# 37

# Light-Weight Indexing of General Bitemporal Data

**Rasa Bliujūtė, Christian S. Jensen, Simonas Šaltenis, and Giedrius Slivinskas**

Most data managed by existing, real-world database applications is time referenced. Data warehouses are good examples. Often, two temporal aspects of data are of interest, namely *valid time*, when data is true in the mini-world, and *transaction time*, when data is current in the database, resulting in so-called bitemporal data. Like spatial data, bitemporal data thus has associated two-dimensional regions. Such data is in part naturally now-relative: some data is currently true in the mini-world or is part of the current database state. So, unlike for spatial data, the regions of now-relative bitemporal data grow continuously. Existing indices, including commercially available indices such as $B^+$- and R-trees, typically do not contend well with even small amounts of now-relative data.

This paper proposes a new indexing technique that indexes general bitemporal data efficiently. The technique eliminates the different kinds of growing data regions by means of transformations and then indexes the resulting stationary data regions with four R*-trees, and queries on the original data are mapped to corresponding queries on the transformed data. Extensive performance studies are reported that provide insight into the characteristics and behavior of the four trees storing differently-shaped regions, and they indicate that the new technique yields a performance that is competitive with the best existing index; and unlike this existing index, the new technique does not require extension of the kernel of the DBMS.

# 1 Introduction

Two temporal aspects of data are fundamental—valid time and transaction time [SA85] [JS96]. The valid time of a database tuple is the time when the tuple is true in the modeled reality, the mini-world. A tuple's transaction time is the time during which the tuple is current in the database. These temporal aspects of data are essential in a wide range of existing, real-world applications, including medical, financial, travel, and multimedia applications. For example, real-world (non-textbook!) banking databases do record at which times the (past and present) balances of an account apply. Transaction time is essential in applications where trace-ability or accountability are important. Data with both valid and transaction time associated is termed bitemporal.

Although several dozen data models and temporal query languages have been proposed and although the new SQL standard has an associated Temporal Part [SNO96], the major DBMSs provide little support for temporal data management. As a result, each new database application is consigned to solve anew and in an ad-hoc fashion temporal data management problems that could be solved by the DBMS. This paper proposes an efficient bitemporal indexing technique that can be implemented as a layer on top of an existing DBMS, by an independent third-party developer. In this sense, the index is light-weight.

Existing research shows that regular indices such as $B^+$-trees are unsuited for temporal data [ST97], and there has recently been proposed a number of indices for temporal data. The majority are for transaction-time data, and only few support valid-time data. Significantly less research has been done on creating indices for bitemporal data. Spatial indices are obvious candidates for indexing bitemporal data, due to the similarities between bitemporal and spatial data: the combined valid and transaction time of a tuple can be treated as a region in two-dimensional space. Several existing proposals [KTF95, KTF98, BJSS98] are based on the $R^*$-tree [BEC90], a very efficient member of the R-tree family of spatial indices.

The bitemporal indices generally fall short in efficiently supporting *now*-relative data [CLI97], data for which the end of the valid time or/and transaction time tracks the progressing current time. Now-relative data occurs naturally in most real-world databases. For example, consider the recording of a new employee in a company's database. The time when the employee starts working (valid-time interval begin) is known, but it is unknown when the employee 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 of a tuple is the time when it is inserted into the database. Since we do not know when the tuple will stop being current, the transaction-time end extends to the current time. Two of the existing indices, the 2-R index and the Bitemporal R-tree [KTF98], efficiently support now-relative transaction time, but not now-relative valid time.

Only the GR-tree [BJSS98] supports both now-relative valid time and now-relative transaction time, and thus general bitemporal data, efficiently. But no existing DBMSs support this index, and adding it to a DBMS such as DB2, Oracle, or Sybase would require an extension of the DBMS's kernel. The indexing technique proposed in this paper achieves a performance that is comparable to the GR-tree's, and it may be implemented on-top of any DBMS that supports R-trees, as does, e.g., Informix.

The reliance of R-trees on (minimum) bounding rectangles does not combine well with growing regions. We propose to overcome this problem by applying transformations to the growing now-relative bitemporal data regions that render them stationary and thus amenable to R-tree indexing. Growing regions come in four kinds, depending on whether valid- or/and transaction-time end values are fixed or track the current time, and each kind of regions has its own transformation and is indexed with its own R*-tree. The resulting index is termed the 4-R index. Queries on the index are transformed into four separate queries, one for each tree. The approach may be seen as a generalization of the approach underlying the 2-R index [KTF95] or as an "extreme" case of the GR-tree.

Consisting of four R*-trees, the 4-R index is quite complex. To provide a proper understanding of its properties and behavior, a substantial portion of the paper covers results of quite extensive performance studies of the index, thus providing detailed insight into the specifics of the index and its constituent trees.

The presentation is structured as follows. First, Section 2 briefly describes important concepts and introduces two-dimensional bitemporal regions. Section 3 surveys the existing work related to the indexing of bitemporal data. The 4-R index, including data and query transformations, is described in Section 4. Section 5 presents performance studies. The final section offers conclusions and directions for future work. An appendix proves the correctness of the data and query transformations.

## 2 General Bitemporal Data and Its Representation

As a foundation for understanding the challenges of indexing bitemporal data, this section first describes in more detail the nature of bitemporal data, then characterizes the different kinds of two-dimensional bitemporal data regions.

As mentioned in the previous section, valid time captures when a tuple is true in the modeled reality, and transaction time captures when a tuple is current in the database [SA85, JS96]. These two temporal aspects are orthogonal in that each could be recorded independently, and each has specific properties associated with it. The valid time of a tuple 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 tuple cannot extend beyond the current time and cannot be changed.

TQuel's four-timestamp format [SNO87] (4TS) is the most popular format for bitemporal data representation. With this format, each tuple has a number of non-temporal attributes and 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 its information is valid until the current time or if the tuple is part of the current database state. This is represented in the 4TS format by the use of variables, which denote the current time, for the time attributes $VT^{\dashv}$ and $TT^{\dashv}$ [CLI97]. The variable UC ("until changed") is used for $TT^{\dashv}$, and the variable NOW is used for $VT^{\dashv}$. Table 1 exemplifies bitemporal data. The time granularity is a month, and the current time is assumed to be 9/97.

| | Employee | Department | $TT^{\vdash}$ | $TT^{\dashv}$ | $VT^{\vdash}$ | $VT^{\dashv}$ |
|---|---|---|---|---|---|---|
| (1) | John | Advertising | 4/97 | UC | 3/97 | 5/97 |
| (2) | Tom | Management | 3/97 | 7/97 | 6/97 | 8/97 |
| (3) | Jane | Sales | 5/97 | UC | 5/97 | NOW |
| (4) | Julie | Sales | 3/97 | 7/97 | 3/97 | NOW |
| (5) | Julie | Sales | 8/97 | UC | 3/97 | 7/97 |
| (6) | Michelle | Management | 5/97 | UC | 3/97 | NOW |

Table 1: The EmpDep Relation

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 fact 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 $VT^{\vdash} \leq VT^{\dashv}$and $VT^{\vdash} \leq$ 'current time' if $VT^{\dashv}$ is equal to NOW apply to valid time; and the constraints $TT^{\vdash}$ = 'current time' 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 '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).

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 1–4 in Figure 1 illustrate the *bitemporal regions* of Tuples 1–4, respectively.

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

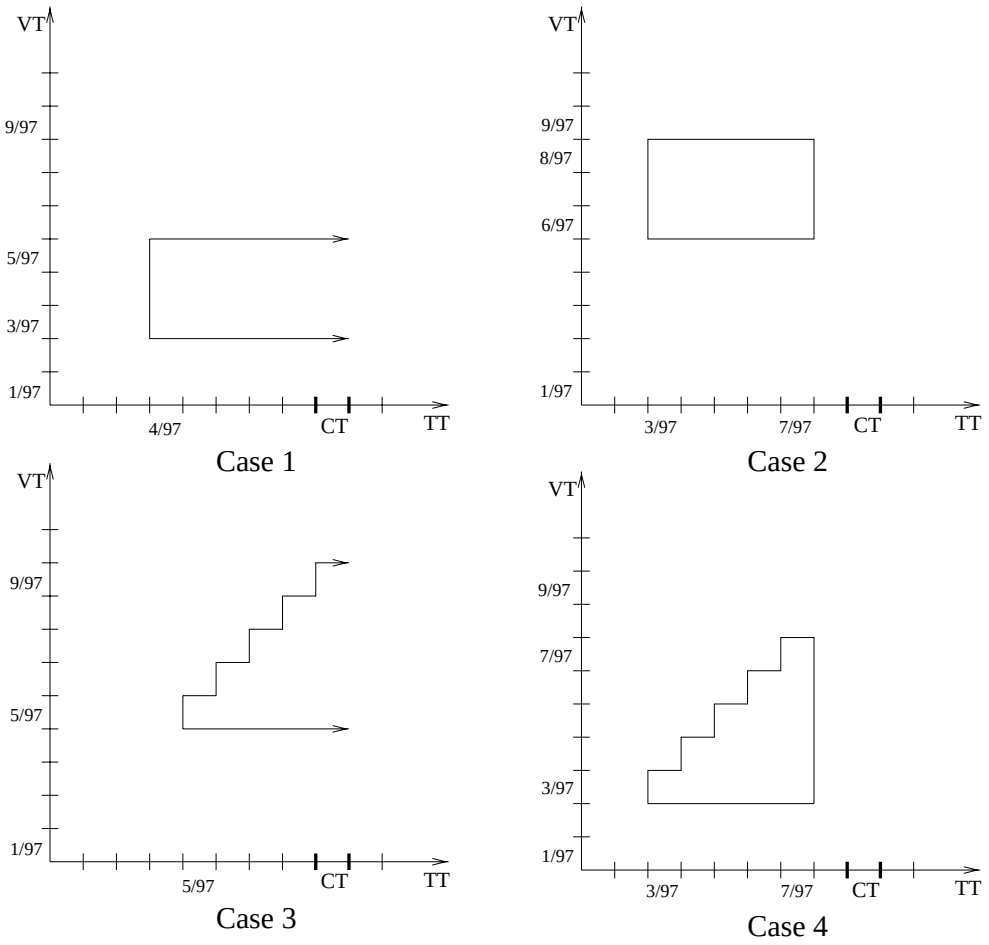


Figure 1: Bitemporal Regions

A now-relative transaction-time interval yields a rectangle that "grows" in the transaction-time direction as time passes (Tuple 1, Case 1). Having both transaction- and valid-time intervals being now-relative yields a stair-shaped region growing in both transaction time and valid time as time passes (Tuple 3, Case 3). If, at some time, a tuple stops being current, the bitemporal region stops growing (Tuples 2 and 4; Cases 2 and 4).

Information may be recorded in the database after it becomes true in the modeled reality. In such situation, if both the transaction- and valid-time intervals are now-relative (Tuple 6), a bitemporal region would be a growing stair-shape with a high first step. 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); 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 of the valid time.

Stated generally, we obtain four combinations of time attributes for which the bitemporal regions are qualitatively different, as illustrated in Figures 1 and 2 where 'tt1', 'tt2', 'vt1', and 'vt2' denote ground values that satisfy the constraints given

above. Stair-shapes having first steps of different heights are not treated as being qualitatively different.

| | $TT^{\vdash}$ | $TT^{\dashv}$ | $VT^{\vdash}$ | $VT^{\dashv}$ | |
|---|---|---|---|---|---|
| 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) |

Figure 2: Possible Combinations of Time Attributes

We have set the context for using spatial indices for indexing bitemporal data. There already exist some indices for bitemporal data that are based on this approach; we discuss them next.

## 3 Overview of the Existing Bitemporal Indices

References [ST97, BER97] provide comprehensive surveys of indices for different types of temporal data. This section focuses solely on the indexing of bitemporal data.

All existing indices for now-relative bitemporal data are based on the idea that bitemporal data can be viewed as a special case of spatial data (recall Figure 1) and that spatial indices can be utilized to index bitemporal data.

Many indices have been developed for spatial data with extent (i.e., non-point data) [SAM90]. One of the most robust such indices 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]. In the R-tree, entries of a leaf-level node store minimum bounding rectangles of spatial regions together with pointers to the data tuples containing those regions. Entries of a non-leaf node store minimum bounding rectangles of child nodes together with pointers to those child nodes. The minimum bounding rectangle of a child node is the rectangle that bounds all entries of that child node. 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 (the dead space is the space in the minimum bounding rectangle not occupied by any of the enclosed rectangles). 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 now-relative bitemporal data, but does not accommodate growing bitemporal regions. The straightforward approach to accommodating growing regions, the *maximum-timestamp approach*, is to represent

these regions using the maximum possible transaction- and valid-time values. As a consequence, their minimum bounding rectangles are static, but the result is also excessive overlap and dead space, as illustrated in Figure 3(a).

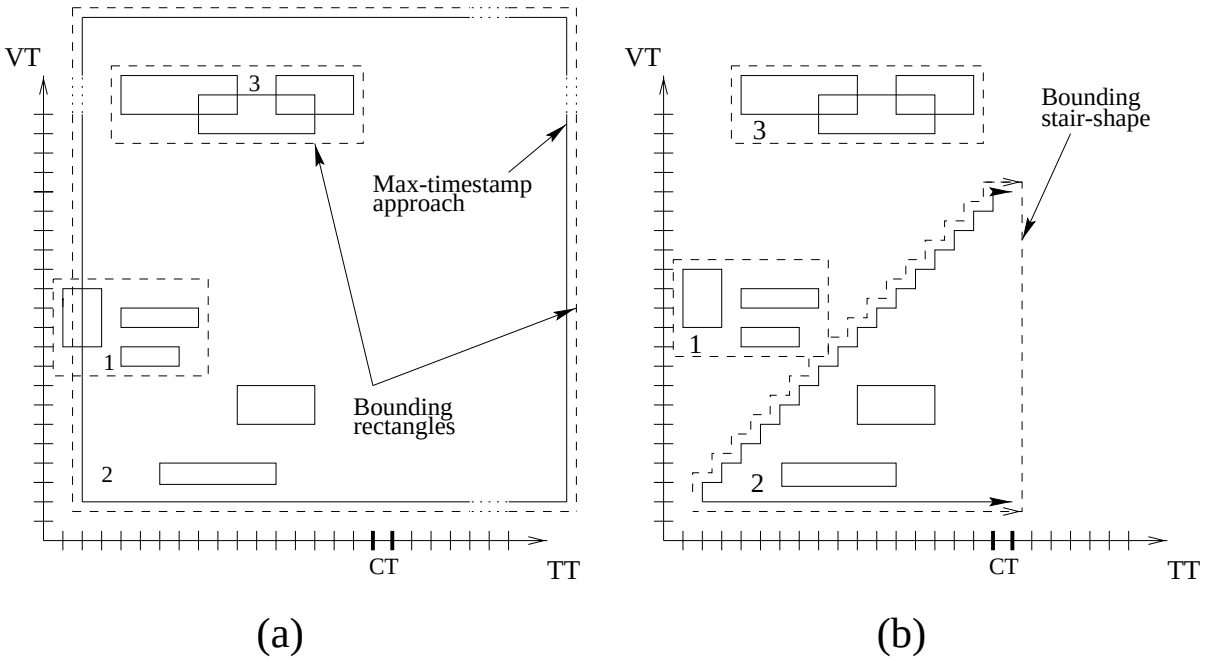


Figure 3: Indexing Growing Bitemporal Regions (a) Using Maximum-Timestamp Values and (b) Using the GR-Tree

Kumar et al. [KTF95, KTF98] propose a new index to handling now-relative transaction time, but do not address now-relative valid time—data regions with an open valid time still must be represented using a maximum-timestamp value. In their index, the 2-R index, they use two R-trees. The front R-tree indexes all growing (i.e., current) rectangles, while the back R-tree indexes all static (i.e., logically deleted) rectangles. Observing that all growing rectangles are in the front tree and that they all extend to the progressing current time, Kumar et al. show that storing only the non-growing transaction-time begin value together with their fixed valid-time interval in the front tree is adequate to support now-relative transaction time. The 2-R index contends well with now-relative transaction time and performs better [BJSS98] than the single, maximum-timestamp R*-tree, but it suffers from the penalty that two trees often have to be searched in a single query, resulting in more disk accesses and diminishing the advantages gained from the decreased overlap. The problem of representing now-relative valid time also remains open in the 2-R index. As discussed already, using the maximum valid-time value for NOW is not promising.

Instead of replacing UC and NOW with maximum time values, the GR-tree [BJSS98] extends the existing R*-tree [BEC90] to allow for the storage of these variables in tree nodes. The GR-tree thus accommodates bitemporal regions (shown

in Figure 1) and uses minimum bounding regions that can be either static or growing and either rectangles or stair-shapes (see Figure 3(b)). It has been shown [BJSS98] that the GR-tree outperforms indices based on the previously described approaches by a factor of 3 or more. The GR-tree is an efficient index, but its implementation requires access to and extension of the DBMS kernel.

Based on the above, we may conclude that two different approaches exist for indexing now-relative bitemporal data. The first approach is to create a new index. The GR-tree, although based on the R*-tree, exemplifies this approach. The second approach is to transform now-relative bitemporal data, in this way eliminating the variables UC and NOW and obtaining static data, and then apply an existing "off-the-shelf" spatial index. This paper explores the latter approach.

Subsequent sections show how now-relative bitemporal data can be efficiently indexed using four R*-trees and employing appropriate data and, consequently, query transformations. The main purpose of the data transformations employed here is to transform growing regions to stationary regions (transformations are used widely in indexing, but always with different purposes).

## 4 The 4-R Index for Now-Relative Bitemporal Data

This section provides a detailed description of the 4-R indexing technique. The idea behind the technique is to apply data transformations that render the continuously growing (now-relative) bitemporal data regions static, upon which existing "off-the-shelf" R-trees may be employed. Specifically, we (1) divide bitemporal data regions into four classes, (2) perform transformations of the regions in each of the classes, thus eliminating any variables, and (3) use separate R-trees for indexing the transformed data regions of each class.

Section 4.1 describes the data transformation. In order to use the index when answering queries, the queries must also be transformed; Section 4.2 presents the query transformation. Section 4.3 concerns the implementation of the 4-R index.

### 4.1 Transformation of Data

In the interest of generality and for our purpose, it is appropriate to model a bitemporal data tuple as a pair of a bitemporal region and a tuple identifier. This corresponds to the information captured at the leaf level of a secondary index such as the R-tree.

The main goal of the transformation of now-relative bitemporal data is to eliminate the variables UC and NOW, so that the data can be indexed with R-trees. To this end, we distinguish between four types of bitemporal data, depending on whether $TT^{\dashv}$ is or is not equal to UC and whether $VT^{\dashv}$ is or is not equal to NOW. For each type, the transformed data is variable-free. The bitemporal region of a

transformed, variable-free bitemporal tuple is always static. Before defining the data transformation, we define the domain of bitemporal data and the domain of variable-free bitemporal data.

**Definition 1** Let the domain of timestamp values be $T$ and the domain of tuple identifiers be *ID*. We then define $D^B$, the domain of bitemporal tuples, and $D^S$, the domain of variable-free bitemporal tuples, as follows.

$$D^B \triangleq \{\langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r\rangle \in T \times T \cup \{\mathrm{UC}\} \times T \times T \cup \{\mathrm{NOW}\} \times ID \mid (\mathrm{TT}_r^{\dashv} = \mathrm{UC} \vee \mathrm{TT}_r^{\vdash} \leq \mathrm{TT}_r^{\dashv}) \wedge (\mathrm{VT}_r^{\dashv} = \mathrm{NOW} \vee \mathrm{VT}_r^{\vdash} \leq \mathrm{VT}_r^{\dashv})\}$$

$$D^S \triangleq \{\langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r\rangle \in T \times T \times T \times T \times ID \mid \mathrm{TT}_r^{\vdash} \leq \mathrm{TT}_r^{\dashv} \wedge \mathrm{VT}_r^{\vdash} \leq \mathrm{VT}_r^{\dashv}\} \square$$

The "*r*" subscripts are used to clearly separate the data rectangles from the query rectangles that will be introduced in the next section.

The data transformation defined next transforms a bitemporal tuple into a variable-free bitemporal tuple augmented by a transformation type.

**Definition 2** Let $R \subseteq D^B$ and define $Type = \{1, 2, 3, 4\}$. Then the 4-R data transformation $\mathcal{T}_D : 2^{D^B} \rightarrow 2^{D^S \times Type}$ is defined as follows.

$$\mathcal{T}_D(R) \triangleq \{\mathcal{T}_r(\langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r\rangle) \mid \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r\rangle \in R\},$$

where

$$\mathcal{T}_r(\langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r\rangle) \triangleq \begin{cases} \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\vdash}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\vdash}, id_r, 1\rangle \\ \qquad \textbf{if } \mathrm{TT}_r^{\dashv} = \mathrm{UC} \wedge \mathrm{VT}_r^{\dashv} = \mathrm{NOW} \\ \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\vdash}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r, 2\rangle \\ \qquad \textbf{if } \mathrm{TT}_r^{\dashv} = \mathrm{UC} \wedge \mathrm{VT}_r^{\dashv} \neq \mathrm{NOW} \\ \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\vdash}, id_r, 3\rangle \\ \qquad \textbf{if } \mathrm{TT}_r^{\dashv} \neq \mathrm{UC} \wedge \mathrm{VT}_r^{\dashv} = \mathrm{NOW} \\ \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r, 4\rangle \\ \qquad \textbf{if } \mathrm{TT}_r^{\dashv} \neq \mathrm{UC} \wedge \mathrm{VT}_r^{\dashv} \neq \mathrm{NOW} \end{cases} \square$$

Four trees, numbered R1 through R4, are created and populated with result tuples according to their types. Figure 4, described next, illustrates the intuition behind the mapping.

Tree R1 indexes the regions that, before the transformation, had non-fixed valid- and transaction-time end values. Knowing that such regions extend to the current time in transaction time ($\mathrm{TT}^{\dashv} = \mathrm{UC}$) and to the line VT = TT ($\mathrm{VT}^{\dashv} = \mathrm{NOW}$), it suffices to represent the regions by two-dimensional points $\langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\vdash}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\vdash}\rangle$, represented in turn by $(\mathrm{TT}^{\vdash}, \mathrm{VT}^{\vdash})$ in the index.

Next, tree R2 indexes regions that had fixed valid-time end values, but non-fixed transaction-time end values prior to the transformation. Because such regions

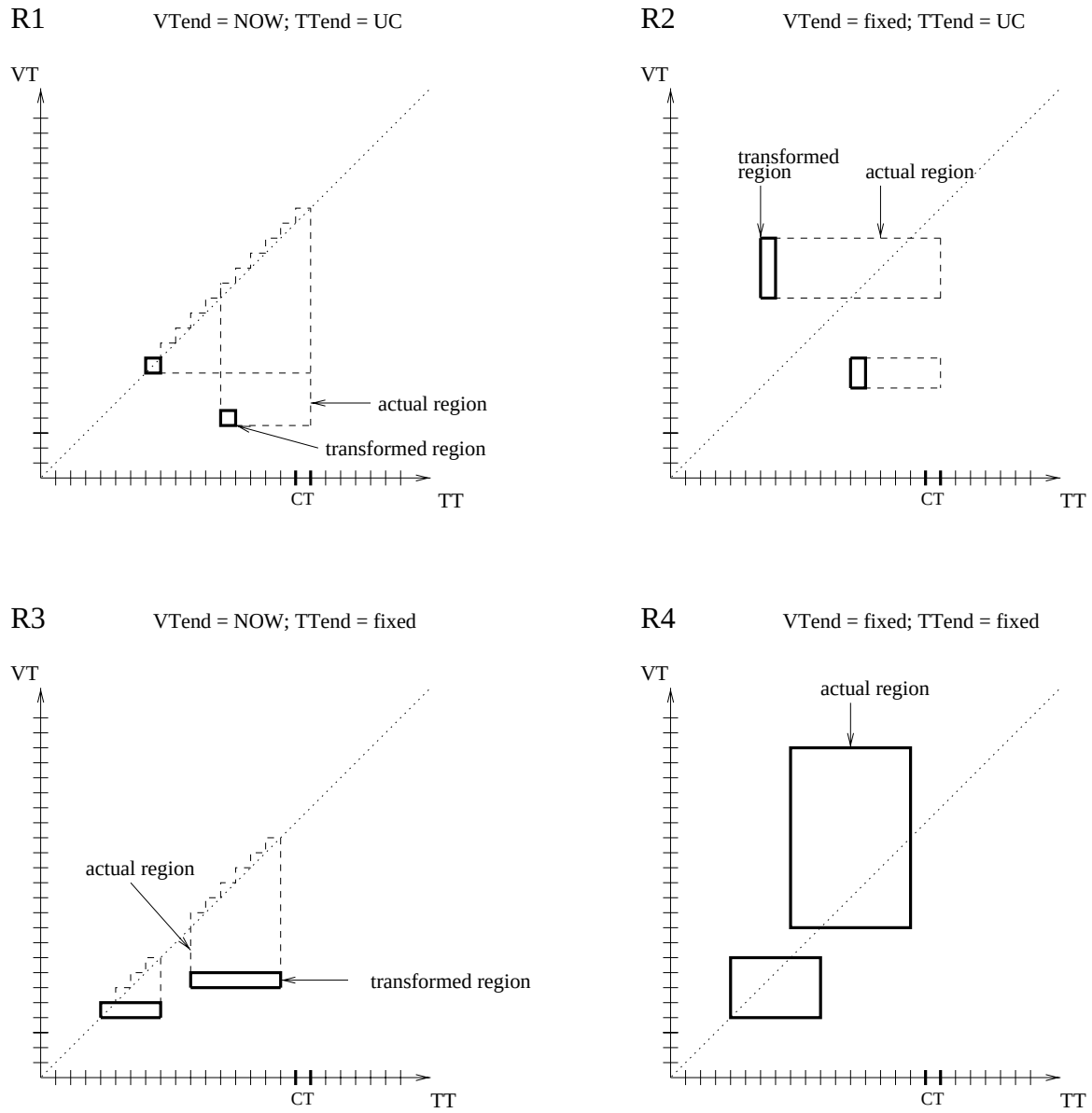


Figure 4: Data Storage in the Four Trees of the 4-R Index

are rectangles that grow in transaction time and extend to the current time in this dimension ($TT^{\dashv} = UC$), we can represent these regions by two-dimensional intervals $\langle TT_r^{\vdash}, TT_r^{\vdash}, VT_r^{\vdash}, VT_r^{\dashv} \rangle$, represented in the index as ($TT^{\vdash}$, $VT^{\vdash}$, $VT^{\dashv}$).

Tree R3 is devoted to regions that, before the transformation, had fixed transaction-time end values, but non-fixed valid-time end values. These regions are all stair-shapes that extend to the line VT = TT ($VT^{\dashv} = NOW$). These may be represented by two-dimensional intervals $\langle TT_r^{\vdash}, TT_r^{\dashv}, VT_r^{\vdash}, VT_r^{\vdash} \rangle$, captured in the index as ($TT^{\vdash}$, $TT^{\dashv}$, $VT^{\vdash}$).

Finally, tree R4 accommodates originally static data regions for which there

is no need for transformation.

The next step is to explore search in the four trees of the 4-R index that accommodate the transformed data. Queries must be transformed as well.

### 4.2 Transformation of Queries

We investigate the most common type of index query, namely the rectangular intersection query. This type of query includes point queries as well as the different kinds of range queries supported by the valid- and transaction-time timeslice operators frequently present in temporal query languages. Let $\langle \mathrm{TT}_q^{\vdash}, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle \in T \times T \times T \times T$ denote the argument rectangle of an intersection query, where $T$ is the domain of timestamps. We will make the reasonable assumptions that $\mathrm{TT}_q^{\vdash} \leq \mathrm{TT}_q^{\dashv}$, $\mathrm{VT}_q^{\vdash} \leq \mathrm{VT}_q^{\dashv}$, and $\mathrm{TT}_q^{\dashv} \leq \mathrm{CT}$ where CT is the value of the current time. The following definition gives the result of an intersection query on bitemporal data.

**Definition 3** Define $\langle \mathrm{TT}_{r_1}^{\vdash}, \mathrm{TT}_{r_1}^{\dashv}, \mathrm{VT}_{r_1}^{\vdash}, \mathrm{VT}_{r_1}^{\dashv} \rangle \cap \langle \mathrm{TT}_{r_2}^{\vdash}, \mathrm{TT}_{r_2}^{\dashv}, \mathrm{VT}_{r_2}^{\vdash}, \mathrm{VT}_{r_2}^{\dashv} \rangle$ by $(\mathrm{TT}_{r_1}^{\vdash} \leq \mathrm{TT}_{r_2}^{\dashv}) \wedge (\mathrm{TT}_{r_1}^{\dashv} \geq \mathrm{TT}_{r_2}^{\vdash}) \wedge (\mathrm{VT}_{r_1}^{\vdash} \leq \mathrm{VT}_{r_2}^{\dashv}) \wedge (\mathrm{VT}_{r_1}^{\dashv} \geq \mathrm{VT}_{r_2}^{\vdash})$. Also let $q = \langle \mathrm{TT}_q^{\vdash}, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle$ and $R \subseteq D^B$. Then an intersection query *Intersect*$^B$ on $R$ with query rectangle $q$ and current time value CT as parameters is defined as follows.

$$
\begin{aligned}
&\mathit{Intersect}^B[q, \mathrm{CT}](R) \triangleq \\
&\quad \{id_r \mid \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r \rangle \in R \wedge \\
&\qquad ((\mathrm{TT}_r^{\dashv} = \mathrm{UC} \wedge \mathrm{VT}_r^{\dashv} = \mathrm{NOW} \wedge \mathrm{TT}_q^{\dashv} \geq \mathrm{VT}_q^{\vdash} \wedge q \cap \\
&\qquad\qquad \langle \mathrm{TT}_r^{\vdash}, \mathrm{CT}, \mathrm{VT}_r^{\vdash}, \mathrm{CT} \rangle) \vee \\
&\qquad\ (\mathrm{TT}_r^{\dashv} = \mathrm{UC} \wedge \mathrm{VT}_r^{\dashv} \neq \mathrm{NOW} \wedge q \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{CT}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv} \rangle) \vee \\
&\qquad\ (\mathrm{TT}_r^{\dashv} \neq \mathrm{UC} \wedge \mathrm{VT}_r^{\dashv} = \mathrm{NOW} \wedge \mathrm{TT}_q^{\dashv} \geq \mathrm{VT}_q^{\vdash} \wedge q \cap \\
&\qquad\qquad \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{TT}_r^{\dashv} \rangle) \vee \\
&\qquad\ (\mathrm{TT}_r^{\dashv} \neq \mathrm{UC} \wedge \mathrm{VT}_r^{\dashv} \neq \mathrm{NOW} \wedge q \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv} \rangle))\} \qquad \square
\end{aligned}
$$

The first line restricts result tuple identifiers to be in the argument tuples. As for Definition 2, each of the next four lines is devoted to one type of bitemporal region. The first disjunct identifies the subset of qualifying growing stair-shapes, the second identifies qualifying growing rectangles, the third, static stair-shaped regions, and the fourth, static rectangles. In cases of stair-shaped regions, in addition to checking the intersection between the bounding rectangle of a region and a query rectangle, it is specified that the lower-right corner of a query must be below or on the VT = TT line (condition $\mathrm{TT}_q^{\dashv} \geq \mathrm{VT}_q^{\vdash}$).

Similarly to Definition 3, we next define the rectangular intersection query on variable-free bitemporal data. The result of this query is independent of the current time.

**Definition 4** Let $q = \langle \mathrm{TT}_q^{\vdash}, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle$ and $S \subseteq D^S$. Then an intersection query $Intersect^S$ on $S$ with a query rectangle $q$ as a parameter is defined as follows.

$$Intersect^S[q](S) \triangleq \{id_r \mid \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r \rangle \in S \wedge q \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv} \rangle\} \quad \square$$

With Definitions 3 and 4 in place, we are now in a position to define the 4-R query transformation, $\mathcal{T}_q$, that goes with the data transformation given in the previous section. The transformation maps an intersection query on the original data to two or four corresponding queries on the transformed data.

**Definition 5** Initially define:

$$\begin{aligned}
&R \subseteq D^B \\
&S = \mathcal{T}_D(R) \\
&S_i = \{\langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r \rangle \mid \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r, i \rangle \in S\}, \\
&\qquad i = 1, 2, 3, 4 \\
&q = \langle \mathrm{TT}_q^{\vdash}, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle \\
&q_1 = \langle 0, \mathrm{TT}_q^{\dashv}, 0, \mathrm{VT}_q^{\dashv} \rangle \\
&q_2 = \langle 0, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle \\
&q_3 = \langle \max(\mathrm{TT}_q^{\vdash}, \mathrm{VT}_q^{\vdash}), \mathrm{TT}_q^{\dashv}, 0, \mathrm{VT}_q^{\dashv} \rangle \\
&q_4 = \langle \mathrm{TT}_q^{\vdash}, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle
\end{aligned}$$

Then the 4-R query transformation $\mathcal{T}_q : [2^{D^B} \rightarrow 2^{ID}] \rightarrow [2^{D^S \times Type} \rightarrow 2^{ID}]$ is defined as follows.

$$\mathcal{T}_q(Intersect^B[q, \mathrm{CT}])(S) \triangleq \begin{cases} \bigcup_{i=1,2,3,4} Intersect^S[q_i](S_i) & \textbf{if } \mathrm{TT}_q^{\dashv} \geq \mathrm{VT}_q^{\vdash} \\ \bigcup_{i=2,4} Intersect^S[q_i](S_i) & \textbf{if } \mathrm{TT}_q^{\dashv} < \mathrm{VT}_q^{\vdash} \end{cases} \quad \square$$

The search spreads to two or four trees and is performed differently in each tree. Figures 5 and 6 illustrate the original search rectangle and the corresponding transformed search rectangle in each tree. A discussion of the search in each of the four trees follows Theorem 1 that states that the combination of the 4-R query and 4-R data transformation yields perfect precision and recall, i.e., is correct. The proof is given in Appendix A.

**Theorem 1** For each $q = \langle \mathrm{TT}_q^{\vdash}, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle$ and each data set $R \subseteq D^B$,

$$Intersect^B[q, \mathrm{CT}](R) = \mathcal{T}_q(Intersect^B[q, \mathrm{CT}])(\mathcal{T}_D(R)). \quad \square$$

Searching tree R1, the argument search rectangle is enlarged to cover the space spanned from the origin of the transaction and valid time to the argument rectangle's top-right corner. Tree R1 contains no data points above the line VT = TT because the original regions encoded by the points in this tree extend only to the line VT = TT. Thus, the transformed search rectangle could also be reduced to

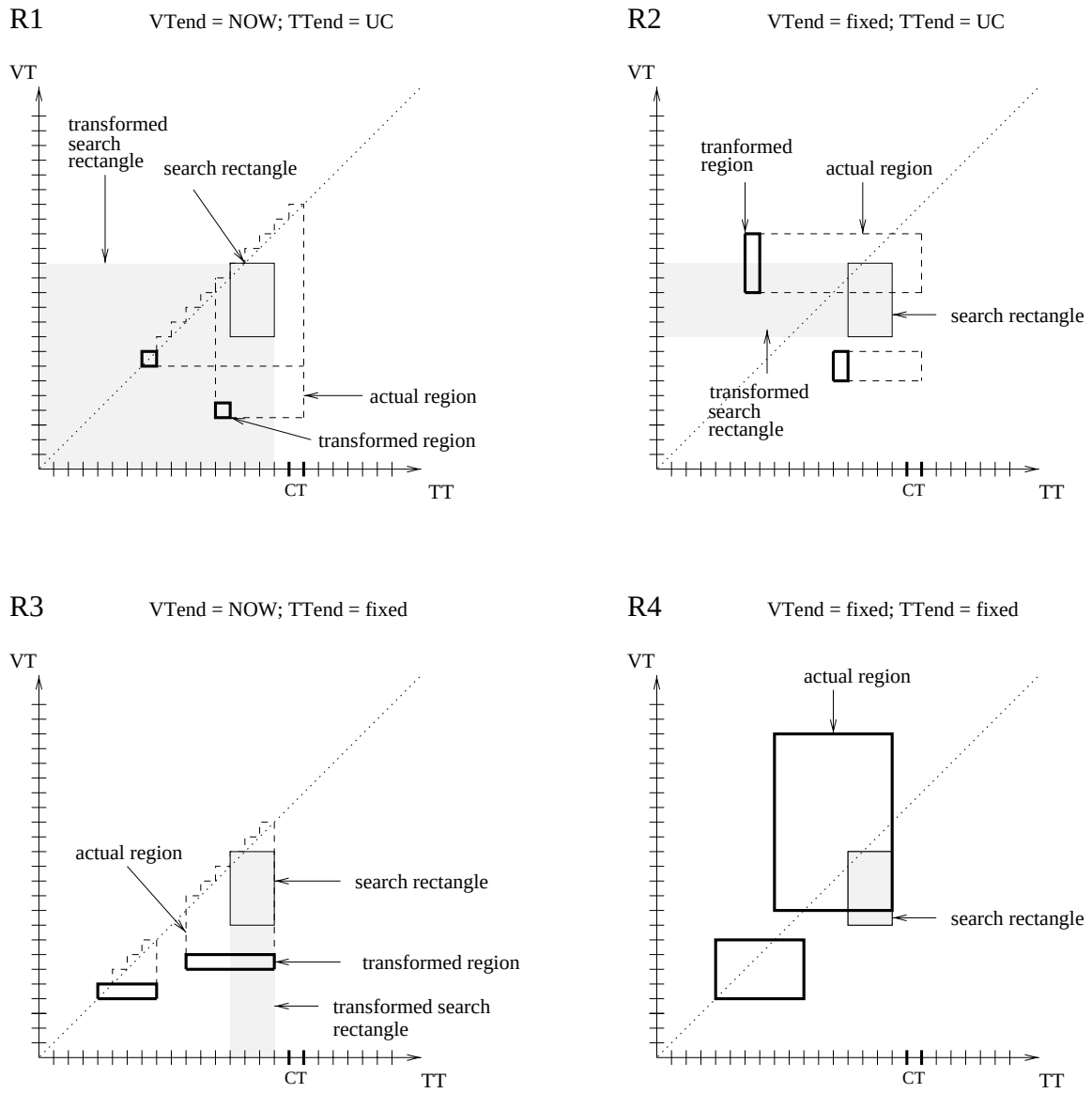


Figure 5: Search in the Four Trees of the 4-R Index

not extend above the line VT = TT without affecting correctness. But this reduction of the search rectangle also does not improve performance because that additional area is empty, so for simplicity, we use the unreduced rectangle.

When searching tree R2, we should look not only for data intervals overlapping with the original search rectangle, but also for intervals to the left of the search rectangle.

The search-rectangle transformation in tree R3 is similar, but now the argument search rectangle is extended downwards instead of to the left, and there is a subtle complication. When part of a search rectangle is below the line VT = TT and another part is above, only the part below this line should be extended downwards;

extending the entire rectangle would yield “false drops” and would thus jeopardize precision. This is illustrated in Figure 6, where extending the entire search rectangle would yield one false drop. In Definition 5, function “max” handles this case.

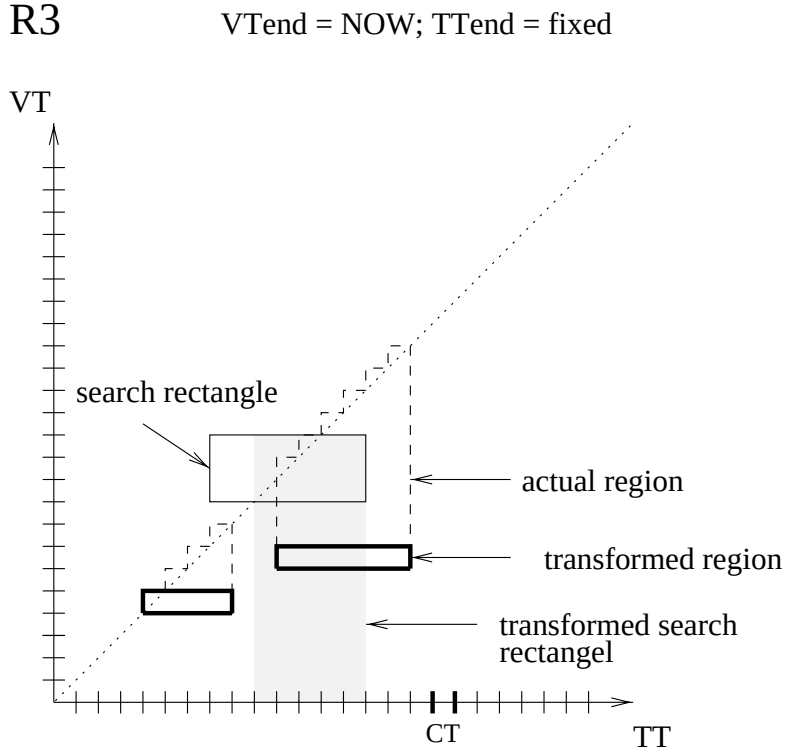


Figure 6: Search Specifics for Tree R3

Since tree R4 indexes untransformed bitemporal rectangles, transformation of the argument search rectangle is unnecessary.

Several noteworthy special cases occur when querying the four trees. In trees R1 and R3, the indexed points and intervals reside only below the line VT = TT, and the regions encoded by these points or intervals also do not extend above this line. Thus, search in these trees is only performed when at least some part of a search rectangle goes below VT = TT.

Another special case is the current-time transaction-timeslice query ($TT_q^{\vdash}$ = $TT_q^{\dashv}$ = CT), which is expected to occur frequently in practice. Current data resides only in trees R1 and R2, so this timeslice query may be restricted to these two trees, ignoring trees R3 and R4. If, in addition, a current time transaction-timeslice query is above the VT = TT line, the only tree to be searched is R2. As a final special case, if such a current-time transaction timeslice has no constraints on valid time, all bitemporal tuples indexed by trees R1 and R2 should simply be returned, and no search is required.

### 4.3 Implementation

An important advantage of the 4-R index is that it may reuse preexisting and efficient R-tree-based indexes such as R*-tree without any modifications, because only points, intervals, and rectangles are indexed—there are no growing bitemporal regions. This means that the 4-R index may be implemented in a layer on top of any DBMS that supports R-trees, with no need for extending the DBMS's query optimizer and recovery and concurrency control subsystems. This renders the index a very practical one.

In addition, the data transformation of bitemporal shapes into intervals and, especially, points may positively impact performance because these simpler shapes take less space and thus lead to trees with higher fanouts of the nodes because more entries fit in one node. The query transformation, where argument rectangles are always enlarged, may have the reverse effect on the performance.

The layer implementing the 4-R technique on top of four R*-trees is responsible for three main tasks.

**insertions** The layer determines into which of the four trees a new data region has to be inserted, and it passes the correct insertion statement to the appropriate tree.

**deletions** The layer determines the tree in which a given data region should be stored, and it passes the correct deletion statement to that tree.

**queries** The layer translates queries into four corresponding queries, each of which is passed to the appropriate tree; and the layer returns the combined result obtained from searching the four trees.

Update is modeled as a deletion followed by an insertion. The algorithms implemented by the layer follow the theory developed earlier in this section and are outlined next.

**Algorithm for Insertion into the 4-R Index**

$$\textbf{insert}(\mathrm{TT}^{\vdash}, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) = \begin{cases} \textbf{insertR1}(\mathrm{TT}^{\vdash}, \mathrm{VT}^{\vdash}) & \textbf{if } \mathrm{TT}^{\dashv} = \mathrm{UC} \wedge \mathrm{VT}^{\dashv} = \mathrm{NOW} \\ \textbf{insertR2}(\mathrm{TT}^{\vdash}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) & \textbf{if } \mathrm{TT}^{\dashv} = \mathrm{UC} \wedge \mathrm{VT}^{\dashv} \neq \mathrm{NOW} \\ \textbf{insertR3}(\mathrm{TT}^{\vdash}, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}) & \textbf{if } \mathrm{TT}^{\dashv} \neq \mathrm{UC} \wedge \mathrm{VT}^{\dashv} = \mathrm{NOW} \\ \textbf{insertR4}(\mathrm{TT}^{\vdash}, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) & \textbf{if } \mathrm{TT}^{\dashv} \neq \mathrm{UC} \wedge \mathrm{VT}^{\dashv} \neq \mathrm{NOW} \end{cases}$$

**Algorithm for Deletion from the 4-R Index**

$$\textbf{delete}(\mathrm{TT}^{\vdash}, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) = \begin{cases} \textbf{deleteR1}(\mathrm{TT}^{\vdash}, \mathrm{VT}^{\vdash}) & \textbf{if } \mathrm{TT}^{\dashv} = \mathrm{UC} \wedge \mathrm{VT}^{\dashv} = \mathrm{NOW} \\ \textbf{deleteR2}(\mathrm{TT}^{\vdash}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) & \textbf{if } \mathrm{TT}^{\dashv} = \mathrm{UC} \wedge \mathrm{VT}^{\dashv} \neq \mathrm{NOW} \\ \textbf{deleteR3}(\mathrm{TT}^{\vdash}, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}) & \textbf{if } \mathrm{TT}^{\dashv} \neq \mathrm{UC} \wedge \mathrm{VT}^{\dashv} = \mathrm{NOW} \\ \textbf{deleteR4}(\mathrm{TT}^{\vdash}, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) & \textbf{if } \mathrm{TT}^{\dashv} \neq \mathrm{UC} \wedge \mathrm{VT}^{\dashv} \neq \mathrm{NOW} \end{cases}$$

**Algorithm for Search in the 4-R Index**

$$
\begin{aligned}
&\textbf{search}(\mathrm{TT}^{\vdash}, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) = \\
&\quad \begin{cases}
\textbf{searchR1}(0, \mathrm{TT}^{\dashv}, 0, \mathrm{VT}^{\dashv}) \cup \\
\textbf{searchR2}(0, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) \cup \\
\textbf{searchR3}(\max(\mathrm{TT}^{\vdash}, \mathrm{VT}^{\vdash}), \mathrm{TT}^{\dashv}, 0, \mathrm{VT}^{\dashv}) \cup \\
\textbf{searchR4}(\mathrm{TT}^{\vdash}, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) \textbf{ if } \mathrm{TT}^{\vdash} \neq \mathrm{CT} \wedge \mathrm{TT}^{\dashv} \geq \mathrm{VT}^{\vdash} \\
\\
\textbf{searchR2}(0, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) \cup \\
\textbf{searchR4}(\mathrm{TT}^{\vdash}, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) \textbf{ if } \mathrm{TT}^{\vdash} \neq \mathrm{CT} \wedge \mathrm{TT}^{\dashv} < \mathrm{VT}^{\vdash} \\
\\
\textbf{searchR1}(0, \mathrm{TT}^{\dashv}, 0, \mathrm{VT}^{\dashv}) \cup \\
\textbf{searchR2}(0, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) \\
\qquad \textbf{if } \mathrm{TT}^{\vdash} = \mathrm{CT} \wedge [\mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}] \neq [t_{min}, t_{max}] \wedge \mathrm{TT}^{\dashv} \geq \mathrm{VT}^{\vdash} \\
\\
\textbf{searchR2}(0, \mathrm{TT}^{\dashv}, \mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}) \\
\qquad \textbf{if } \mathrm{TT}^{\vdash} = \mathrm{CT} \wedge [\mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}] \neq [t_{min}, t_{max}] \wedge \mathrm{TT}^{\dashv} < \mathrm{VT}^{\vdash} \\
\\
\textbf{R1} \cup \textbf{R2} \qquad \textbf{if } \mathrm{TT}^{\vdash} = \mathrm{CT} \wedge [\mathrm{VT}^{\vdash}, \mathrm{VT}^{\dashv}] = [t_{min}, t_{max}]
\end{cases}
\end{aligned}
$$

## 5 Performance

In this section, we compare the search and update performance of four indices: the 4-R index (4-R), the GR-tree, the R-tree (1-R), and the 2-R index (2-R), with the latter two using the maximum-timestamp approach. We first present the strategy used for data and query generation, then discuss the performance results obtained for the data and queries generated using different parameters.

### 5.1 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 is stored in one page. Thus, one node read or write corresponds to one page access (one I/O operation). A buffer of 100 pages is allocated for each index[2]. We include a buffer because Leutenegger and Lopez [LL98] have shown 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

[2] For the 4-R, 4 buffers of 25 pages are allocated, and for the 2-R, 2 buffers of 50 pages are allocated.

marked as "dirty." 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 the search and update performance of the four indices, the same data has to be inserted into the indices, 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—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. First, 4000 insertions are performed in a sequence, and at each later point in time insertion occurs with probability *Ins* and deletion occurs with probability $1 - Ins$.

Several parameters are used for generating the data to be inserted into the indices. The valid-time interval length is uniformly distributed between 0 and the maximum valid-time interval length, *VL*. Alternatively, the valid-time end can be NOW. The percentage of data to be inserted into an index and having valid-time end equal to NOW is denoted as *PNow*. We choose the valid-time begin to be strongly bounded to the data-insertion time. Specifically, it is normally distributed with a mean equal to the insertion time and with some deviation *Dev* that specifies how densely regions inserted into the trees are distributed around the $y = x$ axis. If *Dev* is big and the valid-time end is fixed, the regions are scattered throughout the valid-time universe. (Figure 7 illustrates this point in the GR-tree.) If the valid-time end is equal to NOW, the regions are stair-shapes below and extending to the $y = x$ axis. *Dev* influences how far below the $y = x$ axis the stair-shapes can start.

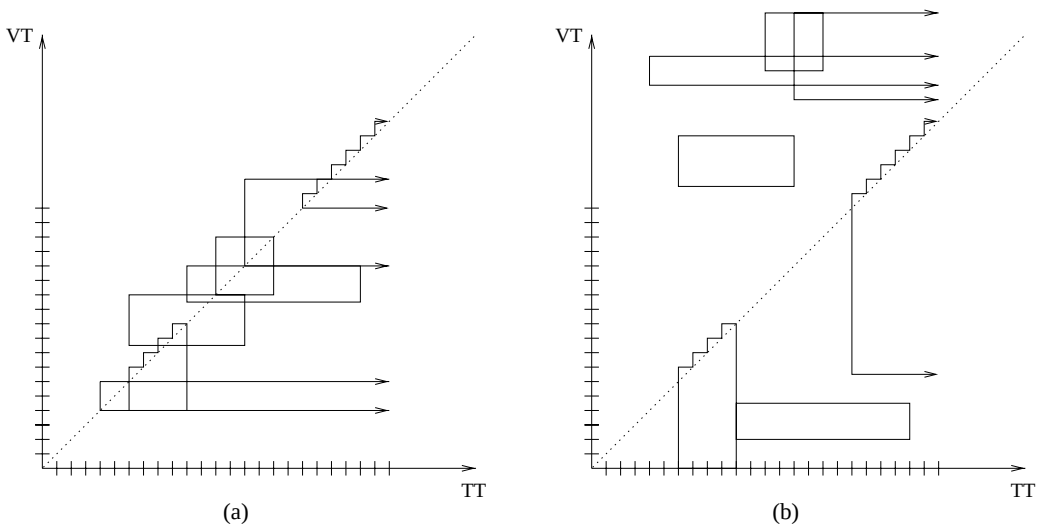


Figure 7: Influence of *Dev* on Data Distribution in the GR-tree; (a) Small and (b) Large *Dev*

A workload also contains queries intermixed with the update operations. We perform bitemporal range, timeslice, and point queries. Parameter *Qcur* denotes the

percentage of "current" queries, i.e., current timeslice and point queries or bitemporal range queries that extend to the current time. Parameters *Qrange*, *Qslice*, and *Qpoint* denote percentages of, respectively, bitemporal range, timeslice, and point queries that will be run. 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 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 summarized in Table 2. In different experiments, the values of these parameters are varied. The values used are given together with the description of each concrete experiment. If the value of some parameter is omitted, the "average" value is used.

| Parameter | Description | Values used | "Average" |
|---|---|---|---|
| *PNow* | percentage of data with valid-time end equal to NOW | 0, 20, 40, 60, 80, 100 | 60 |
| *Ins* | percentage of insertions | 50, 60, 70, 80, 90, 100 | 70 |
| *Dev* | deviation of $VT^{\vdash}$, 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 |
| *Qcur* | percentage of current queries | 0, 25, 50, 75, 100 | 65 |
| *Qrange* | percentage of range queries | 25, 100 | 25 |
| *Qslice* | percentage of timeslice queries | 50, 100 | 50 |
| *Qpoint* | percentage of point queries | 25, 100 | 25 |
| *QmaxI* | maximum valid- and transaction-time intervals given in a query | 1, 100, 300, 500, 1000, 3000 | 300 |

Table 2: Workload Parameters

We intermix queries with update operations in the workload with the aim of measuring search performance throughout the entire index life-time. In the experiments, for each used workload we compute the average I/O cost of update and search operations present in that workload. In each experiment, we also compute pagination, overlap, and dead space in the trees.

The pagination is the percentage of the allocated space that is utilized for data, i.e., it is the percentage of each node's space that on the average is filled with entries. The dead space of a node is the difference between the area of the minimum bounding region of that node and the area of the union of minimum bounding regions of all entries of that node. The dead space of a tree level is the sum of the dead space of each node at that level. The overlap in a node is the difference of the sum

of all areas of that node's entries and the area of the union of all the entries in the node. The overlap in a tree level is the sum of the overlap values of all nodes in that level. The measurement units for dead space and overlap are quadratic time-points, i.e., the same as those for a simple area of a bitemporal region.

## 5.2 Comparison of the Four Bitemporal Indices

In this section, we first discuss general issues in addition to specifics of the four indices that influence their performance. Then we present the results of concrete experiments with the bitemporal indices. For each experiment, the workload parameters are given, search and update performance results are illustrated, and discussions are offered that cover also index pagination, dead space, and overlap.

### General Observations

The number of I/O operations performed during a search depends on index pagination, dead space, and overlap. If nodes are poorly filled, many nodes have to be accessed to retrieve a number of entries. Dead space leads to accessing nodes in vain, eventually finding no qualifying entries. Overlap between nodes leads to I/O-incurring branching of search into several subtrees.

In the 1-R and 2-R, dead space and overlap are excessive because they depend on the maximum-timestamp value, which must be very large in order to exceed any fixed time value used throughout the existence of an index.

Before analyzing the results of the concrete experiments, several issues about the 4-R have to be mentioned. First, if a search rectangle does not extend below the $y = x$ axis, only trees R2 and R4 have to be searched because there will not be qualifying entries in the other two trees. Answering *current* timeslice or point queries, only trees R1 and R2 have to be searched because R3 and R4 do not contain any current data. Manipulating the workload parameters, several extreme cases of the 4-R are possible. If the value of the parameter *Ins*, the percentage of insertions, is 100%, regions will never be deleted, and thus trees R3 and R4, devoted to non-current data, will be empty. If the value of the parameter *PNow*, the percentage of data with valid-time end equal to NOW, is 100%, trees R2 and R4 will be empty because they are devoted to data with fixed valid-time intervals. If *PNow* is 0% then trees R1 and R3 will be empty. Thus, it is appropriate to perform specific experiments to investigate properties of the individual trees of the 4-R index.

### Experiments With Various Data

First, experiments were conducted to find out how varying percentages of data having valid-time end equal to NOW influence search and update performance in the

four indices. Different workloads were constructed changing the value of parameter *PNow*. The search and update performance of the four indices with such data are given in Figures 8(a) and 9(a), respectively.

In general, the GR-tree has the best search performance. The 4-R index does not loose much and even outperforms the GR-tree when *PNow* is 100%, while the two maximum-timestamp-approach-based indices are clearly worse. The problem of the latter two is big overlap and dead space caused by the huge rectangles representing growing bitemporal regions. One of the reasons why the GR-tree is better than the 4-R index is that its pagination (65%) is better than the pagination of the 4-R trees (50% in R3 and 60% in the other trees). Another reason is tree R2 in the 4-R index. This tree indexes data with non-fixed transaction-time intervals (i.e., current data) by physically storing only "vertical lines." Because of the sequential nature of transaction time, the R*-tree algorithms in many cases fail to group these vertical lines into nodes with quadratic minimum bounding rectangles. Often minimum bounding rectangles of the nodes in R2 are long in the valid-time direction. On the other hand, the transformed queries for R2 are not very long in the valid-time direction, but extend to the very beginning of transaction time. Such queries access a lot of nodes with minimum bounding rectangles that are long in valid time, but not many entries (some times none at all) from these nodes qualify for the answer.

However, when *PNow* is near 100%, the 4-R starts to outperform the GR-tree. First, when *PNow* is large, data is concentrated in trees R1 and R3, and the bad performance of the poorly populated R2 does not seriously affect the overall 4-R index performance. Second, with 65% of current queries (we use this number as our average), the majority of the qualifying data is "current" data that resides in R1 and R2. Since R2 is poorly populated, the major part of the qualifying data is retrieved from R1. Tree R1 of the 4-R index has very good selectivity, i.e., usually almost all entries of retrieved nodes qualify for the answer. Consequently, almost the minimal number of nodes have to be accessed during a search, and thus tree R1 significantly contributes to the good performance of the 4-R index.

When *PNow* is low, the 1-R outperforms the 2-R. One of the reasons is the worse pagination of the 2-R (55%, opposed to 65% of the 1-R).

When *PNow* is large, the 2-R outperforms the 1-R. Then the majority of the rectangles in the 1-R extend to the maximum valid-time value; and to the maximum transaction-time value if data is current. The rectangles representing not current data ("short" in the transaction-time direction), however, are mixed with "current" ones in the tree nodes. Thus the selectivity in the 1-R when answering a current query (for which only current rectangles qualify) is not very high. The storing of old and current data in separate trees in the 2-R index shows the advantage.

Concerning updates, the 4-R and 2-R indices achieve better performance than one-tree indices.

Another set of experiments was carried out to investigate how the percentage

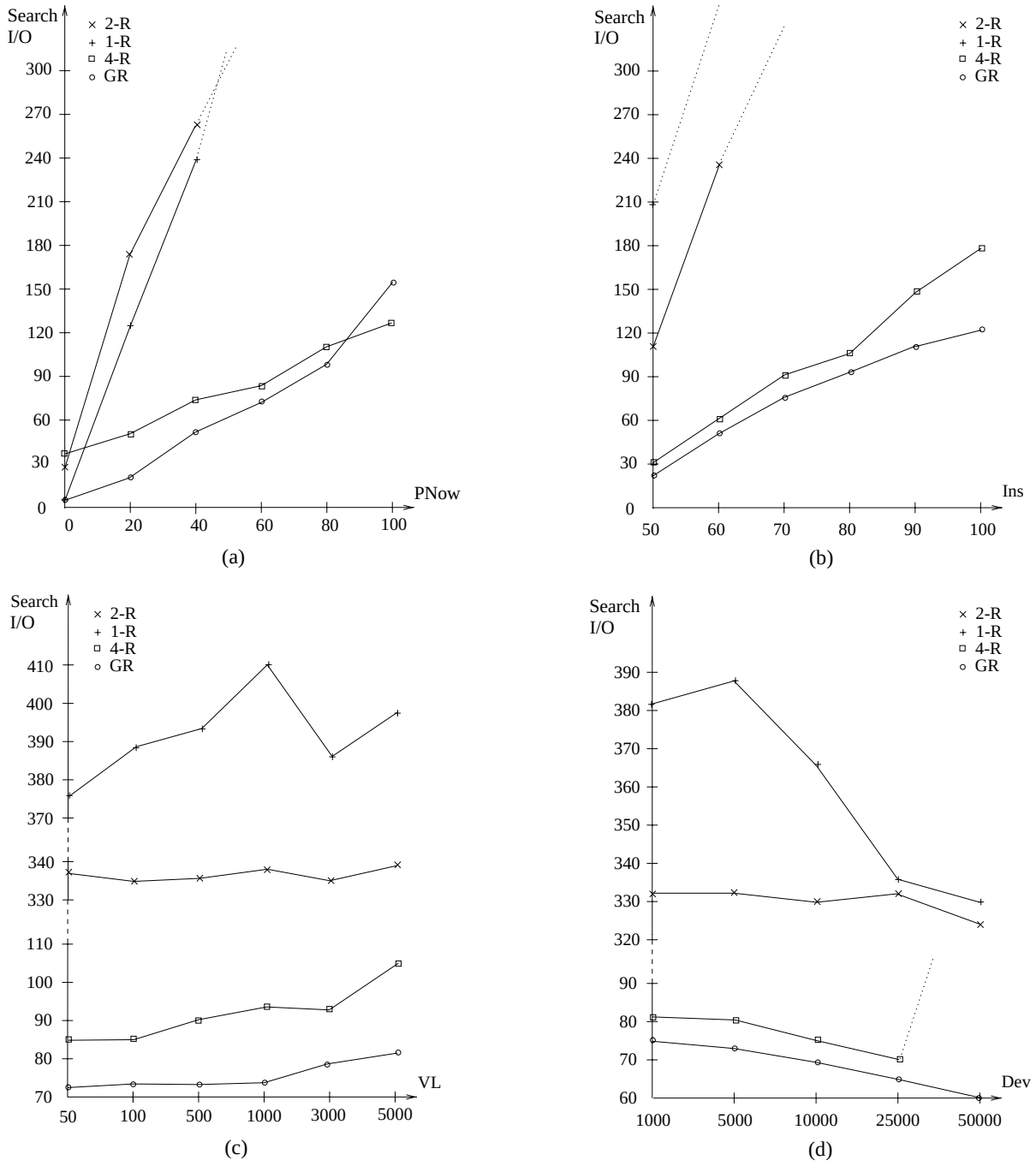


Figure 8: Search Performance for Average Workload Parameters, but (a) Varying *PNow*, (b) *Ins*, (c) *VL*, and (d) *Dev*

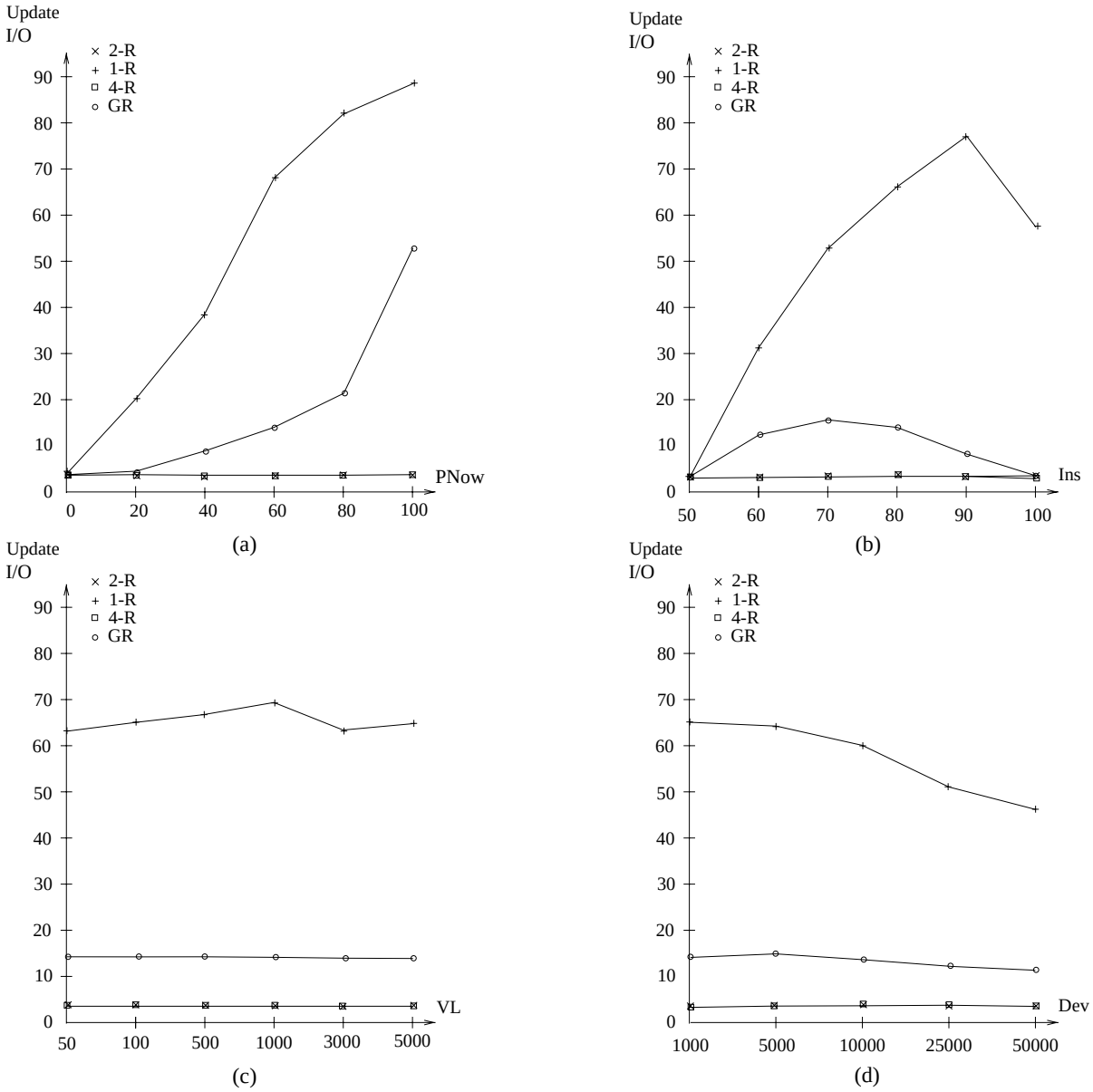


Figure 9: Update Performance for Average Workload Parameters, but (a) Varying *PNow*, (b) *Ins*, (c) *VL*, and (d) *Dev*

of "current" data influences search and update performance in the four indices. In this study, workloads generated with different values of parameter *Ins* were used. Results of the study are given in Figures 8(b) and 9(b).

When the *Ins* value increases, regions are more rarely deleted, current data remains current for a long period, and the amount of current data increases over time. Thus with 65% as the average percentage of current queries (meaning that mostly current data is of interest), when *Ins* increases, more data is retrieved from the indices, resulting in an increasing cost of a search. Overall, the GR-tree has the best performance, but the 4-R is quite close. The difference between the GR-tree and the 4-R index is more visible when *Ins* is close to 100%; the GR-tree then achieves better pagination (70%).

The 4-R and 2-R indices achieve the best update performance also in this study.

Experiments were performed varying also values of parameters *VL* (maximum valid-time interval length) and *Dev* (deviation of VTbegin, when the mean is the insertion time). The results of these experiments are given in Figures 8(c)–(d) and 9(c)–(d), respectively.

### Experiments With Various Queries

For a database where the amount of current data increases over time (using *Ins* equal to 70%, this is the case), the later in the database lifetime a query is issued, the more results are retrieved, which in turn require more I/O operations. We already noticed the impact of the current queries in the previous sections. In this section, we experiment with various kinds of queries. First, we experiment with range, timeslice, point, and "mixed" queries, varying *Qcur*, the percentage of current queries. The results are illustrated in Figures 10 and 11(a). It can be seen that the performance of the 1-R, 2-R, and 4-R indices drops as expected with an increasing amount of current queries. The GR-tree, on the contrary, performs better as the number of current queries increases. Investigating this "phenomenon," we found out the following about the four indices.

The GR-tree nodes containing current data have better pagination than nodes containing non-current data. This is due to the sequential nature of the transaction time. "Current" nodes are filled up by continuously arriving new entries, while new entries are rarely inserted into the "old" nodes, which are left half-full after node splits (the GR-tree split algorithm usually separates older and newer entries). Thus, although the number of retrieved entries increases with an increasing amount of current queries, the qualifying (current) entries are packed into a smaller number of nodes.

In the 4-R index, while the pagination of the trees that contain current data (R1 and R2) is higher than the pagination of trees R3 and R4, the performance is

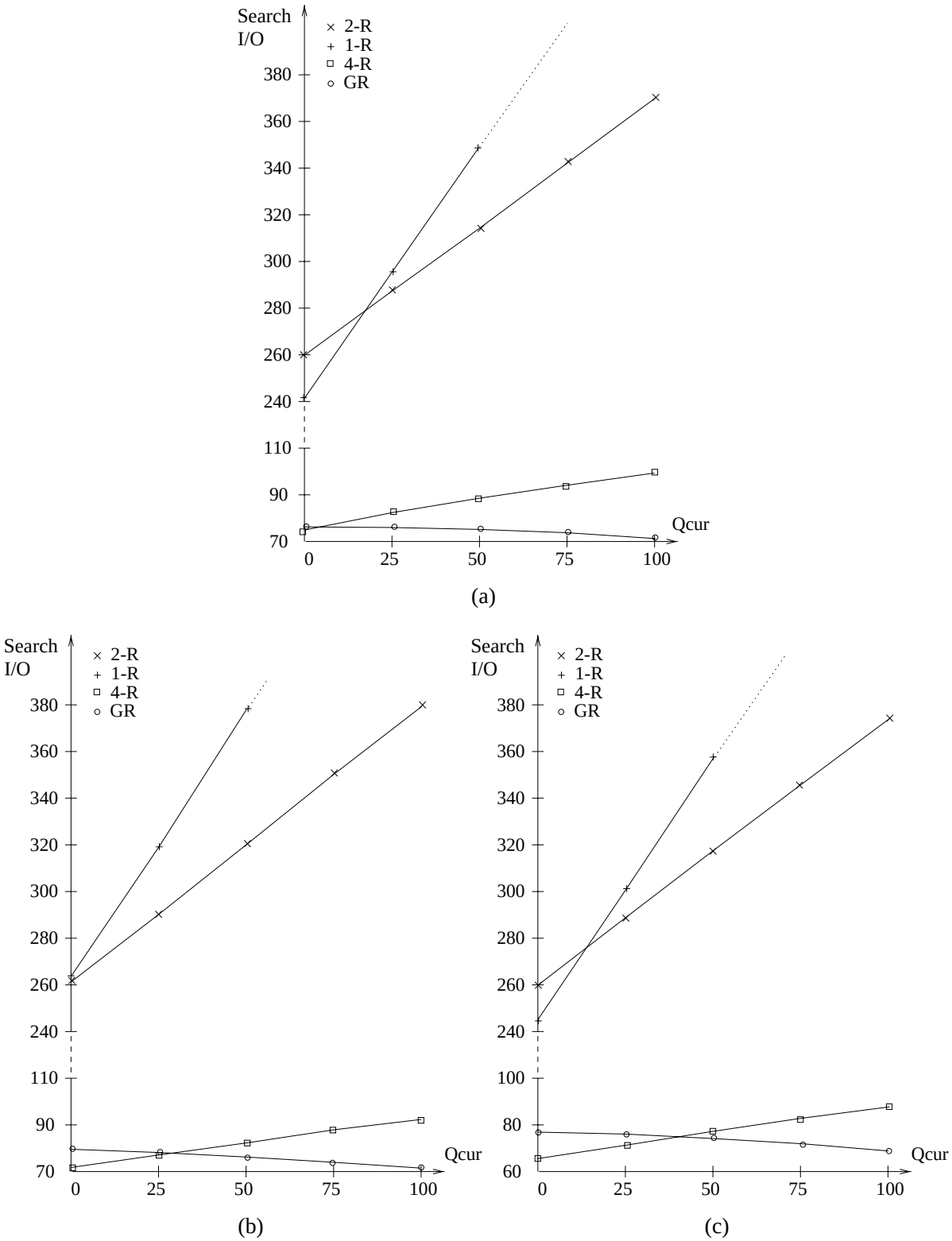


Figure 10: Search Performance for Average Workload Parameters, but Varying *Qcur* for (a) Range Queries, (b) Timeslice Queries, and (c) Point Queries

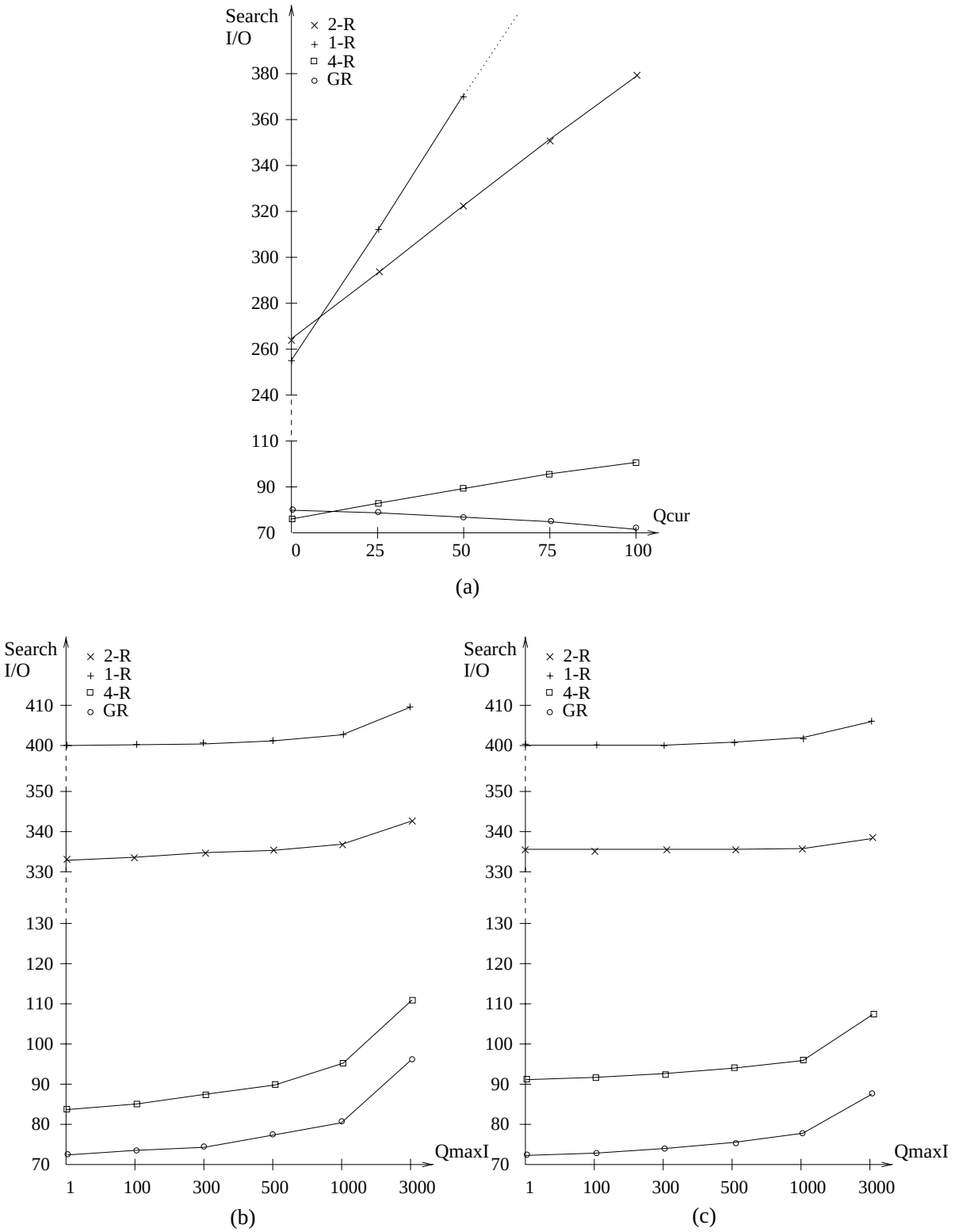


Figure 11: Search Performance for Average Workload Parameters, but Varying *Qcur* for (a) Mixed Queries and Varying *QmaxI* for (b) Range Queries and (c) Timeslice Queries

advertly affected by the bad performance of R2, which in turn is caused by the geometry of the minimum bounding rectangles in this tree (see Section 5.2). The 2-R performs better than the 1-R when the percentage of current queries increases, because current and old data are mixed in the nodes of the 1-R, and the percentage of qualifying entries from each node is low for current queries.

The kind of queries used (range, timeslice, or point queries) practically does not change the results.

The influence of the size of range and timeslice queries was also tested (Figure 11(b) and (c)). As can be expected, bigger queries lead to more entries being retrieved, thus requiring more efforts to perform. This is more pronounced for range queries, which can be expanded to any size in two directions, than for timeslice queries, which can be expanded only in the valid-time direction.

### Experiments With Specialized Data

We experimented with specialized data to understand the properties of the four trees in the 4-R index. We have already observed, in the previous sections, that tree R1 of the 4-R index has very good selectivity, and, in contrast, tree R2 has quite low selectivity.

When *Ins* is 100% (data is never deleted) and *PNow* is 0% (valid-time intervals are fixed), regions representing transformed bitemporal data are solely indexed by tree R2 in the 4-R. The low selectivity in this tree explains why the 4-R index is worse not only than the GR-tree, but also than the R-tree (1-R) (see Figure 12(a)). Note that results of the 2-R index and the 4-R index are the same because the front tree of the 2-R is the same as tree R2 of the 4-R index, since there is no data with now-relative valid-time intervals. The pagination in both the 2-R and the 4-R is 63%, while it is 74% in the GR-tree and 71% in the 1-R.

When *Ins* is 100% (data is never deleted) and *PNow* is 100% (valid-time end is always equal to NOW), regions representing transformed bitemporal data are indexed solely by tree R1 in the 4-R. In tree R1, a rectangle of the transformed query covers a huge area and therefore, during a search, most of the accessed entries qualify for the result. But tree R1 has low pagination (54%) and therefore the 4-R index looses to the GR-tree which has a pagination of 71% (see Figure 12(b)).

## 6 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. Almost none of these support both now-relative valid- and transaction-time intervals, which are accommodated by most of the temporal data models and are natural and meaningful for many kinds of applications. The straightforward R-tree based solution to indexing

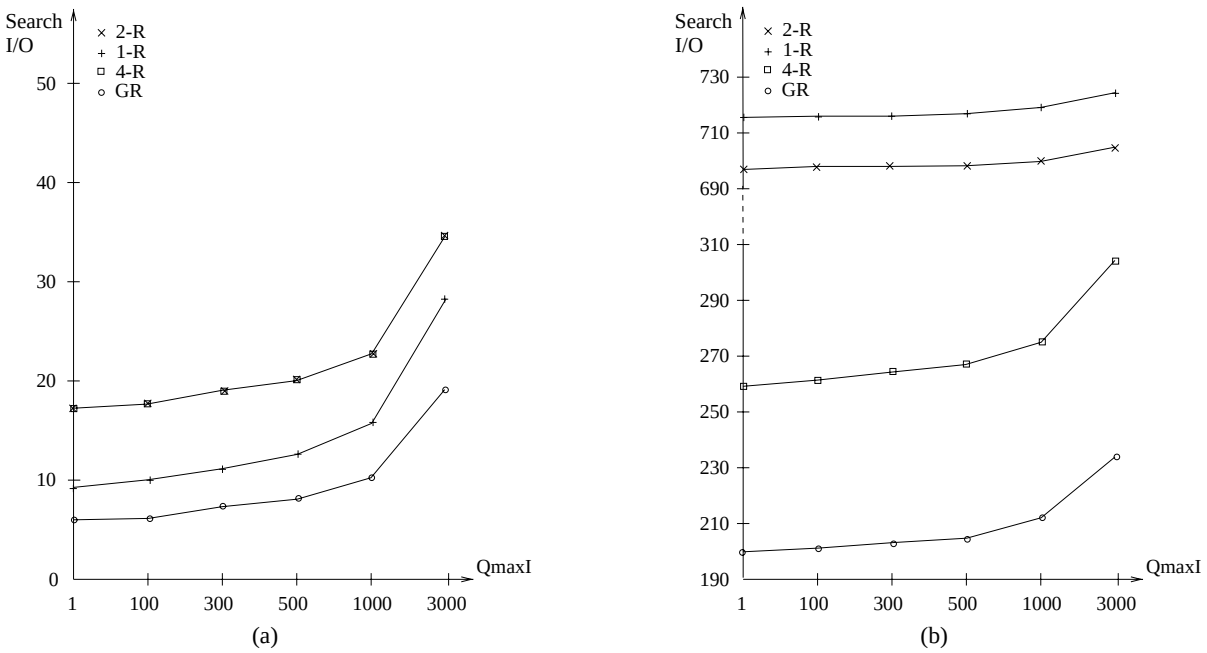


Figure 12: Search Performance for Average Workload Parameters, but *Ins* = 100%, (a) *PNow* = 0% and (b) *PNow* =100%, *Qrange* = 100%, and Varying *QmaxI*

now-relative bitemporal data, the maximum-timestamp approach, is not effective. Another R-tree based index, the GR-tree, employs a special structure and algorithms to contend with now-relative valid- and transaction-time intervals. Although having good performance, the GR-tree is not currently available in any existing DBMSs.

This paper shows how efficient indexing of now-relative bitemporal data can be achieved with little effort by implementing a layer on top of a DBMS supporting R-trees. To make possible the usage of R-trees, the proposed technique applies transformations to bitemporal data regions. Four types of bitemporal regions are distinguished and the transformed counterparts of the bitemporal regions of each of the four types are stored in separate R*-trees. Each query is also transformed into four separate queries, one for each tree.

The proposed index, the 4-R index, may be seen as a generalization of the 2-R index [KTF95], where transformation is used to support now-relative transaction time intervals. In another sense, the 4-R index is a special case of the GR-tree. The insertion algorithm of the GR-tree separates bitemporal data regions of different types into different nodes achieving a tree with groups of nodes storing bitemporal data regions of the same kind. Thus, the 4-R index is an extreme special case of the GR-tree, where such groups of nodes form four different trees.

The detailed performance experiments show that the search performance of the 4-R index is comparable to that of the GR-tree, although usually a little bit worse. In most cases, the difference is not bigger than 25%. On the other hand, the

4-R index shows very good, steady update performance, surpassing the GR-tree's update performance by a big margin.

The performance experiments also reveal some weaknesses of the 4-R index. Most of all, the performance is adversely affected by the unproportionally long, in the valid-time direction, minimum bounding rectangles in tree R2. It seems that what would be desirable is exactly the opposite. Because transformed queries in R2 have relatively long transaction-time extents, the same kind of geometry for minimum bounding rectangles should result in less I/O operations during searches. The same observations hold for tree R3. To achieve the desired geometry of minimum bounding rectangles, the split algorithms of the R*-tree could be modified. This would mean that "off-the-shelf" implementations of the R*-tree cannot be used. To preserve the reusability of "off-the-shelf" technology, the same problem could be addressed by transforming the two-dimensional intervals from R2 and R3 trees into three-dimensional points. Additional query transformations would then also be required.

## Acknowledgements

Curtis E. Dyreson provided insightful comments on an early draft of the paper. This research was supported in part by the Danish Technical Research Council through grant 9700780, by the CHOROCHRONOS project, funded by the European Commission under contract no. FMRX-CT96-0056, and by a grant from the Nykredit Corporation.

## References

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

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

[BJSS98] R. Bliujūtė, C. S. Jensen, S. Šaltenis, and G. Slivinskas. R-tree Based Indexing of Now-Relative Bitemporal Data. To appear in *Proceedings of VLDB* (1998).

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

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

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

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

[JEN98] C. S. Jensen et al. A Consensus Glossary of Temporal Database Concepts. In *Temporal Databases: Research and Practice*, O. Etzion, S. Jajodia, and S. Sripada (eds), Springer-Verlag, pp. 367–405, (1998).

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

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

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

[KTF98] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE TKDE*, 10(1):1–20 (1998).

[LL98] S. T. Leutenegger and M. A. Lopez. The Effect of Buffering on the Performance of R-Trees. *Proceedings of ICDE*, pp. 164–171 (1998).

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

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

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

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

[SNO87] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM TODS*, 12(2):247–298 (1987).

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

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

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

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

## A Correctness of Transformations

Having the definitions given in Section 4, the following theorem holds.

**Theorem 2** For each $q = \langle \mathrm{TT}_q^{\vdash}, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle$ and each data set $R \subseteq D^B$,

$$\mathit{Intersect}^B[q, \mathrm{CT}](R) = \mathcal{T}_q(\mathit{Intersect}^B[q, \mathrm{CT}])(\mathcal{T}_D(R))$$

PROOF: Simplifying Definition 5, we have that $\mathcal{T}_q(\mathit{Intersect}^B[q, \mathrm{CT}])(\mathcal{T}_D(R)) = \bigcup_{i=1,2,3,4} \mathit{Intersect}^S[q_i](S_i) = T_1 \cup T_2 \cup T_3 \cup T_4$. Similarly, according to Definition 3, $\mathit{Intersect}^B[q, \mathrm{CT}](R)$ can be expressed as $Q_1 \cup Q_2 \cup Q_3 \cup Q_4$ where each $Q_i$ is defined by a corresponding disjunct in Definition 3. Note that $Q_1 = Q_3 = \emptyset$ and $T_1 = T_3 = \emptyset$, if $\mathrm{TT}_q^{\dashv} < \mathrm{VT}_q^{\vdash}$. We will prove that $T_i = Q_i$, $i = 1, \ldots, 4$. This actually corresponds to checking correctness of a transformed query in each of the four index trees.

We begin by looking at $T_1$ and $Q_1$. Let us assume that $\mathrm{TT}_q^{\dashv} \geq \mathrm{VT}_q^{\vdash}$ (otherwise both $T_1$ and $Q_1$ are empty). Then, according to Definition 3,

$$Q_1 = \{id_r \mid \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r \rangle \in R \wedge \mathrm{TT}_r^{\dashv} = \mathrm{UC} \wedge \mathrm{VT}_r^{\dashv} = \mathrm{NOW} \wedge q \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{CT}, \mathrm{VT}_r^{\vdash}, \mathrm{CT} \rangle\}.$$

On the other hand, according to Definition 5 and Definition 4,

$$T_1 = \{id_r \mid \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r \rangle \in S_1 \wedge q_1 \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv} \rangle\}.$$

According to the definition of $S_1$ and Definition 2, we have that,

$$T_1 = \{id_r \mid \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv}, id_r \rangle \in R \wedge \mathrm{TT}_r^{\dashv} = \mathrm{UC} \wedge \mathrm{VT}_r^{\dashv} = \mathrm{NOW} \wedge q_1 \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\vdash}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\vdash} \rangle\}.$$

Thus, to prove that $T_1 = Q_1$, we have to prove that

$$\langle \mathrm{TT}_q^{\vdash}, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{CT}, \mathrm{VT}_r^{\vdash}, \mathrm{CT} \rangle \Leftrightarrow \langle 0, \mathrm{TT}_q^{\dashv}, 0, \mathrm{VT}_q^{\dashv} \rangle \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\vdash}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\vdash} \rangle,$$

that is,

$$(\mathrm{TT}_q^{\vdash} \leq \mathrm{CT}) \wedge (\mathrm{TT}_q^{\dashv} \geq \mathrm{TT}_r^{\vdash}) \wedge (\mathrm{VT}_q^{\vdash} \leq \mathrm{CT}) \wedge (\mathrm{VT}_q^{\dashv} \geq \mathrm{VT}_r^{\vdash}) \Leftrightarrow (0 \leq \mathrm{TT}_r^{\vdash}) \wedge (\mathrm{TT}_q^{\dashv} \geq \mathrm{TT}_r^{\vdash}) \wedge (0 \leq \mathrm{VT}_r^{\vdash}) \wedge (\mathrm{VT}_q^{\dashv} \geq \mathrm{VT}_r^{\vdash}).$$

According to constraints put on $q$, $\mathrm{TT}_q^{\vdash} \leq \mathrm{CT}$ and also $\mathrm{VT}_q^{\vdash} \leq \mathrm{CT}$ because we assumed that $\mathrm{TT}_q^{\dashv} \geq \mathrm{VT}_q^{\vdash}$. Thus, the first and the third conditions on both sides of the implications are always satisfied and the others match, proving the implications.

As for $T_1$ and $Q_1$, it can be shown that to prove equality $T_2 = Q_2$, we have to prove that

$$\langle \mathrm{TT}_q^{\vdash}, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{CT}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv} \rangle \Leftrightarrow \langle 0, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\vdash}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\dashv} \rangle,$$

where $r$ is a growing rectangle. Again, expressing the intersection of rectangles by conjunctions, we get:

$$(\mathrm{TT}_q^{\vdash} \le \mathrm{CT}) \wedge (\mathrm{TT}_q^{\dashv} \ge \mathrm{TT}_r^{\vdash}) \wedge (\mathrm{VT}_q^{\vdash} \le \mathrm{VT}_r^{\dashv}) \wedge (\mathrm{VT}_q^{\dashv} \ge \mathrm{VT}_r^{\vdash})$$
$$\Leftrightarrow (0 \le \mathrm{TT}_r^{\vdash}) \wedge (\mathrm{TT}_q^{\dashv} \ge \mathrm{TT}_r^{\vdash}) \wedge (\mathrm{VT}_q^{\vdash} \le \mathrm{VT}_r^{\dashv}) \wedge (\mathrm{VT}_q^{\dashv} \ge \mathrm{VT}_r^{\vdash}).$$

The first condition on both sides of the implications is always satisfied and the others match, thus the implications are proved.

As for $T_1$ and $Q_1$, to prove equality $T_3 = Q_3$, we have to prove that

$$\langle \mathrm{TT}_q^{\vdash}, \mathrm{TT}_q^{\dashv}, \mathrm{VT}_q^{\vdash}, \mathrm{VT}_q^{\dashv} \rangle \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{TT}_r^{\dashv} \rangle$$
$$\Leftrightarrow \langle \max(\mathrm{TT}_q^{\vdash}, \mathrm{VT}_q^{\vdash}), \mathrm{TT}_q^{\dashv}, 0, \mathrm{VT}_q^{\dashv} \rangle \cap \langle \mathrm{TT}_r^{\vdash}, \mathrm{TT}_r^{\dashv}, \mathrm{VT}_r^{\vdash}, \mathrm{VT}_r^{\vdash} \rangle,$$

where $r$ is a static stair-shaped region. Expressing the intersection of rectangles by conjunctions, we get:

$$(\mathrm{TT}_q^{\vdash} \le \mathrm{TT}_r^{\dashv}) \wedge (\mathrm{TT}_q^{\dashv} \ge \mathrm{TT}_r^{\vdash}) \wedge (\mathrm{VT}_q^{\vdash} \le \mathrm{TT}_r^{\dashv}) \wedge (\mathrm{VT}_q^{\dashv} \ge \mathrm{VT}_r^{\vdash})$$
$$\Leftrightarrow (\max(\mathrm{TT}_q^{\vdash}, \mathrm{VT}_q^{\vdash}) \le \mathrm{TT}_r^{\dashv}) \wedge (\mathrm{TT}_q^{\dashv} \ge \mathrm{TT}_r^{\vdash}) \wedge (0 \le \mathrm{VT}_r^{\vdash}) \wedge (\mathrm{VT}_q^{\dashv} \ge \mathrm{VT}_r^{\vdash}).$$

The third condition on the right-hand side of the implications is always true, the second and the fourth conditions on both sides of the implications match and $(\mathrm{TT}_q^{\vdash} \le \mathrm{TT}_r^{\dashv}) \wedge (\mathrm{VT}_q^{\vdash} \le \mathrm{TT}_r^{\dashv}) \Leftrightarrow \max(\mathrm{TT}_q^{\vdash}, \mathrm{VT}_q^{\vdash}) \le \mathrm{TT}_r^{\dashv}$.

Based on Definition 5, it follows that $T_4$ is the result of an untransformed query $q$ on untransformed static rectangles from $D$. It is easy to conclude that $T_4 = Q_4$. □