

Chapter 5

Temporal Access Methods

Traditional databases capture a single (usually the most recent) state of the modeled reality. This implies that only queries about this captured state can be answered. There are however many time-varying applications that deal with historical (past) data, as well as current, or even data about the future. To address these applications *temporal* databases have been proposed. Such databases model reality through the use of two orthogonal time dimensions: the *valid* and *transaction* times. Depending on which time dimension(s) is supported, a temporal database is characterized as *transaction-time*, *valid-time* or *bitemporal*. Supporting temporal data adds extra database functionality since it allows queries about the modeled reality's past, current or future behavior. However, it also creates the novel problem of efficiently managing temporal data. In this chapter we first briefly present the basic characteristics of the two time dimensions and what it means to design indices that support them. While the problem of indexing valid-time databases can be reduced to indexing dynamic collections of interval data (and thus one could use the methods from Chapter 4), indexing transaction-time and bitemporal databases requires new approaches. In particular we present four transaction-time methods (the Snapshot Index, the Time-Split B-tree, the Multiversion B-tree and the Overlapping B-tree) and a bitemporal method (the Bitemporal R-tree). We also discuss temporal hashing, the temporal analogue of external hashing in a transaction-time database.

1. INTRODUCTION

The term “temporal database” refers in general to a database that supports some time domain and is thus able to manage time varying data [Ozsoyoglou and Snodgrass, 1995]. There have been two orthogonal time dimensions proposed in literature, namely, *transaction time* and *valid time* [Jensen et al., 1994]. Transaction time is the time when a fact is stored in the database. Valid time is the time when a fact becomes effective (valid) in reality. In addition, the term *user-defined time* has been introduced to represent an uninterpreted time domain managed by the user; we will

not discuss user-defined time any further. The name “transaction-time” was coined since DBMSs are transactional in nature, i.e., data is entered by the means of transactions.

It is important to realize the orthogonality of the valid and transaction time dimensions. While in various applications we typically assume 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), there are many other applications where this assumption does not hold. Consider for example a sales database, where data about sales at a given day are recorded at the end of the day (when batch processing of all data collected during the day is performed). Each data record represents a sale and has a different valid time (the actual time of the sale) than transaction time (the time this record was entered in the database). Interestingly, note that recorded valid times may represent later times than the transaction which recorded them. For example, a contract may be valid for a period of time that is later than the (transaction) time when this information was entered in the database.

A basic property of transaction time is that it always increases. This is consistent with the serialization order of transactions in a database system. A newly submitted transaction gets a unique id by reading an ever-increasing counter. In a transaction-time database each newly recorded piece of data is timestamped with a new, larger, transaction time. This transaction time can simply be the transaction-id of the transaction that entered this data into the database. 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, like 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.

A valid-time database on the other hand, maintains the entire temporal behavior of an enterprise as best known now. It stores our 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.

Note that updates in a valid-time database can apply to any point in the valid time domain. In contrast, in a transaction-time database, updates happen in increasing transaction time order. Hence valid-time databases are usually termed as supporting “changes anywhere (in the valid time domain)”

while transaction-time databases support “changes in increasing order (of transaction time)”. Since no information about the past is changed, a transaction-time database “appends-only” information.

By supporting both valid and transaction time, a bitemporal 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 it more accurately represents reality.

Critical in the design of any access method is the accurate specification of the problem that needs to be solved. This is particularly important in temporal databases since the problem specification depends dramatically on the time dimension(s) supported. Whether valid and/or transaction time are supported, affects directly the way records are created or updated. To exemplify the distinct characteristics of the transaction and valid time dimensions, we present separate *abstract models* describing the central underlying problem for each kind of temporal database. This discussion has been influenced by [Snodgrass and Ahn, 1986] where the differences between valid and transaction time were introduced and illustrated through various examples. Here we attempt to identify the implications to the index design from the support of each time dimension.

1.1 Transaction-time Index Characteristics

To apprehend the abstract model for a transaction-time database, consider an initially empty collection of objects S . 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 timestamped with the time it occurs (if more than one changes 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 in the rest we concentrate only on object additions or deletions.

An object is termed “*alive*” from the time that it is added in the collection and until (if ever) it is deleted from it. The set $S(t)$ consisting of all alive objects at time t , is termed the *state* of the evolving collection at t . A representative evolution shown as of time $t = 53$ appears in Figure 5.1. 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 during which an object is alive, corresponds to a *lifetime* interval for this object. In Figure 5.1, the lifetime interval for object b is $[2,10)$. An object can have many but non-overlapping such lifetime intervals.

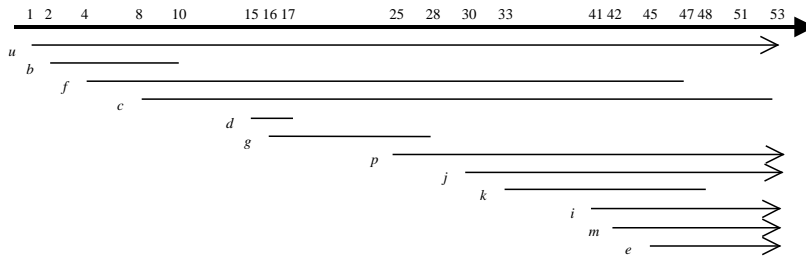


Figure 5.1. An example of a transaction-time evolution.

Note that changes are always applied to the most current state $S(t)$, i.e., past states cannot be changed. That is, at time $t = 17$ the deletion of object d is applied on $S(16) = \{u, f, c, d, g\}$ to create $S(17) = \{u, f, c, g\}$. This implies that at time $t = 54$ we cannot retroactively add an object to state $S(5)$ nor we can change the interval of object d 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 the history of the above evolution is to be stored in a database. Time is always increasing and the past is unchanged. For this reason, 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 $[t, now)$. In this setting, *now* is a variable representing the current transaction time. It is used because 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 but 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, if we design an index for a transaction-time database, this index needs to: (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.

1.2 Valid-time Index Characteristics

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 period the corresponding object was alive in reality. Thus a valid-time

database has a different abstraction. To visualize it, consider a dynamic collection of interval-objects. We use the term *interval-object* to emphasize that the object carries a valid-time interval to represent the validity period of some object property. (In contrast, and to emphasize that transaction-time represents the database activity rather than reality, we term the objects in the transaction-time abstraction as *plain-objects*.)

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, that is, an update is physical (rather than logical). An example of a dynamic collection of object-intervals appears at Figure 5.2. Assume the collection (a) has

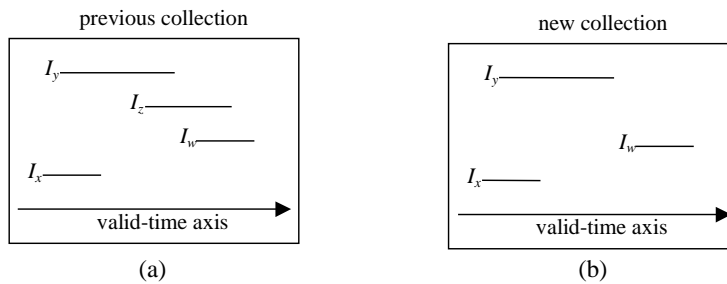


Figure 5.2. Two valid-time databases.

been recorded in some erasable medium and a change happens, namely object I_z is deleted. This change is applied on the recorded data and physically deletes object I_z . The medium now stores collection (b), i.e., collection (a) is not retained. It is interesting to see that if we consider the valid time dimension, changes do not come in increasing time order any-more; rather they can affect any interval in the collection. This implies that we can even correct errors in previously recorded data. However, only a single data state is kept, the one resulting after the correction is applied.

As a real-life example consider the collection of contracts in a company. Each contract has an identity (*contract_no*) and an interval representing the contract's duration or validity. In collection (a) we knew about four contracts in the company. But assume an error was discovered: contract I_z was never a company contract (maybe it was mistakenly entered). Then, this information is permanently deleted from the collection, which now looks as in collection (b).

A valid-time database is suitable for this environment. When an object is added to the collection, it is stored in the database as a record that contains the object's attributes (including its valid-time interval). The time of the record's insertion in the database is *not* recorded. When an object deletion occurs, the corresponding record is physically deleted from the database. If an object attribute is modified, its corresponding record attribute is updated but the previous attribute value is not retained. Hence, a valid-time database keeps only the latest "snapshot" of the data. Querying a valid-time database cannot give any information on the past states of the

database or how the collection evolved; it gives us only the view of reality as “best known now”! For example, by looking at Figure 5.2b (which is the only data available) there is no way to say if we ever had any contract I_z .

The notion of time is now related to the valid-time axis. Given a valid-time point, interval-objects can be classified as past, future or current as related to this point, if their valid-time interval is before, after or contains the given point. Valid-time databases are said to correct errors anywhere in the valid-time domain (past, current or future) because the record of any interval-object in the collection can be changed, independently of its position on the valid-time axis. Note that a valid-time database may store records with the same surrogate but with non-intersecting valid-time intervals. For example, we could add another I_x interval in the collection Figure 5.2b as long as it does not intersect with the existing I_x interval; the new collection will contain both I_x intervals. This is fine since each interval represents object I_x at different times in the valid-time dimension.

From the above discussion, an index used for a valid-time database should: (a) store the latest collection of interval-objects, (b) support addition/deletion/modification changes to this collection, and, (c) efficiently query the interval-objects contained in the collection when the query is asked. Consequently, a valid-time index manages a dynamic collection of intervals and therefore methods from Chapter 4 are applicable.

1.3 Bitemporal Index Characteristics

Reality is more accurately represented if both time dimensions are supported. A bitemporal 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 5.3 offers a conceptual view of a bitemporal database. Instead of a single collection of interval-objects there is a sequence of collections indexed by transaction time. Instead of maintaining a single collection of interval-objects (as a valid-time database) a bitemporal database maintains a sequence of such collections $C(t_i)$ indexed by transaction time. In this environment we can represent how our knowledge about company contracts evolved. In Figure 5.3, 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.

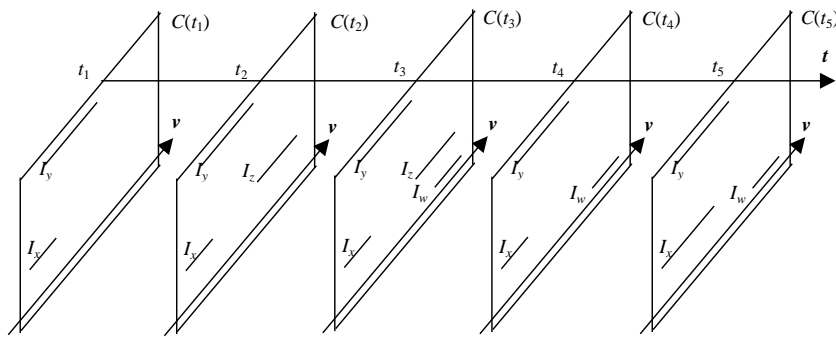


Figure 5.3. A bitemporal database.

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) and valid-time interval (contract duration) and an initial transaction-time interval $[t, now)$. The transaction-time interval's right endpoint 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 $[t_2, t_4)$ since 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 (a) and (b) of Figure 5.2, assuming that it was at time t_4 when the erroneous contract I_z was deleted from the database.

Based on the above discussion, an index for a bitemporal 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 5.3 summarizes the differences among the underlying problems of the various database types. Each collection $C(t_i)$ can be thought on its own as a separate valid-time database. A valid-time database differs from a bitemporal since it keeps *only one* collection of interval-objects (the latest). A transaction-time database differs from a bitemporal 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).

Since both bitemporal and transaction-time databases maintain their past database states they must deal with the management of an ever increasing amount of historical data. Managing such data implies the ability to *efficiently* address queries about it. Note that sequential scans through the stored data can answer any query but this is not efficient, simply because new data is appended. A common practice to organize large amounts of data is the use of indices (or access methods) that will efficiently address some of the most common queries to the database. Traditional indices like the B⁺-tree or the R-tree are not efficient for bitemporal and 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). In the rest of this chapter we concentrate on efficient indices for such databases.

2. TRANSACTION-TIME INDEXING

2.1 Basic Queries and Straightforward Approaches

Common transaction-time queries contain a range predicate on the object surrogates (keys) and an interval predicate on the object transaction time lifetimes. For example, “find all objects with keys in range $[K_1, K_2]$ whose lifetimes intersect interval T ”. We will concentrate on *timeslice* queries, that is, queries where the interval T is reduced to a single transaction time instant t . In particular we will discuss three timeslice queries of interest:

- the *pure-timeslice* query where no key range predicate is defined (“find *all* employees that were working in the company on January 1st, 1999”),
- the *range-timeslice* query where a key range is defined (“find the employees with ids in range $[100, \dots, 900]$ that were working in the company on January 1st, 1999”), and
- the temporal membership query where the key range is reduced to a single key (“find if employee with id 501 was in the company on January 1st, 1999”).

As with any index, the performance of a transaction-time index is characterized by the space used by the index, the update processing (time to update an index about a change) and the query time. In a transaction time environment, the space is a function of n , the total number of changes in the evolution. That is, n is the summation of insertions, deletions and modification updates. For example, if there are 1,000 updates to a database with only one record, n is 1,000. If there are 1,000 insertions to an empty database and no deletions or attribute value modifications, n is also 1,000. Similarly, for 1,000 insertions followed by 1,000 deletions, n is 2,000. Note that n corresponds to the minimal information needed for storing the whole evolution.

[Salzberg and Tsotras, 1999] show a lower bound for answering pure- and range-timeslice queries. 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 a timeslice 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). There is a clear intuition behind this lower bound. Any method definitely needs $O(s/B)$ page I/O's simply to output the answer to the query. Moreover, note that the answer to a query about some time t is affected by changes that happened up to time t ; later changes do not affect state $S(t)$. The $O(\log_B n)$ part is needed to locate the position of time t among the n changes in the evolution. Since these changes are ordered by the time they occurred a logarithmic search suffices. The above bound assumes that a query is answered with the same efficiency independently from which time t it asks for. This is different from cases where the query efficiency

depends on t (for example if queries on more recent times should have faster query response than queries on earlier times).

There are two obvious but clearly inefficient ways to solve timeslice queries, the *copy* and the *log* approaches examined below. For simplicity consider a pure-timeslice query. The copy approach stores the whole current state $S(t)$ for every change instant (i.e., a time instant when at least one change occurred). Then the I/O for answering a pure-timeslice query about time t is minimal: $O(\log_{Bn} + s/B)$. This assumes that the change instants are indexed by a multi-level index (like a B^+ -tree). This index will provide fast access ($O(\log_{Bn})$) to the stored state $S(t')$ where t' is the largest changed instant less or equal to t . Reading the answer takes another $O(s/B)$ I/O's. However, the update time (time to store every state) and the space can be problematic, i.e. $O(n/B)$ per change and $O(n^2/B)$ pages respectively, since in the worst case the state could always be increasing by insertions of new objects.

The log approach keeps all the changes timestamped with the time they occurred in a log. The space requirement is now minimal, just $O(n/B)$ pages. Similarly, the update cost is minimal ($O(1)$), as only the last page of the log is accessed to insert the new change. But the query time of this method suffers. In the worst case one has to search back the whole log, resulting in $O(n/B)$ I/O's query time.

The copy and log solutions provide the two extreme approaches regarding query time and space/update performance. One could create variations of the two by storing copies of some (not all) of the states $S(t)$ and the changes in between the copies. This mixture is bound to behave asymptotically as the copy or log approaches depending of how often copies of $S(t)$ are kept. If infrequent copies are stored then the number of the changes between these copies is large and the method works asymptotically as the log approach; if the number of changes between the copies is kept constant then the method will asymptotically behave like the copy approach.

Fortunately, for the timeslice queries there are indices that achieve the lower query bound using linear space. We call such indices I/O-optimal. In particular, the Snapshot Index [Tsotras and Kangelaris, 1995] uses $O(n/B)$ space, $O(1)$ update per change (in the expected amortized sense due to the use of a hashing scheme) and solves a pure-timeslice query about time t in $O(\log_{Bn} + s/B)$ I/O's; here s is the cardinality of state $S(t)$. The Multiversion B-tree (MVBT) [Becker et al., 1996], and its refinement the Multiversion Access Structure (MVAS) [Varman and Verma, 1997], use $O(n/B)$ space, $O(\log_{Bm})$ update time per change and solve a range-timeslice query about time t in $O(\log_{Bn} + s/B)$ I/O's; here m corresponds to the size of the current state of the evolving set when an update is made on it, while s is the number of objects from $S(t)$ that satisfy the key range query predicate. While both the MVBT and MVAS structures use linear space, the MVAS uses improved merge/split policies, which result to a smaller constant in the space bound.

Obviously, an index that can efficiently address a range-timeslice query can also address a pure-timeslice query with the same efficiency, but not vice-versa. However, such index requires logarithmic update time. This is because updates come in transaction-time order but not necessarily in object key order. To answer range-timeslice queries efficiently, new updates have to be ordered in the key space. In contrast a method that addresses pure-

timeslice queries has to simply provide all objects as of the time of interest with no particular key order resulting in faster updates. Such performance may be needed for applications where the observed set evolves rather rapidly and the database system needs to follow the evolution quickly.

Similarly, an index that can answer range-timeslice queries can also answer temporal membership queries (simply reduce the range to a single key). This will again introduce a logarithmic overhead. [Kollios and Tsotras, 1998] present a better solution for temporal membership queries based on temporal hashing.

2.2 The Snapshot Index

The Snapshot Index [Tsotras and Kangelaris, 1995] concentrates on the pure-timeslice query, i.e., given t find $S(t)$. Conceptually it consists of three data structures:

1. a multilevel index that essentially provides access to the latest change that occurred before or at time t ,
2. a multi-linked structure among the leaf pages of the multilevel index (this structure has the form of a forest of trees where tree nodes correspond to data pages; it is called the *access forest* and helps finding the query answer for any given t), and,
3. a hashing scheme that provides access to object records by key; this scheme is used for update purposes.

A real-world object is represented by a record with a time-invariant surrogate (object id), a time-variant attribute and a semi-closed transaction-time interval of the form: $[start_time, end_time)$. When a new object is added at time t , a new record is created with interval $[t, now]$ and is stored sequentially in a data page. At any given instant there is only one data page that stores (accepts) records and it is called the *acceptor* page. When an acceptor page is created it is also put at the end of a doubly linked list L ; this list enables the creation of the access forest. When the acceptor page becomes full of object records, a new acceptor page is created (and is also added at the end of list L). Up to now, the Snapshot Index resembles a linked “log” of pages that stores object records.

There are four main differences from a regular log: the use of the hashing scheme, the in-place deletion updates, the notion of *page usefulness* and the access forest. The hashing function is used for updating object records when their corresponding object is deleted. At each time, the hashing scheme provides access to the records of all alive objects at that time. Thus, for each new object record created, the hashing function will store the object id of this record together with the address of the acceptor page that stores it. Object deletions are not added at the end of the log. Rather an object deletion results to changing the *end_time* in the interval of the deleted object's record (in-place update). The record of the deleted object is found by the accessing the hashing scheme. After the record of a deleted object is updated, its object id is removed from the hashing scheme. As with alive objects, a record with *end_time* equal to *now* is termed “*alive*” else it is called “*deleted*”.

For I/O optimal performance, the answer to any given query must be clustered to as few pages as possible. This means that each data page accessed for answering a

query about some time t should provide enough part of the answer $S(t)$, i.e., enough records that were alive at t . Without efficient clustering the query time may need to access $O(s)$ data pages (if each accessed data page results in a single alive record from the answer $S(t)$) which is clearly not optimal.

To achieve this objective, the notion of *usefulness* is introduced. In particular, a data page is defined to be *useful* for:

- all time instants, for which it was the acceptor page, or,
- after it ceased being the acceptor page, for all time instants, for which the page contains at least uB alive records.

For all other instants, the page is called *non-useful*. The useful period [$u.start_time$, $u.end_time$) of a page forms a “usefulness” interval for this page. The $u.start_time$ is the time instant the page became an acceptor page. The $u.end_time$ is initially unknown (set to *now*, the variable representing the ever increasing transaction time). If the page becomes non-useful at time t its $u.end_time$ is updated from *now* to t . The *usefulness* parameter u (where $0 < u \leq 1$) is a constant that tunes the behavior of the Snapshot Index.

To answer a pure-timeslice about time t the Snapshot Index needs only access the pages useful at t (or equivalently, those pages that have at least uB records alive at t) plus at most one additional page, the page that was the acceptor at t . This page may contain less than uB records from the answer but the query performance is not affected since there is only one such page per query (at each time there is a single acceptor page).

Finding the useful pages at time t is facilitated by searching through the access forest. We proceed with a description of how the access forest is created. At each time t , list L contains the useful pages at that time. When a new acceptor page is created, it is by definition a useful page thus it is entered at the end of list L . Assume that at some t a data page becomes non-useful. Then its alive records are copied to the current acceptor page. This copying is performed for the following reason: since the Snapshot Index will access only the useful pages for a given time, a query about t will not access this page; however, the records of this page that were alive at t will be found at the page where they were copied (that page was an acceptor at time t so it will be accessed).

In addition, the non-useful data page is removed from list L and is logically placed as a child under the previous data page in the list. Thus while a page remains useful (and is thus in list L) it may acquire a subtree of non-useful pages (thus the name *access forest*). Figure 5.4 illustrates the access forest for a given collection of usefulness intervals. In Figure 5.4a the usefulness interval of each data page as of time 80 is depicted. An open interval at time $t = 80$ represents a data page being useful at that time. Figure 5.4b illustrates the access forest as it is at time $t = 79$ (*now* in this figure corresponds to time 79). A record $\langle \text{page-id, page-usefulness-period} \rangle$ represents each page. Page SP denotes the artificial page on the top of list L . Figure 5.4c illustrates the access forest at time $t = 80$. At that time page C became non-useful (because some deletion reduced the number of alive records in C below the uB threshold). As a result it is removed from L and placed (together with its subtree) under the previous page in the list, page D . The multilevel index and the hashing scheme are not shown.

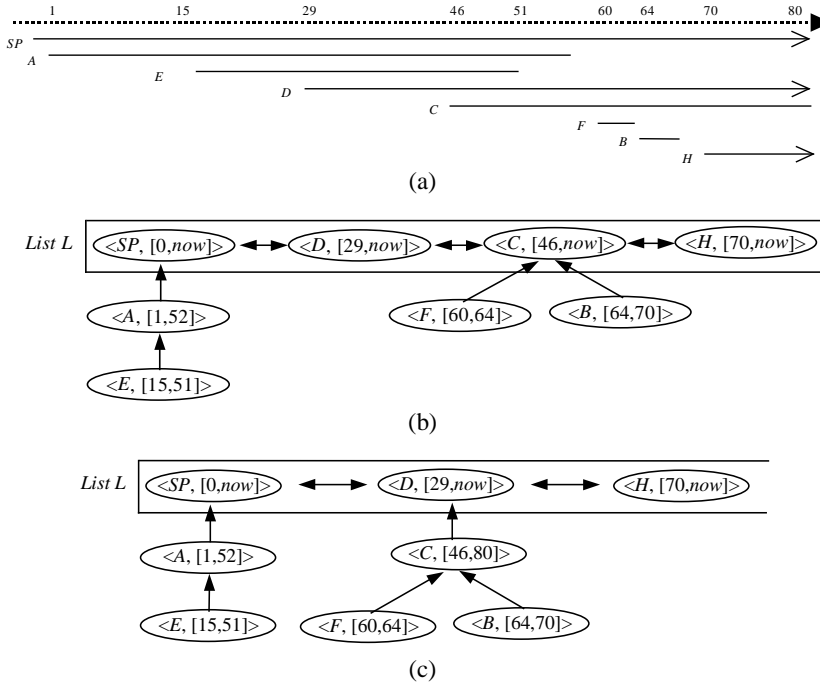


Figure 5.4. The access forest for a given collection of usefulness intervals.

When a page moves under another page, it carries with it its whole subtree (if any). To avoid cases where there is no previous data page in list L , we assume that the first page entered in list L is an “artificial” page that remains useful for ever (page SP in Figure 5.4). This artificial page does not store any data records.

The access forest has the following properties: (a) At each time instant, the root of each tree in the access forest lies in list L . (b) The $u.start_time$ fields of the data pages in a tree are organized in a *preorder* fashion. (c) The usefulness interval of a page, includes all the corresponding intervals of the pages in its subtree. (d) The usefulness intervals $[d_i, e_i]$ and $[d_{i+1}, e_{i+1}]$ of two consecutive children under the same parent page may have one of two orderings: $d_i < e_i < d_{i+1} < e_{i+1}$ or $d_i < d_{i+1} < e_i < e_{i+1}$. A timeslice query for time t is answered in two steps. First, the page that was acceptor at time t is found through the multilevel index. This structure indexes the $u.start_time$ of all the data pages. When a new acceptor page is created, a $\langle u.start_time, page_id \rangle$ record is added to the index; $page_id$ is the unique identifier of this page. Since acceptor pages are created at increasing time instants, the index pages are easily “packed” with records (the index increases only through its right path). Let Y be the acceptor page at time t . Finding Y through the multilevel index takes $O(\log_B n)$ I/O’s. The remaining useful data pages at t are found by traversing the access forest. This traversing is done very efficiently using the access forest properties [Tsotras et al., 1995].

Consider the path p of the access forest that starts at page Y and goes from child to parent until the root of the tree that contains Y is reached. This root may be Y itself if Y is still “useful” when the query is asked. Or, it may be another data page that is still in list L (or it may be artificial page SP). Path p divides the access forest pages in three logical sets: set-1 that contains the pages that are on path p , set-2 with the pages that are on the left of the path, and, set-3 with the pages that are under the path and on its right. From the access forest property (b), set-3 contains pages created after t that clearly should not be checked for usefulness as they became useful after t . All the pages in set-1 (starting with Y itself) were in useful mode at time t ; this is due to property (c).

To complete the search for useful pages, our method must also find which of the pages from set-2 were useful at time t . The search starts from page Y . Then the left sibling of Y (if any) is checked to see if it is useful at t . After this, for any page encountered in useful mode the search continues

- to its subtree starting from its rightmost child, and,
- to its left sibling.

This is repeated at each level. For the pages that are in path p the subtree search starts from the page that is in the path p . When a page is encountered that was not useful at time t the method does not check its left sibling or its subtree as no useful pages can be found towards these directions. The search stops when no new useful page is found. Therefore, to find all the useful pages our method checks at most twice as many pages in the access forest. As an example consider finding the useful pages at time $t = 60$ from Figure 5.4. The correct answer is: F, C, D while the algorithm checks pages: F, C, D, SP and A . For time $t = 50$, the correct answer is: C, D, A, E , and the algorithm checks exactly these pages as well as the artificial page SP .

As a result, the Snapshot Index solves the pure-timeslice query using $O(\log_B n + s/B)$ I/O’s for query time. It can be easily seen that the extra space used by the page copying remains linear to the number of changes [Tsotras and Kangelaris, 1995] thus the space remains $O(n/B)$. The update processing is $O(1)$ update per change (in the expected amortized sense, assuming the use of a dynamic hashing function [Driscoll et al., 1989]). The index performance can be fine-tuned by parameter u . Larger u means faster query time in the expense of more copies. Since more space is available, the answer would be contained in a smaller number of useful pages.

To answer range-timeslice queries the Snapshot Index must first compute the whole timeslice. This is in general the trade-off for the fast update processing provided.

2.3 The Time-Split B-tree

To answer a transaction range-timeslice query efficiently, it is best to cluster data by both transaction time and key within pages. Then “logically” related data for a range-timeslice query are co-located, minimizing the number of accessed pages. Methods in this category are based on some form of a balanced tree whose leaf pages dynamically correspond to regions of the two dimensional transaction time-key space. An example of a page containing a time-key range is shown in Figure

5.5. Each object update (creation, deletion or attribute value change) creates a new *version* for the object. Here, at transaction time instant 5, a new version of the record with key *c* is created. At time 6, a record with key *g* is inserted. At time 7, a new version of the record with key *b* is created. At time 8, both *c* and *f* have new versions and

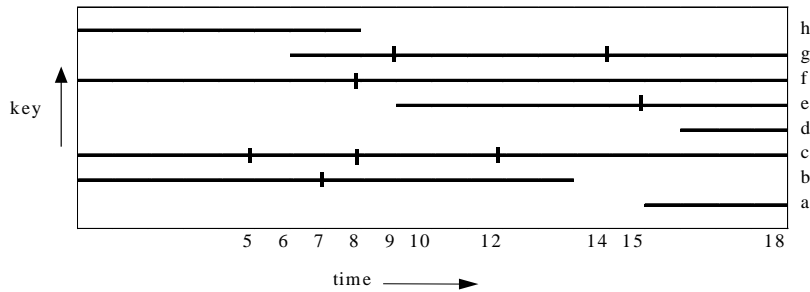


Figure 5.5. Each page is storing data from a time-key range.

record *h* is deleted. Each line segment, whose start and end time are represented by ticks, represents one object version. Each such version corresponds to a record that occupies space in the disk page.

While changes still occur in increasing time order, the corresponding keys on which the changes are applied are not in order. Thus there is a logarithmic update processing per change so that data is placed according to key values in the above time-key space.

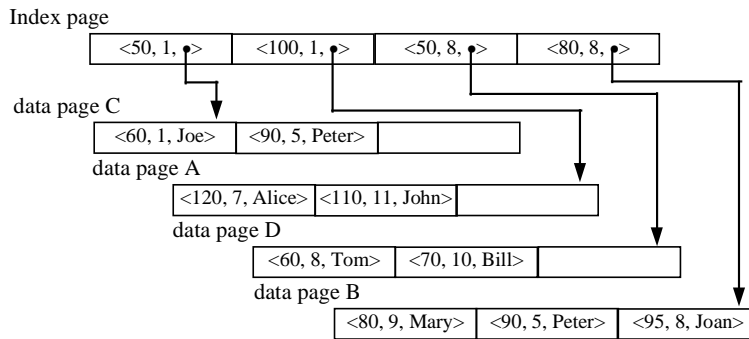
Each node in the TSB-tree describes a rectangle in time-key space. The TSB-tree data nodes contain records of the form $\langle key, time, attr \rangle$. *Key* is the object surrogate, *attr* is the value of a time dependent attribute and *time* corresponds to the transaction time this record was created. It should be noted that the TSB-tree was designed to index temporal applications where the majority of changes are new object additions and object modifications while object deletions are rare. This explains why a single time attribute (instead of a lifetime interval) is used per record. Each object update creates a new record (version) which remains valid until a new version is created. Deletions are still supported: a new record is created at the time of the deletion, carrying a special attribute value that denotes that the corresponding object was actually deleted at that time.

The TSB-tree index nodes contain records that are triplets of the form: $\langle key, time, pointer \rangle$. *Time* and *key* respectively denote the low time value and the low key value for the rectangular region of time-key space accessible by the lower-level node pointed by the pointer. The range of time values of an index term is its time span and the range of key values it key span. An example of a TSB-tree appears in Figure 5.6.

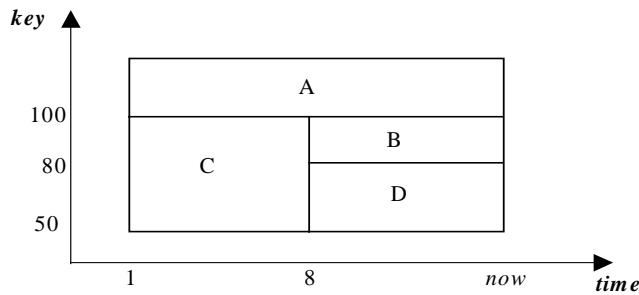
Consider searching for the record of key *k* valid at time *t*. The search begins at the root of the tree. At each index node, we ignore all triplets with times later than the search time. Among the rest of the triplets in a node we find the triplet with the

largest key that is smaller or equal to the search key k . If the node contains many such triplets of key k we find the most recent version (among the non-ignored versions). The pointer in this triplet is followed and the search is repeated until a leaf is reached.

At the leaf (data page) we look for the record with key k and the largest time that is less or equal to t . For example, to find the record with key 60 that is valid at time 7, we ignore all the entries in the index page with $time > 7$. Among the remaining we find the largest key less or equal to 60 with the largest time. This is triplet $\langle 50, 1, C \rangle$. In that data page, the record $\langle 60, 1, \text{Joe} \rangle$ satisfies the query. The TSB-tree is searched similarly for a range-timeslice query with key range $[k_1, k_2]$. Now at each visited node the whole query range is taken into account (i.e., we look for the triplet with the smallest key that is larger than or equal to k_1).



(a) the TSB-tree



(b) the time-key space it indexes

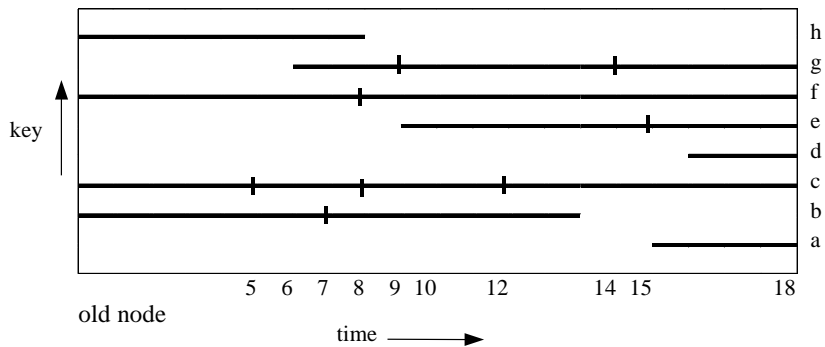
Figure 5.6. An example of the TSB-tree.

To achieve data clustering by time and key, the TSB-tree follows various splitting policies. In particular, when a data page becomes full, if the number of distinct keys in this page is below some threshold (say $2/3$ of the total number of records in the page) then a *time-split* is performed. A time split copies the entries that are valid at the time of the split to a new page (similar to the copying procedure in the Snapshot

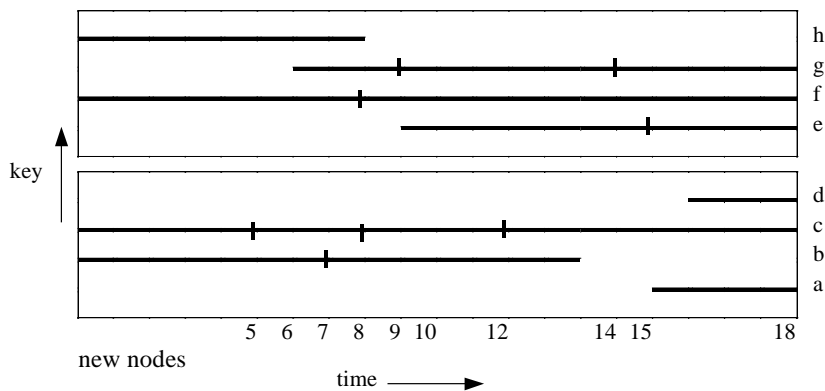
Index). The existence of low number of distinct keys implies that there have been many versions of these keys. However, the new page will carry only the latest versions of each key. The TSB-tree allows to split using other times than the current time, too. Time splits create extra record copies but they allow clustering together records that are valid at the same time. Without this redundancy, long-lived records can only be stored in one place. No matter what strategy is chosen to store such a long-lived record without redundancy, some timeslice queries will be inefficient.

If the number of distinct keys is larger than the threshold then a plain key split is performed, i.e., the records in the page are divided based on their key attribute (and not the time). The existence of a large number of distinct keys implies that the data page had mainly insertions of new records (with very few updates); hence a time split would not help as it would copy most of the records to a new page.

There is however a problem with pure key splits. The decision on the key splits is made based on the alive keys at the time the key split is made. For example in Figure 5.7, the key split is taken at time $t = 18$, when there are six keys alive, thus they are separated three per new page. However, this key range division does not guarantee that the two pages will have enough alive keys for all previous times; at time $t = 15$ the bottom page has only one key alive.



(a) the old TSB node just before the key split



(b) the new nodes created in the TSB tree by the key-split

Figure 5.7. An example of a plain key-split.

Suppose you have a database where most of the changes are insertions of records with a new key. As time goes by, in the TSB-tree, only key splits are made. After a while, queries as of a past time will become inefficient. Every timeslice query will have to visit every node of the TSB-tree since they are all current nodes. Queries as of a recent time, will be efficient since every node will have many alive records. But queries as of the distant past will be inefficient since many of the current nodes will not contain records, which were alive at that distant past time.

In addition the TSB-tree merely posts deletion markers and does not merge nodes which become sparse of alive records. If no merging of nodes is done, and there are many record deletions, a node may contain few current records. This could make current search slower than it should be.

Thus the worst case search time for the TSB-tree can be $O(n/B)$ for a transaction (pure or range) timeslice. Pages may be accessed which have no answers to the query. The MVBT structure presented below solves these problems by:

- always performing a time split before a key split and
- consolidating (a special form of merging) nodes that become sparse of alive records.

Index nodes in the TSB-tree are treated differently from data nodes. The children of index nodes are rectangles in time-key space. So making a time split or key split of an index node may cause a lower level node to be referred to by two parents.

Index node splits in the TSB-tree are restricted in ways which guarantee that current nodes (the only ones where insertions and updates occur) have only one parent [Lomet and Salzberg, 1989]. This parent is a current index node. A time split can be done on any time before the start time of the oldest current child. If time splits were allowed at current transaction time for index nodes, lower level current nodes would have more than one parent. A key split can be done at any current key boundary. This also assures that lower level current nodes have only one parent.

2.4 The Multiversion B-tree

The MVBT approach transforms a timeslice query to a partially persistence problem. In particular, a data structure is called *persistent* [Driscoll et al., 1989] 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 (i.e., changes are applied only to the newest version). In *full* persistence every version can be modified.

The key observation is that partial persistence “suits” nicely with a transaction-time evolution since changes are always applied on the latest state $S(t)$ of the evolving set (recall Figure 5.1). To support key range queries on a given $S(t)$, one could use an ephemeral 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 the ephemeral B^+ -tree took during the evolution of $S(t)$ is clearly inefficient. Instead one should “see” the evolution of the ephemeral B^+ -tree as a partially persistence problem, i.e., as a set of updates that create subsequent versions of the B^+ -tree. [Driscoll et al., 1989] shows how to make an ephemeral linked main-memory data structure partially persistent. [Becker et al., 1996] describes how to make an ephemeral external method, like the B^+ -tree, partially persistent. Below we discuss this approach.

Conceptually the MVBT stores all the states assumed by the ephemeral 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 ephemeral 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 $S(t)$. Thus the MVBT directly represents object deletions. This resembles the transaction-time lifetime used in the Snapshot Index records. Index records in the non-leaf pages of the MVBT maintain the evolution of the corresponding index records of the ephemeral B^+ -tree and are also augmented with insertion-time and deletion-time fields. Therefore each (data or index) record has a transaction interval during which it is called *alive*.

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 is similar to the page usefulness property of the Snapshot Index. Again, 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. 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 ephemeral B^+ -tree. Note that this step is carried out by taking into account the transaction-time intervals of the index and data records visited, i.e., only the latest state of the ephemeral B^+ -tree is traversed. 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 $[t, now)$ 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 as a result of the deletion. The former structural change is called a *page overflow*; the latter is a *weak version underflow* [Becker et al., 1996]. 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 . (We can assume that the deletion-time field of all of x 's alive records is changed to t even though this is not needed in practice). Then the resulting new page has to be incorporated in the structure. Briefly, there are three cases for handling the new page y .

First, if the number of records in y are within a certain specified range, y is directly inserted in the MVBT structure. (This specified range is known a priori. In short, the number of records should be between $q+e$ and $B-e$ where e is a predetermined constant. Constant e works as a buffer that guarantees that a new structural change to the new page y can happen only after at least e new changes). The page insertion is carried out by accessing the parent page of x , marking the index record to x as deleted at the current time t , and inserting a new index record

pointing to page y . (Conceptually this implies that page x is dead, i.e., not accessed for times larger than t). Even though these changes occur in an internal page, they are similar to insertion and deletion of data records in a leaf-page and are handled identically. Similarly, these insertions and deletions can create new changes up the tree to the ancestors and so on.

The second case is if the resulting page y has more records than the specified range; this is called a *strong version overflow* condition and is handled by splitting y into two pages and then accommodating these pages in the structure in a manner similar to the one described above.

The third case is if page y has less records than the specified range; this condition is called a *strong version underflow* and is handled by merging y with another “sibling” page and then accommodating the new page(s) in the MVBT structure. 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 which 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 [Becker et al., 1996]). In general, not many splits propagate to the top so the number of root splits is small. Hence the root* structure can be kept in main memory.

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 ephemeral 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 regular B^+ -tree. This search takes into account 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. Answering a range query on a transaction time interval $P = [t, t']$ is similar. First all roots with transaction interval intersecting P are found. Starting from the first alive root, MVBT pages are recursively accessed provided that they have objects whose lifetime intervals intersect P . Since the MVBT is a graph, some pages are accessible by multiple roots. A method to avoid re-accessing pages is presented in [Van den Bercken et al., 1996].

It can be shown [Becker et al., 1996] that the above splitting/merging policies use space that is still linear to n (i.e., $O(n/B)$). 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 a B^+ -tree on the m elements of $S(t)$. A range-timeslice query takes $O(\log_B n + s/B)$ I/O's.

The MVAS [Varman and Verma, 1997] is similar to the MVBT, however it achieves a smaller constant on the space bound by using better policies to handle the cases when key-splits or merges are performed. There are two main differences in the MVAS policies. The first deals with the case when a node becomes sparse after performing a record deletion. Instead of always consuming a new page (as in the MVBT), the MVAS tries to find a sibling with free space where the remaining alive entries of the time-split page can

be stored. The conditions under which this step is carried out are described in detail in [Varman and Verma, 1997]. The second difference deals with the case when the number of entries in a just time-split node is below the pre-specified threshold. If a sibling page has enough alive records, the MVBT would copy all the sibling's alive records to the sparse time-split page thus "deleting" the sibling page. Instead the MVAS will copy only as many alive records from the sibling page, needed for the time-split page to avoid violating the threshold. The above two modifications reduce the extent of duplication, hence reducing the overall space. As a result the MVAS reduces the worst case storage bound of MVBT by a factor of 2.

2.5 The Overlapping B-tree

As with the MVBT approach the evolving set $S(t)$ is indexed by a B^+ -tree. The intuition behind the Overlapping B-tree [Manolopoulos and Kapetanakis, 1990; Burton et al., 1985] is that the ephemeral B^+ -trees of subsequent versions (states) of $S(t)$ will not differ much. The Overlapping B-tree is thus a graph structure that superimposes many ephemeral B^+ -trees (Figure 5.8). 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 by having index nodes of the new B^+ -tree 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.

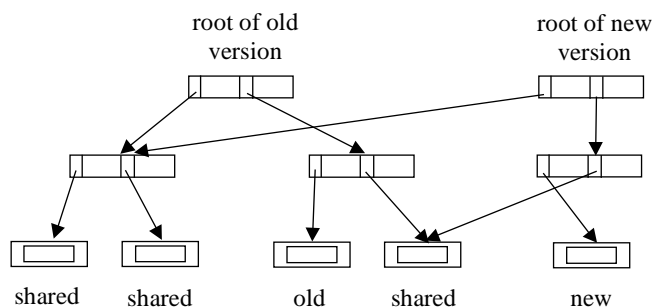


Figure 5.8. The Overlapping B-tree.

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. This is done by timestamping the roots with the transaction time of their creation. These timestamps can be easily indexed on a separate B^+ -tree. This is similar to the root* structure of the MVBT however, the Overlapping B-tree creates a new root per version. After the appropriate root is

found, the search continues by traversing the tree under this root, as if an ephemeral 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.

The advantage of the Overlapping structure is in the simplicity of its implementation. Note that except the roots, the other nodes do not involve any timestamping. Such timestamping 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 [Tzouramanis et al., 1999].

2.6 Temporal Hashing

As already described, transaction-time indices like the TSB-tree, MVBT, MVAS or the Overlapping B-tree, correspond to transaction-time analogues of an ephemeral B^+ -tree. In this section we discuss temporal hashing which is the temporal analogue of ephemeral external hashing. We assume again a collection of objects s that evolves over time and we are interested in answering membership queries for any state $S(t)$ that collection S exhibited. Such membership queries include a temporal predicate as in: "find whether object with identity k was in $S(t)$ ".

As in the ephemeral case (B^+ -tree versus hashing), one could use a temporal index to address temporal membership queries by simply reducing the search key range to a single value k . This however will introduce the usual logarithmic search and update overhead. There are applications where this overhead may be a disadvantage. For example, admission control and security applications are more interested at the whereabouts of a given object at a given time (or interval), instead of a collection of objects. Temporal hashing can also be useful as a partitioning method for expediting temporal join queries. For example, given two time-evolving sets X and Y , join their states at times t_1 and t_2 (that is, find the objects common to states $X(t_1)$ and $Y(t_2)$). Before joining the two large sets, the temporal membership property is used to partition each set into smaller disjoint subsets, in a way that every subset from the first set is joined with only one subset from the second set. A byproduct of partitioning (and thus of the membership property) is query parallelism. That is, the above temporal joins can be performed faster, if the corresponding set partitions are joined in parallel.

Assume that for every time t when $S(t)$ changes (by adding/deleting objects) we could have a good ephemeral dynamic hashing scheme $h(t)$ that maps efficiently the objects in $S(t)$ into a collection of buckets $b(t)$. One straightforward solution to the temporal hashing problem would be to separately store each collection of buckets $b(t)$ for each t . Answering a temporal membership query for object k and time t requires first applying $h(t)$ on k and then accessing the appropriate bucket of $b(t)$. This would provide an excellent query performance (as it uses hashing scheme $h(t)$

for each t), but the space requirements are prohibitively large. Flashing each $b(t)$ on the disk could easily create $O(n^2)$ space.

A better solution is to use again the partial persistence approach. The *partially persistent hashing* of [Kollios and Tsotras, 1998] indexes the evolution of each bucket using the Snapshot Index [Tsotras and Kangelaris, 1995] (one index per bucket ever used). The method behaves as if a separate hashing scheme is available on every state $S(t)$ but uses space linear in (the number of updates) n .

Assume for example that each state $S(t)$ is hashed using a Linear Hashing scheme $h(t)$ (the methodology applies similarly to other ephemeral hashing schemes). There are two basic time-dependent parameters that identify $h(t)$ for each t , namely $i(t)$ and $p(t)$. Parameter $i(t)$ is the round number at time t . The value of parameter $p(t)$ identifies the next bucket to be split.

An interesting property of Linear Hashing is that buckets are reused; when round $i+1$ starts it has double the number of buckets of round i but the first half of the bucket sequence is the same since new buckets are appended in the end of the file.

Let b_{total} denote the longest sequence of buckets ever used during the evolution of $S(t)$ and assume that b_{total} consists of buckets:

$0, 1, 2, \dots, 2^q M - 1$. The above observation implies that for all t , $b(t)$ (the collection of buckets used at time t) is a prefix of b_{total} . In addition $i(t) \leq q, \forall t$.

Consider bucket b_j from the sequence b_{total} ($0 \leq j \leq 2^q M - 1$) and observe the collection of objects that are stored in this bucket as time proceeds. The state of bucket b_j at time t , namely $b_j(t)$, is the set of objects stored in this bucket at t . Let $|b_j(t)|$ denote the number of objects in $b_j(t)$. If all states $b_j(t)$ can somehow be reconstructed for each bucket b_j , answering a temporal membership query for object k at time t can be answered in two steps:

1. find which bucket b_j , object k would have been mapped by the hashing scheme at t , and,
2. search through the contents of $b_j(t)$ until k is found.

The first step requires identifying which hashing scheme was used at time t . The evolution of the hashing scheme $h(t)$ is easily maintained if a record of the form $\langle t, i(t), p(t) \rangle$ is appended to an array H , for those instants t where the values of $i(t)$ and/or $p(t)$ change. Given any t , the hashing function used at t is identified by simply locating t inside the time-ordered H in a logarithmic search. In practice, array H is small and can be kept in main memory.

The second step implies accessing $b_j(t)$. By observing the evolution of bucket b_j we note that its state changes as an *evolving set* by adding or deleting objects. Each such change can be timestamped with the time instant it occurred. Assume that a rehashing occurs at some time t' and results in moving v objects from bucket b_j to b_r . For the evolution of b_j (b_r), this rehashing is viewed as a deletion (respectively, addition) of the v objects at time t' , i.e., all such deletions (additions) are timestamped with the same time t' for the corresponding object's evolution. As a result, the evolution of each bucket can be indexed by a Snapshot Index.

Consider Figure 3.4 (see Chapter 3) that shows an example of an ephemeral Linear Hashing scheme at two different time instants (the scheme uses $m = 5$ buckets, each holding $B = 2$ records). Assume that the addition of key (search_value) 8 happens at

<p>evolution of set S up to time $t=25$:</p> <table border="1"> <thead> <tr> <th>t</th> <th>oid</th> <th>$oper.$</th> </tr> </thead> <tbody> <tr><td>1,</td><td>10,</td><td>+</td></tr> <tr><td>2,</td><td>7,</td><td>+</td></tr> <tr><td>4,</td><td>3,</td><td>+</td></tr> <tr><td>8,</td><td>21,</td><td>+</td></tr> <tr><td>9,</td><td>15,</td><td>+</td></tr> <tr><td>15,</td><td>36,</td><td>+</td></tr> <tr><td>16,</td><td>29,</td><td>+</td></tr> <tr><td>17,</td><td>13,</td><td>+</td></tr> <tr><td>20,</td><td>12,</td><td>+</td></tr> <tr><td>21,</td><td>8,</td><td>+</td></tr> <tr><td>25,</td><td>10,</td><td>-</td></tr> </tbody> </table>	t	oid	$oper.$	1,	10,	+	2,	7,	+	4,	3,	+	8,	21,	+	9,	15,	+	15,	36,	+	16,	29,	+	17,	13,	+	20,	12,	+	21,	8,	+	25,	10,	-	<p>evolution of bucket 0:</p> <table border="1"> <thead> <tr> <th>t</th> <th>oid</th> <th>$oper.$</th> </tr> </thead> <tbody> <tr><td>1,</td><td>10,</td><td>+</td></tr> <tr><td>9,</td><td>15,</td><td>+</td></tr> <tr><td>21,</td><td>15,</td><td>-</td></tr> <tr><td>25,</td><td>10,</td><td>-</td></tr> </tbody> </table>	t	oid	$oper.$	1,	10,	+	9,	15,	+	21,	15,	-	25,	10,	-	<p>records in bucket 0's history:</p> <table border="1"> <thead> <tr> <th>at $t=20$:</th> <th>at $t=21$:</th> <th>at $t=25$:</th> </tr> </thead> <tbody> <tr> <td>$\langle oid, lifespan \rangle$</td> <td>$\langle oid, lifespan \rangle$</td> <td>$\langle oid, lifespan \rangle$</td> </tr> <tr> <td>$\langle 10, [1, now] \rangle$</td> <td>$\langle 10, [1, now] \rangle$</td> <td>$\langle 10, [1, 25] \rangle$</td> </tr> <tr> <td>$\langle 15, [9, now] \rangle$</td> <td>$\langle 15, [9, 21] \rangle$</td> <td>$\langle 15, [9, 21] \rangle$</td> </tr> </tbody> </table>	at $t=20$:	at $t=21$:	at $t=25$:	$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$	$\langle 10, [1, now] \rangle$	$\langle 10, [1, now] \rangle$	$\langle 10, [1, 25] \rangle$	$\langle 15, [9, now] \rangle$	$\langle 15, [9, 21] \rangle$	$\langle 15, [9, 21] \rangle$
t	oid	$oper.$																																																															
1,	10,	+																																																															
2,	7,	+																																																															
4,	3,	+																																																															
8,	21,	+																																																															
9,	15,	+																																																															
15,	36,	+																																																															
16,	29,	+																																																															
17,	13,	+																																																															
20,	12,	+																																																															
21,	8,	+																																																															
25,	10,	-																																																															
t	oid	$oper.$																																																															
1,	10,	+																																																															
9,	15,	+																																																															
21,	15,	-																																																															
25,	10,	-																																																															
at $t=20$:	at $t=21$:	at $t=25$:																																																															
$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$																																																															
$\langle 10, [1, now] \rangle$	$\langle 10, [1, now] \rangle$	$\langle 10, [1, 25] \rangle$																																																															
$\langle 15, [9, now] \rangle$	$\langle 15, [9, 21] \rangle$	$\langle 15, [9, 21] \rangle$																																																															
	<p>evolution of bucket 3:</p> <table border="1"> <thead> <tr> <th>t</th> <th>oid</th> <th>$oper.$</th> </tr> </thead> <tbody> <tr><td>4,</td><td>3,</td><td>+</td></tr> <tr><td>17,</td><td>13,</td><td>+</td></tr> <tr><td>21,</td><td>8,</td><td>+</td></tr> </tbody> </table>	t	oid	$oper.$	4,	3,	+	17,	13,	+	21,	8,	+	<p>records in bucket 3's history:</p> <table border="1"> <thead> <tr> <th>at $t=20$:</th> <th>at $t=21$:</th> <th>at $t=25$:</th> </tr> </thead> <tbody> <tr> <td>$\langle oid, lifespan \rangle$</td> <td>$\langle oid, lifespan \rangle$</td> <td>$\langle oid, lifespan \rangle$</td> </tr> <tr> <td>$\langle 3, [4, now] \rangle$</td> <td>$\langle 3, [4, now] \rangle$</td> <td>$\langle 3, [4, now] \rangle$</td> </tr> <tr> <td>$\langle 13, [17, now] \rangle$</td> <td>$\langle 13, [17, now] \rangle$</td> <td>$\langle 13, [17, now] \rangle$</td> </tr> <tr> <td></td> <td>$\langle 8, [21, now] \rangle$</td> <td>$\langle 8, [21, now] \rangle$</td> </tr> </tbody> </table>	at $t=20$:	at $t=21$:	at $t=25$:	$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$	$\langle 3, [4, now] \rangle$	$\langle 3, [4, now] \rangle$	$\langle 3, [4, now] \rangle$	$\langle 13, [17, now] \rangle$	$\langle 13, [17, now] \rangle$	$\langle 13, [17, now] \rangle$		$\langle 8, [21, now] \rangle$	$\langle 8, [21, now] \rangle$																																				
t	oid	$oper.$																																																															
4,	3,	+																																																															
17,	13,	+																																																															
21,	8,	+																																																															
at $t=20$:	at $t=21$:	at $t=25$:																																																															
$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$																																																															
$\langle 3, [4, now] \rangle$	$\langle 3, [4, now] \rangle$	$\langle 3, [4, now] \rangle$																																																															
$\langle 13, [17, now] \rangle$	$\langle 13, [17, now] \rangle$	$\langle 13, [17, now] \rangle$																																																															
	$\langle 8, [21, now] \rangle$	$\langle 8, [21, now] \rangle$																																																															
	<p>evolution of bucket 5:</p> <table border="1"> <thead> <tr> <th>t</th> <th>oid</th> <th>$oper.$</th> </tr> </thead> <tbody> <tr><td>21,</td><td>15,</td><td>+</td></tr> </tbody> </table>	t	oid	$oper.$	21,	15,	+	<p>records in bucket 5's history:</p> <table border="1"> <thead> <tr> <th>at $t=20$:</th> <th>at $t=21$:</th> <th>at $t=25$:</th> </tr> </thead> <tbody> <tr> <td>$\langle oid, lifespan \rangle$</td> <td>$\langle oid, lifespan \rangle$</td> <td>$\langle oid, lifespan \rangle$</td> </tr> <tr> <td>-</td> <td>$\langle 15, [21, now] \rangle$</td> <td>$\langle 15, [21, now] \rangle$</td> </tr> </tbody> </table>	at $t=20$:	at $t=21$:	at $t=25$:	$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$	-	$\langle 15, [21, now] \rangle$	$\langle 15, [21, now] \rangle$																																																
t	oid	$oper.$																																																															
21,	15,	+																																																															
at $t=20$:	at $t=21$:	at $t=25$:																																																															
$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$	$\langle oid, lifespan \rangle$																																																															
-	$\langle 15, [21, now] \rangle$	$\langle 15, [21, now] \rangle$																																																															

time $t = 21$. Figure 5.9 shows a corresponding evolution of set S , the evolutions of various buckets and their histories (a “+” or a “-” denotes addition or deletion, respectively). At time $t = 21$ the addition of object 8 triggers an overflow. Thus the contents of bucket 0 are rehashed between bucket 0 and bucket 5. As a result object 15 is moved from bucket 0 to bucket 5. For bucket’s 0 evolution this change is considered as a deletion at $t = 21$, however bucket 5 it is an addition of object 15 at the same $t = 21$. Later, at $t = 25$, object 10 is deleted from set S . This updates the lifespan of this object’s corresponding record in bucket 0’s history from $\langle 10, [1, now] \rangle$ to $\langle 10, [1, 25] \rangle$.

Figure 5.9: Evolution of a set and its linear hashing scheme.

Partially persistent hashing answers a temporal membership query about object k at time t , with almost the same query time efficiency (plus a small overhead) as if a separate ephemeral hashing scheme existed on each $S(t)$. First the hashing function used at t is found through array H ; this identifies the bucket b_j where k should have been assigned. Then the contents of this bucket at time t are accessed by the Snapshot Index. Searching the Snapshot Index of bucket b_j takes $O(\log_B n_j + |b_j(t)|/B)$ where n_j corresponds to the number of changes recorded in bucket b_j ’s history. If the Linear Hashing scheme [Kollios and Tsotras, 1998] show that if a controlled ephemeral Linear Hashing scheme is used, partially persistent hashing uses $O(n/B)$ space and an $O(1)$ amortized expected update processing per change.

3. BITEMPORAL INDEXING

There are three approaches that can be used for indexing bitemporal databases. The straightforward one is to have each bitemporal object represented by a “bounding rectangle” created by the object’s valid and transaction-time intervals, and 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 [Kumar et al., 1998]. A bitemporal object with valid-time interval I which is inserted in the database at transaction time t , is represented by a rectangle with a transaction-time interval of the form $[t, now)$. All bitemporal objects which have not been deleted (in the transaction sense) will share the common transaction-time endpoint *now*. Furthermore, intervals that remain unchanged will create long (in the transaction-time axis) rectangles, a reason for further overlapping. A simple bitemporal query that asks for all valid time intervals which at transaction time t_i contained valid time v_j , corresponds to finding all rectangles that contain point (t_i, v_j) .

Figure 5.10 illustrates the bounding-rectangle approach; only the valid and transaction axis are shown. At t_5 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 *now*.

To avoid overlapping, the use of two R-trees has also been proposed [Kumar et al., 1998]. When a bitemporal 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 bitemporal objects whose right transaction endpoint is unknown. If a bitemporal 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 bitemporal objects

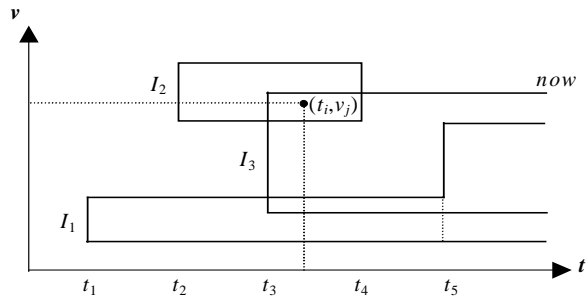


Figure 5.10. The bounding-rectangle approach for bitemporal objects.

with known transaction-time interval. At any given time, all bitemporal 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 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.

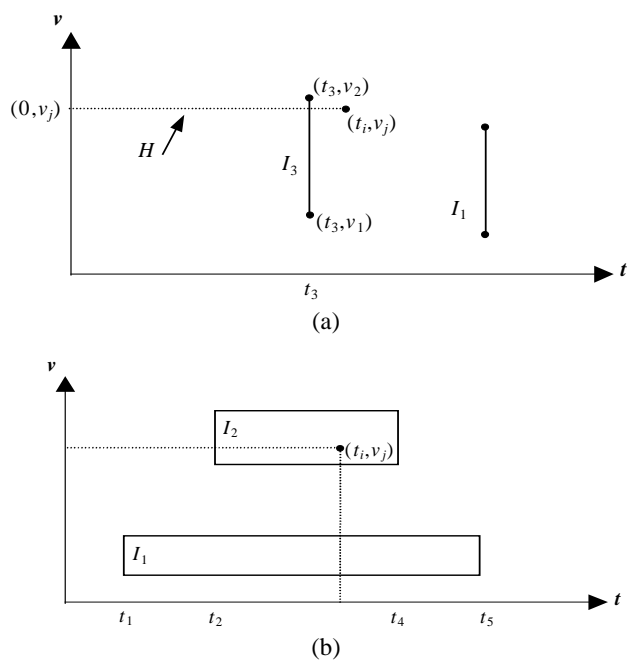


Figure 5.11. The two R-tree methodology for bitemporal data.

In Figure 5.11, the two R-tree methodology for bitemporal data is divided according to whether their right transaction endpoint is known. The scenario of Figure 5.10 is presented here (i.e., after time t_5 has elapsed). The 2-dimensional space (a) is stored in the *front* R-tree while (b) in the *back* R-tree. The query is then translated into an interval intersection and a point enclosure problem, respectively.

A simple bitemporal query that asks for all valid time intervals which at transaction time t_i contained valid time v_j , 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 which intersect a horizontal interval H that starts from the beginning of transaction time and extends until point t_i at height v_j .

A third approach to address bitemporal problems is by using the notion of partial persistence. This solution emanates from the abstraction of a bitemporal database as a sequence of collections $C(t)$ (in Figure 5.3) 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 ephemeral structure implies what queries are answered by the bitemporal structure. Using this approach, the Bitemporal R-tree was introduced in [Kumar et al., 1998] that takes an R-tree and makes it partially persistent.

By “viewing” a bitemporal query as a partial persistence problem, we obtain a *double* advantage. First we disassociate the valid-time requirements from the transaction-time ones. More specifically, the valid time support is provided from the properties of the ephemeral R-tree while the transaction time support is achieved by making this structure partially persistent. 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 on the most current state of the structure and last until updated (if ever) at a later transaction time, thus avoiding the explicit representation of *now*.

In making the ephemeral R-tree partially persistent one could use previous work on partially persistent B⁺-trees, in particular the MVBT [Becker et al., 1996] (and its improvement the MVAS [Varman and Verma, 1997]) or the Time-Split B-tree [Lomet and Salzberg, 1989]. This is because both B⁺-trees and R-trees are multiway-balanced structures that evolve through page splits and merges.

There are however two basic differences in the way the Bitemporal R-tree is updated as compared to the partially persistent B⁺-tree. These differences are:

1. The single order among the stored elements in a B⁺-tree creates an order among the tree's pages, too. Hence, a B⁺-tree page has at most two sibling pages and these are the only possible candidates for this page to merge with, if needed. In comparison, the ephemeral R-tree stores spatial objects and hence the notion of a sibling has to be redefined. Note however that merging in an ephemeral R-tree is not handled explicitly. If a page goes below the lower number of records due to deletions this page is not merged with another page. Instead, the records of the underutilized page are reinserted in the R*-tree variant [Beckmann et al., 1990] to be discussed in Chapter 6. The reinsertion method is not feasible in the Bitemporal R-tree, since a persistent structure “records” all changes that happen in its state. An underutilized page of an R-tree is half full and thus it can cause $O(B)$ record reinsertions. Each record reinsertion could at worst modify the whole path in the R-tree (i.e., logarithmic number of changes). Recording all these changes in the Bitemporal R-tree will require excessive space. To avoid this problem, the Bitemporal R-tree performs merging explicitly. Merging with a sibling may still change the whole path but this will happen once for the under-utilized page. It is an interesting optimization problem to choose with which sibling a page is merged; [Kumar et al., 1998] examines various merging policies.
2. The second difference is with the way insertions and deletions are handled when they do not lead to structural changes. In an ephemeral B⁺-tree, an insertion to a page that has enough empty space is simply performed by adding the new key in the page; no parent page is updated. In an ephemeral R-tree a similar insertion may increase the geometric area covered by the data page. Then the parent page must also be changed, in particular the rectangle of the index record that points to the data page (so that the information about previous data page area is not lost). As this may propagate to the root, an insertion can cause a logarithmic number of updates even though no new page is added on the ephemeral R-tree.

To avoid recording all these changes, the Bitemporal R-tree simply adjusts the current index records in ancestor pages without making copies of these records. Consider a given index record created at time t with some initial rectangle area. At various time instants after t , its rectangle area is subsequently increased (due to non-structural insertions in the pages underneath) but the record's insertion-time remains t . If at a later time t' this index record is (logically) deleted, its transaction interval would be $[t, t')$ and the prevailing rectangle area would be the latest (and largest) this index record received. A query that follows this index record will provide the correct answer for all times in $[t, t')$ since the prevailing rectangle area contains all previous ones. Hence the above policy does not violate the correctness of the Bitemporal R-tree. Since a non-structural deletion can only decrease a page's overall area the Bitemporal R-tree does not adjust ancestor index records (the previous rectangle area contains the new one and queries will still be answered correctly). Among all three bitemporal approaches the Bitemporal R-tree has been shown to be more efficient [Kumar et al., 1998].

4. FURTHER READING

There has been a plethora of temporal access methods in recent years. [Salzberg and Tsotras, 1999] presents a comparison. Other pure-timeslice query methods are: the Append-only Tree [Gunadhi and Segev, 1993], the Time Index [Elmasri et al., 1990] and its variations (Monotonic B-tree [Elmasri et al., 1993], Time Index+ [Kouramajian et al., 1994]), the Differential File approach [Jensen et al., 1991], the Checkpoint Index [Leung and Muntz, 1993], the Archivable Time Index [Verma and Varman, 1994], and the Windows Method [Ramaswamy, 1997]. Range-timeslice query methods include: the Composite Indices of Postgres [Kolovson and Stonebraker, 1989], Segment R-tree [Kolovson and Stonebraker, 1991], the Write-Once B-tree [Easton, 1986], the Persistent B-tree [Lanka and Mays, 1991] and the TP-Index [Shen et al., 1994]. Temporal queries over multiple lines of evolution appear in [Landau et al., 1995]. Another approach for bitemporal indexing appears in [Nascimento et al., 1996]. In [Bliujute et al., 1998] bitemporal indices for data whose valid time extends to now are presented.

REFERENCES

- Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. (1996). An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal*, 5(4):264-275.
- Burton, F.W., Huntbach, M.M., and Kollias, J.G. (1985). Multiple Generation Text Files using Overlapping Tree Structures. *The Computer Journal*, 28(4):414-416.

- Beckmann, N., Kriegel, H.P., Schneider, R., and Seeger, B. (1990). The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 322-331.
- Bliujute, R., Jensen, C.S., Saltenis, S. and Slivinskas, G. (1998). R-Tree Based Indexing of Now-Relative Bitemporal Data. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 345-356.
- Driscoll, J.R., Sarnak, N., Sleator, D., and Tarjan, R.E. (1989). Making Data Structures Persistent. *Journal of Computer and Systems Sciences*, 38(1):86-124.
- Easton, M.C. (1986). Key-sequence Data Sets on Indelible Storage. *IBM Journal on Research and Development*, 30(3):230-241.
- Elmasri, R., Wu, G., and Kim, Y. (1990). The Time Index: an Access Structure for Temporal Data. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 1-12.
- Elmasri, R., Wu, G., and Kouramajian, V. (1993). The Time Index and the Monotonic B⁺-tree. In *Temporal Databases: Theory, Design, and Implementation*, by Tansel, A., et al. (eds.), pages 433-456. Benjamin/Cummings.
- Gunadhi, H. and Segev, A. (1993). Efficient Indexing Methods for Temporal Relations. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):496-509.
- Jensen, C.S., et al. (1994). A Consensus Glossary of Temporal Database Concepts. *ACM SIGMOD Record*, 23(1):52-64.
- Jensen, C.S., Mark, L., and Roussopoulos, N. (1991). Incremental Implementation Model for Relational Databases with Transaction Time. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):461-473.
- Kollios, G. and Tsotras, V.J. (1998). Hashing Methods for Temporal Data, University of California at Riverside, Department of Computer Science, TR UCR_CS_98_01. Available from http://www.cs.ucr.edu/publications/tech_reports/
- Kolovson, C. and Stonebraker, M. (1989). Indexing Techniques for Historical Databases. In *Proceedings of the 5th IEEE International Conference on Data Engineering*, pages 127-137.
- Kolovson, C. and Stonebraker, M. (1991). Segment Indexes: Dynamic Indexing Techniques for Multi-dimensional Interval Data. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 138-147.
- Kumar, A., Tsotras, V.J., and Faloutsos, C. (1998). Designing Access Methods for Bitemporal Databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):1-20.
- Kouramajian, V., Kamel, I., Elmasri, R., and Waheed, S. (1994). The Time Index+: an Incremental Access Structure for Temporal Databases. In *Proceedings of the 3rd International Conference on Information and Knowledge Management*, pages 296-303.
- Landau, G.M., Schmidt, J.P. and Tsotras, V.J. (1995) Historical Queries Along Multiple Lines of Time Evolution. *The VLDB Journal* 4(4): 703-726.
- Lanka, S. and Mays, E. (1991). Fully Persistent B⁺-trees. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 426-435.
- Leung, T.Y.C and Muntz, R.R. (1993). Stream Processing: Temporal Query Processing and Optimization. In *Temporal Databases: Theory, Design, and Implementation*, by Tansel, A., Clifford, J., Gadia, S.K., Jajodia, S., Segev, A., and Snodgrass R. (eds.), pages 329-355, Benjamin/ Cummings.
- Lomet, D. and Salzberg, B. (1989). Access Methods for Multiversion Data. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 315-324.
- Manolopoulos, Y. and Kapetanakis, G. (1990). Overlapping B⁺-trees for Temporal Data. In *Proceedings of the 5th Jerusalem Conference on Information Technology*, pages 491-498.

- Nascimento, M., Dunham, M.H., and Elmasri, R. (1996). M-IVTT: a Practical Index for Bitemporal Databases. In *Proceedings of the 7th International Conference on Database and Expert Systems Applications*, pages 779-790.
- Ozsoyoglu, G. and Snodgrass, R. (1995). Temporal and Real-Time Databases: a Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513-532.
- Ramaswamy, S. (1997). Efficient Indexing for Constraint and Temporal Databases. In *Proceedings of the 6th International Conference on Database Theory*, pages 419-431.
- Snodgrass R. and Ahn, I. (1986). Temporal Databases. *IEEE Computer*, 19(9):35-42.
- Shen, H., Ooi, B.C., and Lu, H. (1994). The TP-Index: a Dynamic and Efficient Indexing Mechanism for Temporal Databases. In *Proceedings of the 10th International Conference on Data Engineering*, pages 274-281.
- Salzberg, B. and Tsostras, V.J. (1999). A Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, to appear. Also available as TimeCenter TR-18, <http://www.cs.auc.dk/research/DBS/tdb/TimeCenter/publications2.html>
- Tsostras, V.J., Gopinath, B., and Hart, G.W. (1995). Efficient Management of Time-Evolving Databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):591-608.
- Tsostras, V.J. and Kangelaris, N. (1995). The Snapshot Index: an I/O-Optimal Access Method for Timeslice Queries. *Information Systems*, 20(3):237-260.
- Tzouramanis, T., Manolopoulos, Y., and Lorentzos, N. (1999). Overlapping B⁺-trees: an Implementation of a Transaction Time Access Method. *Data and Knowledge Engineering*, 29 (3):381-404.
- Van den Bercken, J. and Seeger, B. (1996). Query Processing Techniques for Multiversion Access Methods. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 168-179.
- Verma, R.M. and Varman, P.J. (1994). Efficient Archivable Time Index: a Dynamic Indexing Scheme for Temporal Data. In *Proceedings of the Conference on Computer Systems and Education*, pages 59-72.
- Varman, P.J. and Verma, R.M. (1997). An Efficient Multiversion Access Structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391-409.