that the information "John lived at Pos1" 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 lives at Pos3" from 5/97 until the current time, that we recorded this belief on 5/97, and that this remains part of the current state. Note that while both UC and NOW refer to the current time (in transaction time and valid time, respectively) the valid-time end of NOW is constrained by the transaction time. For example, in Tuple 3, for each time point t between 5/97 and the current time, the valid-time interval extends from from 5/97 to t.

Considering again Tuple 3, the valid-time end being equal to NOW means that we currently do not believe that Jane will live at Pos3 next month (on 10/97). Sometimes such an assumption is too pessimistic. For example, there can exist a restriction that a person can only move with a month's notice. Then we would believe that Jane would live at Pos3 also next month. To record this type of knowledge, Clifford et al. [6] propose to use NOW $+ \Delta$ in the valid-time end attribute. The offset Δ may be any integer, positive or negative. For example, the latter case can be useful if a regulation states that a person has two months to report a change of address. Then we would not know with certainty that Jane currently lives at Pos3, but would be certain that she lived there two months ago. Tuples 6 and 7 exemplify the usage of positive and negative offsets.

Specific constraints apply to insertions, deletions, and modifications. When inserting a new tuple, the natural constraint

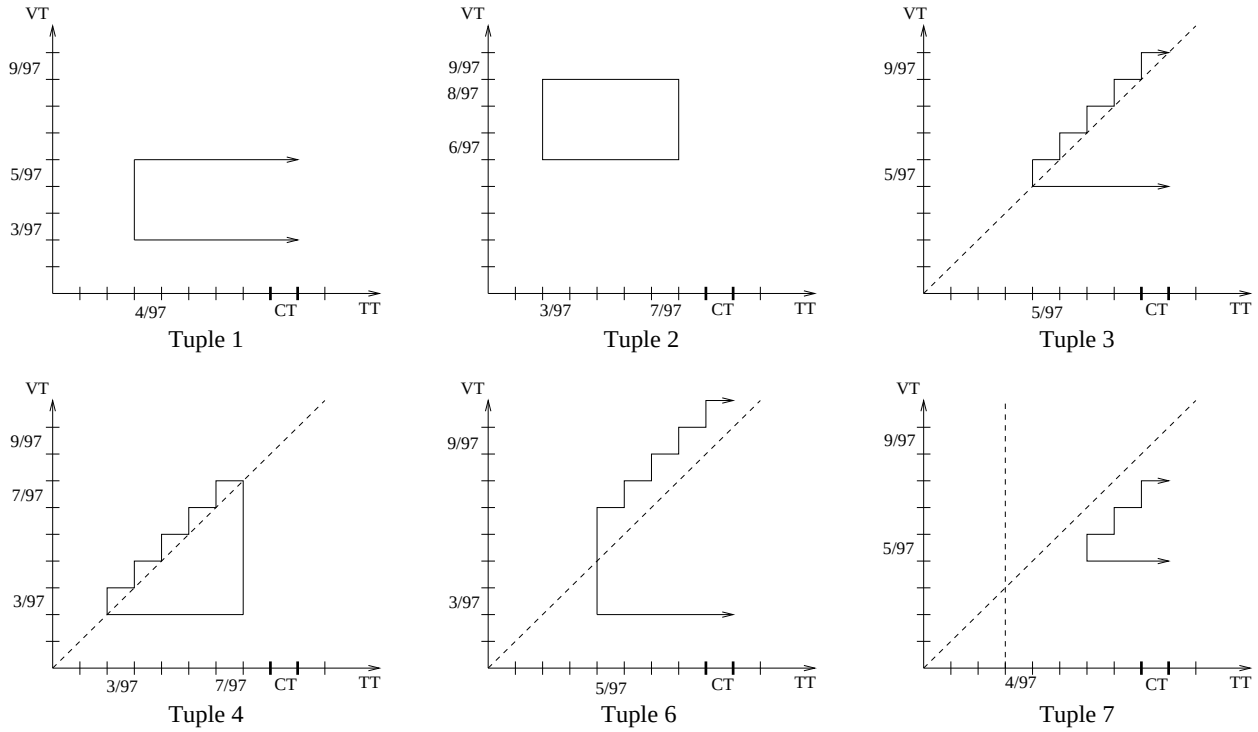

Fig. 1. Bitemporal regions of tuples from Fig. 1

$VT^{\vdash} \leq VT^{\dashv}$ applies to valid time, unless $VT^{\dashv}$ is of the form NOW $+ \Delta$ (see below); and the constraints $TT^{\vdash} =$ CT and $TT^{\dashv} =$ UC apply to transaction time. Any *current* database tuple can be deleted or modified. Deleting a tuple, the $TT^{\dashv}$ value UC is changed to the fixed value CT $- 1$[1], eliminating the tuple from the current state (e.g., Tuple 2), but retaining it in the database. A modification is modeled as a deletion followed by an insertion (e.g., an update led to Tuples 4 and 5).

The temporal aspects of a tuple can be represented graphically by a ("bitemporal") region in the 2-D space spanned by transaction and valid time [11]. Figure 1 visualizes the *bitemporal regions* of the tuples in Table 1. As shown, a now-relative transaction-time interval yields a rectangle that "grows" in the transaction time direction as time passes (Tuple 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).

The condition $VT^{\vdash} < TT^{\vdash} + \Delta$ yields a stair shape with a high first step (Tuple 6). If, on the other hand, $VT^{\vdash} > TT^{\vdash} + \Delta$, the valid-time interval is "illegal" initially, as its end time is larger than its start time. Such a region has no extent for the first $VT^{\vdash} - (TT^{\vdash} + \Delta)$ time units after the time of its insertion; it appears only in the future. Such is the case for Tuple 7, where, although $TT^{\vdash} = 4/97$, the actual region appeared only at 7/97. Specifically, from 4/97 to 7/97, we supposed that Scott did not live, lives, or will live at Pos6; in other words, no valid-time interval is associated with this tuple at these transaction times. For these times, no queries, as supported by the index, will return this tuple – only the tuples with "legal" valid-time intervals are returned. Nevertheless, it is convenient to be able to insert such future-related information.

If at some time a tuple is deleted and thus stops being current, the bitemporal region ceases to grow (Tuples 2 and 4). If $VT^{\vdash} > TT^{\vdash} + \Delta$ and the tuple is deleted before its region appears, the region will never have an extent.

In general, we obtain six combinations of time attributes for which the bitemporal regions are qualitatively different. The combinations are listed in Fig. 2, where 'tt1,' 'tt2,' 'vt1,' and 'vt2' denote ground values that satisfy the constraints given above and the offset Δ is an integer.

The spatial representation of the bitemporal extents of tuples suggests the use of some spatial index as the basis for a bitemporal index. As mentioned in Sect. 1, this approach may also facilitate the incorporation of spatial dimensions into the resulting bitemporal index.

3 Index structure

The new index has the same overall structure as the well-known R-tree [9] (and R*-tree [2]). As for the R-tree, each internal node is a record of index entries, each of which is a pair of a pointer to a node at the next level in the tree and a region that encloses all regions in the node pointed to. But unlike in the R-tree, the enclosing region is not simply a minimum bounding 4-D hyper-rectangle (in the sequel, we will simply use "rectangle" for both two and four dimensions). This section discusses the design of the bounding regions used in the R^{ST}-tree.

The spatial and temporal dimensions may be considered separately, as a 4-D region is the product of its two 2-D spa-

[1] We use closed intervals and let [$TT^{\vdash}$, $TT^{\dashv}$] denote the interval that includes $TT^{\vdash}$ and $TT^{\dashv}$.

Table 2. Possible combinations of time attributes

	$TT^{\vdash}$	$TT^{\dashv}$	$VT^{\vdash}$	$VT^{\dashv}$	Conditions	Examples
Case 1	tt1	UC	vt1	vt2		Tuples 1 and 5
Case 2	tt1	tt2	vt1	vt2		Tuple 2
Case 3	tt1	UC	vt1	NOW $+ \Delta$	(vt1 $\leq$ tt1 $+ \Delta$)	Tuples 3 and 6
Case 4	tt1	tt2	vt1	NOW $+ \Delta$	(vt1 $\leq$ tt1 $+ \Delta$)	Tuple 4
Case 5	tt1	UC	vt1	NOW $+ \Delta$	(vt1 $>$ tt1 $+ \Delta$)	Tuple 7
Case 6	tt1	tt2	vt1	NOW $+ \Delta$	(vt1 $>$ tt1 $+ \Delta$)	

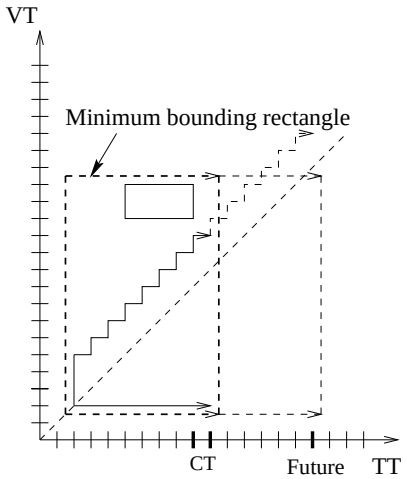

Fig. 2. A "hidden," growing stair shape

tial and bitemporal regions. For the spatial dimensions, the bounding region is simply a rectangle, as in the R-tree. We thus proceed to consider only the temporal regions, and we let the terms *minimum bounding rectangle* and *minimum bounding region* mean the projections of the corresponding 4-D region into the bitemporal hyper-plane.

In its leaf nodes, the R^{ST}-tree records the exact geometry of the bitemporal regions indexed (recall Sect. 2). The following format is used for a leaf-node index entry.

$$(TT^{\vdash}, TT^{\dashv}, VT^{\vdash}, VT^{\dashv}/\Delta, \textit{now-flag}, \langle \textit{spatial part} \rangle, \langle \textit{ptr} \rangle)$$

Here, the first three components are the attributes introduced in the previous section, and they may obtain the same values as described there. The fourth and fifth components compactly encode the values of the $VT^{\dashv}$ attribute. A value of the form NOW $+ \Delta$ is captured by setting the *now-flag* and storing Δ in $VT^{\dashv}/\Delta$; other values are stored in this attribute, without the *now-flag* set. Variable UC is represented as a special, reserved value from the domain of timestamps.

Ideally, the bounding region of an entry of a non-leaf node should enclose the regions pointed to as tightly as possible, and this property should remain even if any of the enclosed regions are growing. This implies that bounding region must also be capable of growing. In addition, the bounding regions should be compact and simple to manipulate.

Selecting a type of bounding bitemporal region that satisfies these requirements is not trivial. For example, consider the situation illustrated in Fig. 2, where a growing stair shape is placed together with a non-growing (termed "static" for short) rectangle in a minimum bounding rectangle growing in transaction time. One day, the stair shape will outgrow its bounding rectangle, making the tree invalid. We call a growing stair shape "hidden," if it is placed in a node with rectangles that are, at least partly, above the highest point of the growing stair shapes in the node.

As a first step in designing bounding regions for the tree that also handle this problem, we consider four types of bounding regions that may be used.

Following the R-tree, the minimum bounding regions may be rectangles, which now may also be growing in transaction time only or in both directions. Such rectangles may then use UC for $TT^{\dashv}$ and NOW $+ \Delta$ for $VT^{\dashv}$. Then the $TT^{\dashv}$ value of UC and the $VT^{\dashv}$ value of NOW $+ \Delta$ would represent a rectangle growing in both directions. In leaf node entries, this combination of timestamp values represents a growing stair shape. This interpretation may also be chosen for non-leaf nodes. We consider these two first types of bounding regions in connection with the above-mentioned "hidden" stair shapes.

When we employ bounding rectangles that grow in both directions, a bounding rectangle for a set of regions that includes one or more growing stair shapes has initially its $VT^{\dashv}$ set to NOW $+ \Delta$, where Δ is large enough for the rectangle to enclose all rectangles above the stair shapes. This is illustrated in Fig. 3a, where the regions from Fig. 2 are being bounded. This approach has the disadvantage that the bounding rectangles are not minimum after some time, making the index unnecessarily cover areas, termed "dead space," not covered by any data regions and also leading to increased overlap among bounding rectangles.

The use of a growing stair shape large enough to bound all regions is illustrated in Fig. 3b. At the current time, such a bounding stair shape may have a larger area than the corresponding bounding rectangle (cf. Figs. 3a,b). However, after some time, the area of the rectangle becomes larger than the area of the stair shape. It can be proven that for any set of bitemporal regions that can be bounded with either a rectangle growing in both directions or a growing stair shape, the rectangle will eventually outgrow the stair shape.

The last two types of bounding regions derive from the GR-tree [3], where hidden stair shapes are handled using a flag *hidden* in the non-leaf node entries. This enables regions with hidden stair shapes and rectangles above the diagonal to initially be bounded with a rectangle growing only in transaction time. This idea may also be applied here, but due to the

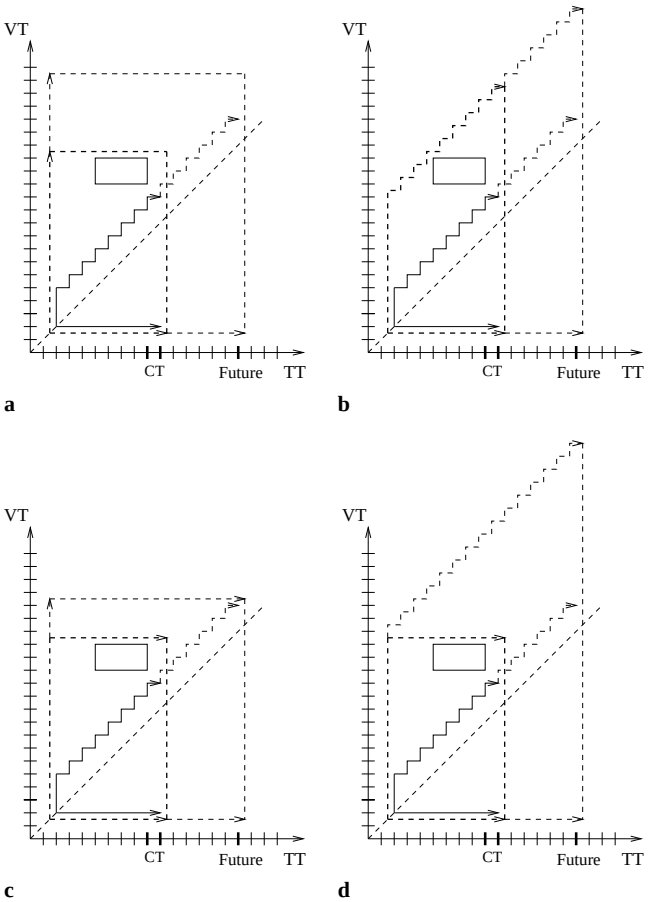

Fig. 3. a A bounding rectangle, **b** a bounding stair shape, **c** a bounding rectangle using the flag *hidden*, and **d** a bounding stair shape using the flag *hidden*

R^{ST}-tree's more general bitemporal regions, the flag *hidden* should be accompanied by an offset Δ_{max}, which is the maximum offset of the hidden stair shapes. Then, whenever the flag *hidden* of a rectangle is set and the current time plus Δ_{max} exceeds the $VT^{\dashv}$ value for this rectangle, $VT^{\dashv}$ may be adjusted to continue to bound the stair shapes that have outgrown its rectangle.

In effect, we turn the rectangle, which grows only in transaction time, into a rectangle growing in both directions or into a growing stair shape. In the first case, $VT^{\dashv}$ is set to NOW $+ \Delta_{max}$. This leads to a bounding rectangle that is minimal at all times, see Fig. 3c.

In the second case, where the original rectangle is replaced by a growing stair shape, $VT^{\dashv}$ is set to NOW $+ \Delta$, where Δ is the difference between the old value of $VT^{\dashv}$ and the value of $TT^{\vdash}$. The resulting growing stair shape fully encloses the rectangle it replaces, see Fig. 3d. Creating a bounding stair shape this large is necessary because we cannot know what regions the original rectangle is bounding without accessing the child node. Some of those regions may be located in the upper-left corner. The large bounding stair shapes is the main disadvantage of this approach.

The two types of bounding regions that employ the *hidden* flag have the disadvantage of a larger and more complex entry structure. This might be justified if the flag is used frequently. But our experiments with the GR-tree [3] show that the flag is not used very often (because the insertion algorithms obtain "good" groupings of regions).

In conclusion, we use the second kind of bounding region, illustrated in Fig. 3b, in the R^{ST}-tree. With this choice, the structure of a non-leaf and a leaf node are exactly the same, and the kinds of bounding regions available is exactly the same as the kinds of bitemporal data regions possible. This homogeneity simplifies the algorithms associated with the tree. In general, a bounding region ($TT^{\vdash}$, $TT^{\dashv}$, $VT^{\vdash}$, $VT^{\dashv}$) for a set of bitemporal regions r_i is computed as shown in Fig. 3, where $VT^{\dashv}$ is subsequently encoded in the $VT^{\dashv}/\Delta$ and *now-flag* attributes given for leaf nodes earlier in this section.

Note that the argument to function $area$ denotes a static stair shape where the function is used first and a static rectangle were it is used the second time.

The computations of $TT^{\vdash}$, $TT^{\dashv}$, and $VT^{\vdash}$ are trivial. The value of $VT^{\dashv}$ determines whether the region is a stair shape or a rectangle. To avoid creating unnecessary growing stair shapes, we set $VT^{\dashv}$ to NOW $+ \Delta_b$ only if at least one growing stair shape exists among the regions(the condition $\exists i\ (TT^{\dashv}_{r_i} = \text{UC} \wedge VT^{\dashv}_{r_i} = \text{NOW} + \Delta)$), or if all the regions are static (the condition $\forall i\ (TT^{\dashv}_{r_i} \neq \text{UC})$) and a minimum bounding stair shape (also static) has a smaller area than a minimum bounding rectangle (the last condition).

If a stair-shaped bounding region must be created, Δ_b is set to be large enough to enclose all upper-left corners of rectangles ($\Delta_b \geq VT^{\dashv}_{r_i} - TT^{\vdash}_{r_i}$) and all stair shapes ($\Delta_b \geq \Delta_{r_i}$). Otherwise, the upper bound of the bounding rectangle ($VT^{\dashv}_b$) is set large enough to enclose both rectangles and stair shapes. In this case, the stair shapes are all static, and each of them extends in valid time up to $TT^{\dashv}_{r_i} + \Delta_{r_i}$.

It should be noted that growing stair shapes with $VT^{\vdash} > TT^{\vdash} + \Delta$ (Fig. 1, Tuple 7) are treated exactly the same as all other growing stair shapes. When bounding such stair shapes, the minimum bounding region is set so that it encloses all regions that will appear in the future, although it may not be minimal for some time.

4 Index algorithms

This section describes the algorithms that serve to maintain the index structure just described. Section 4.1 offers a road map of the algorithms. Section 4.2 extends the heuristics used in the R*-tree insertion algorithm to support the new types of data regions. Sections 4.3–4.5 then describe the insertion algorithm, and Sect. 4.6 shows how the index can be tuned to support specific types of queries. Section 4.7 compares the algorithmic complexity of the new index algorithms to the corresponding algorithms of the R*-tree.

4.1 Road map of algorithms

The overall structure of the R^{ST}-tree is the same as that of the R*-tree. As a result, the R*-tree's search, deletion, and inser-

$$
\begin{aligned}
\mathrm{TT}^{\vdash} &= \min_i(\mathrm{TT}^{\vdash}_{r_i}) \\
\mathrm{VT}^{\vdash} &= \min_i(\mathrm{VT}^{\vdash}_{r_i}) \\
\mathrm{TT}^{\dashv} &= \begin{cases} \mathrm{UC} & \textbf{if } \exists i\ (\mathrm{TT}^{\dashv}_{r_i} = \mathrm{UC}) \\ \max_i(\mathrm{TT}^{\dashv}_{r_i}) & \textbf{otherwise} \end{cases} \\
\mathrm{VT}^{\dashv} &= \begin{cases} \mathrm{NOW} + \Delta_b & \textbf{if } \exists i\ (\mathrm{TT}^{\dashv}_{r_i} = \mathrm{UC} \wedge \mathrm{VT}^{\dashv}_{r_i} = \mathrm{NOW} + \Delta) \vee \\ & \quad (\forall i\ (\mathrm{TT}^{\dashv}_{r_i} \neq \mathrm{UC}) \wedge \\ & \quad\ area(\mathrm{TT}^{\vdash}, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{NOW} + \Delta_b) < area(\mathrm{TT}^{\vdash}, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}_b)) \\ \mathrm{VT}^{\dashv}_b & \textbf{otherwise} \end{cases} \\
\Delta_b &= \max(\max_{i:\mathrm{VT}^{\dashv}_{r_i} \neq \mathrm{NOW}+\Delta}(\mathrm{VT}^{\dashv}_{r_i} - \mathrm{TT}^{\vdash}_{r_i}), \max_{i:\mathrm{VT}^{\dashv}_{r_i} = \mathrm{NOW}+\Delta}(\Delta_{r_i})) \\
\mathrm{VT}^{\dashv}_b &= \max(\max_{i:\mathrm{VT}^{\dashv}_{r_i} \neq \mathrm{NOW}+\Delta}(\mathrm{VT}^{\dashv}_{r_i}), \max_{i:\mathrm{VT}^{\dashv}_{r_i} = \mathrm{NOW}+\Delta}(\mathrm{TT}^{\dashv}_{r_i} + \Delta_{r_i}))
\end{aligned}
$$

Fig. 4. Computation of a bounding region for a set of bitemporal regions r_i

tion algorithms may be used for the new index, provided that these algorithms employ a set of algorithms that manipulate the new kinds of regions presented in Sect. 3. This set includes an algorithm that determines whether a pair of regions overlap and algorithms that compute the area and margin of a region, the intersection of a pair of regions, and the minimum bounding region of a node. Indeed, the R$^{\mathrm{ST}}$-tree reuses the R*-tree's search and deletion[2] algorithms, which have been adapted to use these low-level algorithms. This leaves only insertion unaccounted for.

The insertion algorithm is central because it is responsible for maintaining an efficient tree. The R*-tree's insertion algorithm is likely to be inefficient for the new types of data and bounding regions in the R$^{\mathrm{ST}}$-tree. Before improving this algorithm, we describe its general working.

Given a new entry, the R*-tree's insertion algorithm starts at the root node and traverses the tree downwards until it reaches a leaf node. At each visited node, it uses the ChooseSubtree algorithm to choose one subtree among the subtrees rooted at this node and then proceeds to the root of the chosen subtree. ChooseSubtree evaluates each subtree rooted at a node with respect to the new entry, and it chooses one subtree according to a set of heuristics.

If a new entry is to be inserted into a leaf node that is full, an overflow occurs. If, during the insertion of the new entry, this is the first overflow at a given level in the tree, the algorithm RemoveTop is invoked; otherwise, the Split algorithm is invoked. RemoveTop[3] removes p entries from a node and reinserts them. The entries to be removed are chosen according to heuristics. In the worst case, where all these entries are reinserted into the same node or they overflow some other node, the Split algorithm is used instead of RemoveTop. The split of a node can result in an overflow of a parent node. If this happens, the described procedure is repeated for the parent node.

The algorithm Split distributes the entries of an over-full node into two groups, forming two nodes. Because the number of possible distributions is exponential in the number of entries, only a subset of all possible distributions is considered by the Split algorithm. Again, the choice of the best distribution is made according to heuristics.

[2] It should be noted that the logical deletion presented in Sect. 2 is implemented in the index as a deletion of an old region followed by an insertion of a new one with a fixed $\mathrm{TT}^{\dashv}$ value.

[3] This algorithm implements forced re-insertion, introduced in [2].

The insertion algorithm of the R$^{\mathrm{ST}}$-tree has the same structure as that of the R*-tree, but the heuristics that drive the decisions in the ChooseSubtree, Split, and RemoveTop algorithms are tailored to take into account the growth of bounding regions throughout the tree.

4.2 Heuristics and time parameterization

The R*-tree uses three basic heuristics in its insertion algorithm.

- H1. The volumes of bounding hyper-rectangles should be minimized.
- H2. The overlap (volume of intersection) among bounding rectangles should be minimized.
- H3. The margin of bounding rectangles should be minimized.

Recent analytical studies of the performance of R-trees validate these heuristics [13,19,23]. For a 4-D rectangle with side lengths given by (s_1, s_2, s_3, s_4), the margin is given by the following sum: $\Sigma_i s_i + \Sigma_{i<j} s_i s_j + \Sigma_{i<j<k} s_i s_j s_k$ (where $1 \leq i, j, k \leq 4$) [13]. The intuition underlying the margin heuristic is covered in Sect. 4.6.

In the R$^{\mathrm{ST}}$-tree, the same three types of heuristics are used in the tree algorithms, but because the quantities of volume, overlap, and margin are functions of time, the insertion algorithm should consider not only the current values of these, but also their future evolutions.

Let us investigate Heuristic H1. The evolution of the volume of a bounding region depends on the following four characteristics:

i. The type of the region:
   (1) Static regions (rectangles and stair shapes),
   (2) growing rectangles, and
   (3) growing stair shapes.
ii. The rate of growth of the region's volume.
iii. The time period from now to the moment when the region starts to appear.
iv. The volume of the region at the current time.

We say that an entry is of type *t* if the region in the entry is of type *t*. Similarly, a node is of type *t* if its bounding region is of type *t*.

Let us assume that no updates are performed after the current time and that the goal is to minimize the volume of the

bounding regions (Heuristic H1) over all times starting from the current time and extending indefinitely far into the future. Then, the characteristics of regions should be prioritized in the order given above. It is easy to see why, under these assumptions, this is the only suitable order of prioritization.

The types of the regions should be considered first. Highest priority should be given to static rectangles and static stair shapes, independently of their volume because any growing rectangle or stair shape will eventually outgrow a static rectangle or a static stair shape. For the same reason, growing rectangles (which grow only in transaction time) should be given a higher priority than growing stair shapes.

The second characteristic of a region is its rate of growth. Among two growing rectangles, the one that is narrower in the spatial dimensions or in valid time should be given highest priority because it grows by a smaller amount at each time unit. Similarly, a growing stair shape with a larger projection onto the spatial dimensions will eventually build up more area than a stair shape with a smaller projection.

Third, if two growing regions have the same type and growth rate, it may be possible to prioritize them according to the times when they appear. The later a region begins it existence, the more preferable it is.

Static regions are only compared using the fourth characteristic, exactly as is done in the R*-tree.

Heuristic H2, which minimizes overlap between regions, can also be extended in a similar way to address the growth of regions. This is done by considering the same four characteristics as above, but for the regions obtained by intersecting overlapping regions. Finally, the presented scheme can be applied similarly to the margin heuristic (H3).

As noted, the a prioritization scheme such as the one given here is based on the assumption that the index is queried for an indefinite time and that no assumptions are made about future insertions and deletions. In fact, these quite general assumptions underlie the algorithms of most access methods. However, due to the presence of growing data regions, in a realistic scenario where the index is updated continuously, a less strict prioritization of the four characteristics of regions may be desirable. For example, a very small, but growing rectangle is preferable over a very large static rectangle if the growing rectangle is updated and becomes static before it outgrows the static one.

The above prioritization scheme is implemented in the R^{ST}-tree in a relatively straightforward and *flexible* (to accommodate realistic scenarios) manner by introducing a *time parameter p* in the insertion algorithm, which then computes the areas of regions as of *p* time units into the future (for some *p*). If a sufficiently large time parameter is used when computing and comparing areas, we effectively obtain results that follow the above-described scheme. Relaxed prioritizations are achieved by using smaller time-parameter values. Which time-parameter values to use is investigated experimentally (Sect. 5.2) and depends mostly on the intensity of updates.

Experiments with the GR-tree [3] show that its performance is substantially boosted when strict heuristics are used that look only at node types and group regions of the same type, trying single-mindedly to avoid introducing growing rectangles and, especially, growing stair shapes. While experimenting with different designs of the insertion algorithm, we noticed that these heuristics not only do *not* work for spatiotemporal data, but even have a negative effect.

To understand why, consider the situation in a node split algorithm where there is a choice between having one "large" bounding, growing stair shape and one static bounding region versus having two "smaller" bounding, growing stair shapes. With only two temporal dimensions, the former choice is clearly preferable; the "large" stair shape will most probably not be much larger than one of the two "smaller" stair shapes, and these two overlap substantially. This is so because all growing stair shapes follow the $VT^{\dashv} = \text{NOW}$ diagonal and have equal transaction-time end values. Introducing two spatial dimensions, the spatial extents of the two "smaller" stair shapes may be non-overlapping and far apart in the spatial dimensions. Forcing them into one bounding stair shape may produce a truly large growing stair shape.

Summarizing, the time parameter is a simple and flexible way to extend the R*-tree heuristics. Using the time parameter, the four characteristics of regions, be it bounding regions or the regions that occur as intersections between bounding regions, do not have to be inspected explicitly. The next sections explain in detail the workings of the Split, RemoveTop, and ChooseSubtree algorithms.

4.3 The ChooseSubtree algorithm

The ChooseSubtree algorithm is used for deciding where to insert a new entry. The algorithm chooses one subtree among the subtrees rooted at a node and then repeats the procedure for that subtree until it reaches a leaf node. To optimize the overlap heuristic (H2), the R*-tree's ChooseSubtree algorithm considers the enlargement of the overlap between the bounding regions of the subtrees that would result from inserting the new entry in a subtree.

To determine this overlap enlargement, the overlaps between a subtree's current minimum bounding region and the minimum bounding regions of all the other subtrees are determined. Then the subtree's minimum bounding region is extended with the new entry, and the overlaps are determined anew. This is done for all subtrees, and ChooseSubtree proceeds with the subtree where including the new entry yields the *least overlap-area enlargement*. Ignoring the spatial dimensions, Fig. 5 gives an example, where inclusion of a new entry in node 1 is chosen.

Ties are resolved by choosing the node whose minimum bounding region requires the *least area enlargement*, and further ties are resolved by choosing the node whose minimum bounding region has the *smallest area* with the new entry enclosed.[4]

The R^{ST}-tree employs the above sketched R*-tree's ChooseSubtree algorithm [2], with the exception that com-

[4] For non-leaf nodes, overlap area enlargement is not considered – only area enlargement and area are considered.

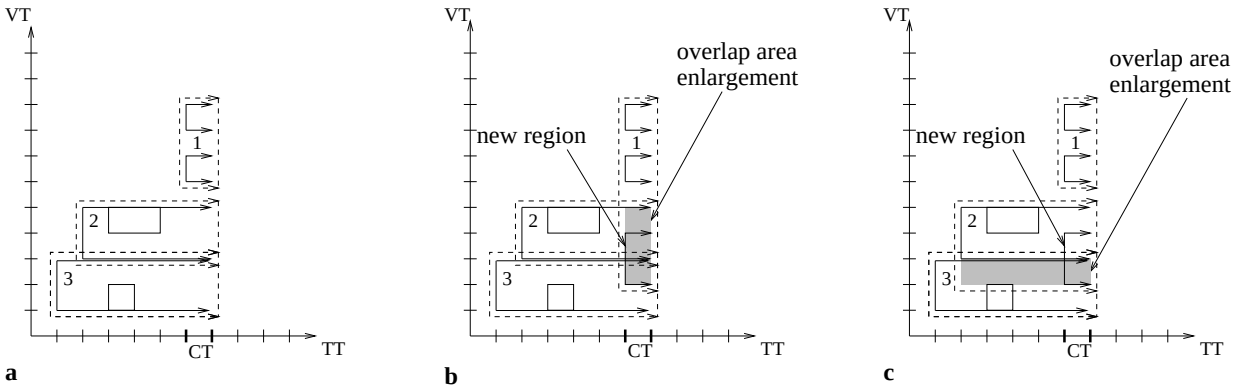

Fig. 5. a Initial bounding regions, **b** insertion of a new entry into node 1, and **c** insertion of a new entry into node 2

putations of overlap enlargement, area enlargement, and area are time-parameterized. Fig. 6 shows the result of applying ChooseSubtree with a later time-parameter value. Now, node 2 is the best choice for the new entry, as it yields a smaller overlap area enlargement than inclusion in node 1, and its area enlargement is smaller than that of node 3.

4.4 The split algorithm

A node split algorithm can be characterized by the set of distributions of entries into two nodes that it considers, and by the criteria it employs for selecting one (the "best") of these candidate distributions.

The subset of all possible distributions considered by the R*-tree's Split algorithm is selected as follows. Along each of the axes, entries of the over-full node are sorted according to their bottom and top values. Then, assuming two dimensions, for each of the four sortings, a total of $M-2m+2$ distributions are considered, where M is the maximum number of entries in the node, and m is the minimum allowed number of entries in the node. 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 best distribution is selected based on the three heuristics H1–H3, introduced in Sect. 4.2. For the pair of bounding rectangles resulting from a distribution of entries, we use *area-value* for the sum of their volumes, *overlap-area* for the volume of their intersection, and *margin-value* for the sum of their margins. Using Heuristic H3 (minimum margin-value), one axis is selected. Then Heuristics H1 and H2 are used considering only the distributions along this axis.

The original R*-tree split algorithm

S1 For each axis, sort the rectangles by their lower then by their upper value and determine all distributions as described above. Compute the sum of margin-values for all distributions for each axis.
S2 Let the axis with the minimum sum of margin-values be the split axis.
S3 Along the split axis, choose the distribution with the minimum overlap-value. Resolve ties by choosing the distribution with the minimum area-value.

The R$^{\mathrm{ST}}$-tree makes this algorithm time-parameterized and introduces the generation of one more candidate distribution after Step S2. The new distribution is generated by trying to split entries of an over-full node so that the resulting bounding regions are of the best types possible. At the same time, the algorithm aims to not distribute entries of the same type into two different nodes. This procedure is described in detail next.

Each of the two nodes produced by a split may be bounded by a region of one of the three types defined in Sect. 4.2, leaving six possible pairs of types. These pairs are prioritized according to their "goodness," as defined in Fig. 7. Stated mathematically, 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:

$$
\begin{aligned}
&(\mathit{type}(x_1) \neq \mathit{type}(y_1) \vee \mathit{type}(x_2) \neq \mathit{type}(y_2)) \wedge \\
&((\mathit{type}(x_1) \leq \mathit{type}(y_1) \wedge \mathit{type}(x_2) \leq \mathit{type}(y_2)) \vee \\
&\ (\mathit{type}(x_1) < \max(\mathit{type}(y_1), \mathit{type}(y_2)) \wedge \\
&\ \ \mathit{type}(x_2) < \max(\mathit{type}(y_1), \mathit{type}(y_2)))).
\end{aligned}
$$

The generation of the additional distribution is based on this ranking of pairs of node types.

Generation of the additional candidate distribution

S2.1 Among the six pairs of types of bounding regions, select the pair (t_1, t_2) such that, according to the conditions given in Fig. 7: (a) it is possible to achieve this pair of bounding-region types when dividing the entries of the over-full node into two nodes; and (b) no other pair with a higher priority can be achieved. Let $t_1 \leq t_2$, and name the node bounded with a region of type t_1 and t_2 as N_1 and N_2, respectively. Let $\mathcal{S}$ contain all entries of the over-full node.
S2.2 Move to N_2 all entries from $\mathcal{S}$ that cannot be put into N_1 because of the type of its bounding region. (Growing rectangles cannot be put into static regions, and growing stair shapes cannot be put into growing rectangles or static regions.)
S2.3 Let $\mathcal{S}_t$ denote all entries from $\mathcal{S}$ of type t. If $|\mathcal{S}_1| \leq M-m+1$, move $\mathcal{S}_1$ into N_1. Next, if $|\mathcal{S}_2|+|N_1| \leq M-m+1$, move $\mathcal{S}_2$ into N_1. Here, we try not to distribute entries of the same type into two different nodes.
S2.4 If $|\mathcal{S}| = 0$ (i.e., **S2.3** succeeded), goto **S3**; (N_1, N_2) is a new distribution.
S2.5 If $|N_1| = 0 \wedge |N_2| = 0$, goto **S3**; no new distribution

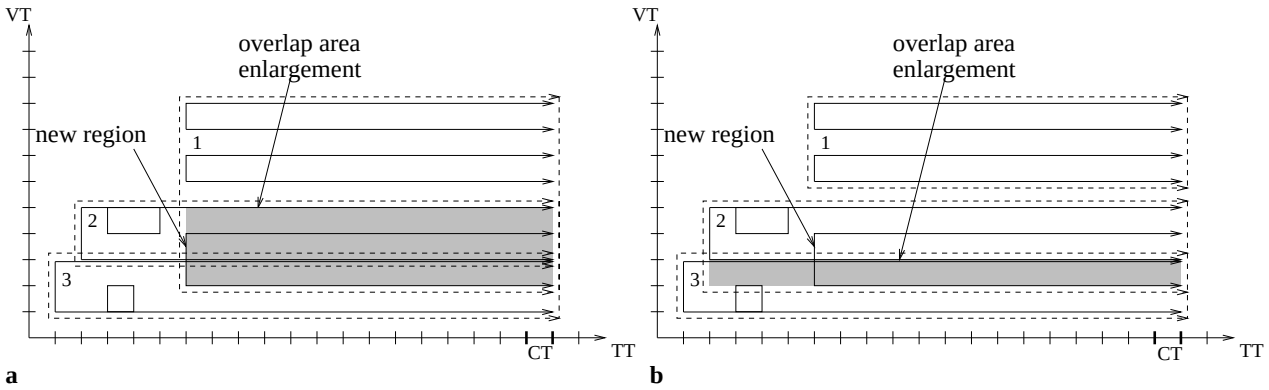

Fig. 6. a Case b from Fig. 5 after a period of time and **b** case c from Fig. 5 after a period of time

was generated (all entries were of the same type).
S2.6 If $|N_1| = 0$, pick a "seed" entry e from $\mathcal{S}$ for Guttman's quadratic *Distribute* algorithm [9] such that its inclusion into N_2 would enlarge that node's minimum bounding region the most. Put e into N_1. Goto **S2.8**.
S2.7 If $|N_2| = 0$, pick a seed entry e from $\mathcal{S}$ and put it into N_2.
S2.8 Apply Guttman's quadratic *Distribute* algorithm.

The above algorithm uses a time-parameterized version of Guttman's quadratic *Distribute* algorithm. Section 5 studies the impact on query performance of considering this additional distribution.

4.5 The remove-top algorithm

The original R*-tree RemoveTop algorithm sorts the entries of an over-full node by the distances of their centers from the center of the minimum bounding rectangle of the over-full node and then removes and reinserts a certain percentage of the entries with the largest distances.

A time-parameterized version of this algorithm could be used for the R$^{\mathrm{ST}}$-tree, but performance experiments show that a RemoveTop algorithm based on the heuristic of volumes works better for spatiotemporal data. The R$^{\mathrm{ST}}$-tree employs an algorithm of quadratic complexity in the number of entries in a node. This algorithm repeatedly removes the entry that, when removed, shrinks the time-parameterized volume of the minimum bounding region the most.

4.6 Prioritizing space versus time

The insertion algorithm described in the previous sections aims to produce a tree that yields high performance of spatiotemporal intersection queries. The heuristics used so far have been based on the assumption that intersection queries are square on average, i.e., all the dimensions (temporal and spatial) are constrained by intervals of approximately the same length.

Due to the quite different semantics of the dimensions involved, this may not always be a good assumption. In some applications, most queries may be much more restrictive on the spatial dimensions than on the temporal dimensions. For example, queries in a cadastral system msaay retrieve the current knowledge of the full history of ownership of some piece of land. In other applications, queries may be most restrictive on the temporal dimensions. Specifically, timeslice queries, which specify time points in the temporal dimensions, have very natural semantics and are often important. For example, a query in a demographic database may retrieve the population for a county, as it was two years ago, as currently best known.

Non-square queries may be due just to the different units of measurement used for the spatial and temporal dimensions. For example, if the granularity of time is 1 s and the granularity of space is 1 m, and most queries are formulated in days and meters, the queries will be long in the temporal dimension. The solution to the problem of non-square queries addresses the issues of different semantics and granularities of the spatial and temporal dimensions.

In the extreme, if a substantial number of queries offer no restrictions on either space or time, additional, separate temporal and spatial indices may be introduced for these queries. Whether the likely increase in query performance obtained from using these specialized indices justifies the extra cost of update and the extra storage space occupied will depend on the specific application requirements.

In order to obtain a versatile spatiotemporal index that may serve well the full spectrum queries – ranging from having only spatial constraints to having only temporal constraints – it is desirable to introduce a mechanism into the R$^{\mathrm{ST}}$-tree that allows it to be tuned to support better either spatially or temporally restrictive queries.

In any R-tree-based index, one dimension can be prioritized over the others by specifically favoring minimum bounding rectangles that are narrow in this dimension and long in the other dimensions. In Fig. 8, a 2-D space is considered. The two sets of minimum bounding rectangles cover the same areas and do not overlap. Scenario (b) favors queries restrictive in the x dimension and not in the y dimension.

Analytical studies also offer some insight into the effect of the shapes of the bounding regions on query performance [13,19]. For a 4-D query with side lengths (q_1, q_2, q_3, q_4), the estimated number of page accesses is proportional to

Priority	Region 1	Region 2	Enabling conditions
1	Type 1	Type 1	GR + GS = 0
2	Type 1	Type 2	GS = 0 and 0 < GR <= k
3	Type 2	Type 2	GS = 0 and GR > k
4	Type 1	Type 3	GS > 0 and GR + GS <= k
5	Type 2	Type 3	0 < GS <= k and GR + GS > k
6	Type 3	Type 3	GS > k

Type 1 Type 2 Type 3 k = M + 1 - m

GR: Number of growing rectangles GS: Number of growing stair shapes

Fig. 7. Pairs of bounding-region types and enabling conditions

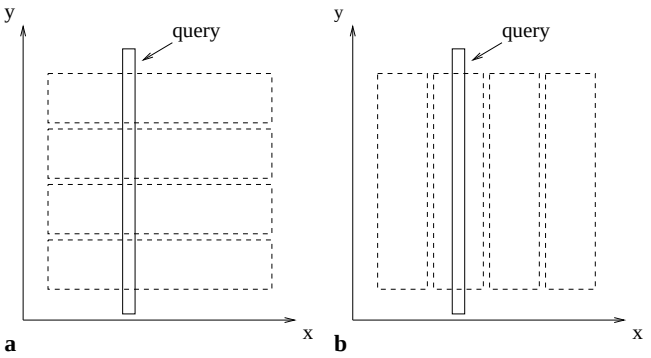

Fig. 8. Geometries of minimum bounding rectangles

$\Sigma_m(\prod_{i=1}^{4}(s_{m,i}+q_i))$, where $s_{m,i}$ is the side length of bounding rectangle s_m in the i-th dimension and m ranges over all bounding rectangles of the tree. Expanding the product in this formula, it can be seen that the notion of margin, as defined in Sect. 4.2, plays a significant role. The formula suggests that non-rectangular queries may be supported better by weighing the components of the margin expression. However, in R*-tree-based indices, the margin-value is used only as a secondary heuristic and only in the node split algorithm, and so a prioritization scheme based on margin will not be effective.

We propose a simple way to prioritize the dimensions in an R-tree-based index, which works with the existing tree algorithms. For each n-dimensional rectangle, instead of considering the extents $(\Delta x_1, \Delta x_2, \ldots, \Delta x_n)$, weighted extents $((\Delta x_1)^{\alpha_1}, (\Delta x_2)^{\alpha_2}, \ldots, (\Delta x_n)^{\alpha_n})$ are used. If all α_i are equal to one, none of the dimensions are prioritized. The priority of dimension i is increased by setting α_i to a value greater than 1, and the priority of dimension i lowered by setting α_i to a value smaller than 1. Setting α_i to 0 makes the algorithms disregard the dimension.

Following this scheme the $\mathrm{R^{ST}}$-tree uses a single parameter $\alpha \in [-1..1]$. The volume of a 4-D region r is then computed as follows.

$$\mathit{vol}(r) = \begin{cases} \mathit{bitemp\_area}(r)^{1+\alpha} \cdot \mathit{spatial\_area}(r) \textbf{ if } \alpha \leq 0 \\ \mathit{bitemp\_area}(r) \cdot \mathit{spatial\_area}(r)^{1-\alpha} \textbf{ otherwise} \end{cases}$$

where *bitemp_area* is the area of the region's time-parameterized bitemporal extent and *spatial_area* is the area of its spatial extent.

The margin-value computation in the formula presented in Sect. 4.2 is updated similarly, so that the temporal and spatial extents are weighted with $1-\alpha$ or $1+\alpha$. It follows that α values less than 0 favor spatially selective queries, as, for example, queries in a cadastral system asking to retrieve the full history of a specific area of land. The values of α greater than 0 favor temporally restrictive queries, such as timeslice queries that ask to retrieve a complete map of land ownership as it was at some specific time in the past. As mentioned above, α values of -1 and 1 turn the $\mathrm{R^{ST}}$-tree into a spatial and a temporal index, respectively. Section 5 investigates the effects of using different α values.

4.7 Algorithmic complexity

It is appropriate to consider how the complexity of the proposed index algorithms relates to the complexity of the corresponding algorithms of the R*-tree, on which the new index is based.

In the worst case, the Split algorithm relies on Guttman's quadratic distribute algorithm. Thus, the Split, as well as RemoveTop, algorithms have a quadratic complexity in the number of entries in a node. This is more that $O(n \log n)$ complexity of the R*-tree Split and RemoveTop algorithms.

We performed a number of experiments to measure the CPU running times of the different parts of the insertion algorithm. These studies show that the combined running time of the Split and RemoveTop algorithms constitutes less than one percent of the total CPU time. This is as could be expected, for the following reasons. The number of entries in a leaf node of the tree was approximately 200 in our experiments. This means that, on average, it takes at least 100 insertions to produce a split (or a call to the RemoveTop algorithm). On the other hand, each insertion calls the ChooseSubtree algorithm as many times as there are levels in the tree. Thus, the vast majority of the CPU time is spent in the ChooseSubtree algorithm, and the complexity of this algorithm in the $\mathrm{R^{ST}}$-tree is the same as in the R*-tree – quadratic in the number of entries in a node.

In practice, the average running time of the ChooseSubtree algorithm can be substantially reduced by first determining whether there are any entries in a node that completely enclose a new entry. If so, the step of looking for minimum overlap enlargement can be skipped, and this is the step with quadratic complexity.

5 Performance studies

Section 5.1 details the experimental setup and data generation. Section 5.2 then describes the effect of the introduction of the time parameter and the additional split distribution in the tree, and Sect. 5.3 studies the use of the α-parameter. Finally, Sect. 5.4 compares the $\mathrm{R^{ST}}$-tree with its only competitor, a minimally adapted R*-tree.

5.1 Experimental setup and data generation

The studies use an implementation of the $\mathrm{R^{ST}}$-tree that is based on the Generalized Search Tree Package, GiST [10]. The page size (and tree node size) is set to 4k bytes, and a page buffer of 200 k bytes, i.e., 50 pages, is used [16] where the root of a tree is pinned and the least-recently-used page replacement policy is employed. Nodes changed during an index operation are marked as "dirty" in the buffer and are written to disk at the end of the operation or when they otherwise have to be removed from the buffer.

The performance studies are based on so-called workloads that intermix queries and update operations on the index, thus simulating index usage across a period of time. The reminder of this section describes the generation of workloads.

The workloads simulate the evolving spatiotemporal aspects of a set of objects. Each change to an object is characterized by three parameters: the time duration for which the previous spatial value of the object was valid (*VL*), the displacement of the center of the spatial value relative to its previous center, and the change of the area of the spatial value relative to its previous area. These parameters account for the spatial and valid-time aspects of objects and are adapted from the GSTD algorithm [26]. These spatial and valid-time values are augmented with transaction times when they are stored in the database, and GSTD is extended to make it possible to simulate the storage of these changing values of objects in the database. Insertions may be retroactive or predictive [12]. A retroactive insertion occurs when the valid-time begin of an inserted record is less than the time of insertion (transaction-time begin), and a predictive insertion occurs when the valid-time begin exceeds the time of insertion.

A large number of parameters may be introduced that control the generation of workloads. The present experimental studies attempts to discern and put focus on the novel aspects of spatiotemporal data. One central such aspect is the concept of a *history*. A history captures the current knowledge about the (spatial) evolution of a single object. An object's history is composed of a number of triples consisting of a spatial value, a valid-time interval, and a transaction-time interval; the valid-time intervals meet, yielding at most one spatial value at each point in time, and for the transaction-time intervals, $\mathrm{TT}^{\dashv} = \mathrm{UC}$.

The spatial value associated with the first valid-time interval in a history is taken from a set of spatial values generated using the À La Carte spatial data generator [7]. This set contains 2,000 rectangles generated in an area spanning 20,000 units in the x and y dimensions. The spatial density is set to 1, and the distribution of the bottom-left corners of the rectangles is uniform. Point data is obtained by taking the bottom-left corners of the rectangles.

The spatial value associated with any other valid-time interval in a history is generated from the spatial value associated with its predecessor valid-time interval, by extending this value by Δx and Δy, if it is a rectangle, and by adding $(\Delta x, \Delta y)$ to it if it is a point. The values of Δx and Δy are uniformly distributed over $[-\mathit{SpChange}, \mathit{SpChange}]$. This enables the modeling of shrinking/expanding rectangles and moving points.

The history of an object is *active* if it has a spatial extent with a valid time that includes the current time. For active histories, the interval having the largest valid-time begin is made now-relative. Histories that are not active contain no now-relative valid-time intervals. Valid-time interval lengths are uniformly distributed between 0 and *VL*. For now-relative intervals, the Δ-offset is normally distributed with mean 0 and deviation 100, and its valid-time begin is normally distributed with mean equal to the insertion time (transaction-time begin) and deviation $Dev = 500$.

The transaction-time intervals in the triples are obtained by simulating database modifications. To accomplish this, the workload generator maintains a list of the currently active histories. Each history has associated the time when the next insertion for the history will occur, along with the pair of the spatial value and valid-time interval to be inserted. The list is ordered on the insertion times.

An insertion is done by updating the valid-time end of the most recent entry in the history from being now-relative to being static, so that it meets the beginning of the new entry. This is accomplished by logically deleting the original now-relative data, inserting the data with the new static interval, and finally inserting the new, now-relative data.

After an insertion, a new, planed entry is generated together with its insertion time. This new entry's valid-time begin is normally distributed with deviation Dev and mean equal to its planned insertion time, which, in turn, is generated so that after inserting the new entry, the length of a previous interval will not exceed *VL*. This way, parameter *VL* captures how often objects change, and parameter Dev controls the correlation between transaction and valid time.

In addition to generating operations that insert new entries into active histories, index operations corresponding to one of three actions are generated at each time point:

1. Introducing a new object, thus starting a new history.
2. Ending an active history.
3. Updating a history.

In our experiments, workloads start at time $\mathit{TStart} = 2000$. For the first 2000 time units, at each time point, we introduce a new object. After this initialization, a history may be started, ended, and updated with probability 0.1, 0.1, and 0.8, respectively, at each time point.

A history is started by generating a new pair of a spatial value and a valid-time interval along with an insertion time. A history is ended by updating the valid-time end of the last interval from being now-relative to being static. A history is updated by either inserting a new valid-time interval somewhere between the beginning and the current end of the history, or deleting some existing valid-time interval from the history. The spatial part of an inserted interval is obtained by changing the spatial part of the interval in the history just before it using Δx and Δy. Special action is necessary to ensure that the valid-time intervals in a history continue to meet after an update. Following the insertion of a new interval, all intervals fully covered by it are logically deleted, and partially covered

Table 3. Workload parameters

Parameter	Description	Values Used
SpChange	Maximum value used for Δx and Δy.	0, 40, **200**, 1000, 5000
VL	Maximum valid-time interval length.	250, 500, **1000**, 2000, 4000
$T : S$	Ratio between the temporal and spatial parts of a query.	$1:1000$, $1:10$, **1 : 1**, $10:1$, $1000:1$

intervals are updated. Following the deletion of an interval, the valid-time end of the previous interval, if it exists, is set to the valid-time end of the deleted interval.

Each workload used in an experiment contains 200,000 logical deletions and insertions, so that after running the workload, 200,000 data items are stored in the index. Depending on parameter *VL*, this results in workloads ranging approximately from 8,000 to 34,000 time units in length. A workload also contains 10 queries for each 200 insertions or deletions. The queries are intersection queries represented as 4-D rectangles with square spatial and temporal projections. The valid-time begin of a query is uniformly distributed between 0 and the largest valid-time begin of an entry in the index. For half of the queries, the transaction-time end is set to the current time; for the others, it is uniformly distributed between *TStart* and the current time. The 4-D volumes of queries are uniformly distributed between 1 and $4 \cdot 10^{12}$. The volumes are divided into bitemporal and spatial areas according to the ratio $T : S$, which ranges from $1 : 1000$ for temporally restrictive queries to $1000 : 1$ for spatially restrictive queries (see Table 3).

In addition, we experimented with bitemporal and spatial point queries as extreme cases of temporally and spatially restrictive queries. For these queries, the 4-D volume of a query varied from 1 to 10^8. Another query type that is very likely to be seen in real-world applications is the transaction timeslice, which retrieves the history of some region in space as current in the database at some specific time point. In our experiments we constrain the region of space to a point.

The parameters that control the generation of workloads are given in Table 3. Standard parameter values are given in bold-face. These are the values used if a parameter was not varied in an experiment. In each experiment, we measured the average number of disk I/O operations per single query in a workload.

5.2 Effects of the time-parameterization and the additional split distribution

Compared to the R*-tree, the R$^{\mathrm{ST}}$-tree includes a time parameter and considers one additional distribution of the entries in an over-full node, when it is split. This section studies the effects of these additions.

It was pointed out in Sect. 4.2 that using a moderately large time-parameter value might yield a well-performing tree in a realistic setting. In addition, the characteristics of the data may affect the utility of different time parameters.

A set of experiments was performed to determine the performance resulting from using different time parameters for different workloads. The results were that the performance is not very sensitive to differences in the time parameter value, but that a range of values yield quite similar performance; only the extreme values of 0 and "very large" tend to decrease the query performance (up to 100% for some workloads), especially for rectangle data (as opposed to point data). As a result, we fixed the time-parameter value at 5,000 in all other experiments.

The experiments also show that what is a good time-parameter value is somewhat dependent on the average number of operations per time unit, which, in turn, depends on the *VL* parameter. This should be expected. The time-parameter value, used when splitting a node, should intuitively be dependent on for how long the resulting nodes will exist before being deallocated or reorganized by other splits or multiple insertions and deletions. The smaller the number of operations per time unit, the longer an average node survives, which tends to favor a large time-parameter value.

Switching the attention to the additional distribution considered in the split algorithm, a set of experiments aimed to determine how often this distribution is actually chosen by the split algorithm. If it is chosen quite often, it is interesting to see how its availability improves query performance. The experiments used workloads with different types of queries along with settings for the α parameter that favor the particular type of queries (see Sect. 5.3). Each workload was run with and without the additional distribution available.

For α values favoring queries with $T : S = 1 : 1$ and spatially restrictive queries, i.e., $\alpha \leq 0$, the additional distribution is chosen very infrequently, and so its presence is insignificant. For the workloads with temporally restrictive queries, the additional distribution is chosen in up to 19% of all splits, in turn improving the query performance by 10% to 25% (see Fig. 9).

5.3 Space-time prioritization

Section 4.6 presented the α-parameter, the use of which aimed at tailoring the index to different types of queries. A set of experiments was performed in order to investigate the effect of this parameter on the performance of different types of queries. Eight workloads with different types of queries were run using different α values.

As shown in Figs. 10 and 11, $\alpha = 0$ is best for workloads with neutral queries ($T : S = 1 : 1$). This is as expected. (Note that the workload with $T : S = 1 : 1$ queries appears in both coordinate systems in each figure.)

Interestingly, for spatially restrictive queries, α values of -1.0 and 0, for rectangle data, and -0.5 and 0, for point data,

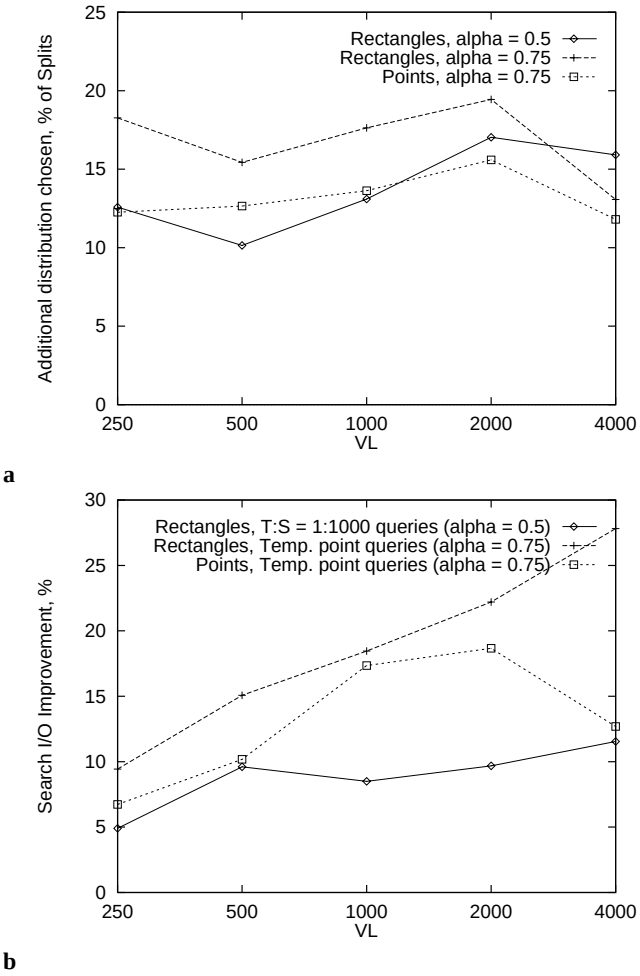

Fig. 9. a Percentage of splits choosing the additional distribution and **b** the resulting query performance improvement (in %) for temporally restrictive queries

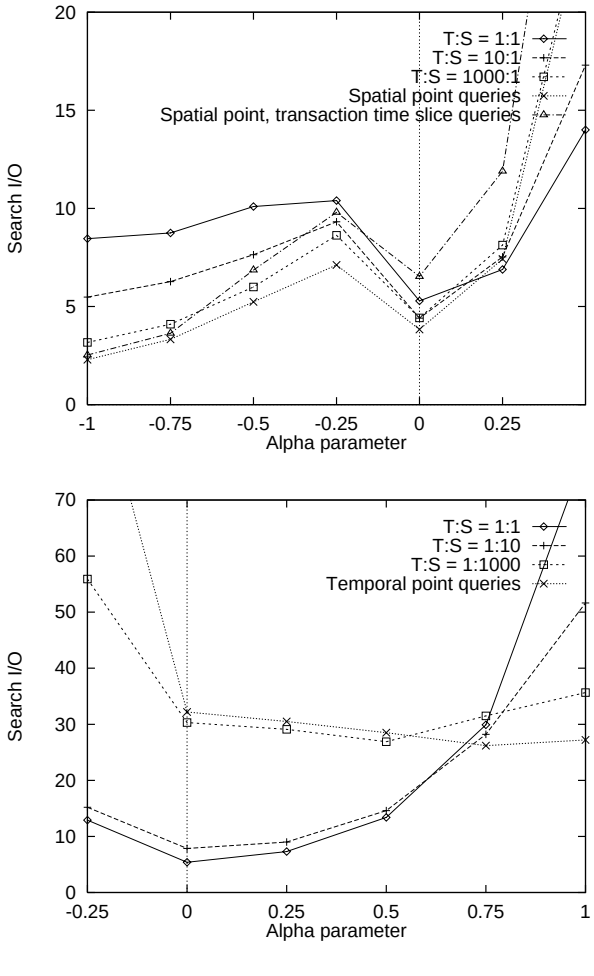

Fig. 10. Query performance for varying α values, rectangle data

outperform other values almost independently from $T : S$ ratio in the queries. For $\alpha < 0$, most of the bounding regions in a tree are growing stair shapes. Only when $\alpha = 0$, the insertion algorithm starts to group growing rectangles, growing stair shapes, and static regions into different nodes and this pays off even for spatially restrictive queries. Nevertheless, values larger than 0 penalize the spatially restrictive queries.

For temporally restrictive queries, depending on the $T : S$ ratio, the optimal α value ranges from 0 to 0.75. The asymmetrical nature of these "spatial" and "temporal" results is as expected and reflects the very different geometries of bitemporal versus spatial regions. In contrast to average spatial regions, bitemporal regions tend to be quite long in transaction time. In addition, the distribution of regions in the bitemporal plane is different from their distribution in the spatial plane.

Figures 10 and 11 also show that temporally restrictive queries require much more I/O than do spatially restrictive queries. This is because spatially restrictive queries are much more selective, which is due to the specifics of our workloads. Intuitively, a temporally restrictive query will retrieve one version of each object, while a spatially restrictive query will retrieve all versions of a few objects. In the workloads, each object has a much smaller number of versions than the total number of objects.

When the mix of queries posed against an $\mathrm{R^{ST}}$-tree does not exhibit a strong temporal restrictiveness or when it is not known which mix to expect, a neutral α value of 0 is preferable. A value of ca. 0.25 is preferable for temporally restrictive queries. For temporal point queries that are non-restrictive in the spatial dimensions, a value of ca. 0.75 shows the best performance.

5.4 Comparison with straightforward R-tree-based indexing*

Replacing all occurrences of UC and NOW $+ \Delta$ in the data by the maximum time values for each dimension renders the data static and enables a standard 4-D R*-tree, followed by a filtering step, to index the spatiotemporal data considered here. The filtering step reverts to the correct temporal extent and reapplies the query predicate, thus eliminating false drops.

Figure 12 compares the $\mathrm{R^{ST}}$-tree to this straightforward solution, without considering the filtering step. Based on the reults shown in Figs. 10 and 11, for $T : S = 1000 : 1$ queries,

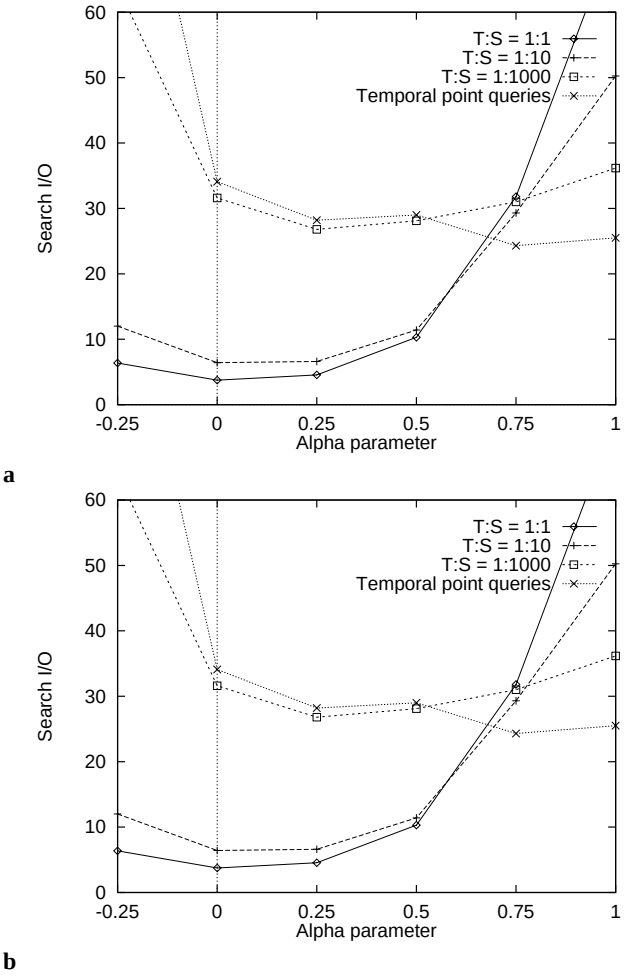

Fig. 11. Query performance for varying α values, point data

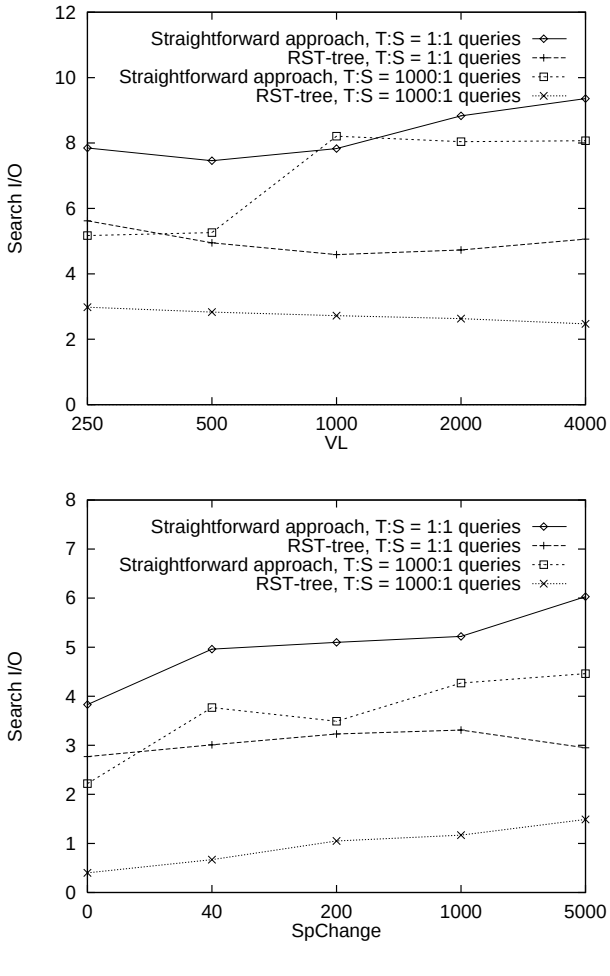

Fig. 12. Query performance for **a** rectangle data with varying *vl* and **b** point data with varying *SpChange*

we used optimal settings of $\alpha = -1$ for rectangle data and $\alpha = -0.5$ for point data.

When comparing Fig. 12 with Figs. 10 and 11, one should take into account that Figs. 10 and 11 correspond to the settings of $VL = 1000$ and $SpChange = 200$. For example, the results for the R$^{\mathrm{ST}}$-tree with $T : S = 1000 : 1$ queries for point data, as reported by the lowest curve in Fig. 12b, can be found in part in Fig. 11 as follows. In Fig. 12b, $\alpha = -0.5$ is used, and in Fig. 11, $SpChange = 200$ is used, so the third data point of the lowest curve in Fig. 12b corresponds to the third data point of the $T : S = 1000 : 1$ curve (with square data points) in the top diagram in Fig. 11.

In our experiments, the R*-tree generally requires from approximately 50% to 100% more I/O operations than the R$^{\mathrm{ST}}$-tree for neutral queries and temporally restrictive queries (not shown in Fig. 12); for spatially restrictive queries, the R*-tree generally requires from approximately 50% to 300% more I/O operations. These numbers are quite significant, especially given that the workloads used in these experiments are quite favorable towards the R*-tree.

The difference in performance among the two indices is highly dependent on the distribution of the queries. As more and more queries reach above the VT $=$ TT primary diagonal in the bitemporal plane, the more false drops will occur in the result obtained from the R*-tree; retrieving these incurs extra I/O not incurred by the R$^{\mathrm{ST}}$-tree. The workloads used in the experiments reported here are very favorable towards the straightforward approach in this regard. For example, the workload used for the data points in Fig. 12a for $VL = 4000$ produced only ca. 9% false drops, but still incurred 100% more I/O operations.

While not illustrated here, the performance of the R*-tree is also highly dependent on the number of now-relative spatiotemporal extents; the more of these, the worse the performance. The R$^{\mathrm{ST}}$-tree does not experience a similar effect.

6 Related proposals

Spatiotemporal indices can be divided into two groups – those that assume discrete motion of objects and those that assume continuous motion. Discretely moving objects are assumed to remain stationary and of non-changing shapes in-between explicit updates to the database (or samples of the objects' movements). In some cases, such as multimedia presentations, objects are not assumed to move at all, they may only appear

and disappear. In other applications, objects are assumed to move continuously, and the database is expected to record interpolated positions of the objects for the time points in-between the samples. As mentioned in the introduction, this paper addresses the problem of indexing discrete movement, and so in this section we mainly consider indices supporting discrete movement.

The previously proposed spatiotemporal indices for discretely moving points or regions [17,24] assume only one time dimension, adopting one of two approaches. The first approach is to use overlapping index structures. These index spatial objects at different time instances and save space by sharing the unchanged parts of the indices. Examples include the HR-tree [18], the MR-tree [28], and the overlapping quadtree [27]). The second approach is to add time to an existing spatial index, as was time an additional spatial dimension. For example, the 3D R-tree [25] indexes the history of evolution of a set of 2-D discretely moving points. The so-called 2+3 R-tree [17] additionally uses a 2D R-tree to index the most recent state of a set of objects. This way, it supports now-relative data in one time dimension.

The overlapping index structures have the disadvantage of using a lot of space, especially if objects change positions frequently. In addition, it is unclear whether this approach may be generalized to two time dimensions. Most importantly, in contrast to the R^{ST}-tree, none of the above-mentioned approaches support now-relative bitemporal data.

The R^*-tree [2] has previously served as the outset for an index [3] that supports now-relative bitemporal data (albeit a somewhat less general kind than the one supported here). As does the R^{ST}-tree, the GR-tree indexes bitemporal data as 2-D regions, which grow continuously if the data is now-relative. Both indices exploit a particular and quite unique property of the R^*-tree, namely that it partitions the data itself. Most other multidimensional access methods, e.g., Quadtrees, partition the embedding space. Unless transformations are employed [4], the continuous growth of the data regions caused by now-relative data renders it difficult to extend the latter type of access method to index now-relative data.

A different treatment of temporal and spatial dimensions has been pursued by Kleiner and Lipeck [14], who use a 3-D R^*-tree to index 2-D objects that move, or change, discretely. Specifically, they experiment with different heuristics in the insertion algorithms. For example, instead of the volume of a minimum bounding cube, they use a quantity that is equal to the product of the minimum bounding cube's extents in the two spatial dimensions plus the temporal extent squared. The authors mention the possibility of prioritizing (or scaling) of the temporal dimension with respect to the spatial dimensions.

For completeness, four recent proposals [1,5,15,20] that index continuous data deserve mention. Unlike the index proposed here, these proposals capture neither the valid nor the transaction time of data; rather, the continuity occurs because they index the current positions of continuously moving objects. The first proposal [15] applies the so-called duality data-transformation to address this indexing problem. The second [1] uses data-transformation, kinetic, and partial-persistence techniques. The third [20] and fourth [5] proposals take the R^*-tree as their outset. While these proposals share this general outset with the R^{ST}-tree, the specifics of these and the present proposal are very different and reflect the differences in the problems being solved. Although designed for continuously moving objects, the proposal of Cai and Revesz [5] could possibly be used to index the valid-time history of discretely moving objects, by assuming zero velocities between the updates of the objects' positions.

7 Conclusions and research directions

This paper addresses the emerging need for efficient support for spatiotemporal data in databases by proposing a new, versatile spatiotemporal indexing technique, termed the R^{ST}-tree, for indexing discretely changing spatial extents-points or rectangles. The R^{ST}-tree solves a new problem: it differs from all previously proposed spatiotemporal indices in that it provides support for both transaction time and valid time, termed bitemporal support, and it accommodates general, now-relative transaction-time and valid-time intervals.

The paper details how to address the special features of the temporal dimensions by using novel bounding regions in the index that grow continuously with the advancement of time and that also take into account the spatial dimensions. Most prominently, these new bounding regions are accompanied by a new set of insertion algorithms. In addition, the tree offers the ability to weigh the spatial and temporal aspects of data, so that either queries that are very selective in the spatial or temporal dimensions are supported more efficiently. The GiST package is used in the implementation of the index. Performance comparisons with a straightforward, R^*-tree-derived spatiotemporal index demonstrate that the new bounding regions and algorithms significantly increase performance. It is also demonstrated that the mechanism for assigning weights makes it possible to better support spatially and temporally selective queries.

In future work, we plan to further investigate support for different types of spatiotemporal queries. Experiments show that an α-parameter of -1.0 seems to provide the best performance for spatially selective queries. A spatial index may be competitive in such cases. In general, given a spatiotemporal query, we would like to be able to determine which existing spatial, temporal, or spatiotemporal index is the better one and to know how the new index compares to the more specialized, existing indices. In addition, we hope to be able to obtain and experiment with real or partly real data, which may yield additional insights.

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.

The authors wish to thank the anonymous referees for their insightful comments, which clarified the presentation.

References

1. Agarwal PK, Arge L, Erickson J (2000) Indexing moving points. Proc. 2000 PODS Conference, pp. 175–186
2. Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. Proc. 1990 ACM SIGMOD Conference, pp. 322–331
3. Bliujūtė R, Jensen CS, Šaltenis S, Slivinskas G (1998) R-tree based indexing of now-relative bitemporal data. Proc. 1998 VLDB Conference, pp. 345–356
4. Bliujūtė R, Jensen CS, Šaltenis S, Slivinskas G (2000) Light-weight indexing of general bitemporal data. Proc. 2000 SSDBM Conference, pp. 125–138
5. Cai M, Revesz P (2000) Parametric R-tree: an index structure for moving objects. Proc. 2000 COMAD Conference, pp. 57–64
6. Clifford J, et al (1997) On the semantics of "Now" in databases. ACM TODS 22(2):171–214
7. Günther O, et al (1998) Benchmarking spatial joins à la carte. Proc. 1998 SSDBM Conference, pp. 32–41
8. Güting RH, et al (2000) A foundation for representing and querying moving objects. ACM TODS, 25(1):1–42, 2000
9. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. Proc. 1984 ACM SIGMOD Conference, pp. 47–57
10. Hellerstein JM, Naughton JF, Pfeffer A (1995) Generalized search trees for database systems. Proc. 1995 VLDB Conference, pp. 562–573
11. Jensen CS, Snodgrass R (1996) Semantics of time-varying information. Inf Syst 21(4):311–352
12. Jensen CS, Snodgrass R (1994) Temporal specialization and generalization. IEEE TKDE 6(6):954–974
13. Kamel I, Faloutsos C (1993) On packing R-trees. Proc. 1993 CIKM Conference, pp. 490–499
14. Kleiner C, Lipeck UW (2000) Efficient index structures for spatio-temporal objects. Proc. 2000 DEXA Conference, pp. 881–888
15. Kollios G, Gunopulos D, Tsotras VJ (1999) On indexing mobile objects. Proc. 1999 PODS Conference, pp. 261–272
16. Leutenegger ST, Lopez MA (1998) The effect of buffering on the performance of R-trees. Proc. 1998 ICDE Conference, pp. 164–171
17. Nascimento MA, Silva JRO, Theodoridis Y (1999) Evaluation of access structures for discretely moving points. Proc. 1999 International Workshop on Spatio-Temporal Database Management, Lecture Notes in Computer Science, vol. 1678. Springer, Berlin Heidelberg New York, 1999, pp. 171–188
18. Nascimento MA, Silva JRO (1998) Towards historical R-trees. Proc. 1998 ACM Symposium on Applied Computing, pp. 235–240
19. Pagel BU, Six HW, Toben H, Widmayer P (1993) Towards an analysis of range query performance in spatial data structures. Proc. 1993 PODS Conference, pp. 214–221
20. Šaltenis S, Jensen CS, Leutenegger S, Lopez M (2000) Indexing the positions of continuously moving objects. Proc. 2000 ACM SIGMOD Conference, pp. 331–342
21. Snodgrass RT, Ahn I (1995) A taxonomy of time in databases. Proc. 1995 ACM SIGMOD Conference, pp. 236–246
22. Snodgrass RT (1987) The temporal query language TQuel ACM TODS 12(2):247–298
23. Theodoridis Y, Sellis T (1996) A model for the prediction of R-tree performance. Proc. 1996 PODS Conference, pp. 161–171
24. Theodoridis Y, Sellis T, Papadopoulos AN, Manolopoulos Y (1998) Specifications for efficient indexing in spatiotemporal databases. Proc. 1998 SSDBM Conference, pp. 123–132
25. Theodoridis Y, Vazirgiannis M, Sellis T (1998) Spatio-temporal indexing for large multimedia applications. Proc. 1998 IEEE Conference on Multimedia Computing and Systems, pp. 441–448
26. Theodoridis Y, Silva JRO, Nascimento MA (1999) On the generation of spatiotemporal datasets. Proc. 1999 International Symposium on Spatial Databases, pp. 147–164
27. Tzouramanis T, Vassilakopoulos M, Manolopoulos Y (1998) Overlapping linear quadtrees: a spatio-temporal access method. Proc. 1998 International Symposium on Advances in Geographic Information Systems, pp. 1–7
28. Xu X, Han J, Lu W (1990) RT-tree: an improved R-tree index structure for spatiotemporal databases. Proc. 1990 International Symposium on Spatial Data Handling, pp. 1040–1049