# BI-TEMPORAL INDEXING

Mirella M. Moro
Universidade Federal do Rio Grande do Sul
Porto Alegre, RS, Brazil
http://www.inf.ufrgs.br/~mirella/

Vassilis J. Tsotras
University of California, Riverside
Riverside, CA 92521, USA
http://www.cs.ucr.edu/~tsotras

## SYNONYMS

Bi-temporal Access Methods

## DEFINITION

A bi-temporal index is a data structure that supports both temporal time dimensions, namely, transaction-time (the time when a fact is stored in the database) and valid-time (the time when a fact becomes valid in reality). The characteristics of the time dimensions supported imply various properties that the bi-temporal index should have to be efficient. As traditional indices, the performance of a temporal index is described by three costs: (i) storage cost (i.e., the number of pages the index occupies on the disk), (ii) update cost (the number of pages accessed to perform an update on the index; for example when adding, deleting or updating a record), and (iii) query cost (the number of pages accessed for the index to answer a query).

## HISTORICAL BACKGROUND

Most of the early work on temporal indexing has concentrated on providing solutions for transaction-time databases. A basic property of transaction-time is that it always *increases*. Each newly recorded piece of data is time-stamped with a new, larger, transaction time. The immediate implication of this property is that previous transaction times *cannot* be changed. Hence, a transaction-time database can “rollback” to, or answer queries for, any of its previous states.

On the other hand, a valid-time database maintains the entire temporal behavior of an enterprise as best known now. It stores the current knowledge about the enterprise's past, current or even future behavior. If errors are discovered about this temporal behavior, they are corrected by modifying the database. In general, if the knowledge about the enterprise is updated, the new knowledge modifies the existing one. When a correction or an update is applied, previous values are not retained. It is thus not possible to view the database as it was before the correction/update.

By supporting both valid and transaction time, a bi-temporal database combines the features of the other temporal database types. While it keeps its past states, it also supports changes anywhere in the valid time domain. Hence, the overlapping and persistent methodologies proposed for transaction-time indexing (for details see chapter on Transaction-Time Indexing) can be applied [6, 9, 5]. The difference with transaction-time indexing is that the underlying access method should be able to dynamically manage intervals (like an R-tree, a quad-tree etc.). For a worst-case comparison of temporal access methods, the reader is referred to [7].

## SCIENTIFIC FUNDAMENTALS

When considering temporal indexing, it is important to realize that the valid and transaction time dimensions are *orthogonal* [8]. While in various scenarios it may be assumed that data about a fact is entered in the database at the same time as when it happens in the real world (i.e., valid and transaction time coincide), in practice, there are many applications where this assumption does not hold. For example, data records about the sales that occurred during a given day are recorded in the database at the end of the day (when batch processing of all data collected during the day is performed). Moreover, a recorded valid time may represent a later time instant than the transaction time when it was recorded. For example, a contract may be valid for an interval that is later than the (transaction) time when this information was entered in the database. The above properties are critical in the design of a bi-temporal access method since the support of both valid and transaction time affects directly the way records are created or updated. Note that the term “interval” is used here to mean a “convex subset of the time domain" (and not a "directed duration"). This concept has also been named a “period”; in this discussion however, only the term “interval” is used.

The reader is referred to the entry on Transaction-Time Indexing, in which a transaction time database was abstracted as an evolving collection of objects; updates arrive in increasing transaction-time order and are always applied on the latest state of this set. In other words, previous states cannot be changed. Thus a transaction-time database represents and stores the database activity; objects are associated with intervals based on this database activity. In contrast, in the chapter on Valid-Time Indexing, a valid-time database was abstracted as an evolving collection of interval-objects, where each interval represents the validity interval of an object. The allowable changes in this environment are the addition/deletion/ modification of an interval-object. A difference with the transaction-time abstraction is that the collection's evolution (past states) is *not* kept. Note that when considering the valid time dimension, changes do not necessarily come in increasing time order; rather they can affect *any* interval in the collection. This implies that a valid-time database can correct errors in previously recorded data. However, only a single data state is kept, the one resulting after the correction is applied.

A bi-temporal database has the characteristics of both approaches. Its abstraction maintains the evolution (through the support of transaction-time) of a dynamic collection of (valid-time) interval-objects. Figure 1 offers a conceptual view of a bi-temporal database. Instead of maintaining a single collection of interval-objects (as a valid-time database does) a bi-temporal database maintains a sequence of such collections $C(t_i)$ indexed by transaction-time. Assume that each interval $I$ represents the validity interval of a contract in a company. In this environment, the user can represent how the knowledge about company contracts evolved. In Figure 1, the $t$-axis ($v$-axis) corresponds to transaction (valid) times. At transaction time $t_1$, the database starts with interval-objects $I_x$ and $I_y$. At $t_2$, a new interval-object $I_z$ is recorded, etc. At $t_5$ the valid-time interval of object $I_x$ is modified to a new length.

When an interval-object $I_j$ is inserted in the database at transaction-time $t$, a record is created with the object's surrogate (contract_no $I_j$), a valid-time interval (contract duration), and an initial transaction-time interval [$t$, $UC$). When an object is inserted, it is not yet known if it (ever) will be updated. Therefore, the right endpoint of the transaction-time interval is filled with the variable $UC$ (Until Changed), which will be changed to another transaction time if this object is later updated. For example, the record for interval-object $I_z$ has transaction-time interval [$t_2$, $t_4$), because it was inserted in the database at transaction-time $t_2$ and was "deleted" at $t_4$. Note that the collections $C(t_3)$ and $C(t_4)$ correspond to the collections $C_a$ and $C_b$ of Figure 1 in the Valid-Time Indexing chapter, assuming that at transaction-time $t_4$ the erroneous contract $I_z$ was deleted from the database.

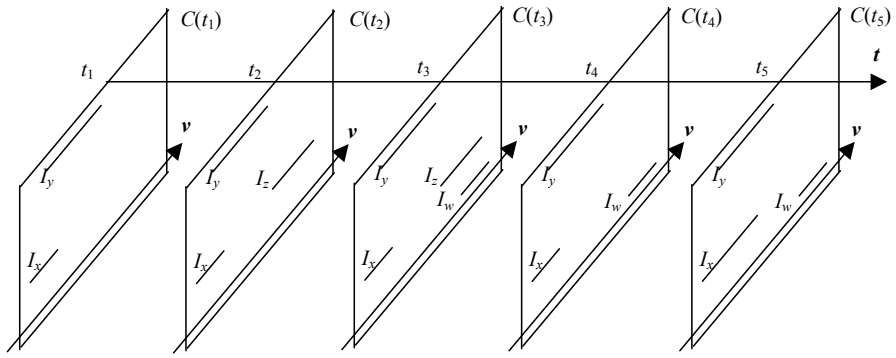


Figure 1. A bi-temporal database.

Based on the above discussion, an index for a bi-temporal database should: (i) store past states, (ii) support addition/deletion/modification changes on the interval-objects of its current logical state, and (iii) efficiently access and query the interval-objects on any state.

Figure 1 summarizes the differences among the various database types. Each collection $C(t_i)$ can be thought of on its own, as a separate valid-time database. A valid-time database differs from a bi-temporal

database since it keeps *only one* collection of interval-objects (the latest). A transaction-time database differs from a bi-temporal database in that it maintains the history of an evolving set of *plain*-objects instead of *interval*-objects. A transaction-time database differs from a conventional (non-temporal) database in that it also keeps its *past* states instead of only the latest state. Finally, the difference between a valid-time and a conventional database is that the former keeps *interval*-objects (and these intervals can be queried).

There are three approaches that can be used for indexing bi-temporal databases.

**Approach 1:** The first one is to have each bi-temporal object represented by a "bounding rectangle" created by the object's valid and transaction-time intervals, and to store it in a conventional multi-dimensional structure like the R-tree. While this approach has the advantage of using a single index to support both time dimensions, the characteristics of transaction-time create a serious overlapping problem [5]. A bi-temporal object with valid-time interval *I* that is inserted in the database at transaction time *t*, is represented by a rectangle with a transaction-time interval of the form [*t*, *UC*). All bi-temporal objects that have not been deleted (in the transaction sense) will share the common transaction-time endpoint *UC* (which in a typical implementation, could be represented by the largest possible transaction time). Furthermore, intervals that remain unchanged will create long (in the transaction-time axis) rectangles, a reason for further overlapping. A simple bi-temporal query that asks for all valid time intervals that at transaction time $t_i$ contained valid time $v_j$, corresponds to finding all rectangles that contain point ($t_i$, $v_j$).

Figure 2 illustrates the bounding-rectangle approach; only the valid and transaction axis are shown. At $t_5$, the valid-time interval $I_1$ is modified (enlarged). As a result, the initial rectangle for $I_1$ ends at $t_5$, and a new enlarged rectangle is inserted ranging from $t_5$ to *UC*.

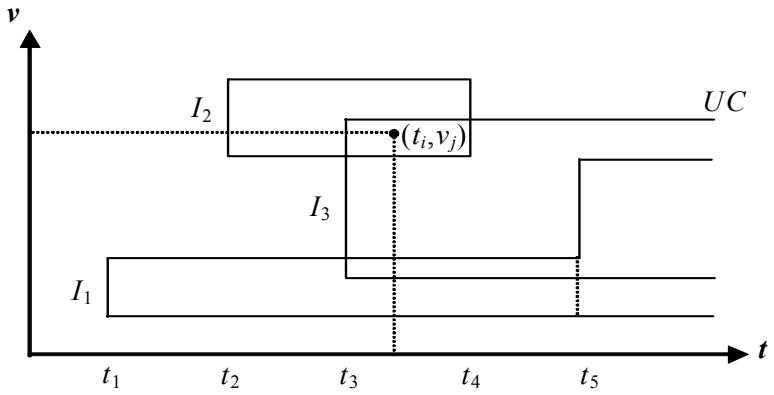


Figure 2. The bounding-rectangle approach for bi-temporal objects.

**Approach 2:** To avoid overlapping, the use of two R-trees has also been proposed [5]. When a bi-temporal object with valid-time interval *I* is added in the database at transaction-time *t*, it is inserted at the *front* R-tree. This tree keeps bi-temporal objects whose right transaction endpoint is unknown. If a bi-temporal object is later deleted at some time $t' > t$, it is physically deleted from the front R-tree and inserted as a rectangle of height *I* and width from *t* to *t'* in the *back* R-tree. The back R-tree keeps bi-temporal objects with known transaction-time interval. At any given time, all bi-temporal objects stored in the front R-tree share the property that they are alive in the transaction-time sense. The temporal information of every such object is thus represented simply by a vertical (valid-time) interval that "cuts" the transaction axis at the transaction-time when this object was inserted in the database. Insertions in the front R-tree objects are in increasing transaction time while physical deletions can happen anywhere on the transaction axis.

In Figure 3, the two R-trees methodology for bi-temporal data is divided according to whether their right transaction endpoint is known. The scenario of Figure 2 is presented here (i.e., after time $t_5$ has elapsed). The query is then translated into an interval intersection and a point enclosure problem. A simple bi-temporal query that asks for all valid time intervals which contained valid time $v_j$ at transaction time $t_i$, is answered with two searches. The back R-tree is searched for all rectangles that contain point ($t_i$, $v_j$). The front R-tree is searched for all vertical intervals that intersect a horizontal interval $H$ that starts from the beginning of transaction time and extends until point $t_i$ at height $v_j$.

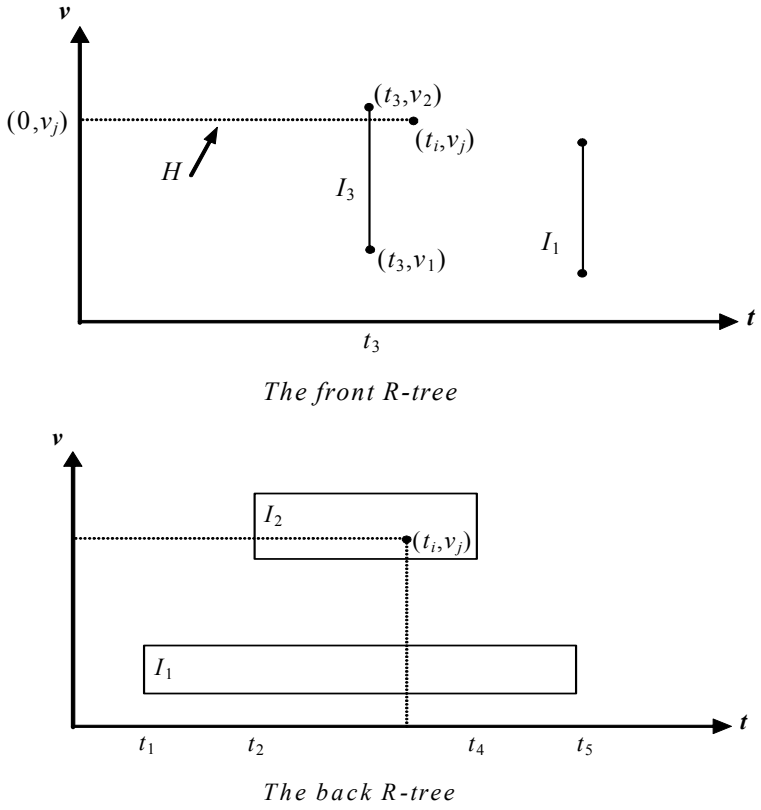


Figure 3. The two R-tree methodology for bi-temporal data.

When an R-tree is used to index bi-temporal data, overlapping may also incur if the valid-time intervals extend to the ever-increasing *now*. One approach could be to use the largest possible valid-time timestamp to represent the variable *now*. In [2] the problem of addressing both the *now* and *UC* variables is addressed by using bounding rectangles/regions that increase as the time proceeds. A variation of the R-tree, the GR-tree is presented. The index leaf nodes capture the exact geometry of the bi-temporal regions of data. Bi-temporal regions can be static or growing, rectangles or stair-shapes. Two versions of the GR-tree are explored, one using minimum bounding rectangles in non-leaf nodes, and one using minimum bounding regions in non-leaf nodes. Details appear in [2].

**Approach 3:** Another approach to address bi-temporal problems is to use the notion of partial persistence [4,1]. This solution emanates from the abstraction of a bi-temporal database as a sequence of collections $C(t)$ (in Figure 1) and has two steps. First, a good index is chosen to represent each $C(t)$. This index must support dynamic addition/deletion of (valid-time) interval-objects. Second, this index is made partially persistent. The collection of queries supported by the interval index structure implies which queries are answered by the bi-temporal structure. Using this approach, the Bi-temporal R-tree that takes an R-tree and makes it partially persistent was introduced in [5].

Similar to the transaction-time databases, one can use the “overlapping” approach [3] to create an index for bi-temporal databases. It is necessary to use an index that can handle the valid-time intervals and an overlapping approach to provide the transaction-time support. Multi-dimensional indexes can be used for supporting intervals. For example, an R-tree or a quad-tree. The Overlapping-R-tree was proposed in [6],

where an R-tree maintains the valid time intervals at each transaction time instant. As intervals are added/deleted or updated, overlapping is used to share common paths in the relevant R-trees. Likewise, [9] proposes the use of quad-trees (which can also be used for spatiotemporal queries).

There are two advantages in "viewing" a bi-temporal query as a "partial persistence" or "overlapping" problem. First, the valid-time requirements are disassociated from the transaction-time ones. More specifically, the valid time support is provided from the properties of the R-tree while the transaction time support is achieved by making this structure "partially persistent" or "overlapping". Conceptually, this methodology provides fast access to the $C(t)$ of interest on which the valid-time query is then performed. Second, changes are always applied to the most current state of the structure and last until updated (if ever) at a later transaction time, thus avoiding the explicit representation of variable *UC*. Considering the two approaches, overlapping has the advantage of simpler implementation, while the partial-persistence approach avoids the possible logarithmic space overhead.

## KEY APPLICATIONS

The importance of temporal indexing emanates from the many applications that maintain temporal data. The ever increasing nature of time imposes the need for many applications to store large amounts of temporal data. Accessing such data specialized indexing techniques is necessary. Temporal indexing has offered many such solutions that enable fast access.

## CROSS REFERENCES

Temporal Databases, Transaction-Time Indexing, Valid-Time Indexing, B+-tree, R-tree

## RECOMMENDED READING

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer (1996). An Asymptotically Optimal Multiversion B-Tree. VLDB J. 5(4): 264-275.

[2] R. Bliujute, C.S. Jensen, S. Saltenis and G. Slivinskas (1998). R-Tree Based Indexing of Now-Relative Bitemporal Data. VLDB Conference, pp: 345-356.

[3] F. W. Burton, M.M. Huntbach, and J.G. Kollias (1985). Multiple Generation Text Files Using Overlapping Tree Structures. Comput. J. 28(4): 414-416.

[4] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan (1989). Making Data Structures Persistent. J. Comput. Syst. Sci. 38(1): 86-124.

[5] A. Kumar, V.J. Tsotras, and C. Faloutsos (1998). Designing Access Methods for Bitemporal Databases. IEEE Transactions on Knowledge Data Engineering 10(1): 1-20.

[6] M.A. Nascimento and J.R.O. Silva (1998). Towards historical R-trees. Proceedings of SAC, pages 235-240

[7] B. Salzberg and V.J. Tsotras (1999). A Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, 31(2): 158-221.

[8] R.T. Snodgrass and I. Ahn (1986). Temporal Databases. IEEE Computer 19(9): 35-42.

[9] T. Tzouramanis, M. Vassilakopoulos and Y. Manolopoulos (2000). Overlapping Linear Quadtrees and Spatio-Temporal Query Processing. Comput. J. 43(4): 325-343.