TRANSACTION-TIME 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

Transaction-Time Access Methods

DEFINITION

A transaction-time index is a temporal index that enables fast access to transaction-time datasets. In a traditional database, an index is used for selection queries. When accessing transaction-time databases, selection queries also involve the transaction-time dimension. The characteristics of the transaction-time axis imply various properties that such temporal index should have to be efficient. As with traditional indices, the performance 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*. This is consistent with the serialization order of transactions in a database system. 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 (since every new change must be stamped with a new, larger, transaction time). This is useful for applications where every action must be registered and maintained unchanged after registration, as in auditing, billing, etc. Note that a transaction-time database records the history of a database activity rather than "real" world history. As such it can “rollback” to, or answer queries for, any of its previous states.

Consider, for example, a query on a temporal relation as it was at a given transaction time. There are two obvious, but inefficient approaches to support this query, namely the “copy” and “log” approaches. In the “copy” approach, the whole relation is “flushed” (copied) to disk for every transaction for which a new record is added or modified. Answering the above rollback query is simple: the system has to then query the copy that has the largest transaction-time less or equal to the requested time. Nevertheless, this approach is inefficient for its storage and update costs (the storage can easily become quadratic to the number of records in the temporal database and the update is linear, since the whole temporal relation needs to be flushed to disk for a single record update). In contrast, the “log” solution simply maintains a log of the updates to the temporal database. Clearly, this approach uses minimal space (linear to the number of updates) and minimal update cost (simply add an update record at the end of the log), but the query time is prohibitively large since the whole log may need to be traversed for reconstructing a past state of the temporal database. Various early works on transaction-time indexing behave asymptotically like the “log” or the “copy” approaches. For a worst-case comparison of these methods see [7]. Later on, two methodologies were proposed to construct more efficient transaction-time indices, namely the (i) *overlapping* [3,10] and (ii) (partially) *persistent* approaches [6,9,1]. These methodologies attempt to combine the benefits of the fast query time from the “copy” approach with the low space and update costs of the “log” approach.

SCIENTIFIC FUNDAMENTALS

The distinct properties of the transaction-time dimension and their implications to the index design are discussed through an example; this discussion has been influenced by [8]. Consider an initially empty collection of objects. This collection evolves over time as changes are applied. Time is assumed discrete and always increasing. A change is the addition or deletion of an object, or the value change of an object's attribute. A real life example would be the evolution of the employees in a company. Each employee has a surrogate (*ssn*) and a salary attribute. The changes include additions of new employees

(as they are hired or re-hired), salary changes or employee deletions (as they retire or leave the company). Each change is time-stamped with the time it occurs (if more than one change happen at a given time, all of them get the same timestamp). Note that an object attribute value change can be simply "seen" as the artificial deletion of the object followed by the simultaneous rebirth (at the same time instant) of this object having the modified attribute value. Hence, the following discussion concentrates only on object additions or deletions.

In this example, an object is called "alive" from the time that it is added in the collection until (if ever) it is deleted from it. The set *s(t)*, consisting of all alive objects at time *t*, forms the state of the evolving collection at *t*. Figure 1 illustrates a representative evolution shown as of time *t* = 53. Lines ending to ">" correspond to objects that have not yet been deleted at *t* = 53. For simplicity, at most one change per time instant is assumed. For example, at time *t* = 10 the state is *s*(10) = *{u, f, c}.* The interval created by the consecutive time instants an object is alive is the "lifetime" interval for this object. 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. In Figure 1, the lifetime interval for object *b* is [2,10). An object can have many non-overlapping lifetime intervals.

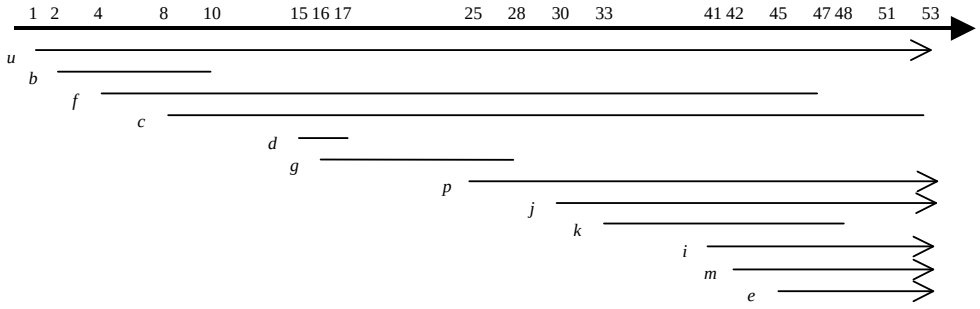

Figure 1. An example of a transaction-time evolution.

Note that in the above evolving set example, changes are always applied to the current state *s(t),* i.e., past states *cannot* be changed. That is, at time *t* = 17, the deletion of object *d* is applied to *s*(16) = *{u, f, c, d, g}* to create *s*(17) = *{u, f, c, g}.* This implies that, at time *t* = 54, no object can be retroactively added to state *s*(5), neither the interval of object *d* can be changed to become [15, 25). All such changes are not allowed as they would affect previous states and not the most current state *s*(53).

Assume that all the states *s(t)* of the above evolution need to be stored in a database. Since time is always increasing and the past is unchanged, a transaction time database can be utilized with the implicit updating assumption that when an object is added or deleted from the evolving set at time *t*, a transaction updates the database system about this change at the same time, i.e., this transaction has commit timestamp *t*. When a new object is added in the collection at time *t*, a record representing this object is stored in the database accompanied by a transaction-time interval of the form *[t, UC).* In this setting, *UC* (Until Changed) is a variable representing the fact that at the time the object is added in the collection, it is not yet known when (if ever) it will be deleted from it. If this object is later deleted at time *t'*, the transaction-time interval of the corresponding record is updated to *[t, t').* A real-world object deletion is thus represented in the database as a "logical" deletion: the record of the deleted object is still retained in the database, accompanied by an appropriate transaction-time interval. Since the past is kept, a transaction-time database conceptually stores, and can thus answer queries about, any past state *s(t)*.

Based on the above discussion, an index for a transaction-time database should have the following properties: (a) store past logical states, (b) support addition/deletion/modification changes on the objects of the current logical state, and (c) efficiently access and query any database state.

Since a fact can be entered in the database at a different time than when it happened in reality, the transaction-time interval associated with a record is actually related to the process of updating the database (the database activity) and may not accurately represent the times the corresponding object

was valid in reality. Note that a valid-time database has a different abstraction, which can be visualized as a dynamic collection of “interval-objects”. The term interval-object is used to emphasize that the object carries a valid-time interval to represent the temporal validity of some object property. Reality is more accurately represented if both time dimensions are supported. 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. The reader is referred to the Valid-time Indexing and Bi-temporal Indexing entries of this encyclopedia for further reading.

Traditional indices like the B^+-tree or the R-tree are not efficient for transaction-time databases because they do not take advantage of the special characteristics of transaction time (i.e., that transaction time is always increasing and that changes are always applied on the latest database state). There are various index proposals that are based on the (partially) *persistent* data-structure approach; examples are the Time-Split B-tree (TSB) [6], the Multiversion B-tree (MVBT) [1], the Multiversion Access Structure [11], the Snapshot Index [9], etc. It should be noted that all the above approaches facilitate "time-splits": when a page gets full, current records from this page are copied to a new page (this operation is explained in detail below). Time-splits were first introduced in the Write-Once B-tree (WOBT), a B-tree index proposed for write-once disks [5]. Later, the Time-Split B-tree used time-splits for read-write media and also introduced other splitting policies (e.g., splitting by other than the current time, key splits etc.) Both the WOBT and TSB use deletion markers when records are deleted and do not consolidate pages with few current records. The MVBT uses the time splitting approach of the WOBT, drops the deletion markers and consolidated pages with few current records. It thus achieves the best asymptotic behavior and it is discussed in detail below. Among the index solutions based on the *overlapping* data-structure approach [2], the Overlapping B-tree [10] is used as a representative and discussed further.

For the purposes of this discussion, the so-called *range-timeslice* query is considered, which provides a key range and a specific time instant selection. For example: “find all objects with keys in range [K_1, K_2] whose lifetimes contain time instant t”. This corresponds to a query “find the employees with ids in range [100, .., 500] whose entries were in the database on July 1st, 2007”. Let n be the total number of updates in a transaction-time database; note that n corresponds to the minimal information needed for storing the whole evolution. [7] presents a lower bound for answering a range-timeslice query. In particular, any method that uses linear space (i.e, $O(n/B)$ pages, where B is the number of object records that fit in a page) would need $O(\log_B n + s/B)$ I/O's to answer such a query (where an I/O transfers one page, and s corresponds to the size of the answer, i.e., the number of objects that satisfy the query).

The Multiversion B-tree (MVBT): The MVBT approach transforms a timeslice query to a partial persistence problem. In particular, a data structure is called *persistent* [4] if an update creates a new version of the data structure while the previous version is still retained and can be accessed. Otherwise, if old versions of the structure are discarded, the structure is termed *ephemeral*. *Partial* persistence implies that only the newest version of the structure can be modified to create a new version.

The key observation is that partial persistence “suits” nicely transaction-time evolution since these changes are always applied on the latest state $s(t)$ of the evolving set (Figure 1). To support key range queries on a given $s(t)$, one could use an ordinary B^+-tree to index the objects in $s(t)$ (that is, the keys of the objects in $s(t)$ appear in the data pages of the B^+-tree). As $s(t)$ evolves over time through object changes, so does its corresponding B^+-tree. Storing copies of all the states that the B^+-tree took during the evolution of $s(t)$ is clearly inefficient. Instead, one should “see” the evolution of the B^+-tree as a partial persistence problem, i.e., as a set of updates that create subsequent versions of the B^+-tree.

Conceptually, the MVBT stores all the states assumed by the B^+-tree through its transaction-time evolution. Its structure is a directed acyclic graph of pages. This graph embeds many B^+-trees and has a number of root pages. Each root is responsible for providing access to a subsequent part of the B^+-tree’s evolution. Data records in the MVBT leaf pages maintain the transaction-time evolution of the corresponding B^+-tree data records (that is, of the objects in $s(t)$). Each record is thus extended to include an interval [*insertion-time*, *deletion-time*), representing the transaction-times that the corresponding object was inserted/deleted from $s(t)$. During this interval the data-record is termed *alive*. Hence, the MVBT directly represents object deletions. Index records in the non-leaf pages of the MVBT maintain the

evolution of the corresponding index records of the B^+-tree and are also augmented with insertion-time and deletion-time fields.

Assume that each page in the MVBT has a capacity of holding B records. A page is called *alive* if it has not been *time-split* (see below). With the exception of root pages, for all transaction-times t that a page is alive, it must have at least q records that are alive at t ($q < B$). This requirement enables clustering of the alive objects at a given time in a small number of pages, which in turn will minimize the query I/O. Conceptually, a data page forms a rectangle in the time-key space; for any time in this rectangle the page should contain at least q alive records. As a result, if the search algorithm accesses this page for any time during its rectangle, it is guaranteed to find at least q alive records. That is, when a page is accessed, it contributes enough records for the query answer.

The first step of an update (insertion or deletion) at the transaction time t locates the target leaf page in a way similar to the corresponding operations in an ordinary B^+-tree. Note that, only the latest state of the B^+-tree is traversed in this step. An update leads to a *structural change* if at least one new page is created. *Non-structural* are those updates which are handled within an existing page.

After locating the target leaf page, an insert operation at the current transaction time t adds a data record with a transaction interval of [t, *UC*) to the target leaf page. This may trigger a structural change in the MVBT, if the target leaf page already has B records. Similarly, a delete operation at transaction time t finds the target data record and changes the record's interval to [insertion-time, t). This may trigger a structural change if the resulting page ends up having less than q alive records at the current transaction time. The former structural change is called a *page overflow*, and the latter is a *weak version underflow* [1]. Page overflow and weak version underflow need special handling: a *time-split* is performed on the target leaf-page. The time-split on a page x at time t is performed by copying to a new page y the records alive in page x at t. Page x is considered *dead* after time t. Then the resulting new page has to be incorporated in the structure [1].

Since updates can propagate to ancestors, a root page may become full and time-split. This creates a new root page, which in turn may be split at a later transaction time to create another root and so on. By construction, each root of the MVBT is alive for a subsequent, non-intersecting transaction-time interval. Efficient access to the root that was alive at time t is possible by keeping an index on the roots, indexed by their time-split times. Since time-split times are in order, this root index is easily kept (this index is called the root* in [1]). In general, not many splits propagate to the top, so the number of root splits is small and the root* structure can be kept in main memory. If this is not the case, a small index can be created on top of the root* structure.

Answering a range-timeslice query on transaction time t has two parts. First, using the root index, the root alive at t is found. This part is conceptually equivalent to accessing $s(t)$ or, more explicitly, accessing the B^+-tree indexing the objects of $s(t)$. Second, the answer is found by searching this tree in a top-down fashion as in a B^+-tree. This search considers the record transaction interval. The transaction interval of every record returned or traversed should include the transaction time t, while its key attribute should satisfy the key query predicate. A range-timeslice query takes $O(\log_B n + s/B)$ I/O's while the space is linear to n (i.e., $O(n/B)$). Hence, the MVBT optimally solves the range-timeslice query. The update processing is $O(\log_B m)$ per change, where m is the size of $s(t)$ when the change took place. This is because a change that occurred at time t traverses what is logically a B^+-tree on the m elements of $s(t)$.

The Overlapping B-tree: Similar to the MVBT approach, the evolving set $s(t)$ is indexed by a B^+-tree. The intuition behind the Overlapping B-tree [10, 2] is that the B^+-trees of subsequent versions (states) of $s(t)$ will not differ much. A similar approach was taken in the EXODUS DBMS [3]. The Overlapping B-tree is thus a graph structure that superimposes many ordinary B^+-trees (Figure 2). An update at some time t creates a new version of $s(t)$ and a new root in the structure. If a subtree does not change between subsequent versions, it will be shared by the new root. Sharing common subtrees among subsequent B^+-trees is done through index nodes of the new B^+-tree that point to nodes of previous B^+-tree(s). An update will create a new copy of the data page it refers to. This implies that a new path is also created leading to the new page; this path is indexed under the new root.

To address a range-timeslice query for a given transaction time *t*, the latest root that was created before or at *t* must be found. Hence, roots are time-stamped with the transaction time of their creation. These timestamps can be easily indexed on a separate B^+-tree. This is similar to the MVBT root* structure; however, the Overlapping B-tree creates a new root per version. After the appropriate root is found, the search continues traversing the tree under this root, as if an ordinary B^+-tree was present for state $s(t)$.

Updating the Overlapping B-tree involves traversing the structure from the current root version and locating the data page that needs to be updated. Then a copy of the page is created as well as a new path to this page. This implies $O(\log_B m)$ I/O's per update, where *m* is the current size of the evolving set. An advantage of the Overlapping structure is in the simplicity of its implementation. Note that except the roots, the other nodes do not involve any time-stamping. Such time-stamping is not needed because pages do not have to share records from various versions. Even if a single record changes in a page, the page cannot be shared by different versions; rather a new copy of the page is created. This, however, comes at the expense of the space performance. The Overlapping B-tree occupies $O(n \log_B n)$ pages since, in the worst case, every version creates an extra tree path. Further performance results on this access method can be found in [10].

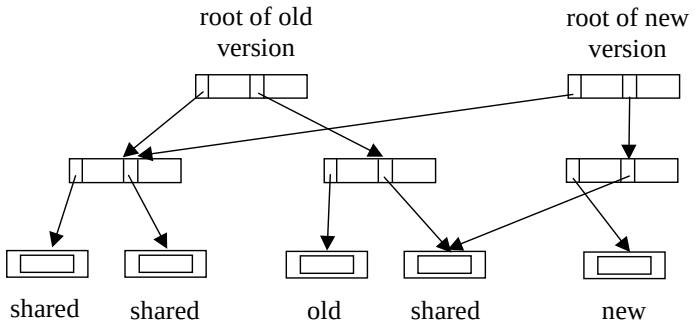

Figure 2. The Overlapping B-tree.

KEY APPLICATIONS

The characteristics of transaction-time make such databases ideal for applications that need to maintain their past; examples are: billing, accounting, tax-related etc.

CROSS REFERENCES

Temporal Databases, Transaction-time, Valid-time, Valid-Time Indexing, Bi-Temporal Indexing, B+-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] 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.

[3] M.J. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita: Object and File Management in the EXODUS Extensible Database System. VLDB 1986: 91-100.

[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] M.C. Easton: Key-Sequence Data Sets on Inedible Storage. IBM Journal of Research and Development 30(3): 230-241 (1986)

[6] D. Lomet and B. Salzberg (1989). Access Methods for Multiversion Data. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 315-324.

[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] V.J. Tsotras and N. Kangelaris (1995). The Snapshot Index: An I/O-optimal access method for timeslice queries. Inf. Syst. 20(3): 237-260.

[10] T. Tzouramanis, Y. Manolopoulos, and N.A. Lorentzos (1999). Overlapping B+-Trees: An Implementation of a Transaction Time Access Method. Data Knowledge Engineering. 29(3): 381-404.

[11] P.J. Varman and R.M. Verma (1997). An Efficient Multiversion Access Structure. IEEE Transactions on Knowledge Data Engineering. 9(3): 391-409.