A Comparison of Access Methods for Temporal Data

Betty Salzberg and Vassilis J. Tsotras

June 13, 1997

TR-18

A TimeCenter Technical Report

Title A Comparison of Access Methods for Temporal Data

Author(s) Betty Salzberg and Vassilis J. Tsotras

Publication History June 1997. A TimeCenter Technical Report.

Abstract:

This paper attempts a comparison of different indexing techniques which have been proposed for supporting efficient access to temporal data. The comparison is based on a collection of important performance criteria that include the consumed space, the update processing and the query time for representative queries. Since it was not possible to compare actual method implementations against the same data sets, the comparison is based on worst case analysis, hence no assumptions on data distribution or query frequencies are made. When a number of methods have the same asymptotic worst case behavior, features in the methods which affect average case behavior are discussed. Additional criteria examined are the pagination of an index, the ability to cluster related data together and the ability to efficiently separate old from current data (so that larger archival storage media such as write-once optical disks can be used). The purpose of the paper is to identify the difficult problems in accessing temporal data and provide a good description of how the different methods aim to solve them. A general lower bound for answering basic temporal queries is also introduced.

1. Introduction

Conventional databases work in terms of a single logical state. Using transactions the database evolves from one consistent state to the next, while the previous state is discarded after a transaction commits; as a result, there is no memory with respect to prior states of the data. Such databases capture a single snapshot of reality (also called *snapshot* databases) and are insufficient for those applications that require the support of past, current or even future data. What is needed is a *temporal* database [SA86] since it fully supports the storage and querying of time varying data.

Research in temporal databases has shown an immense growth in recent years [TK96]. Various aspects of temporal databases have been examined [OS95], including temporal data models, query languages, access methods, etc. Prototype efforts appear in [B95]. In this paper we provide a comparison of proposed *temporal access methods*, i.e., indexing techniques for temporal data. We attempt to identify the problems in the area together with the solutions given by each method.

A taxonomy of time in databases has been developed in [SA85]. Specifically, *transaction time*, *valid time* and *user-defined time* have been proposed. Transaction and valid time are two orthogonal time dimensions. Transaction time is the time when a fact is stored in the database. It is consistent with the serialization order of transactions (i.e., it is monotonically increasing) and can be implemented using the commit times of transactions [S94]. Valid time denotes the time when a fact becomes effective (valid) in reality. User-defined time is an uninterpreted time domain managed by the user and therefore we will not discuss it further.

The term “temporal database” refers in general to a database that supports some aspect of time, not counting the user-defined time. Depending on the time dimension(s) supported, there are three kinds of temporal databases: *transaction-time*, *valid-time* and *bitemporal* [J+94].

This research was partially supported by NSF grants IRI-9303403, IRI-9111271, IRI-9509527 and by the New York State Science and Technology Foundation as part of its Center for Advanced Technology program

A transaction-time database records the history of a database activity rather than real world history. As such, it can "rollback" to one of its previous states. Since previous transaction times cannot be changed (every change is stamped with a new transaction time), there is no way to change the past. This is useful for applications in auditing, billing etc. A valid-time database 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 in this temporal behavior, they are corrected by modifying the database. When a correction is applied, previous values are not retained. It is thus not possible to view the database as it was before the correction. A bitemporal database combines the features of the other two types. It more accurately represents reality and allows for retroactive as well as postactive changes.

The tuple-versioning temporal model [LJ88, NA87] is used in this paper. Under this model the database is a set of records (tuples) that store the versions of real-life objects. Each such record, has a time-invariant *key* (surrogate) and, in general, a number of time-variant *attributes*; for simplicity we assume that each record has exactly one time varying attribute. In addition it has one or two *intervals*, depending on which types of time are supported. Each interval is represented by two attributes: *start_time* and *end_time*.

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. This has resulted in much confusion in the past regarding the design of temporal access methods. To exemplify the distinct characteristics of the transaction and valid time dimensions, a separate *abstraction*, describing the central underlying problem for each kind of temporal database, is used.

The query performance of the examined methods is compared in the context of various temporal queries. In order to distinguish among the various kinds of queries, we use a general *temporal query classification* scheme [SJ96]. The paper also introduces *lower* bounds for answering basic temporal queries. Each lower bound assumes a disk-oriented environment and describes the minimal I/O needed for solving the query if the space consumption is kept minimal. Access methods that achieve a matching upper bound for a temporal query are also indicated.

Among the discussed methods, the ones that support transaction time (either in a transaction-time or in a bitemporal environment) assume a *linear* transaction-time evolution [OS95]. This implies that a new database state is created by updating <u>only</u> the current database state. Another option is the so-called *branching* transaction time [OS95]. Branching methods are not discussed further except noting that related problems are investigated in [DSST89], [LM91], [SL95] and [LST95]. In particular, in [DSST89] and [LM91] a new state can be created by updating any of the past states; each new state gets a separate version identifier by which it can be directly accessed. In [SL95] version identifiers are replaced by (branch identifier, timestamp) pairs. Both a tree access method and a forest access method are proposed for these branched versions. [LST95] deals with multiple lines of evolutions created by splitting the database state: at some time instant the state of the database can be split into independently evolving sub-states etc. Instead of version identifiers timestamps are used. The same timestamp can appear in many different evolution lines, thus concurrent states at parallel evolutions can be related through queries.

Recently, other kinds of temporal, in particular *time-series*, queries have appeared in literature [AFS93, FRM94, JMM95]. Given are a pattern and a time-series (an evolution) and the typical query asks for all those times that a similar pattern appeared in the series. The search involves some distance criterion that qualifies when a pattern is similar to the given pattern. The distance criterion guarantees no false dismissals (false alarms are eliminated afterwards). Whole pattern matching [AFS93] and sub-matching [FRM94] queries have been examined. Such time-series queries are *reciprocal* in nature to the temporal queries addressed here (which usually provide a time instant and ask for the pattern at that time) and are not covered in this paper.

The rest of the paper is organized as follows: section 2 specifies the basic problem underlying each of the three temporal databases. We categorize a method as transaction-time, valid-time and bitemporal, depending on which time dimension(s) it most efficiently supports. Section 3 presents the items on which our comparison was based including the lower bounds. Section 4 discusses in more detail the basic characteristics that a good transaction or bitemporal access method should have. The examined methods are presented in section 5. The majority of them falls in the transaction-time category which comprises the bulk of this paper (subsection 5.1). Within the transaction-time category we further classify methods according to what queries they more efficiently support (key-only, time-only or time-key methods) A table summarizing the worst case performance characteristics of the transaction-time methods is also included. For completeness we also cover valid-time and bitemporal methods in subsections 5.2 and 5.3 respectively. We conclude the paper with a discussion on remaining open problems.

2. Problem Specification

The following discussion is influenced by [SA86] where the differences between valid and transaction time were introduced and illustrated through various examples. Here we attempt to identify the implications to the access method design from the support of each time dimension.

To visualize a transaction-time database consider first an initially empty set of objects that evolves over time as follows. Time is assumed to be discrete and described by a succession of consecutive nonnegative integers. Any change is assumed to occur at a time indicated by one of these integers. A change is the addition or deletion of an object or the value change of the 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 hired or re-hired), salary changes or employee deletions (as they retire or leave the company). Since an attribute value change can be represented by the artificial deletion of the object followed by the simultaneous rebirth of this object having the modified attribute, we may concentrate on object additions or deletions. Such an evolution appears in Figure 1. An object is *alive* from the time that it is added in the set and until (if ever) it is deleted from the set. The state of the evolving set at time t, namely $s(t)$, consists of all the alive objects at t. Note that changes are always applied to the most current state $s(t)$, i.e., past states cannot be changed.

Assume that the history of the above evolution is 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 on the evolving set at time t a record representing this object is stored in the database accompanied by a transaction-time interval of the form [t, *now*). *now* is a variable representing the current transaction time and is used because at the time the object is born its deletion time is yet unknown. If this object is later deleted at time t', the transaction-time interval of the corresponding record is updated to $[t, t')$. An object deletion in the evolving set is thus represented as a "logical" deletion in the database (the record of the deleted object is still retained in the database but with a different transaction *end_time*).

Since a transaction-time database system keeps both current and past data, it is natural to introduce the notion of a logical database state as a function of time. We should therefore

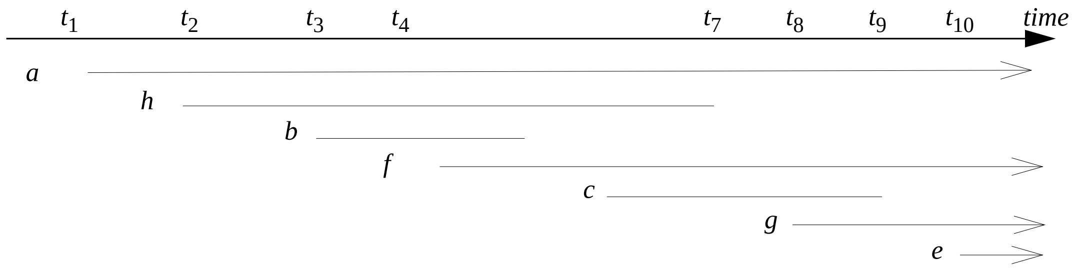


Figure 1: An example evolution where changes occur in increasing time order. The evolution is depicted as of time t_{10}. Lines ending to '>' correspond to objects that have not yet been deleted. At t_{10} state $s(t_9) = \{a,f,g\}$ is updated by the addition of object *e* to create state $s(t_{10}) = \{a,f,g,e\}$.

distinguish between the database system and logical database state. (This is not required in traditional database systems as there always exist exactly one logical database state---the current one). The logical database state at time *t* consists of those records whose transaction time interval contains *t*. Under the implicit updating assumption, the logical database state is equivalent to the state *s(t)* of the observed evolving set. Since an object can be reborn there may be many records (or versions) that are stored in the database system representing the history of the same object. But all these records correspond to disjoint transaction-time intervals in the object's history and each such record can belong to a single logical database state.

To summarize, an access method for a transaction-time database needs to: (a) store its past logical states, (b) support addition/deletion/modification changes on the objects of its current logical state, and, (c) efficiently access and query the objects in any of its states.

In general, a fact can be entered in the database at a different time than when it happened in reality. This implies that 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.

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 are the addition/deletion/ modification of an interval-object, but the collection's evolution (past states) is <u>not</u> kept. An example of a dynamic collection of object-intervals appears at Figure 2.

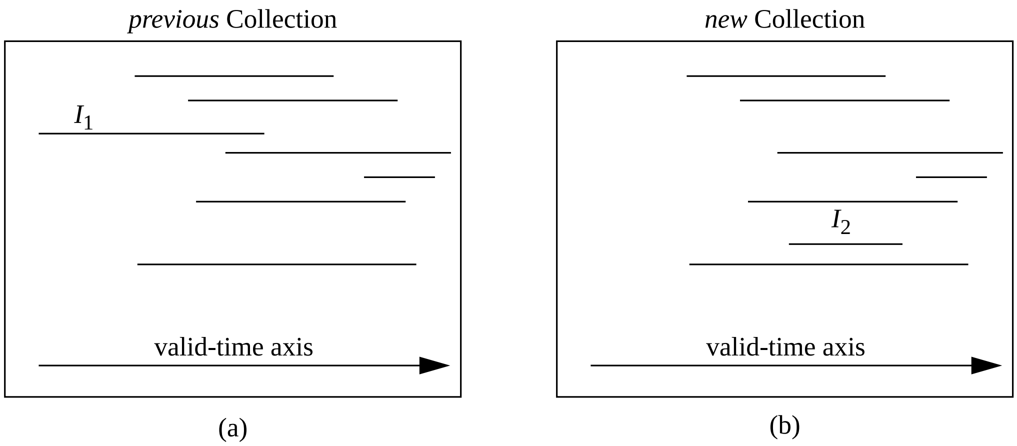


Figure 2: Two states of a dynamic collection of interval-objects. Only the valid-time intervals of the objects are shown. The new collection (b) is created from the previous collection (a) after deleting object I_1 and adding object I_2. Only the new (latest) collection is retained.

As a real-life example consider the collection of contracts in a company. Each contract has an identity (*contract_no*), an *amount* attribute and an interval representing the contract's duration or validity. Assume that when a correction is applied only the corrected contract is kept.

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 kept. 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. The valid-time database keeps only the latest "snapshot" of the collection of interval-objects. Querying a valid-time database cannot give any information on the past states of the database or how the collection evolved. Note that the database may store records with the same surrogate but with non-intersecting valid-time intervals.

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 (alive) 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.

An access method for a valid-time database should therefore: (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.

Reality is more accurately represented if both time dimensions are supported. The abstraction of a bitemporal database can be viewed as keeping the evolution (through the support of transaction-time) of a dynamic collection of (valid-time) interval-objects. Figure 3 (taken from [KTF95b]) 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. If each interval-object represents a company contract we can now represent how our knowledge about such contracts evolved. When an interval-object is inserted in the database at transaction-time t a record is created with the object's surrogate (contract_no), attribute (contract amount) and valid-time interval (contract duration) and an initial transaction-time interval [t, *now*). The transaction-time interval endpoint will be changed to another transaction time if this object is later updated. For example, the record for interval-object I_2 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. Such a scenario happens if at time t_4 we realize that a contract was wrongly inserted at the database.

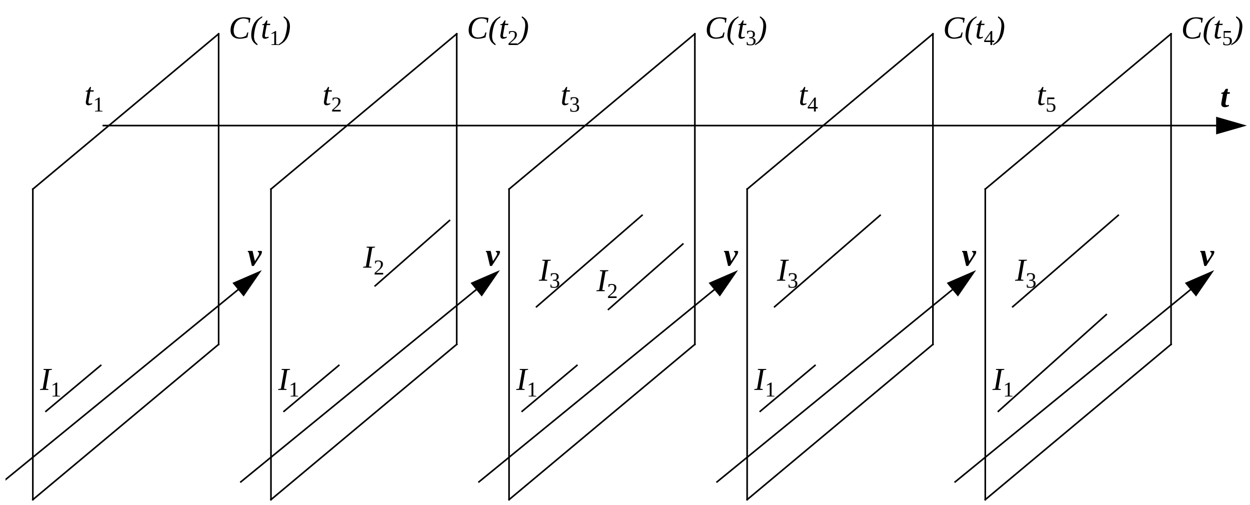


Figure 3: A conceptual view of a bitemporal database. The t-axis (v-axis) corresponds to transaction (valid) times. Only the valid-time interval is shown from each interval-object. At transaction time t_1 the database recorded that interval-object I_1 is added on collection $C(t_1)$. At t_5 the valid-time interval of object I_1 is modified to a new length.

A bitemporal access method should: (a) store its past logical states, (b) support addition/deletion/modification changes on the interval-objects of its current logical state, and, (c) efficiently access and query the interval-objects on any of its states.

Figure 3 is helpful in summarizing the differences among the underlying problems of the various database types. 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 valid-time database differs from a bitemporal since it keeps only one collection of interval-objects (the latest). Each collection *$C(t_i)$* can be thought on its own as a separate valid-time database. A transaction-time database differs from a (traditional) snapshot database in that it also keeps its past states instead of only the latest state. Finally, the difference between a valid-time and a snapshot database is that the former keeps interval-objects (and these intervals can be queried).

Most of the presented methods directly support a single time-dimension. We categorize methods that take advantage of the increasing time ordered changes as transaction-time access methods, since these are main characteristics of transaction-time. The bulk of this paper deals with transaction-time methods. Fewer approaches deal with valid-time access methods and even less with the bitemporal methods category (methods which support both time dimensions on the same index).

3. Comparison Items

This section elaborates on the items used in comparing the various access methods. We start with the various queries examined and proceed with the other criteria.

3.1 Queries

From a query perspective a valid-time and a transaction-time database are simply a collection of intervals. Figures 1 and 2(a) or 2(b) differ on how these intervals were created (which is important to the update and space performance of the access method) and what is their meaning (which is important for the application). Hence, for single time databases (valid or transaction) queries are of similar form. Since most methods assume transaction-time characteristics we discuss first queries in this domain, i.e., interval *T* below corresponds to a transaction-time interval and "history" is meant on the transaction-time axis. The examined queries can be categorized in the following classes:

(I) Given a contiguous interval *T*, find all objects alive during this interval.

(II) Given a key range and a contiguous time interval *T*, find the objects with keys in the given range and which are alive during interval *T*.

(III) Given a key range find the history of the objects in this range.

A special case of class (I) is when interval *T* is reduced to a single transaction time instant *t*.

This query has been termed the *transaction pure-timeslice*. In the company employee example this query is "find all employees working at the company at time *t*". It is usually the case that an access method which efficiently solves the timeslice query is also efficient for the more general interval query; we therefore consider the timeslice query as a good representative of class (I) queries.

Similarly for class (II), special cases include combinations where the key range and/or the transaction time interval, contain a single key and a single time instant respectively. For simplicity, we will consider the representative case when the time interval is reduced to a single transaction time instant; this is the *transaction range-timeslice* query ("find the employees working at the company at time *t* and whose ssn belongs in range *K*").

From class (III) we choose the special case that the key range is reduced to a single key as in: "find the salary history of employee with ssn *k*". This is the *transaction pure-key* query. If employee *k* ever existed the answer would be the salaries of that employee, else the answer is empty. In some methods, an instance of an employee object must be provided in the query and its previous salary history is found (this is because these methods need to include a time predicate in their search). This special pure-key query (termed the pure-key with time predicate) is of the form: "find the salary history of employee *k* who existed at time *t*."

Query class (I) can be thought as a special case of class (II) when no key range is specified and class (III) a special case of (II) when no interval is specified (rather, all times in history are of interest). As some of the proposed methods are better suited for answering queries from a particular class we discuss all three classes separately. If an access method as originally presented does not address queries from a given class but we feel that such queries could be addressed with a slight modification which does not affect the method's behavior we indicate so.

For valid-time databases we can similarly define the valid-time pure-timeslice ("find all contracts valid at time *v*"), valid-time range-timeslice ("find all contracts with numbers in range *K* and which are valid at *v*") etc. A bitemporal database enables queries in both time dimensions: "find all contracts that were valid on *v* = January 1, 1994, as recorded in the database at transaction time *t* = May 1, 1993". From all contracts in the collection *C(t)* for *t* = May 1, 1993, the query retrieves only the contracts that would be valid on Jan. 1, 1994.

The selection of the above query classes is definitely not complete, but contains basic, non-trivial queries. In particular, classes (I) and (II) relate to *intersection* based queries, i.e., the answer consists of objects whose interval contains some query time point or in general intersects a query interval. Depending on the application, other queries may be of importance. For example, find all objects with intervals before or after a query time point/interval, or all objects with intervals contained in a given interval [BO95, NDK96], etc.

To distinguish among the various temporal queries a three-entry notation, namely: *Key/Valid/ Transaction* [SJ96], will be alternatively used. This notation specifies which object attributes are involved in the query and in what way. Each entry is described as a "point", "range", "*," or "-". A "point" for the *Key* entry means that the user has specified a single value to match the object key; "point" for the *Valid* or *Transaction* entry implies a single time instant is specified for the valid or transaction-time domain. "range" indicates a specified range of object key values for the *Key* entry, or, an interval for the *Valid/Transaction* entries. A "*" means that any value is accepted in this entry, while "-" means that the entry is not applicable for this query. For example, "*/-/point" denotes the transaction pure-timeslice query, "range/point/-" is the valid range timeslice query and "point/-/*" is the transaction pure-key query. In a bitemporal environment the query "find all the company contracts that were valid on v = January 1, 1994, as recorded in the database during transaction time interval T: May 1-May 20, 1993" is an example of a "*/point/range" query. As presented, the three-entry notation deals with intersection queries but can be easily extended through the addition of extra entry descriptions to accommodate before/after and other kinds of temporal queries.

3.2 Access Method Costs

The performance of an access method is characterized by three costs: (1) the storage space used to physically store the data records and the structures of the access method, (2) the update processing (the time to update the method's data structures about a change that took place), and, (3) the query time for each of the basic queries.

An access method has two modes of operation: in the Update mode data is inserted, altered or deleted while in the Query mode queries are specified and answered using the access method. For a transaction-time access method the input for an update consists of a *time* instant t and all the changes that occurred on the data on that instant. A change is further specified by the unique *key* of the object it affects and the kind of *change* (addition, deletion or attribute modification). The access method's data structure(s) will then be updated to include the new change. Similar is the input to a bitemporal access method where the time of the change is also specified together with the changes and the interval-object(s) affected. The input to a valid-time access method simply contains the changes and the interval-object(s) affected.

For a transaction or a bitemporal method the space is a function of *n*, the total number of changes in the evolution, i.e., *n* is the summation of insertions, deletions and modification updates. 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 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 evolution's past. We assume that the total number of transaction instants is also $O(n)$. This is a natural assumption since every real computer system can process a possibly large but limited number of updates per transaction instant.

In a valid-time method, the space is a function of *l*, the number of interval-objects currently stored in the method, i.e., the size of the collection. For example, in both 2(a), 2(b) *l* is seven.

The query time of a method is a function of the answer size *a*. We use *a* to denote the answer size of a query in general.

Since temporal data can be large (especially in transaction and bitemporal databases), a good solution should use space efficiently. A method with fast update processing can be utilized even with a quickly changing real world application. In addition, fast query times will greatly facilitate the use of temporal data.

The basic queries that we examine can be considered as special cases of classical problems in computational geometry for which efficient in-core (main memory) solutions have been provided [CT92]. It should be mentioned that the general computational geometry problems support physical deletions of intervals. Hence they are more closely related to the valid-time database environment. The valid pure-timeslice query ("*/point/-") is a special case of the dynamic interval management problem. The best in-core bounds for the dynamic interval management are provided by using the Priority-Search tree data structure of [McC85], yielding $O(l)$ space, $O(\log l)$ update processing per change and $O(\log l + a)$ query time (all logarithms are base-2). Here *l* is the number of intervals in the structure when the query/update is performed. The range-timeslice query is a special case of the orthogonal segment intersection problem for which a solution using $O(l \log l)$ space, $O(\log l \log\log l)$ update processing and $O(\log l \log\log l + a)$ query time has been provided in [M84]; another solution [McC85] that uses a combination of the Priority-Search tree and the Interval Tree [E83] yields $O(l)$ space, $O(\log l)$ update processing and $O(\log^2 l + a)$ query time.

The problems addressed by transaction or bitemporal methods are related to work on persistent data structures [DSST89]. In particular, [DSST89] shows how to take an in-core “ephemeral data structure” (meaning that past states are erased when updates are made) and convert it to a “persistent data structure” (where past states are maintained). A “fully persistent” data structure allows updates to all past states. A “partially persistent” data structure allows updates only to the most recent state. Because of the properties of transaction time evolution, transaction and bitemporal access methods can be thought as disk extensions of partially persistent data structures.

3.3 Index Pagination and Data Clustering

In a database environment the cost of a computation is not based on how many main memory slots are accessed or how many comparisons are made (as it is the case with in-core algorithms) but instead on how many pages are transferred between main and secondary memory. In our comparison this is very crucial as the bulk of data will be stored in secondary storage media. It is therefore natural to use an I/O complexity cost ([KRVV93]) that measures the number of disk accesses for updating and query answering. Two important considerations regarding the I/O complexity of the query time are: *index pagination* and *data clustering*.

Index pagination deals with the issue of how well index nodes of a method are paginated. Since the index is used as a means to search for and update the data, its pagination greatly affects the performance of the method. As an example, a B^+-tree is a well-paginated index as it requires $O(\log_B r)$ page accesses for searching or updating r objects, using pages of size B. The reader should be careful with the notation: $\log_B r$ is itself an $O(\log_2 r)$ function only if B is considered a constant. For an I/O environment B is another problem variable. Thus $\log_B r$ represents a $\log_2 B$ speedup over $\log_2 r$ which for I/O complexity is a great improvement. Transferring a page takes about 10 msec on the fastest disk drives; in contrast, comparing two integers in main memory takes about 5 nsec. Accessing pages also uses CPU time. The CPU cost of reading a page from the disk is about 2000 instructions [GR93].

Data clustering can also substantially improve the performance of an access method. If data records that are “logically” related for a given query can also be stored physically close, then the query is optimized as fewer pages are accessed. Consider for example an access method that can cluster the data in such a way that answering the transaction pure-timeslice query takes $O(\log_B n + a/B)$ page accesses. This method is more I/O efficient than another method which solves the same query in $O(\log_B n + a)$ page accesses. Both methods use a well-paginated index (which corresponds to the logarithmic part of the query). However, in the second method each data record that belongs to the answer set may be stored on a separate page, thus requiring a much larger number of page accesses for solving the query.

Data can be clustered by the time dimension only, where data records that have been “alive” for the same time periods are collocated, or by both time and key range, or by key range only. Note that a clustering strategy that optimizes a given class of queries may not work for another query class; for example, a good clustering strategy for pure-key queries would store all the versions of a particular key in the same page; however this strategy would not work for pure-timeslice queries as the clustering objective is different.

Clustering is in general more difficult to maintain in a valid-time access method because of its dynamic behavior. The answer to a valid-time query depends on the collection of interval-objects currently contained in the access method; this collection changes as valid-time updates are applied. Even though some good clustering may have been achieved for some collection, it may not be as efficient for the next collection that is produced after a number of valid-time updates. In contrast, in transaction or bitemporal access methods the past is not changed, so an efficient clustering can be retained more easily, despite updates.

Any method which clusters data (a primary index) and uses, say, $O(\log_B n + a/B)$ pages for queries can also be used (less efficiently) as a secondary index by replacing the data records with pointers to pages containing data records, thus using $O(\log_B n + a)$ pages for queries. The distinction between methods used as primary indexes and methods used as secondary indexes is one of efficiency, not of algorithmic properties.

We use the term “primary index” only to mean that the index controls the physical placement of data. For example, a primary B^+-tree has data in the leaves. A secondary B^+-tree has only keys and references to data pages (pointers) in the leaves. Primary indexes need not be on primary keys of relations. Many of the methods do expect a unique non-time varying key for each record; we do not attempt to discuss how these methods might be modified to cluster records by non-unique keys.

3.4 Migration of Past Data to Another Location

Methods that support transaction time maintain all their past states, a property that can easily result in excessive amounts of data (even for methods that support transaction time in the most space-efficient way). In comparing such methods it is natural to introduce two other comparison considerations: (a) whether or not past data can be separated from the current data, so that the smaller collection of current data can be accessed more efficiently, and, (b) whether data is appended sequentially to the method and never changed so that Write-Once Read-Many (WORM) devices might be used.

On WORMs, one must burn into the disk an entire page with a checksum (the error rate is high, so a very long error-correcting code must be appended to each page.). Thus, once a page is written, it cannot be updated. It should be noted that since the WORM devices are themselves random access media, any access method which can use WORM devices can also be used with magnetic disks only. There are no access methods which are restricted to the use of WORMs.

3.5 Lower Bounds on I/O Complexity

We first establish a lower bound on the I/O complexity of basic transaction-time queries. The lower bound is obtained using a comparison based model in a paginated environment and applies to the transaction pure-timeslice ("*/-/point"), range-timeslice ("range/-/point") and pure-key (with time predicate, or a "point/-/range") query. Any method that solves such a query in linear (*O(n/B)*) space needs at least $\Omega(\log_B n + a/B)$ I/Os to solve it.

Since *a* corresponds to the query answer size, any method cannot do better than *O(a/B)* I/O's to provide the answer; *a/B* is the minimal number of pages where this answer can be stored. We proceed with the justification of the logarithmic part of the bound. Since the range-timeslice query is more general than the pure-timeslice query, we first show that the pure-timeslice problem is reduced to the "predecessor" problem for which a lower bound is then established [TK95]. A similar reduction can be proved for the pure-key query with time predicate.

The predecessor problem is defined as following: Given an ordered set *P* of *N* distinct items, and an item *k*, find the largest member of set *P* that is less than or equal to *k*. For the reduction of the pure-timeslice problem assume that set *P* contains integers $t_1 < t_2 < \ldots < t_N$ and consider the following real-world evolution: at time t_1 a single real-world object with name (*oid*) t_1 is created, and lives until just before time t_2, i.e., the lifespan of object t_1 is $[t_1, t_2)$. Then, real-world object t_2 is born at t_2 and lives for the interval $[t_2, t_3)$, and so on. Therefore, at any time instant t_i the state of the real-world system is a single object with name t_i. Hence the *N* integers correspond to $n = 2N$ changes in the above evolution. Consequently, finding the whole timeslice at time *t* reduces to finding the largest element in set *P* that is less or equal to *t*, i.e., the predecessor of *t* inside *P*.

We will show that in the comparison based model, and in a paginated environment the predecessor problem needs at least $\Omega(\log_B N)$ I/O 's. The assumption is that each page contains B items and there is no charge for a comparison within a page. Our argument is based on a decision tree proof. Let the first page be read and assume that the items read within that page are sorted (in any case sorting inside one page is free of I/O 's). By exploring the entire page using comparisons, we can only get B+1 different answers concerning item k. These correspond to the B+1 intervals created by the B items. No additional information can be retrieved. Then a new page is retrieved that is based on the outcome of the previous comparisons of the first page, i.e., a different page is read every B+1 outcomes. In order to determine the predecessor of k the decision tree must have N leaves (as there are N possible predecessors). As a result, the height of the tree must be $\log_B N$. Thus any algorithm solving the paginated version of the predecessor problem in the comparison model needs at least $\Omega(\log_B N)$ I/O 's.

If there was a faster than $O(\log_B n + a/B)$ method for the pure-timeslice problem using $O(n/B)$ space, then we would have invented a method that solves the above predecessor problem in less than $O(\log_B N)$ I/O 's.

Observe that the lower bound was shown for the query time of methods using linear space, irrespectively of the update processing. If the elements of set P are given in order, one after the other, $O(1)$ time (amortized) per element is needed in order to create an index on the set that would solve the predecessor problem in $O(\log_B N)$ I/O 's (more accurately, since no deletions are needed, we only need a fully paginated, multilevel index that increases on one direction). If these elements are given out of order, then $O(\log_B N)$ time is needed per insertion (B-tree index). In the transaction pure timeslice problem ("*/-/point") time is always increasing and $O(1)$ time for update processing per change is enough and clearly minimal. Thus we call a method *I/O optimal* for the transaction pure-timeslice query if it achieves $O(n/B)$ space and $O(\log_B n + a/B)$ query time using constant updating.

Similarly, for the transaction range-timeslice problem ("range/-/point"), we call a method *I/O optimal* if it achieves $O(\log_B n + a/B)$ query time, $O(n/B)$ space and $O(\log_B m)$ update processing per change. m is the number of alive objects when the update takes place. The logarithmic processing is needed because the range-timeslice problem requires ordering keys by their value. Changes arrive in time order but out of key order and there are m alive keys on the latest state among which an update has to choose.

For the transaction pure-key with time predicate the lower bound for query time is $\Omega(\log_B n + a/B)$, since the logarithmic part is needed to locate the time predicate in the past and *a/B* I/O's are required to provide the answer in the output.

The same lower bound holds for bitemporal queries since they are at least as complex as transaction queries. For example, consider the "*/point/point" query which is specified by a valid time *v* and a transaction time *t*. If the valid-time interval of each interval object extends from $-\infty$ to ∞ in the valid-time domain, finding all interval objects that at *t* where intersecting *v* reduces to finding all interval-objects in collection *C(t)* (since all of them would contain the valid instant *v*). However this is the "*/-/point" query.

Since from a query perspective a valid and a transaction-time database are both collections of intervals, a similar lower bound applies for the corresponding valid-time queries (by replacing *n* by *l*, the number of interval-objects in the collection). For example, any algorithm solving the "*/point/-" query in *O(l/B)* space needs at least $\Omega(\log_B l + a/B)$ I/O's query time.

4. Issues in Efficient Method Design for Transaction/Bitemporal Data

A common problem to all methods that support the transaction time axis is how to efficiently store large amounts of data. We first consider the transaction pure-timeslice query and show why obvious solutions are not efficient. Similarly, we discuss the transaction pure-key and range-timeslice queries. Bitemporal queries follow. The problem of separating past from current data (and the use of WORM disks) is also examined.

4.1 The Transaction Pure-Timeslice Query

There are two straightforward solutions to the transaction pure-timeslice query ("*/-/point") that will serve as two extreme cases in our comparison; we denote them as the "copy" and "log" approaches.

The "copy" approach stores a copy of the transaction database state *s(t)* (timeslice) for each transaction time that at least one change occurred. These copies are indexed by time *t*. Access to a state *s(t)* is performed by searching for time *t* on a multilevel index on the time dimension. Since changes arrive in order, this multilevel index is clearly paginated. The closest time that is less or equal to *t* is found with $O(\log_B n)$ page accesses. An additional *O(a/B)* I/O time is needed to output the copy of the state, where *a* denotes the number of "alive" objects in the accessed database state.

The major disadvantage of the "copy" approach is with the space and update processing requirements. The space used can in the worst case be proportional to $O(n^2/B)$. This happens if the evolution is mainly composed of "births" of new objects. The database state is thus enlarged continuously. If the size of the database remains relatively constant due to deletions and insertions balancing out, and if there are p records on average, the space used is $O(np/B)$.

The update processing is $O(n/B)$ per change instant in a growing database and $O(p/B)$ per change instant in a non-growing database, as a new copy of the database has to be stored at each change instant. The "copy" approach provides a minimal query time. However, since the information stored is much more than the actual changes, the space and update requirements suffer.

A variation on the copy approach stores a list of ADDRESSES of records which are "alive" at each time when at least one change occurred. The total amount of space used is smaller than if the records themselves are stored in each copy. However, the asymptotic space used is still $O(n^2/B)$ for growing databases and $O(np/B)$ for databases whose size does not increase significantly over time. This will mean most records have $O(n)$ references in the index. "n" does not have to be very large before the index is several times the size of the record collection. In addition, by thus changing from a primary to a secondary unclustered structure, $O(a)$ not $O(a/B)$ pages must be accessed to output the copy of the a alive records (after the usual $O(\log_B n)$ accesses to find the correct list).

In the remainder of this paper, we will not be considering any secondary indexes. Indexes which are described as secondary by their authors will be treated as if they were primary indexes in order to make a fair comparison. Secondary indexes never cluster data in disk pages and thus always lose out in query time. Recall that by "primary" index we mean only an index which dictates the physical location of records, not an index on "primary key." Secondary indexes can only cluster references to records, not the records themselves.

In an attempt to reduce the quadratic space and linear updating of the "copy" approach, the "log" approach stores only the changes that occur in the database timestamped by the time instant on which they occurred. The update processing is clearly reduced to $O(1)$ per change, as this history management scheme appends the sequence of inputs in a "log" without any other processing. The space is similarly reduced to the minimal $O(n/B)$. Nevertheless, this straightforward approach will increase the query time to $O(n/B)$, since in order to reconstruct a past state the whole "log" may have to be searched.

Combinations of the two straightforward approaches are possible; for example a method could keep repeated timeslices of the database state and "logs" of the changes between the stored timeslices. If repeated timeslices are stored after some bounded number of changes, this solution is equivalent to the "copy" approach, since it is equivalent to using different time units (and therefore changing only the constant in the space complexity measure). If the number of changes between repeated timeslices is not bounded, the method is equivalent to the "log" approach, as it corresponds to a series of logs. We will use the two extreme cases to characterize the performance of the examined transaction-time methods. Some of the proposed methods are equivalent to one of the two extremes. However, it is possible to combine the fast query time of the first approach with the space and update requirements of the second.

In order for a method to answer the transaction pure-timeslice ("*/-/point") query efficiently, data must at least be clustered according to its transaction time behavior. Since this query asks for all records "alive" at a given time, this clustering can be based only on the transaction time axis, i.e., records that are existing on the same time should be clustered together, independently of their key values. We call access methods that cluster by time only, as (transaction) *time-only* methods. There are methods that cluster by both time and key; we call them (transaction) *time-key* methods. They optimize queries that involve both time and key predicates, like the transaction range-timeslice query ("range/-/point"). Clustering by time only can lead to constant update processing per change; thus a good time-only method can "follow" its input changes "on-line". In contrast, clustering by time and key would need some logarithmic update as changes arrive in time order but not in key order; some appropriate placing of a change is needed based on the key it is applied on.

4.2 The Transaction Pure-Key Query

The "copy" and the "log" solutions could be used for the pure-key query ("point/-/*"). However they are both very inefficient. The "copy" method will use too much space, no matter what query it is used for. In addition, finding a key in a timeslice implies either that one uses linear search or that there is some organization on each timeslice (such as an index on the key). The "log" approach will require running from the beginning of the log to the time of the query, keeping the most recent version of the record with that key. This is still $O(n/B)$ time.

A better solution to this query is to store the history of each key separately, i.e. cluster data by key only. This creates a (transaction) *key-only* method. Since at each transaction time instant there exists at most one "alive" version of a given key, the versions of the same key can be linked together. Access to a key's (transaction-time) history can be implemented by a hashing function (which must be dynamic hashing as it has to support additions of new keys) or a balanced multiway search tree (B-tree). The hashing provides constant access (in the expected amortized sense) while the B-tree logarithmic access. Note that hashing does not guarantee against pathological worst cases while the B-tree does. Hashing cannot be used to obtain the history for a range of keys (as in the general class (III) query). After the queried key is identified, its whole history can be retrieved (forward or backward reconstruction using the list of versions).

To answer a pure-key query with time predicate ("point/-/range"), the list of versions of each key can be further organized in a separate array indexed by transaction time. Since updates are appended at the end of such an array, a simple paginated multilevel index can be implemented on each array to expedite searching. Then a query of the form: "provide the history of key *k* after (before) time *t*", is addressed by first finding *k* (using the hashing or the B-tree) and then locating the version of *k* that is closest to transaction time *t* using the multilevel index on *k*'s versions. This takes $O(\log_B n)$ time (each array can be $O(n/B)$ large).

The above straightforward data clustering by key only is efficient for class III queries but is not efficient for any of the other two classes. For example, to answer a "*/-/point" query, each key ever created in the evolution must be searched for being "alive" at the query transaction time and it takes logarithmic time for searching in each key's version history.

4.3 The Transaction Range-Timeslice Query

If records that are "logically" related for a given query can also be stored physically close, then the query is optimized as fewer pages are accessed. Therefore, to answer a "range/-/point" query efficiently, it is best to cluster by transaction time <u>and</u> key within pages. This is very similar to spatial indexing. But it has some special properties.

If the time-key space is partitioned into disjoint rectangles, one for each disk page and only one copy of each record is kept, long-lived records (records with long transaction-time intervals) would have to be collocated with many short-lived ones that cannot all fit on the same page. We cannot thus partition the space without allowing duplicate records. One therefore is reduced to either making copies (*data duplication*), allowing overlap of time-key rectangles (*data bounding*) or mapping records represented by key, (transaction) start_time and end_time, to points in 3-dimensional space (*data mapping*) and using a multidimensional search method.

Time-key spaces do not have the "density" problem of spatial indexes. Density is defined as the largest overlap of spatial objects at a point. There is only one version of each key at a given time so the time-key objects (line segments in time-key space) never overlap. This makes data duplication a more attractive option than in spatial indexing, especially if the amount of duplication can be limited as in [E86, LS90a, LM91, BGO+93, VV95].

Data bounding may force single-point queries to use backtracking as there is not a unique path to a given time-key point. In general, for the data-bounding approach, temporal indexing has worse problems than spatial indexing because long-lived records are likely to be common. In a data-bounding structure, such a record will be stored in a page with a long time-span and some key range. Every timeslice query in that timespan must access that page even though the long-lived record may be the only one alive at the search time (the other records in the page are alive at another part of the timespan). The R-tree based-methods [S87, KS89, KS91] use data bounding.

The third possibility, data mapping, will map a record to three (or more) coordinates: its transaction start_time, end_time, and key(s) and then use a multiattribute point index. Here records with long transaction-time intervals would be clustered with other records with long intervals as their start and end times would be close. Records with short transaction-time intervals would be clustered with other records with short intervals if they were alive at nearby times. This would be efficient for most queries as the long-lived records would be the answers to many queries. The pages with short-lived records would effectively partition the answers to different queries; most such pages would not be touched for a given timeslice query. However, there are special problems because many records may still be current and have growing lifetimes (i.e., transaction-time intervals extending to *now*). This approach is further discussed in the end of section 5.1.3.

Naturally, the most efficient methods for the transaction range-timeslice query are the ones that combine the time and key dimensions. In contrast, by using a (transaction) time-only method, the whole timeslice for the given transaction time is first reconstructed and then the records with keys outside the given range are eliminated. This is clearly inefficient, especially if the requested range is a small part of the whole timeslice.

4.5 Bitemporal Queries

An obvious approach would be to index bitemporal objects on a single time axis (transaction or valid time) and use a single time access method. For example, if a transaction access method is utilized, a bitemporal “*/point/point” query is answered in two steps. First all bitemporal objects existing at transaction time *t* are found. Then the valid time interval of each such object is checked whether it includes valid time *v*. This approach is inefficient because very few of the accessed objects may actually satisfy the valid-time predicate.

If both axes are utilized, an obvious approach is an extended combination of the "copy" and "log" solutions. This approach stores copies of the collections *C(t)* (Fig. 3) at given transaction-time instants and a log of changes between copies. Together with each collection *C(t)*, an access method (for example an R-tree [G84]) that indexes the objects of this *C(t)* is also stored. Conceptually it is like storing snapshots of R-trees and changes between them. While each R-tree enables efficient searching on a stored collection *C(t)*, the approach is clearly inefficient because the space or query time increases dramatically depending on the frequency of snapshots.

The *data bounding* and *data mapping* approaches can also be used in a bitemporal environment. However, the added (valid-time) dimension provides an extra reason for inefficiency. For example, the bounding rectangle of a bitemporal object consists of two intervals (Figure 4; taken from [KTF95a]). A "*/point/point" query is translated into finding all rectangles that include the query point (t_i, v_j). An R-tree [G84] could be used to manage these rectangles. However, the special characteristics of transaction time (many rectangles may extend up to *now*) and the inclusion of the valid-time dimension, increase the possibility of extensive overlapping which in turn reduces the R-tree query efficiency [KTF95b].

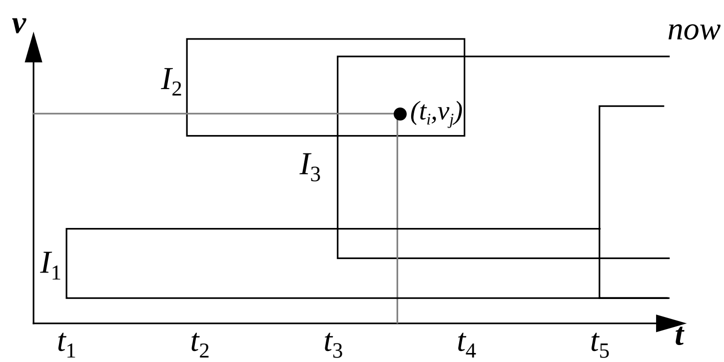


Figure 4: The bounding-rectangle approach for bitemporal queries (the key dimension is not shown). The evolution of Fig.3 is depicted, as of (transaction) time $t > t_5$. The modification of interval I_1 at t_5 ends the initial rectangle for I_1 and inserts a new rectangle from t_5 to *now*.

4.6 Separating Past from Current Data and the use of WORM disks

In transaction or bitemporal databases, it is usually the case that access to current data is more frequent than to past data (in the transaction-time sense). In addition, since the bulk of data in these databases is due to the historical part, it would be advantageous to use a higher capacity, but slower access medium for the past data, such as optical disks. First, the method should provide for natural separation between current and past data. There are two ways to achieve this separation: (a) With the “manual” approach, a process will vacuum all records that are “dead” (in the transaction-time sense) when the process is invoked; this vacuuming process can be invoked at any time. (b) With the “automated” approach, where such “dead” records are migrated to the optical disk due to a direct cause from the evolution process (for example during an update). The total I/O involved is likely to be smaller than in a manual method, since it is piggybacked on I/O which is necessary for index maintenance in any case (such as the splitting of a full node).

Even though Write-Many Read-Many optical disks are available, the WORM optical disks are still the main choice for storing large amounts of archival data; they are less expensive, have larger capacities and usually have faster write transfer times. Since the contents of WORM disk blocks cannot be changed after their initial writing (due to an added error-correcting code) data that is to be appended on a WORM disk should not be allowed to change in the future. Since on the transaction axis the past is not changed, past data can be written on the WORM disk.

We emphasize again that methods which can be used on WORM disks are not “WORM methods”-- they can also be used on magnetic disks. Thus the question of separation of past and current records can be considered regardless of the availability of WORM disks.

5. Method Classification and Comparison

This section provides a concise description of the methods we examine. Since it was practically impossible to run simulations for all methods on the same collections of data and queries, our analysis is based on worst case performance. Various access method proposals provide a performance analysis that may have strong assumptions about the input data (uniform distribution of data points, etc.) and may very well be that under those constraints the proposed method works quite well. Our purpose however was to categorize the methods without any assumption on the input data or the frequency of queries asked. Obviously the worst case analysis may penalize a method for some very unlikely scenarios; to distinguish against likely worst cases we call such scenarios *pathological* worst cases. We shall also point out some features which may affect average case behavior without necessarily affecting worst-case behavior.

We first describe transaction-time access methods. These methods are further classified to key-only, time-only and time-key, based on the way data is clustered. In the key-only methods we study the *Reverse Chaining*, *Accession Lists*, *Time Sequence Arrays* and *C-lists*. Among the time-only we examine: the *Append-only Tree*, the *Time-Index* and its variations (*Monotonic B-Tree, Time-Index+*), the *Differential File* approach, the *Checkpoint Index*, the *Arhivable Time Index*, the *Snapshot Index* and the *Windows Method*. In the time-key category we present: the POSTGRES Storage System and the use of *Composite Indexes*, the *Segment-R Tree*, the *Write-Once B-Tree*, the *Time-Split B-Tree*, the *Persistent B-Tree*, the *Multiversion B-Tree*, the *Multiversion Access Structure* and the *Overlapping B-Tree*. A *comparison table* (Table B) is included in the end of the section with a summary of each method's worst case performance. We then proceed with the valid-time access methods where we discuss the *Metablock Tree*, the *External Segment Tree*, the *External Interval Tree* and the *MAP21* methods. The bitemporal category describes the *M-IVTT*, the *Bitemporal Interval Tree* and the *Bitemporal R-Tree*.

5.1 Transaction-time Methods

In this category we have included methods which assume that changes arrive in increasing time order, a characteristic of transaction time. This property greatly affects the update processing of the method. If "out of order" changes (a characteristic of valid-time) were to be supported, the updating cost becomes much higher (practically prohibitive).

5.1.1 Key-only Methods

The basic characteristic of transaction key-only approaches is the organization of evolving data by key (surrogate), i.e., all versions that a given key assumes are "clustered" together logically or physically. Such organization makes these methods more efficient for transaction pure-key queries. In addition, the approaches considered here correspond to the earliest solutions proposed for time evolving data.

Reverse chaining was introduced in [B82] and further developed in [LDE+84]. Under this approach, previous versions of a given key are linked together in reverse chronological order. The idea of keeping separate stores for current and past data was also introduced. Current data is assumed to be queried more often, so by separating it from past data, the size of the search structure is decreased and queries for current data become faster.

Each version of a key is represented by a tuple (which includes the key, attribute value and a lifespan interval) augmented with a pointer field that points to the previous version (if any) of this key. When a key is first inserted into a relation, its corresponding tuple is put into the *current store* with its previous-version pointer being *null*. When the attribute value of this key is changed, the version existing in the current store is moved to the *past store* with the new tuple replacing it in the current store. The previous-version pointer of the new tuple points to the location of the previous version in the past store. Hence a chain of past versions is created out of each current key. Tuples are stored in the past store without necessarily being clustered by key.

Current keys are indexed by a regular B^+-tree ("front" B^+-tree). The chain of past versions of a current key is accessed by following previous-version pointers starting from the current key. If a current key is deleted, it is removed from the B^+-tree and is inserted in a second B^+-tree ("back" B^+-tree) which indexes the latest version of keys that are not current. The past version chain of the deleted key is still accessed from its latest version stored in the "back" B^+-tree. If a key is "reborn" it is reinserted in the "front" B^+-tree. Subsequent modifications of this current key create a new chain of past versions. It is thus possible to have two chains of past versions, one starting from its current version and one from a past version, for the same key. Hence queries about the past are directed to both B^+-trees. If the key is later deleted again, its new chain of past versions is attached to its previous chain by appropriately updating the latest version stored in the "back" B^+-tree.

Clearly this approach uses $O(n/B)$ space, where n denotes the number of changes and B is the page size. The number of changes corresponds to the number of versions for all keys ever created. When a change occurs (such as a new version of *key* or the deletion of *key*) the "front" B^+-tree (current store) has first to be searched to locate the current version of *key*. If it is a deletion, the "back" B^+-tree is also searched to locate the latest version of *key*, if any. Hence the update processing of this method is $O(\log_B n)$ since the number of different keys can be similar to the number of changes.

To find all previous versions of a given *key*, the "front" B^+-tree is first searched for the latest version of *key*; if *key* is in the current store, its pointer will provide access to recent past versions of *key*. Since version lists are in reverse chronological order, one has to follow such a list until a version number (transaction timestamp) that is less or equal to the query timestamp is found. The "back" B^+-tree is then searched for older past versions. If a denotes all past versions of *key* the query time is $O(\log_B n+a)$ since versions of a given key could in the worst case be stored in different pages. This can be improved if *cellular chaining, clustering* or *stacking* is used [AS88]. If each collection of versions for a given key is clustered in a set of pages, but versions of distinct keys are never on the same page, query time would be $O(\log_B n+a/B)$ but space utilization would be $O(n)$ pages (not $O(n/B)$) as the versions of a key may not be enough to justify the use of a full page.

Reverse chaining can be further improved by the introduction of *accession lists* [AS88]. An accession list clusters together all version numbers (timestamps) of a given key. Each timestamp is associated with a pointer to the accompanying tuple which is stored in the past store (or to a cluster of tuples). Thus instead of searching a reverse chain until a given timestamp, one can search an index of the chain's timestamps. As timestamps are stored in chronological order on an accession list, finding the appropriate version of a given key would take $O(\log_B n + \log_B a)$. The space and update processing remain as before.

While the above structure can be efficient for a transaction pure-key query, answering pure- or range-timeslice queries is problematic. For example to answer a "*/-/point" query that is satisfied only by some keys, one has to search the accession lists of all keys ever created.

Another early approach proposed the use of *Time Sequence Arrays (TSA)* [SK86]. Conceptually, a TSA is a two dimensional array with a row for each key ever created; each column represents a time instant. The *(x,y)* entry stores the value of key *x* at time *y*. Static (the data set has been fully collected) and dynamic (the data set is continuously growing--as in a transaction-time environment) are examined. If this structure is implemented as a two-dimensional array the query time is minimal (just access the appropriate array entry), but the update processing and space are prohibitive ($O(n)$ and $O(n^2)$ respectively). One could implement each row as an array keeping only values when there was a change; this is conceptually the same solution as the reverse chaining with accession lists. A solution based on a multidimensional partitioning scheme is proposed in [RS87], but the underlying assumption is that the whole temporal evolution is known in advance before the partitioning scheme is implemented.

The theoretically optimal solution to the transaction pure-key query with time predicate is provided by the *C-lists* of [VV95]. C-lists are similar to accession lists in that they cluster together the versions of a given key. There are two main differences. First, access to each C-list is provided through another method, the Multiversion Access Structure [VV95] (in short "MVAS"; the MVAS is discussed later with the time-key methods). Second, C-list maintenance is more complicated: splitting/ merging of C-list pages is guided by the page splitting/merging of the MVAS (for details we refer to [VV95]). If there are *m* "alive" keys in the structure, updating takes $O(\log_B m)$. The history of key *k* before time *t*, is found in $O(\log_B n + a/B)$ I/Os, which is optimal. An advantage of C-lists is that they can be combined with the MVAS structure to create a method that answers optimally both the range-timeslice and pure-key with time predicate queries. However, the additional complexity (an extra B+-tree is needed together with double pointers between the C-lists and the MVAS) limits its practicality.

5.1.2 Time-only Methods

Most time-only methods timestamp changes (additions, deletions etc.) by the transaction time they occurred and append them in some form of a "history log". Since no clustering of data according to keys is made, such methods optimize "*/-/point" or "*/-/range" queries. Because changes arrive in chronological order, ideally a time-only method can provide constant update processing (as the change is simply appended at the end of the "history log"); this advantage is important in applications where changes are frequent and the database has to "follow" these changes in an *on-line* fashion. For efficient query time, most methods use some index on the top of the "history log" that indexes the (transaction) timestamps of the changes. Because of the time-ordered changes, the cost of maintaining this (paginated) index on the transaction time-axis is minimal, amortized $O(1)$ per change.

While organizing data only by its time behavior provides for very fast updating, it is not efficient for answering transaction range-timeslice queries. In order to use time-only methods for such queries, one suggestion [EWK93, GS93] is to employ a separate *key index*, whose leaves point to predefined key "regions". A key region could be a single key or a collection of keys (either a sub-range of the key space or a relation). The history of each "region" is organized separately, using an individual time-only access method (such as the Time Index or the Append-Only Tree). The *key index* will direct a change of a given key to update the method that keeps the history of the key's region. However, after the region is found, the placement of this key in the region's access method, is based only on the key's time behavior (and not any more on the key itself).

To answer transaction range-timeslice queries one has to search the history of each region that belongs to the query range. The range-timeslice is thus constructed by creating the individual timeslices for every region in the query range. If R is the number of regions, the key index would add $O(R/B)$ space and $O(\log_B R)$ update processing to the performance of the individual historical access methods. The query time for the combination of the key index and the time-only access methods would be $O(Mf(n_i, t, a_i))$, where M is the number of regions that fall in the given query range, and $f(n_i, t, a_i)$ is the time needed in each individual region r_i ($i = 1,..., m$) to perform a timeslice query for time t (n_i and a_i correspond to the total number of changes in r_i and the number of "alive" objects from r_i at time t, respectively). For example, if the Time-Index [EWK90] is used as the access method in each individual region, then: $f(n_i, t, a_i) = O(\log_B n_i + a_i/B)$.

There are three drawbacks with this approach: (1) if the query key range is a small subset of a given region, the whole region's timeslice is reconstructed, even if most of its objects may not belong to the query range and thus do not contribute to the answer, (2) if the query key range contains many regions, all these regions have to be searched, even if they may contribute no "alive" objects at the transaction time of interest *t*, and, (3) for every region examined, at best, a logarithmic search is performed in order to locate *t* among the changes recorded in the region. To put this in perspective, imagine replacing a multi-attribute spatial search structure with a number of collections of records from predefined key ranges in one attribute and then organizing each key range by some other attribute.

To answer general pure-key queries of the form: "find the salary history of employee named *k*", an index on the key space can be utilized. This index keeps the latest version of a key while key versions are linked together. Since the key space is separate from the time space, such an index is easily updated. In some methods this index has the form of a B^+-Tree and is also facilitated for the transaction range-timeslice queries (like the Surrogate Superindex used in the AP-Tree [GS93] and the Archivable Time Index [VV94]) or it has the form of a hashing function as in the Snapshot Index [TK95]. A general method is to link records to any one copy of the most recent distinct past version of the record. We continue with the presentation of various time-only methods.

5.1.2.1 The Append-Only Tree

The Append-Only Tree (*AP-Tree*) is a multi-way search tree that is a hybrid of an *ISAM* Index and a B^+-Tree. It was proposed as a method to optimize event-joins [SG89, GS93]. Here we examine it as an access method for the query classes of section 3.1. Each tuple is associated with a (*start_time*, *end_time*) interval. The basic method indexes the start_times of tuples. Each leaf node has entries of the form: (*t*, *b)* where *t* is a time instant and *b* is a pointer to a bucket that contains all tuples with start_time greater than the time recorded in the previous entry (if any) and less than or equal to *t*. Each non-leaf node indexes nodes at the next level (Figure 5).

In the AP-Tree insertions of new tuples arrive in increasing start_time order. Based on this, we consider it as a transaction-time method. It is also assumed that the end_times of tuples are known when a tuple is inserted in the access method. In that case the update processing is $O(1)$ since the tuple is inserted ("appended") on the rightmost leaf of the tree. (This is somewhat similar to the procedure used in most commercial systems for loading a sorted file to a multilevel index [S88], except that insertions are now successive instead of batched). If end_times are not known at

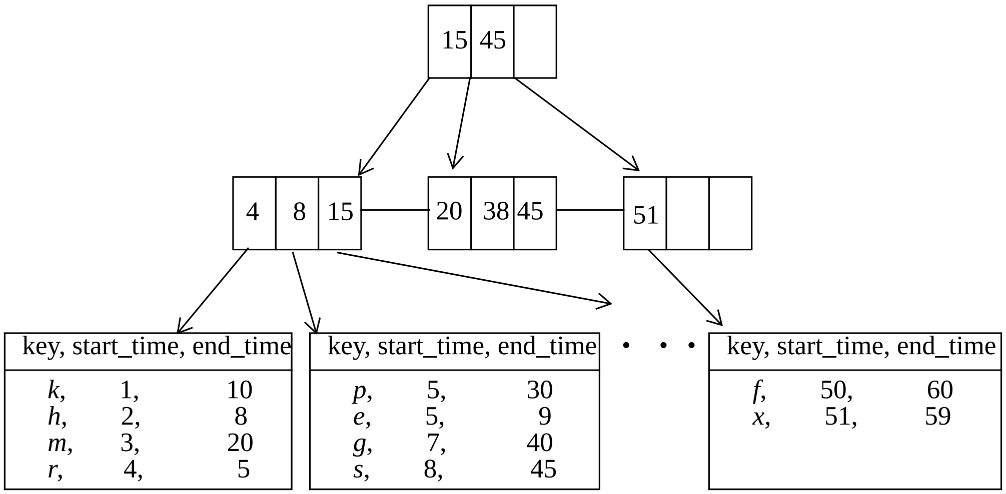


Figure 5: The Append-Only Tree. Leaves include only the *start_time* fields of intervals. Each leaf points to file pages, with records ordered according to the *start_time* field. New records are added only at the rightmost leaf of the tree. It is assumed that both endpoints are known for the intervals in this figure.

insertion but are updated later (as in a transaction-time environment), the index has to be searched for the record that is updated. If the start_time of the updated record is given in the input then this search is $O(\log_B n)$. Otherwise, one could use a hashing function that stores only the alive objects and for each such object it points in its position in the AP-Tree (this however is not discussed in the original paper).

To answer a transaction pure-timeslice query for time *t*, the AP-Tree is first searched for the leaf that contains *t*. All intervals on the "right" of this leaf have start_times that are larger than *t* and thus should not be searched further. However, all intervals on the left of this leaf (i.e. the data file from the beginning until *t*) have to be checked for "containing" *t*. Such a search can be as large as $O(n/B)$ since the number of intervals in the tree is proportional to the number of changes in the evolution. Of course, if we assume that the queries are randomly distributed over the entire transaction-time range, on the average half of the leaf nodes must be searched. The space is $O(n/B)$.

For answering transaction pure-key and range-timeslice queries the nested *ST-Tree* has been proposed [GS93]. This method facilitates a separate B^+-Tree index (called *Surrogate Superindex*) on the keys (surrogates) ever inserted in the database. A leaf node of such a tree contains entries of the form: (key, p_1, p_2) where p_1 is a pointer to an AP-Tree (called the *Time Subindex*) that organizes the evolution of the particular key and p_2 is a pointer to the latest version of a key. This approach solves the problem of updating intervals by key (just search the Surrogate Superindex for the key of the interval; then this key's Time Subindex will provide the latest version of this interval, i.e., the version to be updated). The *ST-Tree* approach is conceptually equivalent to *reverse chaining* with an index on each *accession list* (however due to its relation to the AP-Tree we included it in the time-only methods).

The update processing is now $O(\log_B S)$ where S denotes the total number of keys (surrogates) ever created (S is itself $O(n)$). Note that there may be key histories with just one record. For the space to remain $O(n/B)$ unused page portions should be shared by other key histories. This implies that the versions of a given key may reside in separate pages. Answering a pure key query then takes $O(\log_B S + a)$ I/Os. The given key can be found with a logarithmic search on the Surrogate Superindex and then its a versions are accessed but at worst each version may reside in a distinct page. For a transaction range-timeslice query whose range contains K keys (alive or not at t) the query time is $O(K\log_B n)$ as each key in the range has to be searched. When the range is the whole key space, that is, to answer a transaction pure-timeslice query for time t, one has to perform a logarithmic search on the Time Subindex of each key ever created. This takes $O(S\log_B n)$ time.

The basic AP-Tree does not separate past from current data so transferring to a write-once optical disk may be problematic. One could start transferring data to the write-once medium in start_time order, but this could also transfer long-lived tuples that are still current (alive) and may be later updated. The ST-Tree does not have this problem as data are clustered by key; the history of each key represents past data that can be transferred to an optical medium.

If the AP-Tree is used in a valid-time environment, interval insertions, deletions or updates may happen anywhere in the valid-time domain. This implies that the index would not be as compact as in the transaction domain where changes arrive in order, but it would behave as a B-Tree. If for each update, only the key associated with the updated interval is provided, the whole index may have to be searched. If the start_time of the updated interval is given, a logarithmic search is needed. Since the l valid intervals are sorted by start_time a “*/point/-” query takes $O(l/B)$ I/O’s. For “range/point/-” queries, the ST-Tree must be combined with a B-Tree as its Time Subindex. Updates are logarithmic (by traversing the Surrogate Superindex and the Time Subindex). A valid range timeslice query whose range contains K keys takes $O(K\log_B l)$ I/O’s since every key in the query range must be searched for being alive at the valid query time.

5.1.2.2 The Time Index

The Time Index, proposed in [EWK90, EKW91], is a B^+-Tree based access method on the time axis. In the original paper the method was proposed for storing valid-times. It makes however the assumption that changes arrive in increasing time order and that physical deletions rarely occur.

Since these are basic characteristics of the transaction-time dimension we consider the Time-Index in the transaction-time category. There is a B^+-Tree that indexes a linearly ordered set of time points, where a time point (referred also as *indexing* point in [EWK90]) is either the time instant where a new version is created or the next time instant after a version is deleted. Thus a time point corresponds to the time instant of a change (for deletions it is the next time instant after the deletion). Each entry of a leaf node of the Time Index is of the form: *(t, b)* where *t* is a time point and *b* is a pointer to a bucket. The pointer of a leaf's first entry points to a bucket that holds all records that are "alive" (i.e., a snapshot) at this time point; the rest of the leaf entries point to buckets that hold incremental changes (Figure 6). As a result, the Time Index does not need to know in advance the end_time of an object (which is an advantage over the AP-Tree).

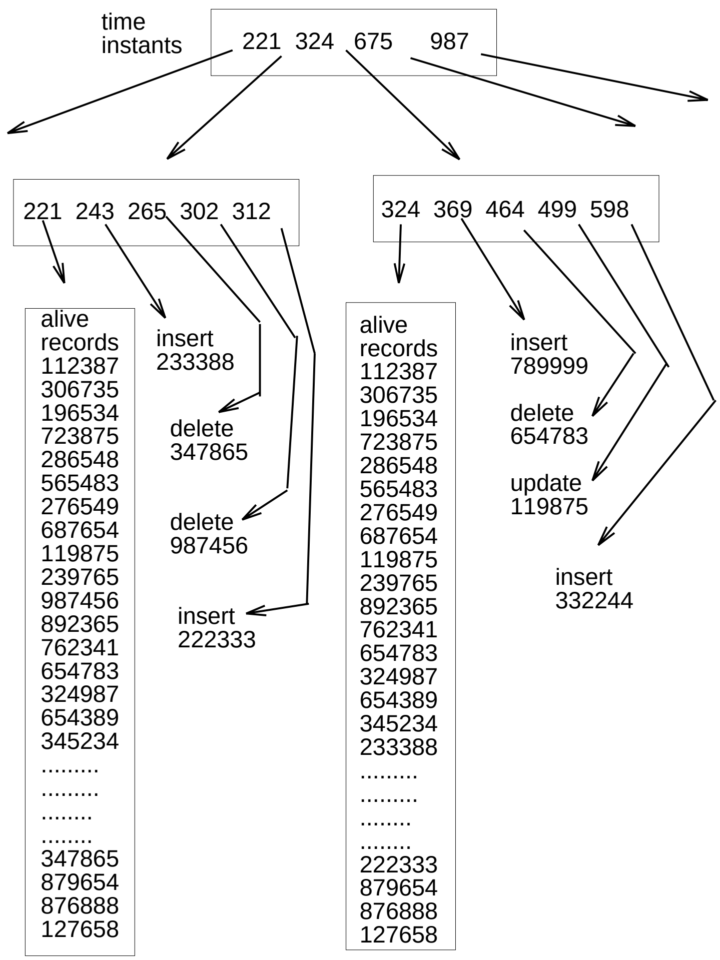


Figure 6: The Time Index. Each first leaf entry holds a full timeslice while the next entries keep incremental changes.

The Time Index was originally proposed as a secondary index. We shall treat it as a primary index here in order to make a fair comparison to other methods, as explained in section 4.1. This makes the search estimates competitive with the other methods without changing the worst case asymptotic space and update formulas.

Since in a transaction environment changes occur in increasing time order, new nodes are always added on the rightmost leaf of the index. This can produce a more compact index than the B+ tree used in the original paper, called the *Monotonic B⁺-Tree* [EWK93]. (The Monotonic B⁺-Tree insertion algorithm is similar to that of the AP-tree).

To answer a transaction pure-timeslice query for some time *t*, one has to search the Time Index for *t*; this will lead to a leaf node that "contains" *t*. The past state is reconstructed by accessing all the buckets of entries of this leaf node that contain timestamps that are less or equal to *t*. If we assume that the number of changes that can occur at each time instant is bounded (by some constant) the query time of the Time Index is $O(\log_B n + a/B)$. After the appropriate leaf node is found in logarithmic time, the answer *a* is reconstructed by reading leaf buckets. The update processing and space can be as large as $O(n/B)$ and $O(n^2/B)$ respectively. Therefore, this method is conceptually equivalent to the "copy" approach of section 4.1 (the only difference is that copies are now made after a constant number of changes).

Answering a transaction range-timeslice query with the Time-Index requires reconstructing the whole timeslice for the time of interest and then selecting only the tuples in the given range. To answer range-timeslice queries more efficiently, the Two-Level Attribute/Time Index (using predefined key regions) has been proposed [EWK90]. Assuming that there are *R* predefined key regions (and *R* is smaller than *n*), the update processing and space remain $O(n/B)$ and $O(n^2/B)$ respectively, since most of the changes can happen to a single region. Answering a "*/-/point" query would mean creating the timeslices for all *R* ranges, even if a range does not contribute to the answer. Thus the pure-timeslice query time is proportional to $\sum_{i=1}^{R} \log_B n_i + a_i / B$, where n_i and a_i correspond to the total number of changes in individual region r_i and the number of "alive" objects from r_i at time *t*, respectively. This can be as high as $O(R\log_B n + a)$, since each region can contribute a single tuple to the answer. Similarly, for "range/-/point" queries the query time becomes $O(M\log_B n + a)$ where *M* is the number of regions that fall in the given query range (assuming that the query range contains a number of regions, otherwise a whole region timeslice has to be created).

Pure-key queries are not supported as record versions of the same object are not linked (for example, to answer a query of the form: find all past versions of a given key, one may have to search the whole history of the range where this key belongs).

In [EWK93], it is suggested to move record versions to optical disk when their end times change to a time before *now*. This is under the assumption that the Time Index is being used as a secondary index and that each record version is only located in one place. The leaf buckets therefore contain lists of addresses of record versions.

In order to move full pages of data to the optical disk, a buffer is used in the magnetic disk to collect records as their end times are changed. An optical disk page is reserved for the contents of each buffer page. When a record version is placed in a buffer page, all pointers to it in the Time Index must be changed to refer to its new page in the optical disk. This can require $O(n/B)$ update processing as a record version pointer can be contained in $O(n/B)$ leaves of the Time Index. A method for finding pointers for particular record versions within the lists of addresses in the leaf's first entry, in order to update them, is not given.

Index leaf pages can be migrated to the optical disk only when all their pointers are references to record versions which are on the optical disk or in the magnetic disk buffer used to transfer record versions to optical disk. Since each index leaf page contains the pointers to all record versions which were alive at the time the index page was created, it is likely that many index pages may not qualify for moving to optical disk, because they contain long-lived records.

It is suggested in [EWK93] that long-lived records which are inhibiting movement of index pages also be kept in a magnetic buffer and assigned an optical address so that the index leaf page can be moved. When all the children of an internal index page have been moved to the optical disk, an internal index page can also be moved. However, the number of long-lived record versions can also be $O(n)$. Thus the number of empty optical pages waiting for long-lived object versions to die and having mirror buffers on magnetic disk is $O(n/B)$.

In an attempt to overcome the high storage and update requirements, the Time Index$^+$ [KKEW94] has been proposed. There are two new structures in the Time Index$^+$: the *SCS* and the *SCI* buckets. In the original Time Index, a timeslice is stored for the first timestamp entry of each leaf node. Since sibling leaf nodes may share much of this timeslice, in the Time Index$^+$ odd-even pairs of sibling nodes store their common parts of the timeslice in a shared *SCS* bucket. Even though the *SCS* technique would in practice save considerable space (about half of what was used before), the asymptotic behavior remains the same as of the original Time Index.

Common intervals that span a number of leaf nodes are stored together on some parent index node (similarly to the Segment Tree data structure [B77]). Each index node in the Time Index$^+$ is associated with a range, i.e., the range of time instants covered by its subtree. A time interval I is stored in the highest internal node v such that I covers v's range and does not cover the range of v's parent. All such intervals are kept in the *SCI* bucket of an index node.

By keeping the intervals in this way the quadratic space is dramatically reduced. Observe that now an interval may be stored in at most logarithmic many internal nodes (this is due to the segment tree property [M84]). This implies that the space consumption of the Time Index$^+$ is reduced to $O((n/B) \log_B n)$ space. The authors mention in the paper that in practice there is no need to associate *SCI* buckets to more than two-levels of index nodes. However, if no *SCI* buckets are used in higher levels, the asymptotic behavior would remain similar to the original Time Index.

In addition, it is not clear how the updates are performed when *SCI* buckets are used. In order to find the actual *SCI*s where a given interval will be stored, both endpoints of the interval should be known. Otherwise, if an interval is initially inserted as *(t, now)* it has to be found and updated when at a later time the right endpoint becomes known. This implies that some search structure is needed in each *SCI* which would of course affect the update behavior of the whole structure. Finally, the query time bound remains the same for the Time Index$^+$ as for the original Time Index.

If the original Time-Index (using the regular B+ tree) is used in a valid-time environment, physical object deletions anywhere in the (valid) time domain should be supported. However, a deleted object should removed from all the stored (valid) snapshots. If the deleted object has a long valid-time interval, the whole structure may have to be updated, making such deletions very costly. Similarly, objects can be added anywhere in the valid domain; this implies that all affected stored snapshots have to be updated.

5.1.2.3 The Differential File Approach

While the Differential File Approach [JMR91, JMRS92] does not propose the creation of a new index, we discuss it since it involves an interesting implementation of a database system based on transaction time. In practice, an index can be implemented on top of the differential file approach, however here we assume no such index exists. Changes that occur for a base relation r are stored incrementally and timestamped on the relation's log; this log is itself considered a special relation, called a *backlog*. In addition to the attributes of the base relation, each entry of the backlog contains a triplet: (*time, key, op*). Here *time* corresponds to the (commit) time of the transaction that updated the database about a change that was applied on the base relation tuple with *key* surrogate; *op* corresponds to the kind of change that was applied on this tuple (addition, deletion, or modification operation).

As a consequence of the use of timestamps a base relation is a function of time; thus $r(t)$ is a timeslice of the base relation at time t. A timeslice of a base relation can be stored or computed. Storing a timeslice can be implemented either as a *cache* (where pointers to the appropriate backlog entries are used) or as materialized data (where the actual tuples of the timeslice are kept). Using the cache avoids storing probably long attributes, however some time is needed to reconstruct the full timeslice (Figure 7).

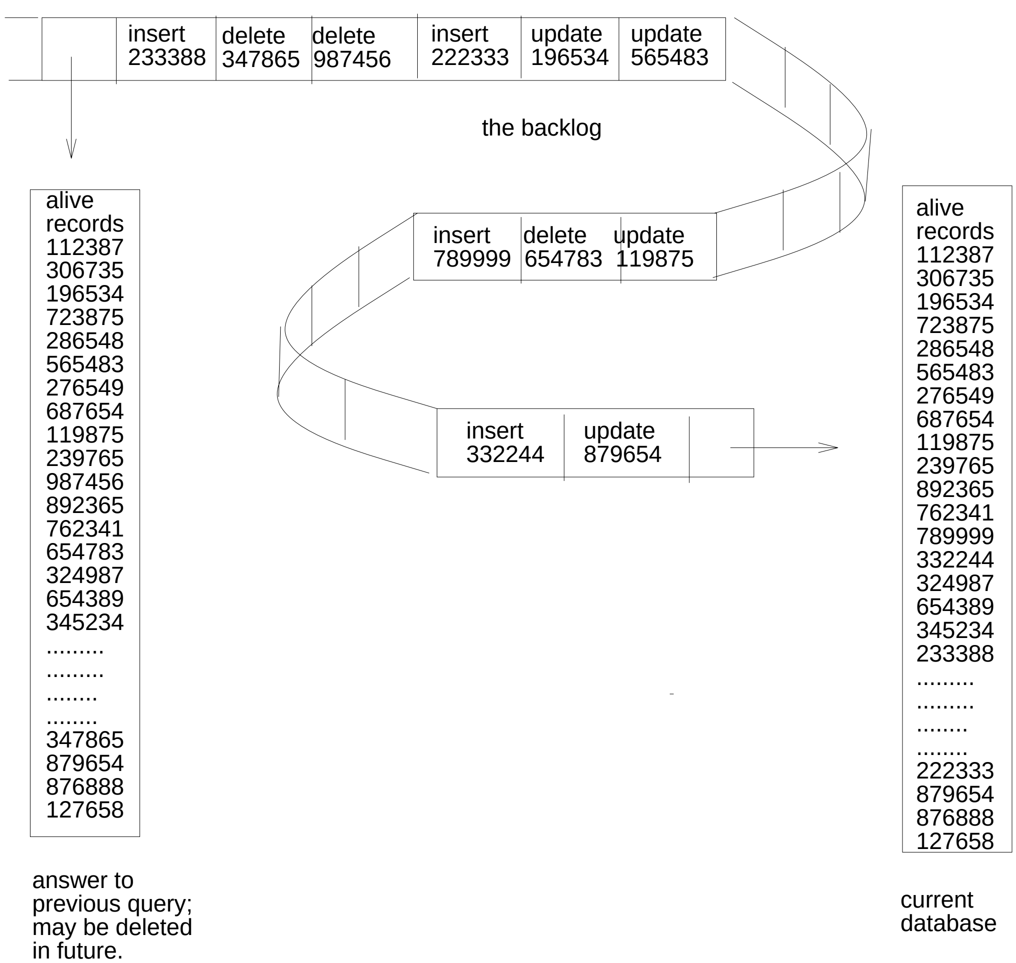


Figure 7: The Differential File approach.

A timeslice can be *fixed* (for example: $r(t_1)$) or *time-dependent* ($r(now\text{-}t_1)$). Time-dependent stored base relations have to be updated; this is done eagerly (changes are directly updating such relations) or lazy (when the relation is requested, the backlog is used to bring it up in the current state). An eager current (*r(now)*) timeslice is like a snapshot relation, that is, a collection of all records that are current.

A time-dependent base relation can also be computed from a previous stored timeslice and the set of changes that occurred in between. These changes correspond to a *differential* file (instead of searching the whole backlog). Differential files are also stored as relations.

For answering "*/-/point" queries, this approach can be conceptually equivalent to the "log" or the "copy" methods, depending on how often timeslices are stored. Consider for example a single base relation r with backlog b_r: if timeslices are infrequent or the distance (number of changes) between timeslices is not fixed, the method is equivalent to the "log" approach where b_r is the history log. The space is $O(n/B)$ and the update processing is constant (amortized) per change, but the reconstruction can also be $O(n/B)$. Conversely, if timeslices are kept with fixed distance, the method would behave similarly to the "copy" approach.

In order to address "range/-/point" queries one has to produce the timeslice of the base relation and then check all of the tuples of this timeslice for being in the query range. Similarly, if the value of a given key is requested as of some time, the whole relation must first be reconstructed as of that time. The history (previous versions) of a key is not kept explicitly as versions of a given key are not connected together.

5.1.2.4 The Checkpoint Index

The Checkpoint Index was originally proposed for the implementation of various temporal operators (temporal joins, parallel temporal joins, snapshot/interval operators, etc.) [LM92a, LM92b, LM93]. Here we take the liberty to consider its behavior if it was used as an access method for transaction-time queries. Timeslices (called *checkpoints*) are periodically taken from the state of an evolving relation. If the query operator is a join, checkpoints from two relations are taken. Partial relation checkpoints based on some key predicate have also been proposed. For simplicity we concentrate on checkpointing a single relation.

The Checkpoint Index assumes that the object intervals are ordered by their start_time. This is a property of the transaction-time environment (Fig. 1). A *stream processor* follows the evolution as time proceeds. When a checkpoint is made at some (checkpoint) instant t, the objects alive at t are stored in the checkpoint. A separate structure, called the *data stream pointer* (DSP), points to the first object born after *t*. Conceptually the DSP provides access to an ordered (by interval start_time) list of objects born between checkpoints. The DSP is needed since some of these objects may end before the next checkpoint so they would not be recorded otherwise. The checkpoint time instants are indexed through a B+-Tree like structure (Figure 8).

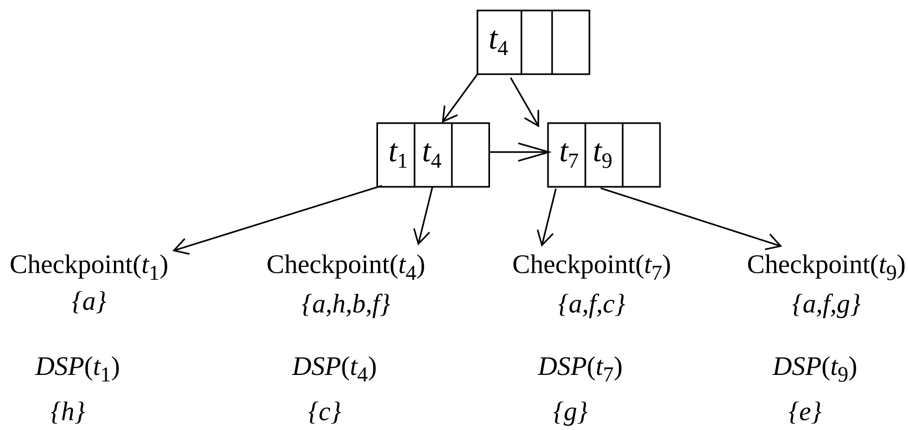


Figure 8: The Checkpoint Index. The evolution at Fig. 1 is assumed.

The performance of the Checkpoint Index for pure-timeslice queries depends on how often checkpoints are taken. On the one extreme, if very few checkpoints are taken the space remains linear *O(n/B)*. Conversely, if checkpoints are kept within fixed distance, the method would behave similarly to the “copy” approach. In general, the DSP pointer may be “reset” backwards in time to reduce the size of a checkpoint (which is an optimization issue).

When an object is deleted, its record has to be found so as to update the end_time. The original presentation of the Checkpoint Index implicitly assumes that the object end_times are known (since the whole evolution or *stream* is known). However a hashing function on the alive objects can be used to solve this problem (as with the AP-Tree). The Checkpoint Index resembles the Differential File and the Time Index in that all keep various timeslices. However instead of simply storing the changes between timeslices, the Checkpoint Index keeps the DSP pointers to actual object records. Hence in the Checkpoint Index, an update to an interval end_time cannot simply be added at the end of a log but it has to update the corresponding object’s record.

To address range-timeslice queries with the Checkpoint Index, the timeslice of the base relation is first produced and then all of the tuples of this timeslice are checked for being in the query range. The history (previous versions) of a given key is not kept explicitly as versions of the same key are not connected together.

The Checkpoint Index could use a transfer policy to an optical medium similar to the one of the Time Index.

5.1.2.5 The Archivable Time Index

The Archivable Time Index [VV94] does not directly index actual transaction time instants but version numbers. The transaction time instant of the first change takes version number 0 and successive changes are mapped to consecutive version numbers. An interval is represented by the version numbers corresponding to its start and end times. A special structure is needed to transform versions to timestamps and vice versa. For the rest we use the terms time instant and version number synonymously.

Let T denote the current time. The method partitions records to *current* and *past*. For the current records (those with unknown end_time) a conventional B+-tree structure is used to index the start_time of their transaction intervals. For the past (records whose end_time is less or equal to T), a more complex structure, the PVAS, is used. Conceptually, the PVAS can be viewed as a logical binary tree of size 2^a ($T \leq 2^a$). Each node in the tree represents a segment of the transaction time space. At T only some of the nodes of the tree would have been created; new nodes are dynamically added on the right path of the tree, as time increases. A node denoted by segment $[i,j]$ where $i < j$, has *span* $(j\text{-}i)$. The root is denoted as $[0,2^a]$. The left child of node $[i,j]$ is node $[i,(i+j)/2]$ and its right child is node $[(i+j)/2, j]$. Hence the span of a node is the sum of the span of its two children. The span of a leaf node is two. Figure 9 (taken from [VV94]) shows an example of the PVAS tree at $T = 55$ with $a = 6$. Node segments appear inside the nodes.

Past records are stored in the nodes of this tree. Each record is stored in exactly one node, the lowest node whose span contains the record's interval. For example, a record with interval [3,16] is assigned to node [0,16]. The nodes of the binary tree are partitioned into three disjoint sets: *passive, active* and *future* nodes. A node is passive if no more records can ever be stored in that node. It is an active node if it is possible for a record with interval ending in T to be stored there. It is a future node if it can only store records whose intervals end after T, i.e., in the future. Initially all nodes begin as future nodes at T=0 then become active and finally they end up as passive nodes, as time proceeds. Node $[i,j]$ becomes active at $T= (i+j)/2$ if it is a leaf, or at $T= (i+j)/2 +1$ otherwise. For example, in Figure 9, for T=55, node [48,64] belongs to the future nodes. This is because any record with interval contained in [48,55] will be stored somewhere in its left subtree. The only records that can be stored in [48,64] have intervals ending after time 55, so they are future records.

Future nodes need not be kept in the tree before becoming active.

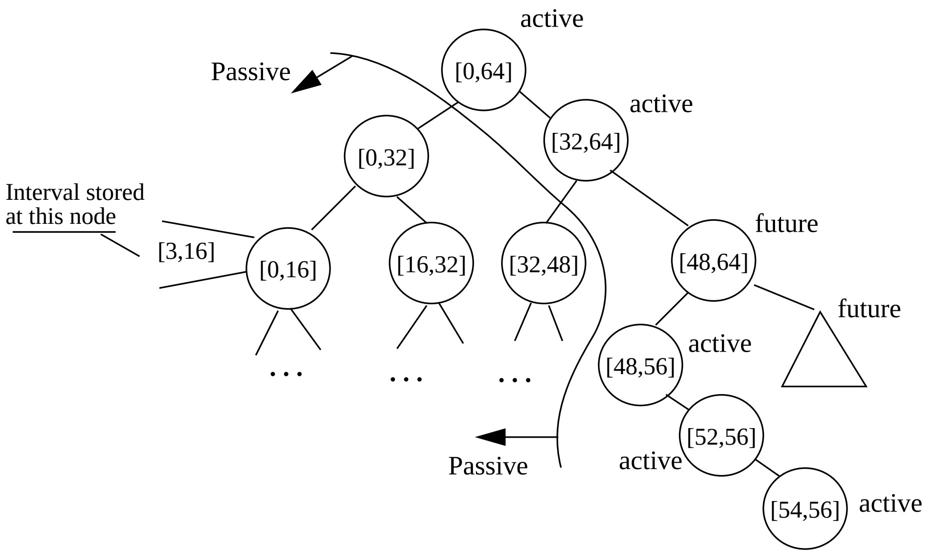


Figure 9: The PVAS binary tree. The current logical time is 55.

Each interval assigned to a PVAS node is stored in two lists, one that stores the intervals in increasing start_time order and one that stores them in increasing end_time order. This is similar to the Interval Tree [E83]. In [VV94] a different structure is used to implement these lists for the active and passive nodes, by exploiting the fact that passive nodes do not get any new intervals after they become passive. In particular, all passive node lists can be stored in two sequential files (the IFILE and the JFILE) a property that provides for good pagination and record clustering. Two dynamic structures, the ITREE (a B-tree structure) and JLISTS (a collection of lists) are used for the active node lists.

The PVAS logical binary tree and its accompanied structures can be placed efficiently into pages (details appear in [VV94]) occupying $O(n/B)$ space. Since the structure does not index record keys, the update assumes that the start_time of the updated record is known; then updating is $O(\log_B n)$. As with most of the other time-only methods, if updates are provided only by the record key, a hashing function can be used to find the start_time of the record before the update proceeds on the PVAS.

To answer a transaction pure-timeslice query both the CVAS and the PVAS are searched. Since the CVAS is ordered on the start_times a logarithmic search will provide which of the current records is born before the query time t. Searching the PVAS structure is more complicated. The search follows a single path down the logical binary tree and the lists of nodes whose span contains t are searched sequentially. Searching each list provides clustered answer, but there maybe

$O(\log_2 n)$ binary nodes whose lists are searched. Since every list access may be a separate I/O, the query time becomes $O(\log_2 n + a/B)$.

Since no record keys are indexed, the method as presented above cannot answer efficiently pure-key queries. For transaction range-timeslice queries the whole timeslice should first be computed. To answer pure-key and range-timeslice queries, [VV94] assumes the existence of another index for various key regions, in a similar way as for the Time-Index.

5.1.2.6 The Snapshot Index

The Snapshot Index [TK95] achieves the I/O-optimal solution to the transaction pure-timeslice problem. It conceptually consists of three data structures: a multilevel index that provides access to the past by time *t*, a multi-linked structure among the leaf pages of the multilevel index that facilitates the creation of the query answer at *t*, and, a hashing function that provides access to records by key, used for update purposes. A real-world object is represented by a record with a time invariant id (object id), a time-variant (temporal) 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 becomes full, a new acceptor page is created. Acceptor pages are added at the end of a linked list (list *L*) as they are created. Up to now, the Snapshot Index resembles a linked "log" of pages that keeps the object records.

There are three main differences from a regular log: the use of the hashing function, the in-place deletion updates and the notion of *page usefulness*. The hashing function is used for updating records about their "deletion". When a new record is created, the hashing function will store the id of this record together with the address of the acceptor page that stores it. Object deletions are not added at the and of the log. Rather they are represented by changing the end_time of the corresponding deleted record. This access is facilitated by the hashing function. All records with end_time equal to *now* are termed "alive" else they are called "deleted".

As pointed out in section 4.1 time-only methods need to order their data by time only and not by time and key. Since data arrives ordered by time, a dynamic hashing function is enough for accessing a record by key (membership test) when updating it. Of course hashing cannot guarantee against pathological worst cases (i.e., when a bad hashing function is chosen). In those cases a B^+ tree on the keys can be used instead of hashing, leading to logarithmic worst case update.

A data page is defined to be *useful* for: (i) all time instants, for which it was the acceptor page, or, (ii) after it ceased being the acceptor page, for all time instants, for which the page contains at least $u \cdot B$ "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 *usefulness* parameter u $(0 < u \leq 1)$ is a constant that tunes the behavior of the Snapshot Index. To answer a pure-timeslice about time t the Snapshot Index will only access the pages useful at t (or equivalently, those pages that have at least $u \cdot B$ records alive at t) plus at most one additional page that was the acceptor page at t. This single page may contain less than $u \cdot B$ records from the answer.

When a useful data page becomes non-useful, its "alive" records are copied to the current acceptor page (this is like a time-split [E86, LS89]). In addition, based on its position in the linked list L, a non-useful data page is removed from L and is logically placed under the previous data page in the list. This creates a multi-linked structure that resembles a forest of trees of data pages and is called the *access forest* (Figure 10). The root of each tree in the access forest lies in list L. The access forest has the following properties: (a) The *u.start_time* fields of the data pages in a tree are organized in a *preorder* fashion. (b) The usefulness interval of a page, includes all the corresponding intervals of the pages in its subtree. (c) 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 $d_i<d_{i+1}<e_i<e_{i+1}$.

Finding the timeslice as of a given time t is reduced to finding the data pages that were useful at time t. This is equivalent to the set-history problem of [TG90, TGH95]. The acceptor page as of t is found through the multilevel index which indexes the *u.start_time* fields of all the data pages. That is, all data pages are at the leaves of the multilevel index (the link list and the access forest are implemented among these leaf pages). Since time is increasing, the multilevel index is "packed" and increases only through its right side. After the acceptor data page at t is located, the remaining useful data pages at t are found by traversing the access forest. This traversing can be done very efficiently using the access forest properties [TK95].

As a result, the Snapshot Index solves the "*/-/point" query optimally: $O(\log_B n + a/B)$ I/Os for query time, $O(n/B)$ space and $O(1)$ update processing per change (in the expected amortized sense, assuming the use of a dynamic hashing function [DKM+88] instead of a B-tree). The number of useful pages depends on the choice of parameter u. Larger u means faster query time (less

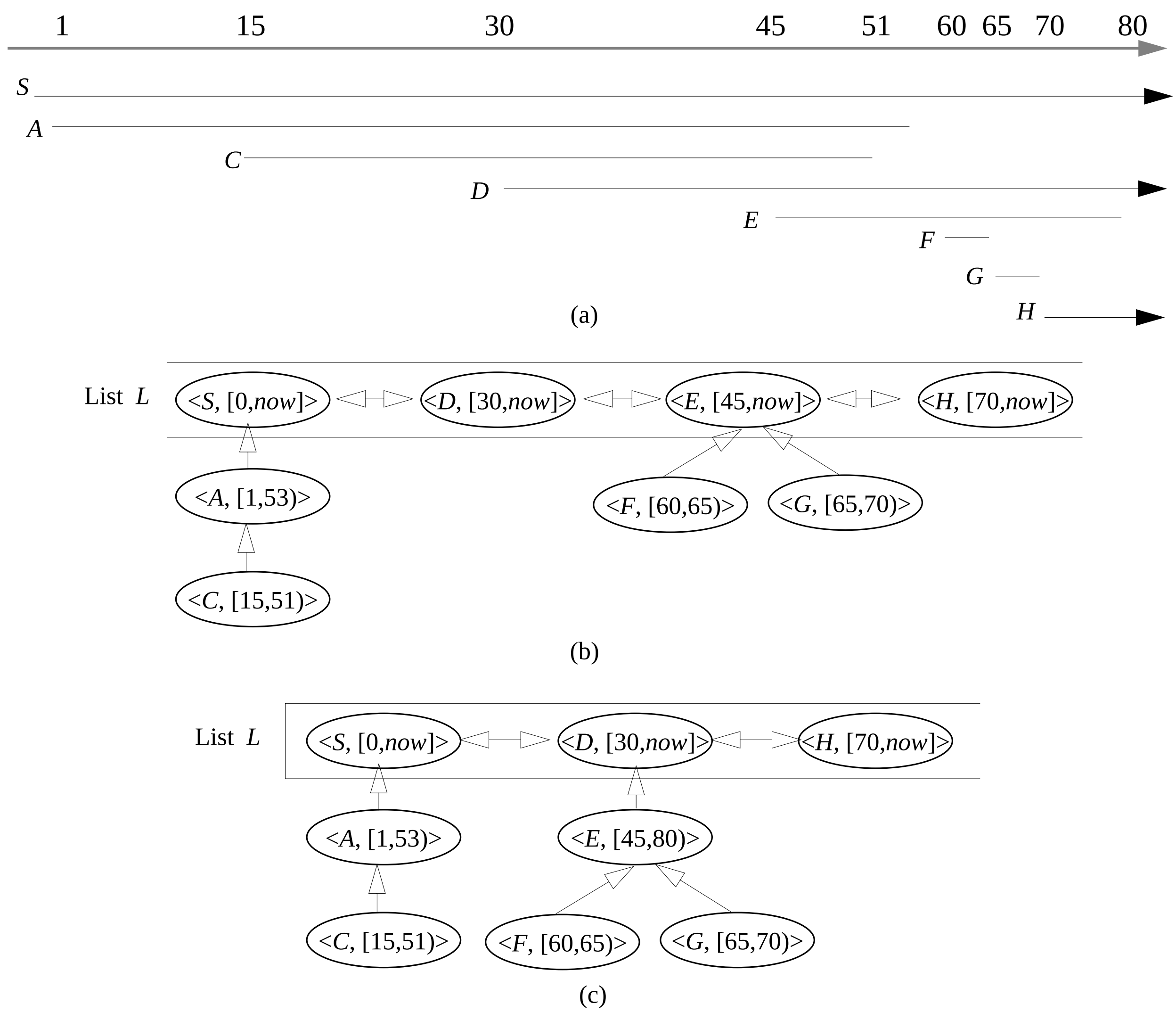


Figure 10: A schematic view of the access forest for a given collection of usefulness intervals. (a)The usefulness interval of each data page as of time 80 is depicted. An open interval at time $t = 80$ represents a data page that is still useful at that time. (b)The access forest as it is at time $t = 79$ (*now* in this figure corresponds to time 79). Each page is represented by a tuple <page-id, page-usefuleness.period>. Page *S* denotes the top of list *L*, while the current acceptor page is always at the end of *L*. (c)The access forest at time $t = 80$. At that time page *E* became non-useful (because some record deletion reduced the number of "alive" records in *E* 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 is not shown.

number of accessed pages) in the expense of additional space (which remains linear to *n/B*). Since more space is available, the answer would be contained in a smaller number of useful pages.

Migration to a WORM disk is possible for each data page that becomes non-useful. Since the parent of a non-useful page in the access forest may still be a useful page, an optical disk page must be reserved for the parent. Observe however, that the Snapshot Index uses a "batched" migration policy which guarantees that the "reserved" space in the optical disk is limited to a controlled small fraction of the number of pages already transferred to the WORM.

Different versions of a given key can be linked together so that pure-key queries of the form: “find all versions of a given key” are addressed in $O(a)$ I/O’s, where a represents now the number of such versions. Since the key space is separate from the transaction time space, the hashing function used to access records by key can keep the latest version of a key, if any. Each key version when updated can be linked to its previous version; thus each record representing a key contains an extra pointer to the record’s previous version. If instead of a hashing function a B-tree is used to access the key space, the bound becomes $O(\log_B S + a)$ where S is the number of different keys ever created.

For answering “range/-/point” queries the Snapshot Index has the same problem as the other time-only methods: the whole timeslice must first be computed. This is in general the trade-off for the fast update processing provided.

5.1.2.7 The Windows Method

Very recently, [R97] provided yet another solution to the “*/-/point” query, the *Windows Method*. This approach has the same performance as the Snapshot Index. It is a paginated version of a data-structure presented in [C86] and which optimally solved the pure timeslice query in main-memory. [R97] partitions the time space in contiguous “windows” and associates with each window a list of all intervals that intersect the window’s interval. Windows are indexed by a B-tree structure (similar to the multilevel index of the Snapshot Index).

To answer a pure timeslice query, the appropriate window that contains this timeslice is first found and then the window’s list of intervals is accessed. Note that the “windows” of [R97] would correspond to one or more consecutive pages in the access-forest of [TK95].

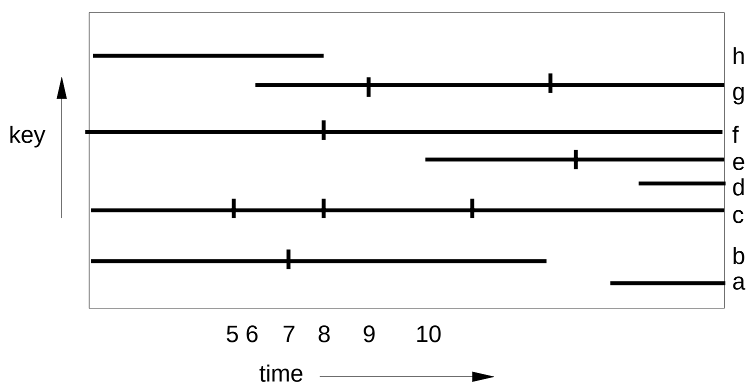
As with the Snapshot Index some objects will appear in many windows (when a new window is created, it gets copies of the “alive” objects from the previous) but the space remains $O(n/B)$. The Windows Method uses the B-tree to also access the objects by key, hence updating is amortized $O(\log_B n)$. If all copies of a given object are linked as proposed in the previous section, all versions of a given key can be found in $O(\log_B n + a)$ I/Os.

5.1.3 Time-Key Methods

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 this query are co-located thus minimizing the number of pages accessed. 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. 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.

An example of a page containing a time-key range is shown in Figure 11. Here, at transaction time instant 5, a new version of the record with key *b* is created. At time 6, a record with key *g* is inserted. At time 7, a new version of the record with key *c* is created. At time 8, both *c* and *f* have new versions and record *h* is deleted. Each line segment, whose start and end time are represented by ticks, represents one record version. Each record version takes up space in the disk page.

There have been two major approaches: methods based on variants of R-Trees [S87, KS89, KS91] and methods based on variants of B^{+}-trees [E86, LS89, LM91, MK90, BGO+93, VV95]. A strong advantage of using R-Tree based methods is that R-trees [G84, SRF87, BKKS90, KF94] can represent additional dimensions on the same index (in principal such a method could support both time dimensions on the same index). A disadvantage of the R-tree based methods is that they cannot guarantee good worst case update and query time performance. However, such worst cases are usually <u>pathological</u> (do not happen often). In practice R-trees have shown good average case performance. Another characteristic of R-tree based methods, is that the end_time of a record's interval is assumed known when the record is inserted in the method, which is not a property of transaction time.



<u>Figure 11:</u> Each page is storing data from a time-key range.

5.1.3.1 R-Tree based methods

The POSTGRES database management system [S87] proposed a novel storage system in which no data is ever overwritten. Rather updates are turned into insertions. POSTGRES timestamps are timestamps of committed transactions. Thus the POSTGRES storage system is a transaction-time access method.

The storage manager accommodates past states of the database on a WORM optical disk (archival system) in addition to the current state that is kept on an ordinary magnetic disk. The assumption is that users will access current data more often than past data, thus the faster magnetic disk is more appropriate for recent data. As past data keeps increasing the magnetic disk will eventually be filled.

As data becomes "old" it migrates to the archival system by means of an asynchronous process, called the *vacuum cleaner*. Each data record has a corresponding interval (*Tmin*, *Tmax*), where *Tmin* and *Tmax* are the commit times of the transactions that inserted and (logically) deleted this record from the database, respectively. When the vacuum cleaner operates, it transfers data whose end time is before some fixed time to the optical disk. The versions of data which reside on an optical disk page have similar end times (*Tmax*), but may have widely varying start times (*Tmin*). Thus pages on the optical disk are as in Figure 12. If such a page is accessed for a query about some "early" time *t*, it may contribute only a single version to the answer, i.e., the answer would not be well clustered among pages.

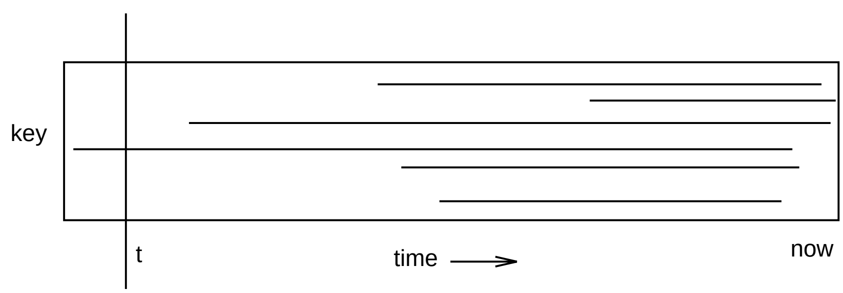


Figure 12: A page storing data with similar end times.

Since data records can be accessed by queries that may involve both time and key predicates, a 2-dimensional R-Tree [G84] access method has been proposed for archival data. POSTGRES assumes that this R-tree is a secondary access method. Pointers to data records are organized according to their key value in one dimension and to their intervals (life-spans) in the other dimension.

The data are written sequentially to the WORM device by the vacuuming process. It is not possible to insert new records in a data page on a WORM device which already contains data. Thus it is not possible to have a primary R-tree with leaves on the optical disk, without changing the R-tree insertion algorithm. However, we will make estimates based on a primary R-tree in keeping with our policy of section 4.1.

For current data, POSTGRES does not specify the indexing used. Whatever this is, queries as of any past time before the most recent vacuum time must access both the current and the historical components of the storage structure. Current records are stored only in the current database and their start times can be arbitrarily far back in the past.

For archival data, (secondary) indexes spanning the magnetic and optical disk are proposed (*combined media* or *composite indexes*). There are two advantages in allowing indexes to span both media: (a) improved search and insert performance as compared to indexes that are completely on the optical medium (such as the *Write-Once Balanced Tree* [E86] and the *Allocation Tree* [V85]), and, (b) reduced cost per bit of disk storage as compared to indexes entirely contained on magnetic disk. Two combined media R-Tree indexes were proposed in [KS89]; they differ on the way index blocks are vacuumed from the magnetic to the archival medium.

In the first approach, the R-Tree is rooted on the magnetic disk and whenever its size on the magnetic disk exceeds some pre-allocated threshold, the vacuuming process starts moving some of the leaf pages to the archival medium. These pages refer to records which have already been moved to the optical disk. Each such record has *Tmax* less than some time value. For each leaf page, the maximum *Tmax* is recorded. The pages with smallest maximum *Tmax* refer to data which was transferred longest ago. These are the pages which are transferred. Following the vacuuming of the leaf nodes the process recursively vacuums all parent nodes that point entirely to children nodes which have already been stored on the archive. The root node however is never a candidate for vacuuming.

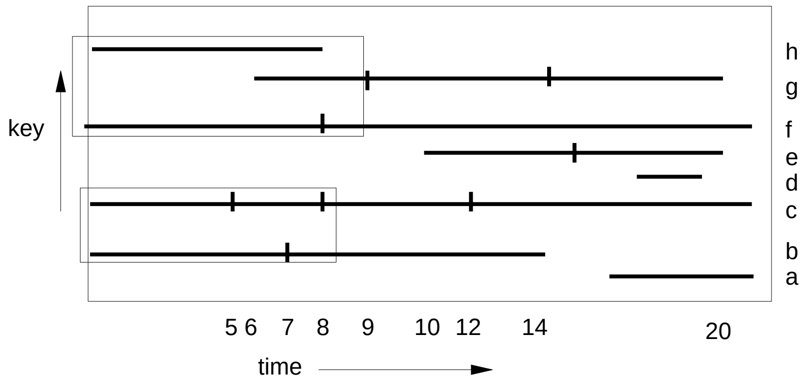
The second approach (*Dual R-Tree*) maintains two R-Trees, both rooted on the magnetic disk. The first is entirely stored on the magnetic disk while the second is stored on the archival disk except for its root (in general except from the upper levels). When the first tree gains the height of the second tree, the vacuuming process will vacuum all the nodes of the first tree, except its root, to the optical disk. References to the blocks below the root of the first tree are inserted in the root of the second tree. Over time, there would continue to be two R-Trees, the first completely on the magnetic disk and periodically archived. Searches are performed by descending both R-Trees.

In analyzing the use of the R-tree as a temporal index, we will speak of records rather than pointers to records. In both approaches a given record is kept only once, therefore the space is clearly linear to the number of changes (the number of data records in the tree is proportional to n). Since the height of the trees is $O(\log_B n)$ each record insertion needs logarithmic time. While on the average searching an R-tree is also logarithmic, in (pathological) worst case this searching can be $O(n/B)$ since the whole tree may have to be traversed due to the overlapping regions. Figure 13 shows the general R-tree method, using overlapping rectangles of time-key space.

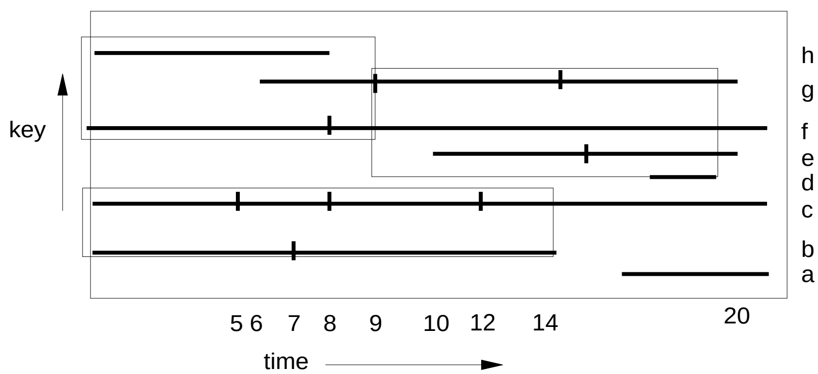
R-Trees are best suited for indexing data that exhibits a high degree of natural clustering in multiple dimensions; then the index can partition data into rectangles so as to minimize both the coverage and the overlap of the entire set of rectangles (i.e., rectangles corresponding to leaf pages and internal nodes). Transaction time databases however, may consist of data whose attribute values vary independently of their transaction time intervals, thus exhibiting only one-dimensional clustering. In addition, in an R-Tree that stores temporal data, page splits cause a good deal of overlap in the search regions of the non-leaf nodes. It was observed that for data records with non-uniform interval lengths (i.e., a large proportion of “short” intervals and a small proportion of “long” intervals), the overlapping is clearly increased, affecting the query and update performance of the index.

Figure 14 shows how long-lived records inhibit the performance of structures which keep only one copy of each record and which keep time-key rectangles. The problem is that a long-lived record determines the length of the time range associated with the page in which it resides. Then even if only one other key value is present, and there are many changes to the record with the other key value in that time range, overlap is required. For example, in Figure 14, the eleventh record version (shown with a dotted line) belongs to the time-key range of this page but it cannot fit since the page has already ten record versions. It will instead be placed in a different page whose rectangle has to overlap this one. The same example also illustrates that the number of leaf pages to be retrieved for a timeslice can be large ($O(a)$) since only a few records may be “alive” (contain the given time value in their interval) for any one page.

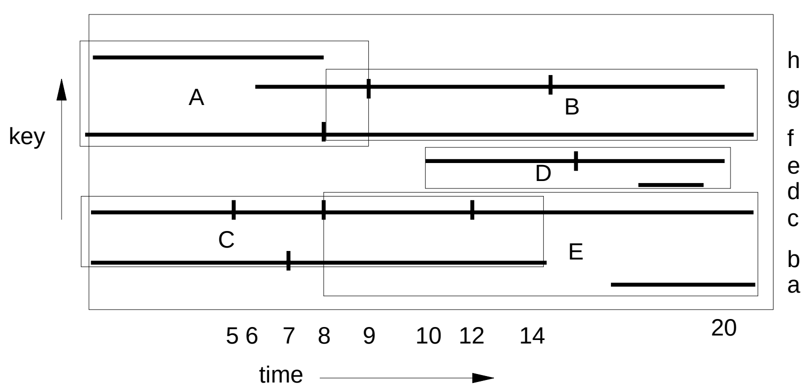
In an attempt to overcome the above problems, the *Segment R-Tree* (*SR-Tree*) was proposed [KS91, K93]. The SR-Tree combines properties of the R-Tree and the *Segment Tree*, a binary tree



Suppose a maximum capacity of 5 record versions in each page
Record versions are entered as they die. At time instant 9, the
records must be split into two pages as illustrated here because
at instant 9 there are six dead record versions.



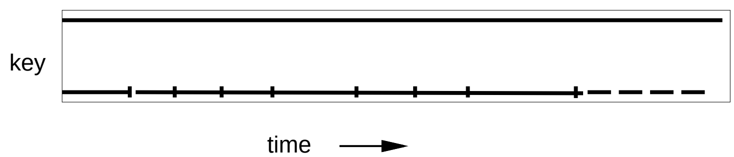
These are possible data page boundaries at the time record version d dies.



This is a possible allocation to R−tree data pages of all versions shown, given that at most 5 and and least 3 record versions must be in each page. The parent node will contain the border coordinates for each of the five children. For example, data node C has borders with time running between 0 and 14 and keys b and c only. The version of record c between 8 and 12 belongs to E. A time slice query at time instant 8 visits A, B, C, and E and obtains one record version from each page.

Figure 13: An example of data bounding as used in R-tree based methods.

a) The first record version is very long−lived. Its time span is the minimum time span this time−key rectangle can have. Here we add one other record, with another key. Now we have minimum boundaries in both time and key.



b) The second record gets many new versions over the time span of the first (long−lived) record. If the page capacity is ten record versions, the eleventh version in this time−key rectangle does not fit. This implies that there will be overlapping with another time−key rectangle, which holds the next version of the second record. This shows where one long−lived record and one record with a different key and many short versions forces time−key rectangles to overlap.

Figure 14: The effect of long-lived records on overlapping.

data structure proposed in [B77] for storing line segments. A Segment Tree stores the interval endpoints on the leaf nodes; each internal node is associated with a “range” that contains all the endpoints in its subtree. An interval *I* is stored in the highest internal node *v* such that *I* covers *v*’s range and does not cover the range of *v*’s parent. Observe that an interval may be stored in at most logarithmic many internal nodes; thus the space is no longer linear [M84].

The SR-Tree (Figure 15) is an R-Tree where intervals can be stored in both leaf and non-leaf nodes. An interval *I* is placed to the highest level node *X* of the tree such that *I* spans at least one of the intervals represented by *X*’s child nodes. If *I* does not span *X*, spans at least one of its children but is not fully contained in *X*, then *I* is fragmented.

Using this idea, long intervals will be placed in higher levels of the R-Tree, thus the SR-Tree tends to decrease the overlapping in leaf nodes (in the regular R-Tree, a long interval stored in a leaf node will “elongate” the area of this node thus exacerbating the overlap problem). One risks having large numbers of spanning records or fragments of spanning records stored high up in the tree. This decreases the fan-out of the index as there is less room for pointers to children. It is suggested to vary the size of the nodes in the tree, making higher-up nodes larger. “Varying the size” of a node means that several pages are used for one node. This adds some page accesses to the search cost.

As with the R-tree, if the record is inserted at a leaf (because it did not span anything) the

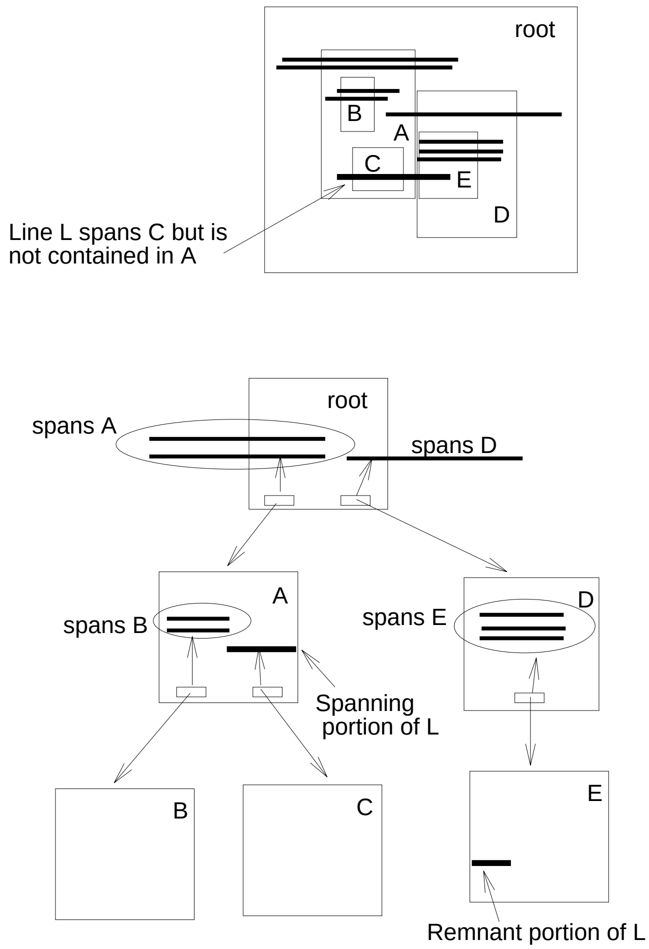


Figure 15: The SR-Tree.

boundaries of the space covered by the leaf node in which it is placed may be expanded. Expansions may be needed on all nodes on the path to the leaf which contains the new record. This may change the spanning relationships since records may no longer span children which have been expanded. In this case, such records are reinserted in the tree, possibly being demoted to occupants of nodes they previously spanned. Splitting nodes may also cause changes in spanning relationships as they make children smaller -former occupants of a node may be promoted to spanning records in the parent.

Similarly with the Segment Tree, the space used by the SR-Tree is no longer linear. An interval may be stored in more than one non-leaf nodes (in the *spanning* and *remnant* portions of this interval). Due to the use of the segment-tree property, the space can be as much as $O(n\log_B n)$.

Inserting an interval still takes logarithmic time. However, due to possible promotions, demotions and fragmentation, insertion is slower than in the R-tree. Even though the segment property tends to reduce the overlapping problem, the (pathological) worst case performance for the deletion and query time remains the same as for the R-Tree organization. The average case behavior is again logarithmic.

To improve the performance of their structure, the authors have also proposed the use of a *Skeleton SR-Tree*, which is an SR-Tree which pre-partitions the entire domain into some number of regions. This pre-partition is based on some initial assumption on the distribution of data and the number of intervals to be inserted. Then the Skeleton SR-Tree is populated with data; if the data distribution is changed the structure of the Skeleton SR-Tree can be changed, too.

An implicit assumption made by all R-Tree based methods is that when an interval is inserted both its *Tmin* and *Tmax* values are known. In practice however this is not true for "current" data. One solution would be to enter all such intervals as (*Tmin*, *now*), where *now* is a variable representing the current time. A problem with this approach is that a "deletion" update that changes the *now* value of an interval to *Tmax*, is implemented by a search for the interval, a deletion of the (*Tmin*, *now*) interval and a re-insertion as (*Tmin*, *Tmax*) interval. Since searches are not guaranteed for worst case performance this approach could be problematic. The deletion of *(Tmin, now)* is a physical deletion which implies the physical deletion of all remnant portions of this interval. A better solution would be to keep the current records in a separate index (probably a basic R-tree). This will avoid the above deletion problem but the worst case performance remains as before.

The pure-key query is addressed as a special case of a range time-interval query, where the range is limited to a key and the time-interval is the whole time axis. Hence all pages that contain the key in their range will be accessed. However, if this key never existed, the search may go through *O(n/B)* pages in (pathological) worst case. If this key has existed, the search will definitely find its appearances but it may also access pages that do not contain any appearances of this key.

If the SR-Tree is used as a valid-time method, then physical deletions of any stored interval should be supported efficiently. As above, the problem with physical deletions emanates from keeping an interval in many remnant segments that all have to be found and physically deleted. Actually, the original SR-Tree paper [KS91] assumes that physical deletions do not happen often.

5.1.3.2 The Write-Once B-Tree

The Write-Once B-tree or WOBT, proposed in [E86], was originally intended for a database which resided entirely on WORMs. However, many variations of this method, the Time-Split B-tree [LS89], the Persistent B-tree [LM91], the Multiversion B-tree [BGO+93] and the Multiversion Access Structure [VV95] have been proposed which may use both a WORM and a magnetic disk, or only a magnetic disk. The WOBT itself can be used either on a WORM or on a magnetic disk. The WOBT is a modification of the B^+-tree given the constraints of a WORM.

The WORM characteristics imply that once a page is written no new data can be entered or updated in the page (since a checksum is burned into the disk). As a result, each new index entry occupies an entire page; for example, if a new index entry takes 12 bytes and a page is 1024 bytes, 99% of the page is empty. Similarly, each new record version is an entire page. Tree nodes are collections of pages, for example a track on a disk. Record versions contain their transaction start time only. A new version with the same key is placed in the same node. Its start time is the end time of the previous version. Nodes represent a rectangle in the transaction time-key space. The nodes partition that space-- each time-key point is in exactly one node.

When a node fills up, it can be split by (current) transaction time or split first by current transaction time and then by key. The choice depends on how many records in the node are current versions at the time of the split. The old node is left in place. (There is no other choice.) The record versions “alive” at the current transaction time are copied to a new node or two new nodes if it is also split by key. There is space for new versions in the new nodes. Deletions of records are handled in the only possible way: a node deletion record is written in a current node and it contains the end time. When the current node is split, the deleted record is not copied. This design enables some clustering of the records in nodes by time and key (after a node split, “alive” records are stored together in a page) but most of the space of most of the optical disk pages is empty (because most new entries occupy whole pages).

When a root node splits, a new root is created. Addresses of consecutive roots and their creation times are held in a “root log” that has the form of a variable-length append-only array. This array will provide access to the appropriate root of the WOBT by time.

If the WOBT is implemented on a magnetic disk, space utilization is immediately improved as it is not necessary to use an entire page for one entry. Pages can be updated, so they can be used for nodes. Space utilization is $O(n/B)$ and range queries are $O(\log_B n+a/B)$, if one disregards record deletion. These bounds are for using the method exactly as described in the paper, except that each node of the tree will be a page on a magnetic disk. In particular, the old node in a split is not moved.

Current records are copied to a new node or to two new nodes. Since deletions are simply handled with a deletion record (which is "seen" by the method as another updated value) the search algorithm is not able to avoid searching pages that may be full of "deleted" records. Therefore, if deletions are frequent, pages that do not contribute in the answer may be accessed.

Since all the B^+-tree based transaction-time methods search data records by both transaction time and key, or, by transaction time only, answering a pure-key query with the WOBT (and the Time-Split B-tree, Persistent B-tree and Multiversion B-tree) requires that a given version (instance) of the key whose previous versions are requested should be also provided by the query. That is, a transaction time predicate should be provided in the pure-key query as for example in: "find the previous salary history of employee *A* who was alive at *t*."

Different versions of a given key can be linked together so that the pure-key query (with time predicate) is addressed by the WOBT in $O(\log_B n + a)$ I/O's. The logarithmic part is spent for finding the instance of employee *A* in version *t* and then its previous *a* versions are accessed using a linked structure. Basically, the WOBT (and the Time-Split B-tree, Persistent B-tree and the Multiversion B-tree) can have backwards links in each node to the previous historical version. This does not use much space, but for records which do not change over many copies, one needs to go back many pages before getting more information. To achieve the above bound, each record needs to keep the address of the last different version of that record.

If such addresses are kept in records, the address of the last different version for each record is available at the time the data node does a time split. Then these addresses can be copied to the new node with their respective records. A record whose most recent previous version is in the node which is split must add that address. A record which is the first version with that key must have a special symbol to indicate this fact. This simple algorithm can be applied to any method which does time splits.

To answer the general pure-key query "find the previous salary history of employee *A*" requires finding if *A* was ever an employee. The WOBT would need to copy "deleted" records when a time split occurs, which implies that the WOBT state carries the latest record for all keys ever created. However this will increase the space consumption. Otherwise, if "deleted" records are not copied, all pages including this key in their key space may have to be searched.

The WOBT used on a magnetic disk still makes copies of records where it does not seem necessary. The WOBT always makes a time split before making a key split. This creates one historical page and two current pages where previously there was only one current page. A B-tree split creates two current pages where there was only one. No historical pages are created. It seems like a good idea to be able to make pure key splits as well as time splits or time-and-key splits. This would make the space utilization better.

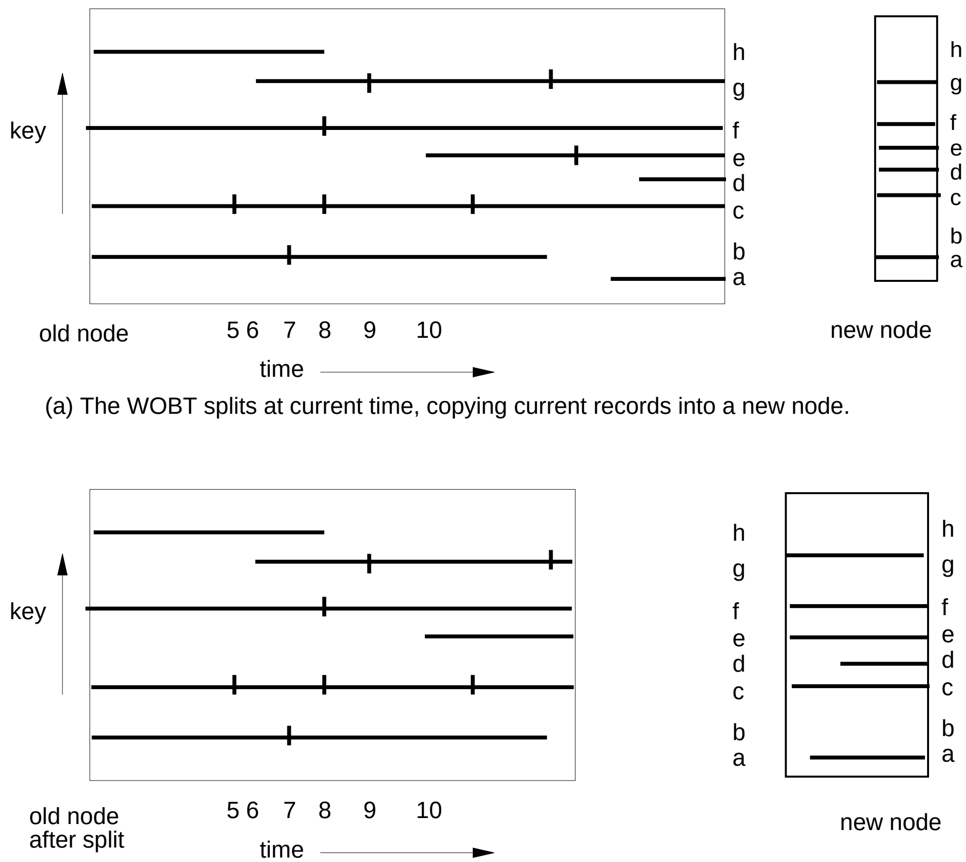
5.1.3.3 The Time Split B-Tree

The Time-Split B-tree (or "TSB-tree") [LS89, LS90a, LS93a] is a modification of the WOBT which allows pure key splits and which keeps the current data in an erasable medium such as a magnetic disk and migrates the data to another disk (which could be magnetic or optical) when a time split is made. This partitions the data in nodes by transaction time and key like the WOBT, but is more space efficient. It also separates the current records from most of the historical records. In addition, the TSB-tree does not keep a "root log". Instead it creates new roots as B^+-trees do, by increasing the height of the tree when the root splits.

When a data page is full and there are less than some threshold value of alive distinct keys, the TSB-tree will split the page by transaction time only. This is the same as what the WOBT did, except now times other than the current time can be chosen. For example, the split time for a data page could be the "time of last update", after which there were only insertions of records with new keys and no updates creating new versions of already existing records. The new insertions, after the time chosen for the split, need not have copies in the historical node. Time splits in the WOBT and in the TSB-tree are illustrated in Figure 16.

Time splitting, whether by current time or by time of last update, enables an automatic migration of older versions of records to a separate historical database. This is to be contrasted with POSTGRES' vacuuming which is "manual" and is invoked as a separate background process which searches through the database for dead records.

It is also to be contrasted with methods which reserve optical pages for pages which cannot yet be moved and maintain two addresses (a magnetic page address and an optical page address) for search for the contents. TSB-tree migration takes place when a time split is made. The current page retains the current contents and the historical records are written sequentially to the optical disk. The new optical disk address and the time of the split are posted to the parent in the TSB-tree. As with B^+-tree node splitting, only the node to be split, the newly allocated node and the parent are affected (also, rarely, a full parent requires further splitting). Since the node is full and is obtaining new data, a split must be made anyway, whether or not the new node is on an optical disk. (This



(a) The WOBT splits at current time, copying current records into a new node.

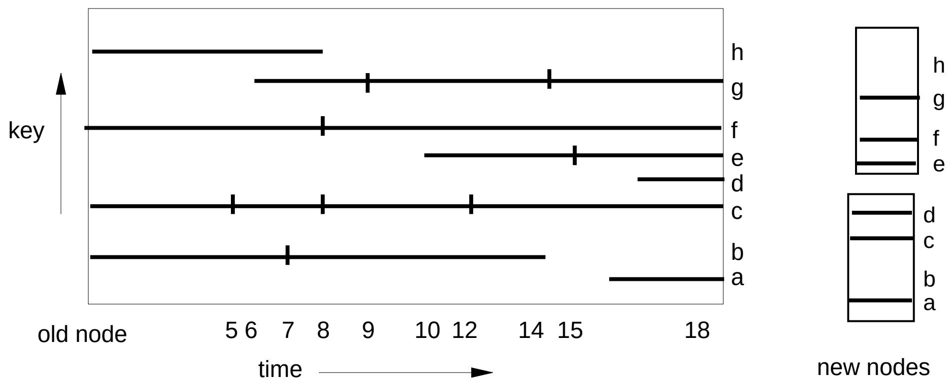
(b) The TSB tree can choose other times to split.

Figure 16: Time splitting in the WOBT and TSB-tree.

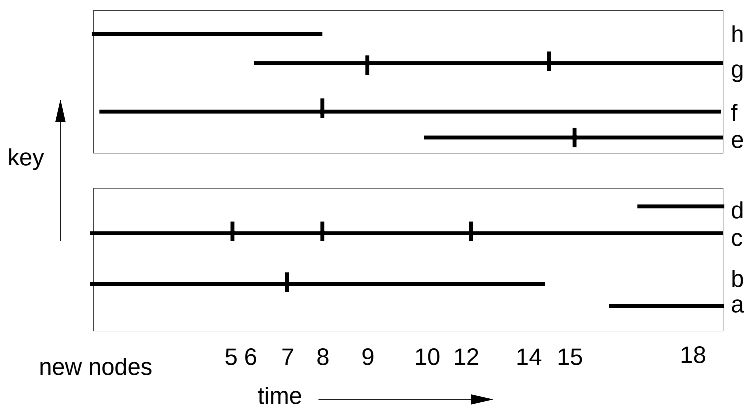
migration to an archive can also be used for media recovery as illustrated in [LS93b].)

Time splitting by other than the current transaction time has another advantage. It can be used in distributed databases where the commit time of a transaction is not known until the commit message is sent from the coordinator. In such a database, an updated record of a PREPARED cohort may or may not have a commit time before the time when an overflowing page containing it must be split. Such a page can only be split by a time before the time voted by the cohort as the earliest time it might commit (see [S94,L93] for details).

Full data pages with a large number of distinct keys currently “alive” are split by key only in the TSB-tree. The WOBT splits first by time and then by key. Similarly with the WOBT, the space usage for the TSB-tree is *O(n/B)*. The constant factor in the asymptotic bound will be smaller for the TSB-tree since it makes less copies of records. Key splitting for the WOBT and the TSB-tree is illustrated in Figure 17.



(a) The WOBT splits data nodes first by time then sometimes also by key.



(b) The TSB−tree can split by key alone.

Figure 17: Key splitting in the WOBT and TSB-tree.

An extensive average case analysis using Markov chains and considering various rates of update versus insertions of records with new keys can be found in [LS90a]. This shows at worst two copies of each record even under large update rates. The split threshold was kept at $2B/3$. (If more than $2B/3$ distinct keys were in the page, a pure key split was made.)

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 17 (b), the key split is taken at time t=18, when there are six keys alive, that 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.

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 now, or 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, as in the WOBT, the TSB-tree merely posts deletion markers and does not merge sparse nodes. If no merging of current nodes is done, and there are many record deletions, a current 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. Other modifications of [E86] discussed in the next section combined with the TSB-tree modifications of [E86] should solve this problem. Basically, when there are too few distinct keys at any time covered by the time-key rectangle of a node to be split, it must be split by time and then possibly by key. Node consolidation should also be supported (to deal with pages that become sparse of alive keys due to deletions).

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. This parent is a current index node. Updates need never be made in historical index nodes, which like historical data nodes can be placed on WORM devices.

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. Index node splitting is illustrated in Figure 18.

Unlike the WOBT (or [LM91] or [BGO+93]), the TSB-tree can move the contents of the

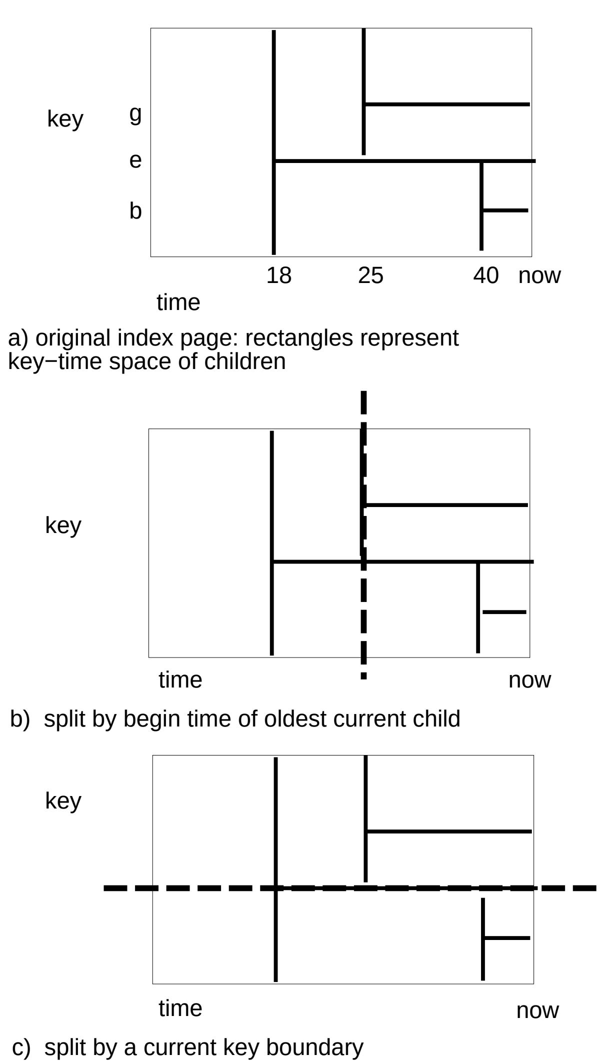


Figure 18: Index node splitting in the TSB-tree.

historical node to another location in a separate historical database without updating more than one parent. No node which might be split in the future has more than one parent. If a node does a time split, the new address of the historical data from the old node can be placed in its unique parent and the old address can be used for the new current data. If it does a key split, the new key range for the old page can be posted along with the new key range and address.

As with the WOBT, pure-key queries with time predicate are addressed in $O(\log_B n + a)$ I/O's, where a represents the size of the answer.

5.1.3.4 The Persistent B-tree

Several methods [LM91, BGO+93, VV95] were derived from a method [DSST89] for general main-memory resident linked data structures. [DSST89] shows how to take an "ephemeral data structure” (meaning that past states are erased when updates are made) and convert it to a “persistent data structure” (where past states are maintained). A “fully persistent” data structure allows updates to all past states. A “partially persistent” data structure allows updates only to the most recent state.

Consider the abstraction of a transaction time database as the “history of an evolving set of plain objects” (Fig. 1). Assume that a B^+-tree is used to index the initial state of this evolving set. If this B^+-tree is made partially persistent we have constructed an access method that supports transaction range-timeslice queries (“range/-/point”). Conceptually, a range-timeslice query for transaction time *t* is answered by traversing the B^+-tree as it was at *t*. Partial persistence suits nicely to transaction time since only the most recent state is updated. Note that the method used to index the evolving set state affects what queries are addressed. For example, to construct a pure-timeslice method, the evolving set state is represented by a hashing function that is made partially persistent. This is another way to “visualize” the approach taken by the Snapshot Index.

Note that a fully persistent access structure can be restricted to the partially persistent case. That is the reason for discussing [DSST89] and [LM91] in this survey.

[LM91] provides a fully persistent B^+-tree. For our purposes we are only interested in the methods presented in [LM91] when reduced to partial persistence. We thus term the <u>partially</u> persistent method of [LM91] as the Persistent B-Tree. The Multiversion B-Tree (or “MVBT”) of [BGO+93] and the MVAS of [VV95] are also partially persistent B^+-trees. The Persistent B-Tree and the MVBT, MVAS support node consolidation (that is, a page is consolidated with another page if it becomes sparse of alive keys due to frequent deletions). In comparison, the WOBT and the TSB-tree are partially persistent B^+-trees which do not do node consolidation (since they aim for applications were data is mainly updated and infrequently deleted). Node consolidation may result in thrashing (consolidating and splitting the same page continually) which results in more space. The MVBT, MVAS disallow thrashing while the Persistent B-Tree does not.

[DSST89], [LM91], [BGO+93] and [VV95] speak of version numbers rather than of timestamps. One important difference between version numbers for partially persistent data and timestamps is that timestamps as we have defined them are transaction time instants when events (changes) are stored. So timestamps are not consecutive integers. But version numbers can be consecutive integers. This has an effect on search operations since [DSST89], [LM91], [BGO+93] and [VV95] maintain an auxiliary structure called *root** which serves the same purpose as the “root log” of the WOBT.

In [DSST89], root* is an array indexed on version numbers. Each array entry has a pointer to the root of the version in question. If the version numbers are consecutive integers, search for the root is $O(1)$. If timestamps are used, search is $O(\log_B n)$. In [LM91], [BGO+93] and [VV95], root* only obtains entries when a root splits. Although root* is thus smaller than it would be if it had an entry for each timestamp, search within root* for the correct root is $O(\log_B n)$.

The use of the root* structure (array) in [LM91, BGO+93, VV95] facilitates faster update processing as the most current version of the B^+-tree is thus separated from most of the previous versions. The most current root can have a separate pointer yielding $O(1)$ access to that root. (Each root corresponds to a consecutive set of versions). If the current version has size m updating is $O(\log_B m)$. Methods that do not use the root* structure have $O(\log_B n)$ update processing.

[DSST89] explains how to make any ephemeral main-memory linked structure persistent. Two main methods are proposed: the *fat node* method and the *node copying* method. The fat node method keeps all the variations of a node in a variable-sized “fat node.” When an ephemeral structure would update the contents of a node, the fat node method would simply append the new values to the old node, with a notation of the version number (timestamp) which does the update. When an ephemeral structure would create a new node, a new fat node is created.

[LM91] applies the fat node method of [DSST89] to the B^+-tree. The fat nodes are collections of B^+-tree pages, each corresponding to a set of versions. Versions can share B^+-tree pages if the records in them are identical for each member of the sharing set. But versions with only one different data record have distinct B^+-tree pages.

Pointers to lower levels of the structure are pointers to fat nodes, not to individual pages within fat nodes. When a record is updated or inserted or deleted, a new leaf page is created. The new leaf is added to the old fat node. If the new leaf contents overflows, the new leaf is split, with the lower-value keys in the old fat node and the higher value keys in a new fat node. When a leaf splits, the parent node must be updated to direct search correctly for part of the new version that is in the new fat node. When a new page is added to a fat node, the parent need not be updated.

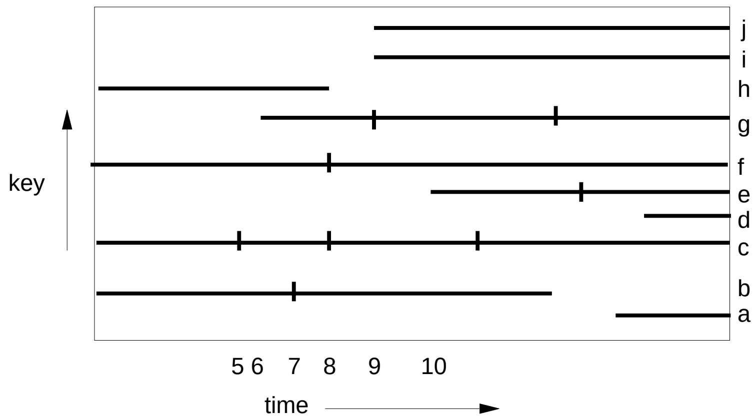
Similarly, when index nodes obtain new values because a fat node child has a split, new pages are allocated to the fat index node. When an index node would split, the parent of the index node obtains a new value. When roots split, a new pointer is put in the array root*, which allows access to the correct (fat) root nodes.

Since search within a fat node would mean fetching all the pages in the fat node until the correct one was found (with the correct version number), [LM91] suggests an auxiliary structure in each fat node of the Persistent B-tree: a *version block*. The version block indicates which page or block in the fat node corresponds to which version number. Figure 19 shows the incremental creation of a version block with its fat node pages. In Figure 20, an update causes this version block to split. The version block is envisioned as one disk page, but there is no reason that it might not become much larger. It may itself have to take the form of a multiway access tree (since new entries are always added at the end of a version block). Search in one version block for one data page could itself be $O(\log_B n)$. For example, if all changes to the database were updates of one B^+-tree page, the fat node would have n B^+-tree pages in it.

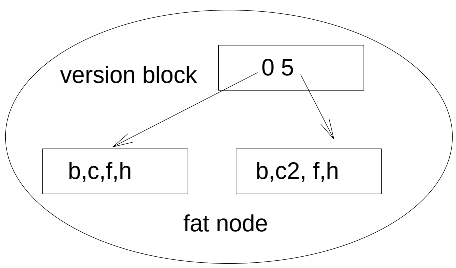
Although search is no longer linear within the fat node, the path from the root to a leaf is at least twice as many blocks as it would be for an ephemeral structure. The height of the tree in blocks is at least twice what it would be for an ephemeral B^+-tree containing the same data as in one of the versions. Update processing is amortized $O(\log_B m)$ where m is the size of the current B^+-tree being updated. Range timeslice search is $O(\log_B n(\log_B m + a/B))$. After the correct root is found, the tree that was current at the time of interest is searched. This tree has height $O(\log_B m)$ and searching each version block in the path is $O(\log_B n)$. A similar bound holds for the pure-timeslice query. Space is $O(n)$ (not $O(n/B)$) since new leaf blocks are created for each update.

To avoid creating new leaf blocks for each update, the “fat field method” is also proposed in [LM91]. Here updates can fit in space of non-full pages. In the general full persistence case, each update must be marked with the version number which created it and with the version numbers of all later versions which delete it. Since we are interested only on partial persistence, this corresponds to the start time and the end time of the update. Fat fields for the Persistent B-tree are illustrated in Figure 21.

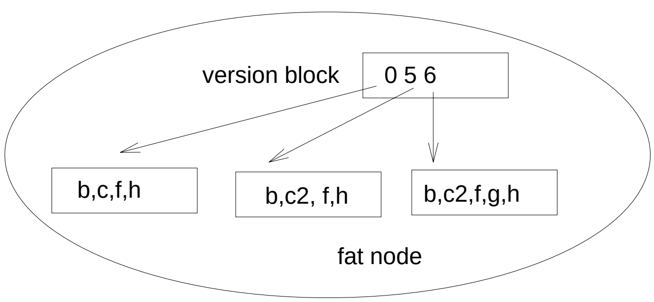
When a page becomes full, the version creating the overflow copies all information which is relevant to that version to a new node. The Persistent B-tree then creates a fat node and a version block. If the new copied node is still overflowing, a key split can be made and then information must be posted to the parent node regarding the key split and the new version. Thus the Persistent B-tree of [LM91] does time splits and time-and-key splits just as in the WOBT. In this variation, space usage is $O(n/B)$, update processing is amortized $O(\log_B m)$ and query time (for both the range and pure-timeslice queries) is $O(\log_B n(\log_B m + a/B))$. The update and query time characteristics remain asymptotically the same as in the fat-node method since the fat-field method



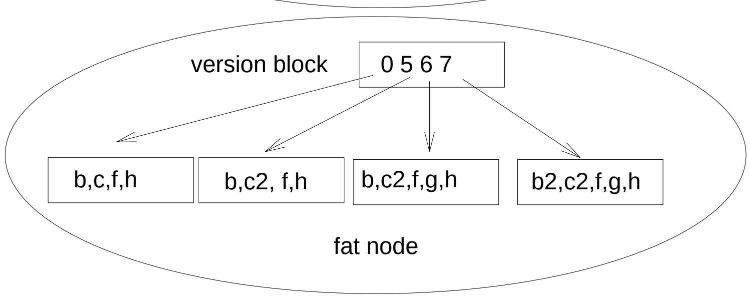
At time 0, the database contains records with keys h, f, c and b. At time 5, a new version is created which has an update to record c. At time 6, record g is inserted. Each time instant when a change occurs corresponds to a new version of the database. We assume a capacity of 5 records per page.



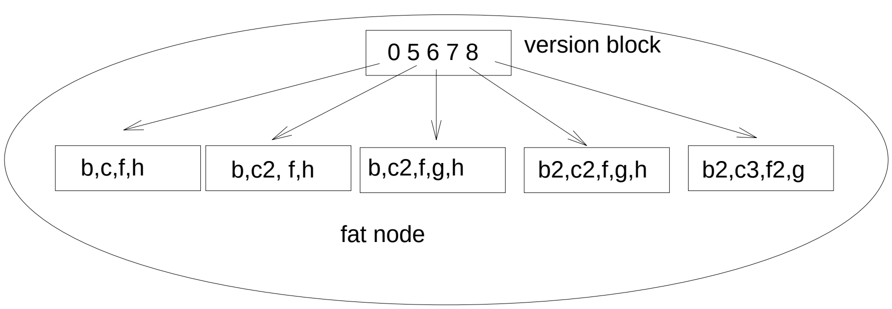
a) Every time a new version of the database is created by a transaction, a new data page is allocated to hold only those records alive after the change. A version block (another disk page) directs search to the correct data page. Here, for times before 5, search goes to the first data page. At and after 5, it goes to the second data page. The record "c2" is the updated version of the record "c". A set of data pages with their version block is called a "fat node."



b) At time 6, a new record, with key "g", is inserted. A new data page is allocated. The version block is updated to direct search at and after 6 to the third data page.

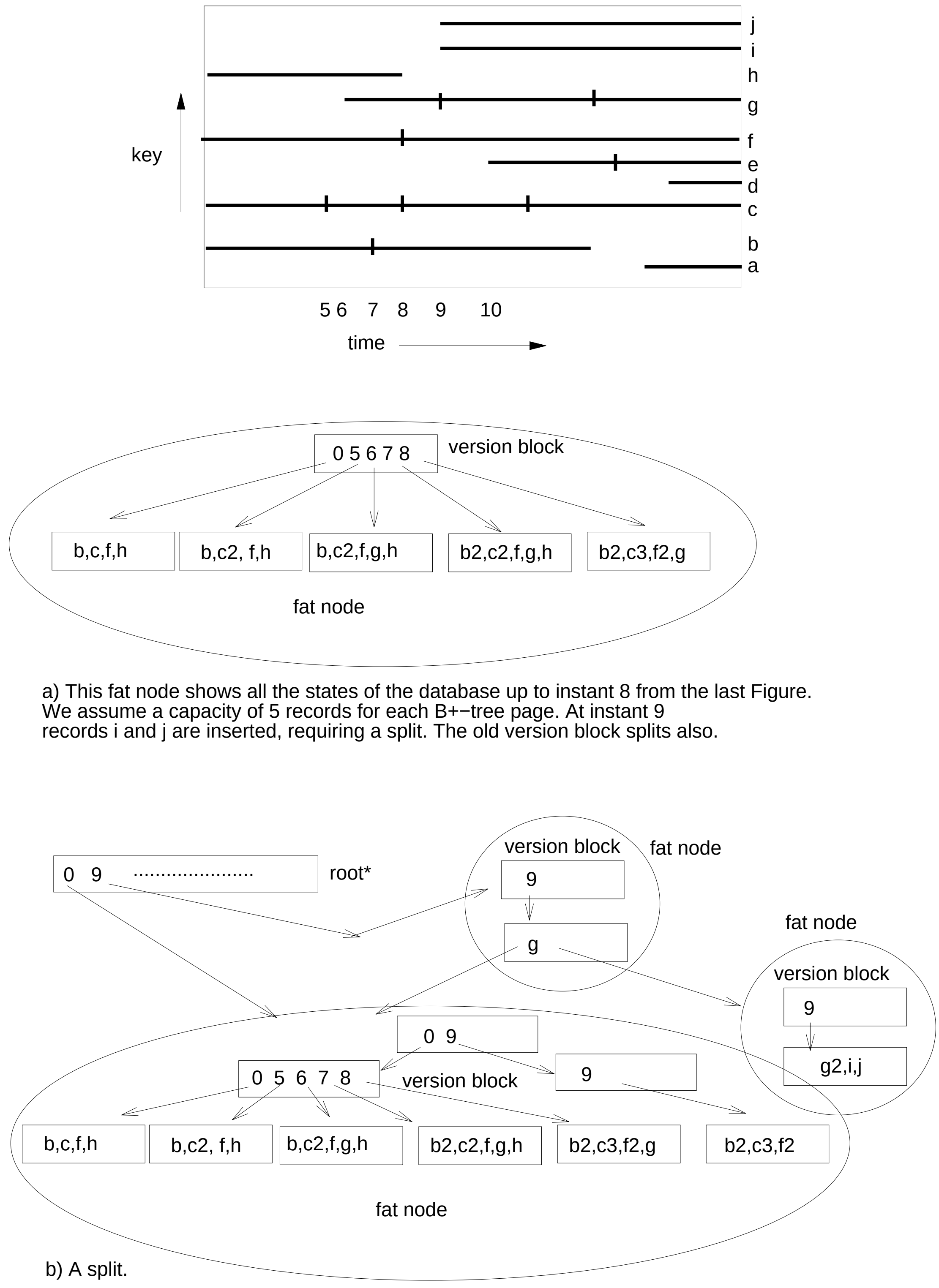


c) This shows the fat node after an update is made on record "b" at time 7.



d) This shows the fat node after an update is made on records with keys "c" and "f" at time 8.

Figure 19: Incremental creation of a fat node in the Persistent B-Tree.



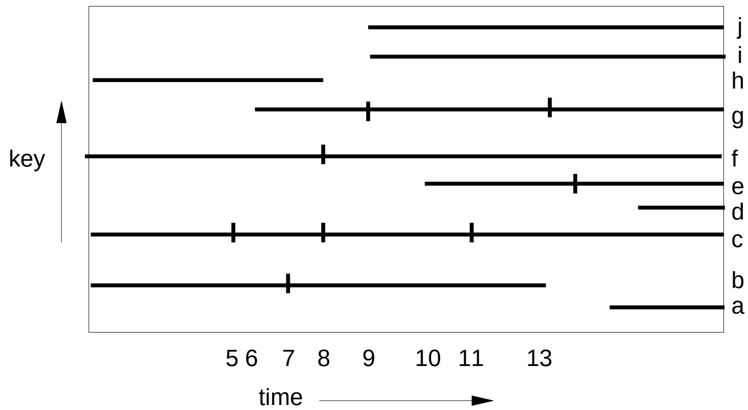
a) This fat node shows all the states of the database up to instant 8 from the last Figure. We assume a capacity of 5 records for each B+−tree page. At instant 9 records i and j are inserted, requiring a split. The old version block splits also.

b) A split.

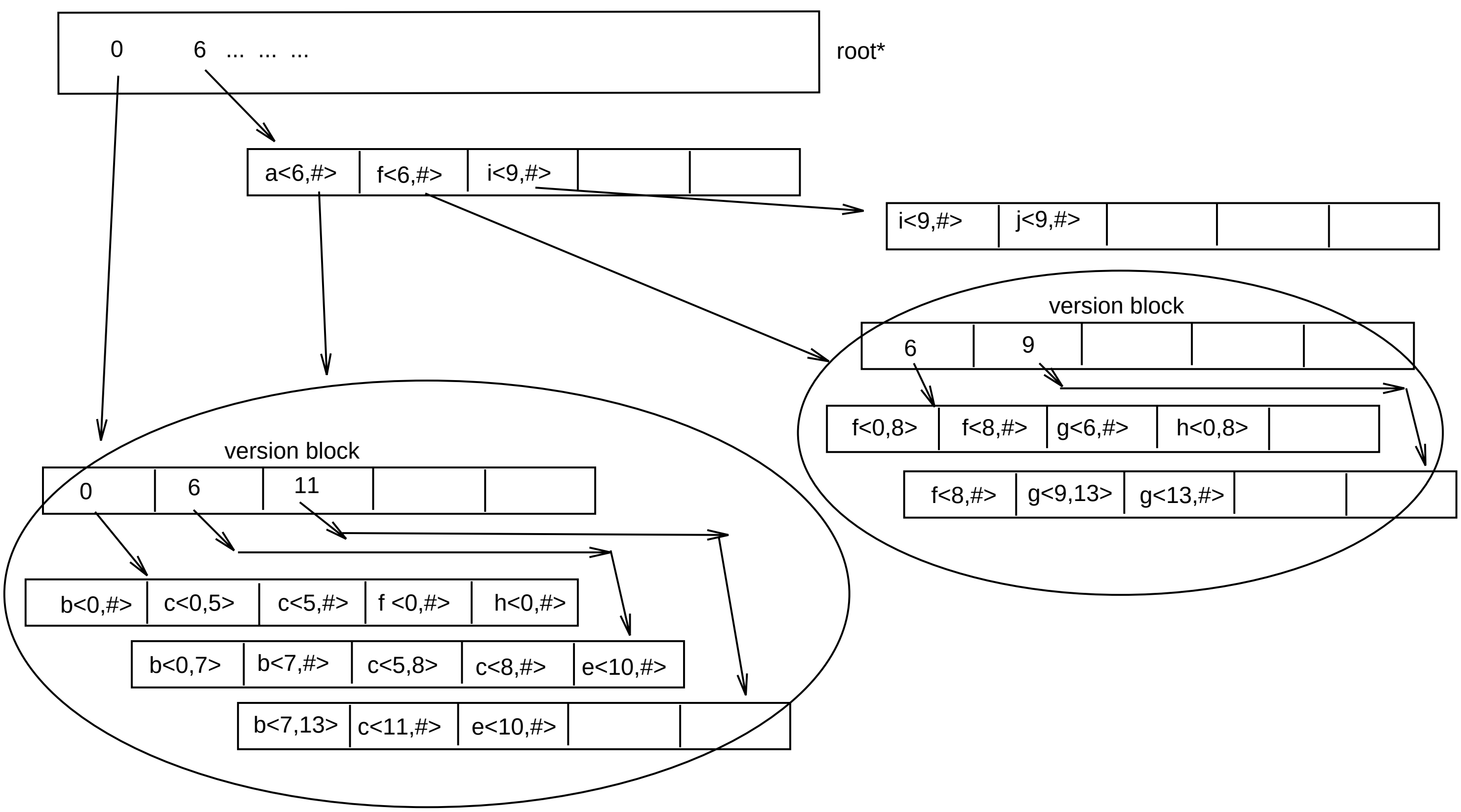
Figure 20: An example of a split on a fat node in the Persistent B-Tree.

still uses version blocks.

If a page becomes sparse, from too many deletions, a node consolidation algorithm is presented.



A copy is made when a page overflows. Page capacity is five records. If there are four or more records in the new node, a key split is made after the copy. The root* structure points to the root for a given timestamp. Fat fields contain a key, a begin time, an end time and data. We shall not show the data. The "now" end time is represented with a # sign.



The fat field method stores records from several versions as long as they fit in a page. At time 5, a new version of record c is placed in the page. The old version gets 5 as its new end time. At time 6, overflow occurs. There are four distinct keys in the new node, so a key split takes place. At time 9, when i and j are inserted and g is updated, another copy and key split occur. This causes the new root to obtain a new index entry. At time 11, a new version of record c causes an overflow with only a copy, not a key split. Fat nodes are created when the first copy operation is made. Search for a given time follows pointers which begin before or at the search time and do not end before the search time. Search in a version block (or in root*) follows the largest time before or equal the search time.

Figure 21: The fat-field method of the Persistent B-Tree.

The version making the last deletion copies all information relative to that version to a new node. Then a sibling node also has its information relative to that version copied to the new node. If it is necessary, the new node is then key-split.

Technically speaking, the possibility of thrashing by continually consolidating and splitting the same node could cause the space usage to become $O(n)$ not $O(n/B)$. This could happen by inserting a record in a node, causing it to time-and-key split, then deleting a record from one of the new current nodes and causing a node consolidation, which creates a new full current node, and so forth. A solution for thrashing appears in [MS81]. Basically, the threshold for node consolidation is made lower than half the threshold for node splitting. Since this is a rather pathological scenario, we will continue to assume that the space usage for the fat-fields variation of the Persistent B-tree is $O(n/B)$.

For moving historical data to another medium, observe that time splitting by current transaction time as performed in the Persistent B-Tree means that nodes cannot be moved once they are created unless all the parents (not just the current one) are updated with the new address of the historical data. Only the TSB-tree solves this problem by splitting index nodes before the time of the earliest start time of their current children. Thus in the TSB-tree when a current node is time-split, the historical data can be moved to another disk. In the TSB-tree, current nodes have only one parent.

Fat nodes are not necessary for partial persistence. This is observed in [DSST89], where "node-copying" for partially persistent structures is discussed.

The reason fat nodes are not needed is that although alive (current) nodes have many parents, only one of them is current. So when a current node is copied or split, only its current parent has to be updated. The other parents will correctly refer to its contents as of a previous version. The fact that new items may have been added does not affect correctness of search. Since nodes are always time split (most recent versions of items copied) by current transaction time, no information is erased when a time split is made.

Both approaches of the Persistent B-Tree use a version block inside each fat-node. If the node in question is never key-split (that is, all changes are applied to the same ephemeral B^+-tree node), new version block pages may be created for this node without updating the parent's version block. Thus when answering a query, all encountered version blocks have to be searched for time *t*. In comparison, the MVBT and MVAS we discuss next use "node-copying" and thus have better asymptotic query time ($O(\log_B n+a/B)$).

5.1.3.5 The Multiversion B-Tree and the Multiversion Access Structure

The Multiversion B-Tree of [BGO+93] and the Multiversion Access Structure of [VV95] provide another approach to partially persistent B^+-trees. Both structures have the same asymptotic behavior but the MVAS improves the constant of MVBT's space complexity. We first discuss the MVBT and then present its main differences with the MVAS.

The MVBT is similar to the WOBT, however it efficiently supports deletions (as in [DSST89] and [LM91]). Supporting deletions efficiently implies use of node consolidation. In addition, the MVBT uses a form of node-copying [DSST89] and disallows thrashing.

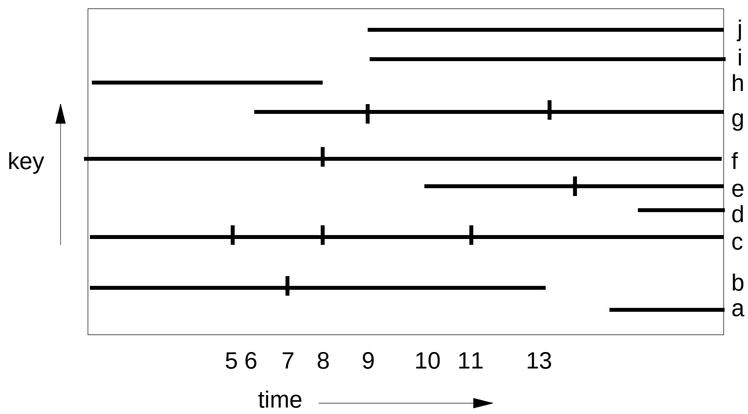
As with the WOBT and the Persistent B-tree, it uses a root* structure. When the root does a time-split, the sibling becomes a new root. Then a new entry is placed in the variable length array root*, pointing to the new root. If the root does a time-and-key split, the new tree has one more level. If a child of the root becomes sparse and merges with its only sibling, the newly merged node becomes a root of a new tree.

Figures 21 and 22 illustrate some of the similarities and differences between Persistent B-tree, the MVBT and the WOBT. To better illustrate the similarities, we picture the WOBT in Figure 22 with end times and start times in each record version. In the original WOBT, end times of records were calculated from the begin times of the next version of the record with the same key. If no such version was in the node, the end time of the record was known to be after the end time of the node.

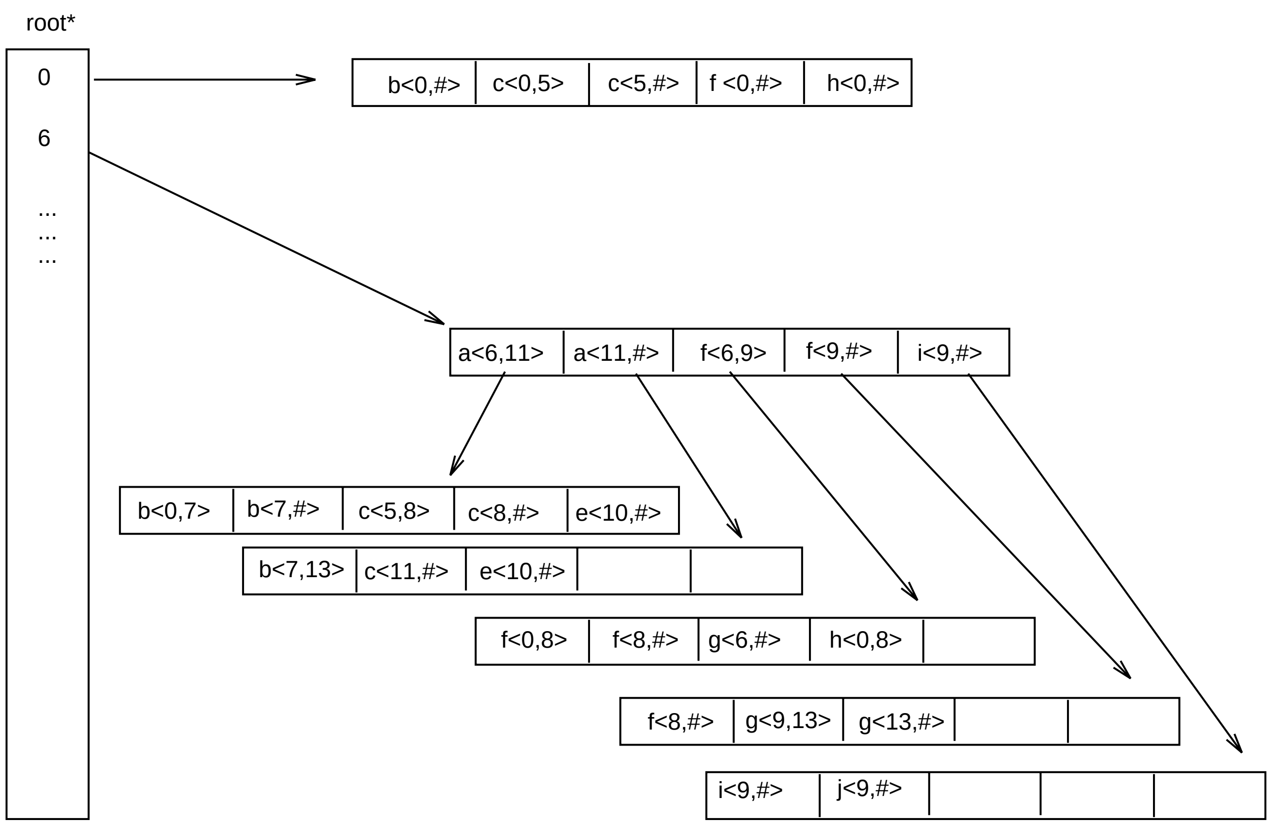
In all three methods, if we have no node consolidations, the data nodes are exactly the same. In all three methods, when a node becomes full, a copy is made of all the records "alive" as of the time of the version making the update which causes the overflow. If the number of distinct records in the copy is above some threshold, the copy is split into two nodes by key.

The Persistent B-tree creates a fat node when a data node is copied. The WOBT and the MVBT do not create fat nodes. Instead, as illustrated in Figure 22, information is posted to the parent of the overflowing data node. A new index entry or two new index entries which describe the split are created. If there is only one new data node, the key used as the lower limit for the overflowing child is copied to the new index entry. The old child pointer gets the time of the copy as its end time and the new child pointer gets the split time as its start time. If there are two new children, they both have the same start time, but one has the key of the overflowing child and the other has the key used for the key split.

A difference between the Persistent B-tree, the WOBT, the MVBT on one hand and the TSB



A copy is made when a page overflows. Page capacity is five records. If there are four or more records in the new node, a key split is made after the copy. The root* structure points to the root for a given timestamp. Records contain a key, a begin time, an end time and data. We shall not show the data. The "now" end time is represented with a # sign.



At time 5, a new version of record c is placed in the page. The old version gets 5 as its new end time. At time 6, overflow occurs. There are four distinct keys in the new node, so a key split takes place. At time 9, when i and j are inserted and g is updated, another copy and key split occur. This causes the new root to obtain two new index entries. At time 11, a new version of record c causes an overflow with only a copy, not a split. There is one new index entry.

Figure 22: The Multiversion B-Tree and the Write-Once B-Tree. (For simplicity of comparison, both the end and start times appear in each record, which is not needed in the original WOBT).

on the other is that the TSB does not have root*. When the only root in the TSB does a time split, a new level is placed on the tree to contain the information about the split. When the root in the MVBT does a time-split, root* obtains a new entry. When the root in the (fat-field) Persistent B-tree does a time split, that root fat node obtains a new page and a new entry in the version block. (Only when the root fat node requires a key split or a merge so that a new root fat node is constructed, does root* obtain a new entry in the Persistent B-tree.)

Another difference with the WOBT is that the MVBT and the Persistent B-tree use a node consolidation algorithm. When a node is sparse, it is consolidated with a sibling by time- splitting (copying) both the sparse node and its sibling and then combining the two copies, possibly key splitting if necessary.

In addition, the MVBT disallows thrashing (splitting and consolidating the same node continually) by suggesting that the threshold for splitting be higher than twice the threshold for consolidation. The Persistent B-tree does not disallow thrashing. This is not an issue with the WOBT, since the WOBT does no node consolidation.

Search in root* for the correct root in MBVT is $O(\log_B n)$. Although the example illustrated in Figure 22 has a small root*, there is no reason why this should always be the case. One need only imagine a database with one data node with records that are continually updated, causing the root (which is also the data node) to continually time-split. So, if the root* becomes too large, a small index has to be created above it.

In the MBVT, transaction range timeslice search ("range/-/point") is $O(\log_B n+a/B)$ since search for the root is itself $O(\log_B n)$. The MVBT has $O(\log_B m)$ amortized update cost (where m denotes now the size of the current B^+-tree on which the update is performed), and $O(n/B)$ space usage. Thus the MVBT provides the I/O-optimal solution to the transaction range timeslice problem. The update cost is amortized because of the updating needed to maintain efficient access to the root* structure.

The WOBT is not optimal because deletions of records can cause pages to be sparsely populated as of some recent times. Thus transaction range-timeslice searches may become inefficient. Insertions in the WOBT and in the MVBT are $O(\log_B m)$ since a special pointer can be kept to the root of the tree containing all current records (which are $O(m)$).

The MVBT uses more space than the WOBT, which in turn uses more space than the TSB-tree. In order to guard against sparse nodes and thrashing the MVBT policies create more replication (the constant in the $O(n/B)$ space worst case bound of the method is about 10.)

Probably the best variation of the WOBT is to use some parameters to decide whether to time split, time-and-key split, key split, time-split and merge or time-split, merge and key-split. These parameters will depend on the minimum number of versions alive at any time in the interval spanned by the page. All of the policies pit disk space usage against query time. A pure key split creates one new page. A time-and-key split creates two new pages: one new historical page and one new current page. The historical page will have copies of the current records, so more copies are made than when pure key splits are allowed. Node consolidation creates at least two new historical pages. However, once a minimum number of records is guaranteed to be alive for any given version in all pages, range-timeslice queries will be $O(\log_B n+a/B)$ and space usage will be $O(n/B)$. The different splitting policies will affect the total amount of space used and the average number of copies of record versions made.

The Multiversion Access Structure (MVAS) [VV95] 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 [VV95]. 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.

Since the WOBT, TSB-tree, Persistent B-tree, MVBT and MVAS are similar in their approach towards solving range-timeslice queries, we summarize their characteristics in Table A. The issues of time-split, key-split, time- and key-split, sparse nodes, thrashing and history migration are closely related.

The pure-key query was not addressed in the works of [DSST89], [LM91] and [BGO+93]; however, the technique that keeps with each record the address of any one copy of the most recent

Table A: The basic characteristics of WOBT, TSB-tree, Persistent B-tree, MVBT and MVAS.

	time split [1]	pure key split [2]	time/key split	sparse node merge	prevent thrashing [3]	root* [4]	history migrate [5]
WOBT	yes	no	yes	no	N.A.	yes	no
TSB−Tree	yes	yes	no	no	N.A.	no	yes
Persistent B−tree	yes	no	yes	yes	no	yes	no
MVBT/MVAS	yes	no	yes	yes	yes	yes	no

Notes

1. All methods time−split (copy) data and index nodes. The TSB−tree can time−split by other than current time.
2. The TSB−Tree does pure key splits. The other methods do time−and−key splits. Pure key splits use less total space, but risk poor performance on past−time queries.
3. Thrashing is repeated merging and splitting of the same node. Only the MBVT prevents thrashing by choice of splitting and merging thresholds. Prevention of thrashing is not needed when there is no merging.
4. The use of root* enables the current tree search to be more efficient by keeping a separate pointer to its root. Past time queries must search within root*, so are not more efficient than methods without root*.
5. Only the TSB−tree has only one reference to current nodes, allowing historical data to migrate.

distinct previous version of the record can avoid going through all copies of a record. The pure-key query (with time predicate) is then addressed in $O(\log_B n + a)$ I/O's, just as proposed for the WOBT (where a represents the number of different versions of the given key).

As discussed in section 5.1.1, [VV95] solves the pure-key query (with time predicate) in optimal $O(\log_B n + a/B)$ query time using C-lists. Despite their extra complexity in maintenance, an advantage of the C-lists is that they can be combined with the main MVAS method.

5.1.3.6 Exodus and Overlapping B^+-Trees

The Overlapping B^+-tree[MK90, BHK85] and the Exodus large storage object [RCDS86] are similar. Here we begin with a B^+-tree. When a new version makes an update in a leaf page, copies are made of the full path from the leaf to the root, changing references as necessary. Each new version has a separate root and subtrees may be shared (Figure 23).

Space usage is $O(n\log_B n)$ since new pages are created for the whole path leading to each data page updated by a new version. Update processing is $O(\log_B m)$ where m is the size of the current tree being updated. Timeslice or range-timeslice query time depends on the time needed to find the correct root. If non consecutive transaction timestamps of events are used it is $O(\log_B n+a/B)$.

Even though pure-key queries of the form “find the previous salary history of employee A who was alive at *t*” (i.e., with time predicate) are not discussed, they can in principle be addressed in the same way as with the other B$^+$-tree based methods by linking together data records.

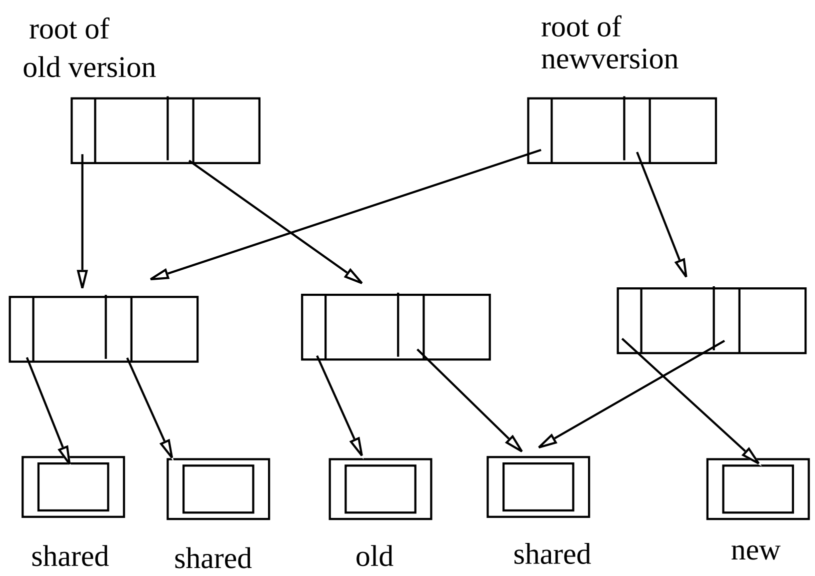


Figure 23: The Overlapping tree/Exodus structure.

5.1.3.7 Multiattribute Indexes

Suppose that the transaction start time, transaction end time and database key are used as a triplet key for a multiattribute point structure. If this structure clusters records in disk pages by closeness in several attributes, one can obtain efficient transaction pure-timeslice and range-timeslice queries, using only one copy of each record.

Records with similar values of start time, end time and key will be clustered together in disk pages. Having both a similar start time and a similar end time means that long-lived records will be in the same page as other long-lived records. These records will be answers to many timeslice queries. Short lived records will only be on the same pages if their short lives are close in time. These will contain many correct answers to timeslice queries with time values in the short interval their entries span. Every timeslice query will access some of the long-lived record pages and a small proportion of the short-lived record pages. Individual timeslice queries will not need to access most of the short-lived record pages as they will not intersect the timeslice.

There are some subtle problems with this. Suppose a data page is split by start time. In one of the pages resulting from the split, all the record versions whose start time is before the split time are stored. This page has an upper bound on start time, implying that no new record versions can be inserted. All new record versions will have a start time after *now*, which is certainly after the split time. Further, if there are current records in this page, their end time will continue to rise, so the lengths of the time spans of records in this page will be variable.

Some will be long and others short. Queries as of current transaction time may only retrieve a few (or no) records from a page which has been limited by an upper bound on start time. This is illustrated in Figure 24. Many such pages may have to be accessed in order to answer a query, each one contributing very little to the answer (that is, the answer is not well clustered in pages).

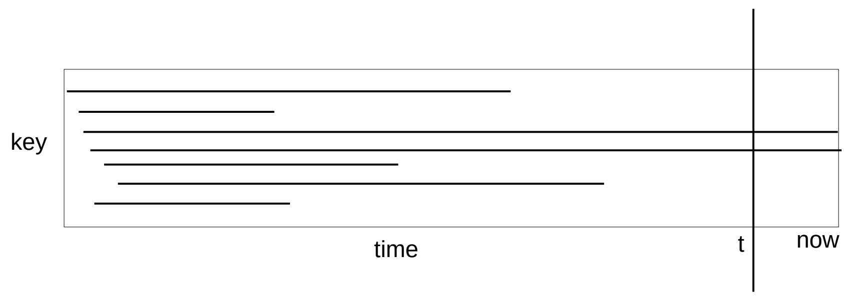


Figure 24: Storing data with similar start_times.

Also, when a new version is created, its start time is often far from the start time of its predecessor (the previous version with the same key). So consecutive versions of the same record are unlikely to be on the same page if start-time splits are used.

Now suppose we decide that splitting by start time is a bad idea and we split only by key or by end time. Splitting by end time enables migration of past data to a WORM disk. However, a query as of a past transaction time may only retrieve a small number of records if the records are placed only by the requirement of having an end time before some cut-off value just as in Figure 12.

Current pages (which have been split by key) can contain versions whose lifetimes are very long and versions whose lifetimes are very short. This also makes past-time queries inefficient.

All of these subtle problems come from the fact that many records are still current and have growing lifetimes and all new record versions have increasing start times. Perhaps if we use a point-based multiattribute index for dead versions only, efficient clustering may be possible. Here newly dead record versions can be inserted in a page with an upper limit on start time because the start times may have been long ago. Items can be clustered in pages by key, nearby start times and nearby end times. No guarantees can be made that a query as of a given time will hit a minimum number of record versions in a page, however. For example, imagine a page with record versions with very short lifetimes all of which are close by, but none of which overlap.

Although no guarantees of worst case search time can be made, the advantages of having only one copy of each record and having no overlapping of time-key space, so that backtracking is not necessary, may make this approach worthwhile at least for "dead" versions. Space usage is thus linear (space would be $O(n/B)$ if in addition the multiattribute method can guarantee that index and data pages have good space utilization). A method for migrating current data to the WORM and of organizing the current data for efficient temporal queries would be needed if the multiattribute method was used only for past data.

5.1.4 Summary

The worst case performance of the transaction-time methods is summarized in Table B. The reader should be cautious when interpreting worst case performance. Sometimes the notation penalizes a method for its performance on a pathological scenario. The footnotes indicate such cases.

Table B: The performance characteristics of the examined transaction-time methods.

Access Method (related section)	*Total Space*	*Update per change*	*Pure-key Query*	*Pure-Timeslice Query*	*Range-Timeslice Query*
AP-Tree (5.1.2.1)	$O(n/B)$	$O(\log_B n)$ [1]	*N/A*	$O(n/B)$	$O(n/B)$
ST-Tree (5.1.2.1)	$O(n/B)$	$O(\log_B S)$ [2]	$O(\log_B S+a)$ [2]	$O(S\log_B n)$ [2]	$O(K\log_B n)$ [3]
Time-Index (5.1.2.2)	$O(n^2/B)$	$O(n/B)$	*N/A*	$O(\log_B n+a/B)$	$O(\log_B n+s/B)$ [4]
Two-level Time Index (5.1.2.2)	$O(n^2/B)$	$O(n/B)$	*N/A*	$O(R\log_B n+a)$ [5]	$O(M\log_B n+a)$ [6]
Checkpoint Index (5.1.2.4) [7]	$O(n/B)$	$O(n/B)$	*N/A*	$O(n/B)$	$O(n/B)$
Archivable Time Index (5.1.2.5) [8]	$O(n/B)$	$O(\log_B n)$	*N/A*	$O(\log_2 n+a/B)$	$O(\log_2 n+s/B)$ [4]
Snapshot Index (5.1.2.6)	$O(n/B)$	$O(1)$ [9]	$O(a)$ [10]	$O(\log_B n+a/B)$	$O(\log_B n+s/B)$ [4]
Windows Method (5.1.2.7)	$O(n/B)$	$O(\log_B n)$	$O(\log_B n + a)$	$O(\log_B n+a/B)$	$O(\log_B n+s/B)$ [4]
R-Trees (5.1.3.1)	$O(n/B)$	$O(\log_B n)$	$O(n/B)$ [11]	$O(n/B)$ [11]	$O(n/B)$ [11]

Access Method (related section)	*Total Space*	*Update per change*	*Pure-key Query*	*Pure-Timeslice Query*	*Range-Timeslice Query*
SR-Tree (5.1.3.1)	$O((n/B)\log_B n)$	$O(\log_B n)$	$O(n/B)$ [11]	$O(n/B)$ [11]	$O(n/B)$ [11]
WOBT (5.1.3.2) [12]	$O(n/B)$	$O(\log_B m)$ [13]	$O(\log_B n+a)$ [14]	$O(\log_B n+a/B)$	$O(\log_B n+a/B)$
TSB-Tree (5.1.3.3)	$O(n/B)$	$O(\log_B n)$	$O(\log_B n+a)$ [14]	$O(n/B)$ [15]	$O(n/B)$ [15]
Persistent B-tree/ Fat Node (5.1.3.4)	$O(n)$	$O(\log_B m)$ [13]	$O(\log_B n \log_B m+a)$ [14,16]	$O(\log_B n(\log_B m + a/B))$ [16]	$O(\log_B n(\log_B m +a/B))$ [16]
Persistent B-tree/ Fat Field (5.1.3.4)	$O(n/B)$	$O(\log_B m)$ [13]	$O(\log_B n \log_B m+a)$ [14,16]	$O(\log_B n(\log_B m +a/B))$ [16]	$O(\log_B n(\log_B m +a/B))$ [16]
MVBT (5.1.3.5)	$O(n/B)$	$O(\log_B m)$ [13]	$O(\log_B n+a)$ [14]	$O(\log_B n+a/B)$	$O(\log_B n+a/B)$
MVAS (5.1.1 & 5.1.3.5)	$O(n/B)$	$O(\log_B m)$ [13]	$O(\log_B n+a/B)$ [14,17]	$O(\log_B n+a/B)$	$O(\log_B n+a/B)$
Overlapping B-Tree (5.1.3.6)	$O(n\log_B n)$	$O(\log_B m)$ [13]	$O(\log_B n+a)$ [14]	$O(\log_B n+a/B)$	$O(\log_B n+a/B)$

1. This is the time needed when the end_time of a stored interval is updated; it is assumed that the start_time of the updated interval is given. If intervals can be identified by some key attribute, then a hashing function could find the updated interval at $O(1)$ expected amortized time. In the original paper it was assumed that intervals can only be added and in increasing start_time order; in that case the update time is $O(1)$.
2. Where S denotes the number of different keys (surrogates) ever created in the evolution.
3. Where K denotes the number of keys in the query key range (which may or may not be alive at the time of interest).
4. Where s denotes the size of the whole timeslice for the time of interest. No separation of the key space in regions is assumed.
5. Where R is the number of predefined key regions.
6. Assuming that a query contains a number of predefined key-regions, M denotes the number of regions in the query range.
7. The performance is under the assumption that the Checkpoint Index creates very few checkpoints and the space remains linear. The update time is $O(n/B)$ since when the end_time of a stored interval is updated, the interval has to be found. As with the AP-Index, if intervals can be identified by some key attribute, then a hashing function could find the updated interval at $O(1)$. The original paper did not deal with this issue since it was implicitly assumed that interval endpoints are known at insertion. If checkpoints are often then the method will behave as the Time Index.
8. For the update it is assumed that the start_time of the updated interval is known. Otherwise, if intervals can be identified by some key, a hashing function could be used to find the start_time of the updated interval. For the range-timeslice query, we assume no extra structure is used. The original paper proposes using an approach similar to the Two-Level Time Index or the ST-Tree.
9. In the expected amortized sense, using a hashing function on the object key space. If no hashing but a B-tree is used then the update becomes $O(\log_B m)$ where m is the size of the current state, on which the update is performed.
10. Assuming as in 9 that a hashing function is used. If a B-tree is used the query becomes $O(\log_B S + a)$ where S is the total number of keys ever created.
11. This is a pathological worst case, due to the non-guaranteed search on an R-tree based structure. In most cases the avg. performance would be $O(\log_B n+a)$. Note that all the R-tree related methods assume both interval endpoints are known at insertion time.
12. Here we assume that the WOBT tree is implemented thoroughly on a magnetic disk, and that no (or infrequent) deletions occur, i.e., just additions and updates.
13. In the amortized sense, where m denotes the size of the current tree being updated.
14. For a pure key query of the form: “find the previous salaries of employee *A* who existed at time *t*”.

15. This is a pathological worst case, where only key splits are performed. If a time-split is performed before a key-split when nodes resulting from a pure key split would have too few records "alive" at the begin time of the node, then the query takes $O(\log_B n+a)$, also assuming infrequent deletions.
16. Where m denotes the size of the ephemeral B+-tree at the time of interest.
17. The pure-key query performance assumes the existence of the *C-lists* on top of the MVAS structure.

5.2 Valid-Time Methods

According to the valid-time abstraction presented in section 2, a valid-time database should maintain a dynamic collection of interval-objects. Very recently, [AV96] presented an I/O optimal solution for the "*/point/-" query. The solution (the *External Interval Tree*) is based on a main-memory data structure, the Interval Tree [E83], that is made external (disk-resident). Valid timeslices are supported in $O(l/B)$ space, using $O(\log_B l)$ update per change (interval addition, deletion or modification) and $O(\log_B l + a/B)$ query time. Here l is the number of interval-objects in the database when the update or query is performed. Even though it is not clear how practical the solution would be (various details are not included in the original paper) the result is very interesting. To optimally support valid timeslices is a rather difficult problem because in a valid-time environment the clustering of data in pages can dramatically change by the updates. Deletions are now physical and insertions can happen anywhere in the valid-time domain. In contrast, in a transaction-time environment objects are inserted in increasing time order and after their insertion they can be "logically" deleted but they are not removed from the database.

A valid timeslice query ("*/point/-") is actually a special case of a 2-dimensional range query. Note that an interval contains a query point v, if and only if, its start_time is less than or equal to v and its end_time is greater than or equal to v. Let us map an interval $I = (x_1, y_1)$ into a point (x_1, y_1) in the 2-dimensional space. Then an interval contains query v, if and only if, its corresponding 2-dimensional point lies inside the box generated by the lines $x = 0$, $x = v$, $y = v$ and $y = \infty$ (Figure 25). Since an interval's end_time is always greater or equal than its start_time, all intervals are represented by points above the diagonal $x = y$. This 2-dimensional mapping is used in the Priority Search Tree [McC85] the data structure which provides the main-memory optimal solution. A number of attempts have been made to externalize this structure [KRVV93, IKO87, BG90].

[KRVV93] uses the above 2-dimensional mapping to address two problems: indexing constraints and indexing classes in an I/O environment. Constraints are represented as intervals that can be added, modified or deleted. The problem of indexing constraints is then reduced to the dynamic interval management problem, i.e., the "*/point/-" query! For solving the dynamic

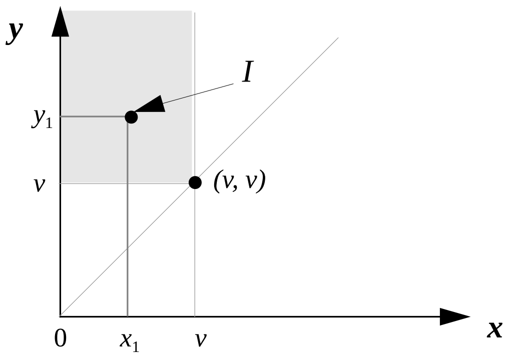


Figure 25: An interval is translated into a point in a 2-dimensional space. Axes x and y represent an interval's starting and ending valid-times. Intervals that intersect valid instant v correspond to the points included in the shaded area.

interval management problem, [KRVV93] introduces a new access method, the *Metablock Tree*, which is a B-ary access method that partitions the upper diagonal of the 2-dimensional space into *metablocks*, each of which with B^2 data points (the structure is rather complex; for details we refer to [KRVV93]). Note however that the Metablock Tree is a *semi-dynamic* structure, since it can support only interval insertions (no deletions). It uses $O(l/B)$ space, $O(\log_B l + a/B)$ query time and $O(\log_B l + (\log_B l)^2/B)$ amortized insertion time. The insertion bound is amortized since the maintenance of a metablock's internal organization is rather complex to be performed after each insertion. Instead, metablock reorganizations are deferred until enough insertions have accumulated. If interval insertions are random, the expected insertion time becomes $O(\log_B l)$.

[IKO87] and [BG90] present two other external implementations of the Priority Search Tree. Both use optimal space $(O(l/B))$; [IKO87] has $O(\log_2 l + a/B)$ query time I/O's for valid timeslices, while [BG90] has $O(\log_B l + a)$ query time.

In [RS94], a new technique called *path caching* is introduced for solving 2-dimensional range queries. This technique is used to turn various main-memory data structures, like the Interval Tree, or the Priority Search Tree [McC85] into external structures. With this approach, the "*/point/-" query is addressed in $O(\log_B l + a/B)$ query time, $O(\log_B l)$ amortized update time (including insertions and deletions) but in $O(\frac{l}{B}\log_2 \log_2 B)$ space.

The above approaches are aimed at good worst case bounds but lead to rather complex structures. Another main-memory data-structure that solves the "*/point/-" query is the Segment Tree [B77] which however uses more than linear space. [BG94] presents the *External Segment Tree* (EST) which is a paginated version of the Segment Tree. We first describe the worst case performance of the EST method. If the endpoints of the valid-time intervals take values from a universe of size V (i.e., there are V possible endpoint values), the EST supports "*/point/-" queries using $O(\frac{l}{B}\log_2 V)$ space, $O(\log_2 V)$ update per change and $O(\log_2 V + a)$ query time. [BG94] presents also an extended analysis of the expected behavior of the EST under the assumption of a uniformly distributed set of intervals of fixed length. It is shown that the expected behavior is much better; the average height of the EST is for all practical purposes small (this affects the logarithmic portion of the performance) and the answer is found by accessing an additional $O(a/B)$ pages.

An advantage of the External Segment Tree is that the method can be modified to also address queries with key predicates (like the “range/point/-” query). This is performed by embedding B-trees in the EST. The original EST structure guides the search to a subset of intervals that contain the query valid time *v* while an embedded B-tree allows to search this subset for whether the query key predicate is also satisfied. For details we refer to [BG94].

Good average case performance could also be achieved by using a dynamic multidimensional access method. If only multidimensional points are supported as in the k-d-B-tree [R84] or the h-B-tree [LS90b], mapping an (interval, key) pair to a triplet consisting of (start_time, end_time, key) as discussed above, would allow the valid intervals to be represented by points in three-dimensional space.

If intervals are represented more naturally, as line segments in a two dimensional key-time space, the cell-tree [G89], the R-tree or one of its variants, the R* [BKKS90] or the R+ [SRF87] could be used. Such solutions should provide good average case performance, but overlapping still remains a problem especially if the interval distribution is highly non-uniform (as observed in [KS91] for R-trees). If the SR-tree [KS91] is utilized for valid-time databases the overlapping is decreased but the method may suffer if there are many interval deletions, since all remnants (segments) of a deleted interval have to be found and physically deleted.

Another possibility would be to facilitate a two level method whose top level indexes the key attribute of the interval objects (using a B^+-tree) while the second level indexes the intervals that share the same key attribute. An example of such method is the ST-Index [GS93]. In the ST-Index there is a separate AP-Tree that indexes the start_times of all valid-time intervals sharing a distinct key attribute value. The problem with this approach is that a “*/point/-” query will have to check all stored intervals for whether they include the query valid-time *v*.

The Time-Index [EWK90] may also be considered for storing valid-time intervals however there are two drawbacks. First, changes can arrive in any order so leaf entries anywhere in the index may have to merge or split thus affecting their relevant timeslices. Second, updating may be problematic as deleting (or adding or modifying the length of) an interval involves updating all the stored timeslices that this interval overlaps.

[NDK96] offers yet another approach to indexing valid-time databases, the *MAP21* structure. A valid-time interval *(x, y)* is mapped to a point $z = x10^s + y$, where *s* is the maximum number of digits needed to represent any time point in the valid-time domain. This is enough to map each interval to a separate point. A regular B-tree is then used to index these points. An advantage of this approach is that interval insertions/deletions are easy using the B-tree. However, to answer a valid timeslice query about time *v* the point closer to *v* is found in the B-tree and then a sequential search for all intervals before *v* is performed. At worse many intervals that do not intersect *v* can be found ([NDK96] assumes that in practice the maximal interval length is known, which limits how far back the sequential search continues from *v*).

Further research is needed in this area. An interesting open problem is whether an I/O optimal solution exists for the "range/point/-" query (valid range timeslices).

5.3 Bitemporal Methods

As mentioned in section 4.5, one way to address bitemporal queries is to fully store some of the $C(t_i)$ collections of Figure 3, together with the changes between these collections. To exemplify searching through the intervals of a stored $C(t_i)$, an access method for each stored $C(t_i)$ is also included. These $C(t_i)$'s (and their accompanying methods) can then be indexed by a regular B-tree on t_i, the transaction time. This is the approach taken in the *M-IVTT* [NDE96]; the changes between stored methods are called "patches" and each stored $C(t_i)$ is indexed by a MAP21 method [NDK96].

The M-IVTT approach can be thought as an extension of the Time-Index [EWK90] to a bitemporal environment. Depending on how often $C(t_i)$'s are indexed the space/update or the query time of the M-IVTT will increase. For example, the space can easily become quadratic if the indexed $C(t_i)$'s are every constant number of changes and each change is the addition of a new interval.

In another approach, the intervals associated with a bitemporal object can be "visualized" as a bounding rectangle which is then stored in a multidimensional index, like the R-tree [G84] (or some of its variants, like the SR-Tree [KS91]). 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 [KTF95b]. All bitemporal objects which have not been "deleted" (in the transaction sense) are represented with a transaction-time endpoint extending to *now* (Figure 4).

To avoid this overlapping, the use of two R-trees (2-R approach) has been proposed [KTF95b].

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', $(t' > t)$ 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 with known transaction-time interval (Figure 26, taken from [KTF95b]). 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.

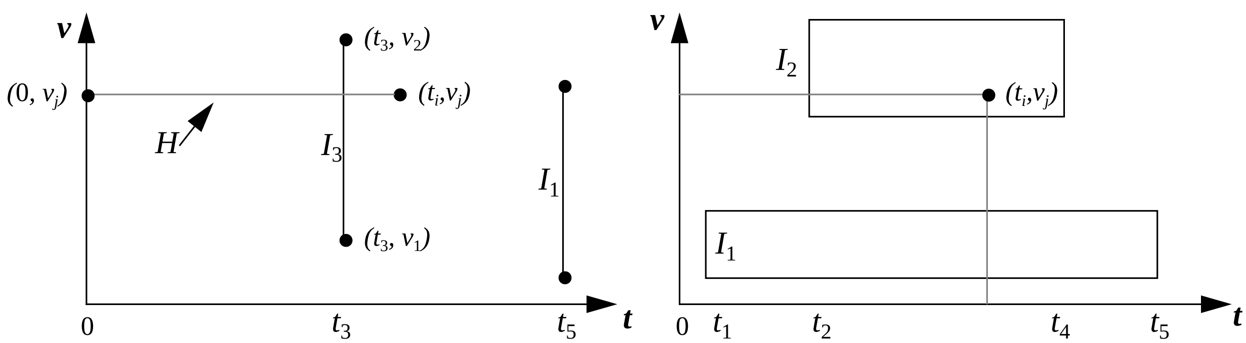


Figure 26: In the 2-R-tree approach bitemporal data is divided according to whether their right transaction endpoint is known. The scenario of Fig. 3 is presented here (i.e., after time t_5 has elapsed). The left 2-dimensional space is stored in the *front* R-tree while the right in the *back* R-tree.

A "*/point/point" query about (t_i, v_j) is then 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. Interval H starts from the beginning of transaction time and extends until point t_i at height v_j (Figure 26). To support "range/range/range" queries, an additional third dimension for the key ranges is added in both R-trees.

The usage of two R-trees is reminiscent of the Dual-Root Mixed Media R-tree proposed in [KS89] as a mixed-media index that stores intervals and consists also of two R-trees. There, new intervals are stored on one R-tree and are gradually moved to the second R-tree. There are however the following differences: (a) in the Dual-Root Mixed Media R-tree intervals inserted have both their endpoints known in advance (which is not a characteristic of transaction-time); (b) both R-trees in [KS89] store intervals with the same format; (c) the transferring of data in the Dual-Root

Mixed Media R-tree is performed in a batched way. When the first R-tree reaches a threshold near its maximum allocated size, a *vacuuming* process completely vacuums all the nodes of the first R-tree (except its root) and inserts them to the second R-tree. In contrast, transferring of a bitemporal object in the 2-R approach is performed whenever this object is deleted in the transaction-time sense. Such a deletion can happen to any currently "alive" object in the front R-tree.

Bitemporal problems can also be addressed by the partial persistence approach; this solution emanates from the abstraction of a bitemporal database as a sequence of history-timeslices *C(t)* (Figure 3) and has two steps. First, a good ephemeral structure is chosen to represent each *C(t)*. This structure must support dynamic addition/deletion of (valid-time) interval-objects. Second, this structure is made partially persistent. The collection of queries supported by the ephemeral structure implies what queries are answered by the bitemporal structure.

The main advantage obtained by "viewing" a bitemporal query as a partial persistence problem is that the valid-time requirements are disassociated from the transaction-time ones. More specifically, the valid time support is provided from the properties of the ephemeral structure 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.

The partial persistence methodology was also used in [LM91, BGO+93, VV95] for the design of transaction-time access methods. For a transaction-time environment the ephemeral structure must support dynamic addition/deletion of plain-objects; hence a B-tree is the obvious choice. For a bitemporal environment two access methods have been proposed: the Bitemporal Interval Tree [KTF95a] which is created by making an Interval Tree [E83] partially persistent (and well paginated), and, the Bitemporal R-Tree [KTF95b] created by making an R-tree partially persistent.

The Bitemporal Interval Tree is designed for the "*/point/point" and "*/range/point" queries. Answering such queries implies that the ephemeral data structure should support point-enclosure and interval-intersection queries, respectively. In the absence of an <u>external</u> ephemeral method that optimally solves these problems [KRVV93, RS94], a main-memory data structure, the Interval Tree (which optimally solves the in-core versions of the above problems) was used and was made partially persistent and well paginated. One constraint of the Bitemporal Interval Tree is that the universe size *V* on the valid domain is known in advance. The method computes "*/point/point" and "*/range/point" queries in $O(\log_B V + \log_B n + a)$ I/O's. The space is $O((n+V)/B)$; the update is amortized $O(\log_B(m + V))$ I/O's per change. Here *n* denotes the total number of changes, *a* is the answer size and m is the number of intervals contained in the current timeslice $C(t)$ when the change is performed.

The Bitemporal R-Tree does not have the valid-universe constraint. It is a method designed for the more general "range/point/point" and "range/range/point" bitemporal queries. For that purpose, the ephemeral data structure must support range point-enclosure and range interval-intersection queries on interval-objects. Since neither a main-memory, nor an external data structure exists with good worst-case performance for this problem, the R*-tree [BKKS90] was used, an access method that has good average-case performance for these queries. As a result, the performance of the Bitemporal R-Tree is bound by the performance of the ephemeral R*-tree. This is because a method created by the partial-persistence methodology behaves asymptotically as the original ephemeral structure.

[KTF95b] contains various experiments comparing the average case performance of the 2-R methodology, the Bitemporal R-tree and the obvious approach which stores bitemporal objects in a single R-tree (the 1-R approach, as in Figure 4). Due to the limited copying introduced by partial persistence, the Bitemporal R-tree uses some small extra space (about double the space used by the 1-R and 2-R methods) but it has much better update and query performance. Similarly, the 2-R approach has in general better performance than the 1-R approach.

It remains an interesting open problem to find the theoretically I/O optimal solutions even for the simplest bitemporal problems, like the "*/point/point" and "*/range/point" queries.

6. Conclusions

We presented a comparison of various temporal access methods. While we have also covered valid-time and bitemporal approaches, the bulk of this paper addresses transaction-time methods as they represent the majority among the published approaches. Since it is practically impossible to run simulations of all methods under the same input patterns, our comparison was based on the worst case performance of the examined methods. Comparison items included the space requirement, the update characteristics and the query performance. The query performance is measured against three basic transaction-time queries, the pure-key, the pure-timeslice and the range-timeslice queries, or, using the three-entry notation, the "point/-/*", the "*/-/point" and the "range/-/point" queries respectively. In addition we addressed problems like index pagination, data clustering and the ability of a method to efficiently migrate data to another medium (like a WORM device). We also introduced a general lower bound for such queries. A method that achieves the lower bound for a particular query is termed I/O-optimal for that query. The worst-case performance of each transaction-time method is summarized in table B. The reader should be cautious when interpreting worst case performance. Sometimes the notation penalizes a method for its performance on a pathological scenario. We have indicated such cases. While table *B* provides a good feeling for the asymptotic behavior of the examined methods, the choice of the appropriate method for the particular application also depends on the application characteristics. In addition, issues such as data clustering, index pagination, migration of data to optical disks etc. may also be more or less important according to the application. While I/O-optimal (and practical) solutions exist for many transaction-time queries, this is not the case for the valid and bitemporal domain. An I/O optimal solution exists for the valid-timeslice query but is mainly of theoretical importance; more work is needed in this area. All examined transaction-time methods support "linear" transaction time. Another promising area of research is the support of branching transaction time.

Acknowledgments:

The idea of doing this survey was proposed to the authors by R. Snodgrass. We would like to thank the anonymous referees for many insightful comments that improved the presentation of this paper. The performance and the description of the discussed methods are based on our understanding of the related papers, hence any error is entirely our own. The second author would also like to thank J.P. Schmidt for many helpful discussions on lower bounds in a paginated environment.

References:

[AFS93] R. Agrawal, C. Faloutsos, A. Swami, "Efficient Similarity Search in Sequence Databases", *Proc. FODO Conference,* 1993.

[AS88] I. Ahn, R. Snodgrass, "Partitioned Storage for Temporal Databases", *Information Systems*, Pergamon Press, Vol. 13, No 4, pp 369-391, 1988.

[AV96] L. Arge, J.S. Vitter, "Optimal Dynamic Interval Management in External Memory", in *Proc.* 37*th IEEE Symp. on Foundations of Computer Science,* Vermont, Oct. 1996.

[B77] J.L. Bentley, "Algorithms for Klee's Rectangle Problems", Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1977.

[B82] J. Ben-Zvi, "The Time Relational Model", Ph.D. Dissertation *UCLA*, 1982.

[B95] M. H. Böhlen, "Temporal Database System Implementations", in *ACM Sigmod Record*, Vol.24, No. 4, pp 53-60 (1995)

[BG90] G. Blankenagel, R.H. Guting, "XP-Trees, External Priority Search Trees", Technical Report, Fern Universitat Hagen, Informatik-Bericht No.92, 1990.

[BG94] G. Blankenagel, R.H. Guting, “External Segment Trees”, *Algorithmica*, Vol.12, No.6, pp 498-532, 1994.

[BGO+93] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, “On Optimal Multiversion Access Structures”, *Proc. Symp. on Large Spatial Databases*, in Lecture Notes in Computer Science, Vol. 692, pp 123-141, Singapore 1993.

[BHK85] F.W. Burton, M.M. Huntbach, J.G. Kollias, “Multiple Generation Text Files using Overlapping Tree Structures”, *The Computer Journal*, Vol. 28, pp 414-416, 1985.

[BKKS90] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, “The R*-tree: An efficient and Robust Access Method for Points and Rectangles”, *Proc. ACM SIGMOD*, pp 322-331, 1990.

[BO95] T. Bozkaya, M. Ozsoyoglu, “Indexing Transaction-Time Databases”, Tech. Rep. CES-95-19, Case Western Reserve University, 1995.

[CT92] Y.J. Chiang, R. Tamassia, “Dynamic Algorithms in Computational Geometry”, *Proceedings of IEEE, Special Issue on Computational Geometry*, Vol 80, No 9, pp 362-381, 1992.

[DKM+88] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer, H. Rohnhert and R. Tarjan, “Dynamic Perfect Hashing: Upper and Lower Bounds”, *Proc.* 29*th IEEE FOCS*, pp. 524-531, 1988.

[DSST89] J.R. Driscoll, N. Sarnak, D. Sleator, R.E. Tarjan, “Making Data Structures Persistent”, *J. of Comp. and Syst. Sci.*, Vol 38, pp 86-124, 1989.

[E83] H. Edelsbrunner, “A new Approach to Rectangle Intersections, Part I&II”, *Int. Journal of Computer Mathematics*, Vol. 13, pp 209-229, 1983.

[E86] M. C. Easton, “Key-sequence data sets on indelible storage”, *IBM J. Res. Development*, Vol.30, No.3, pp 230-241, 1986.

[EKW91] R. Elmasri, Y. Kim, G. Wuu, “Efficient Implementation Techniques for the Time Index”, *Proc.* 7*th IEEE Intern. Conf. on Data Engineering*, pp 102-111, 1991.

[EWK90] R. Elmasri, G. Wuu, Y. Kim, “The Time Index: An Access Structure for Temporal Data”, *Proc.* 16*th Conference on Very Large Databases*, pp 1-12, 1990.

[EWK93] R. Elmasri, G. Wuu, V. Kouramajian, “The Time Index and the Monotonic B^+-tree”, in A.Tansel, J. Clifford, S.K. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (eds.), *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, pp 433-456, 1993.

[FRM94] C. Faloutsos, M. Ranganathan, Y. Manolopoulos, “Fast Subsequence Matching in Time-Series Databases”, *Proc. ACM SIGMOD Conf. on the Management of Data*, pp 419-429, 1994.

[G84] A. Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching”, *Proc. ACM SIGMOD Conf. on the Management of Data*, pp 47-57, 1984.

[G89] O. Gunther, “The Design of the Cell-Tree: An Object-Oriented Index Structure for Geometric Databases”, *Proc.* 5*th IEEE Intern. Conf. on Data Engineering*, pp 598-605, 1989.

[GR93] J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[GS93] H. Gunadhi, A. Segev, “Efficient Indexing Methods for Temporal Relations”, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 3, pp 496-509, 1993.

[IKO87] C. Icking, R. Klein, T. Ottmann, “Priority Search Trees in Secondary Memory”, *Proc. of Graph-Theoretic Concepts in Computer Science, LNCS* 314, pp 84-93, 1987.

[J+94] C.S. Jensen, editor et al., "A Consensus Glossary of Temporal Database Concepts", *ACM SIGMOD Record*, Vol. 23, No. 1, pp. 52-64, 1994.

[JMM95] H.V. Jagadish, A.O. Mendelzon, T. Milo, "Similarity-Based Queries", *Proc.* 14*th ACM PODS, Symp. on Princ. of Database Systems*, pp 36-45, 1995.

[JMR91] C.S. Jensen, L. Mark, N. Roussopoulos, "Incremental Implementation Model for Relational Databases with Transaction Time", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 3, No. 4, pp 461-473, 1991.

[JMRS92] C.S. Jensen, L. Mark, N. Roussopoulos, T. Sellis, "Using Differential Techniques to Efficiently Support Transaction Time", *VLDB Journal*, Vol.2, No.1, pp 75-111, 1992.

[K93] C. Kolovson, "Indexing Techniques for Historical Databases", in A.Tansel, J. Clifford, S.K. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (eds.), *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, pp 418-432, 1993.

[KF94] I. Kamel, C. Faloutsos, "Hilbert R-tree: An Improved R-tree using Fractals", *Proc.* 20*th VLDB Conf.*, pp 500-509, 1994.

[KKEW94] V. Kouramajian, I. Kamel, R. Elmasri, S. Waheed, "The Time Index$^+$: An Incremental Access Structure for Temporal Databases", *Proc.* 3*rd Intern. Conf. on Information and Knowledge Management CIKM*'94.

[KRVV93] P.C. Kanellakis, S. Ramaswamy, D.E. Vengroff, J.S. Vitter, "Indexing for Data Models with Constraints and Classes", *Proc. ACM PODS, Symp. on Princ. of Database Systems*, pp 233-243, 1993.

[KS89] C. Kolovson, M. Stonebraker, "Indexing Techniques for Historical Databases", *Proc.* 5*th IEEE Intern. Conf. on Data Engineering*, pp 127-137, 1989.

[KS91] C. Kolovson, M. Stonebraker, "Segment Indexes: Dynamic Indexing Techniques for Multi-dimensional Interval Data", *Proc. ACM SIGMOD Conf. on the Management of Data*, pp 138-147, 1991.

[KTF95a] A. Kumar, V.J. Tsotras, C. Faloutsos, "Access Methods for Bitemporal Databases", *International Workshop on Temporal Databases*. In *Recent Advances in Temporal Databases*, J. Clifford, A. Tuzhilin (eds.), pp. 235-254, Springer-Verlag, 1995

[KTF95b] A. Kumar, V.J. Tsotras, C. Faloutsos, "Designing Access Methods for Bitemporal Databases", AT&T Bell Laboratories Tech. Rep. No: 112530-950926-11-TM, 1995. To appear in *IEEE Trans. on Knowledge and Data Engineering*.

[L93] D. Lomet, "Using Timestamping to Optimize Commit", *Proc.* 2*nd Intern. Conf. on Parallel and Distributed Information Systems*, pp 48-55, 1993

[LDE+84] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, J. Woodfill, "Designing DBMS Support for the Temporal Database", *Proc. ACM SIGMOD Conf. on the Management of Data*, pp 115-130, 1984.

[LJ88] N.A. Lorentzos, R.G. Johnson, "Extending Relational Algebra to Manipulate Temporal Data", *Information Systems*, Pergamon Press, Vol 13, No. 3, pp 289-296, 1988.

[LM91] S. Lanka, E. Mays, "Fully Persistent B$^+$ Trees", *Proc. ACM SIGMOD Conf. on the Management of Data*, pp 426-435, 1991.

[LM92a] T.Y.C. Leung, R.R. Muntz, "Generalized Data Stream Indexing and Temporal Query Processing", 2*nd Intern. Work. on Res.Issues in Data Eng.: Trans. and Query Proc.*, Feb.1992.

[LM92b] T.Y.C. Leung, R.R. Muntz, "Temporal Query Processing and Optimization in Multiprocessor Database Machines", *Proc.* 18*th VLDB Conf.*, pp. 383-394, 1992.

[LM93] T.Y.C. Leung, R.R. Muntz, "Stream Processing: Temporal Query Processing and Optimization, in A.Tansel, J. Clifford, S.K. Gadia, S. Jajodia, A. Segev and R. Snodgrass (eds.), *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, pp 329-355, 1993.

[LS89] D. Lomet, B. Salzberg, "Access Methods for Multiversion Data", *Proc. ACM SIGMOD Conf. on the Management of Data*, pp 315-324, 1989.

[LS90a] D. Lomet, B. Salzberg, "The Performance of a Multiversion Access Method", *Proc. ACM SIGMOD Conf. on the Management of Data*, pp 353-363, 1990.

[LS90b] D. Lomet, B. Salzberg, "The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance", *ACM Trans. on Database Systems*, Vol.15, No.4, pp 625-658, 1990.

[LS93a] D. Lomet, B. Salzberg, "Transaction-Time Databases", in A.Tansel, J. Clifford, S.K. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (eds.), *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, pp 388-417, 1993.

[LS93b] D. Lomet, B. Salzberg, "Exploiting a History Database for Backup", *Proc.* 19*th Conference on Very Large Data Bases*, pp 380-390, 1993.

[LST95] G.M. Landau, J.P. Schmidt, V.J. Tsotras, "On Historical Queries Along Multiple Lines of Time Evolution", *Very Large Data Bases Journal*, Vol. 4, pp. 703-726, 1995.

[M84] K. Mehlhorn, *Data Structures and Algorithms* 3: *Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, 1984.

[McC85] E.M. McCreight, "Priority Search Trees", *SIAM Journal of Computing*, Vol.14, No 2, pp 257-276, 1985.

[MK90] Y.Manolopoulos, G. Kapetanakis, "Overlapping B^+ Trees for Temporal Data", *Proc. of* 5*th JCIT Conf.*, Jerusalem, Israel, Oct.22-25, pp 491-498, 1990.

[MS91] D. Maier, S.C. Salveter, "Hysterical B-trees", *Information Processing Letters*, Vol. 12, pp 199-202, 1981.

[NA87] S.B. Navathe, R. Ahmed, "A Temporal Relational Model and a Query Language", *Information Sciences*, North Holland, Vol. 49, 1987.

[NDE96] M.Nascimento, M.H. Dunham, R. Elmasri, "M-IVTT: A Practical Index for Bitemporal Databases", *Proc. DEXA'*96, Zurich, Switzerland.

[NDK96] M.Nascimento, M.H. Dunham, V. Kouramajian, "A Multiple Tree Mapping-Based Approach for Range Indexing". *Journal of the Brazilian Computer Society*, Vol.2, No.3, April 1996.

[O85] M.H. Overmars, "Range Searching in a Set of Line Segments", *Proc. ACM Symp. Computational Geometry*, pp. 177-185, 1985.

[OS95] G. Ozsoyoglu, R. Snodgrass, "Temporal and Real-Time Databases: A Survey", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 7, No. 4, pp 513-532, Aug. 1995.

[R84] J.T. Robinson, "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes", *Proc. ACM SIGMOD Conf. on the Management of Data,* pp 10-18, 1984.

[R97] S. Ramaswamy, “Efficient Indexing for Constraint and Temporal Databases”, *Proc. ICDT*’97 *Intern. Conference on Database Theory*, Delphi, Greece, Jan. 8-10, 1997.

[RCDS86] J.E. Richardson, M.J. Carey, D.J. DeWitt, E.J. Shekita, “Object and File Management in the Exodus Extensible System”, *Proc.* 1*2th Conference on Very Large Databases*, pp 91-100, 1986.

[RS87] D. Rotem, A. Segev, “Physical Organization of Temporal Data”, 3*rd IEEE Intern. Conf. on Data Engineering*, pp 547-553, 1987

[RS94] S. Ramaswamy, S. Subramanian, “Path Caching: a Technique for Optimal External Searching”, *Proc.* 13*th ACM Symp. on Principles of Database Systems (PODS)*, pp 25-35, 1994.

[S87] M. Stonebraker, “The Design of the Postgres Storage System”, *Proc.* 1*3th Conference on Very Large Databases*, pp 289-300, 1987.

[S88] B. Salzberg, *File Structures: An Analytic Approach*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[S94] B. Salzberg, “Timestamping After Commit”, *Proc.* 3*rd Intern. Conf. on Parallel and Distributed Information Systems*, pp 160-167, 1994.

[SA85] R. Snodgrass, I. Ahn, “A Taxonomy of Time in Databases”, *Proc. ACM SIGMOD Conf. on the Management of Data*, pp 236-246, 1985.

[SA86] R. Snodgrass, I. Ahn, “Temporal Databases”, *IEEE Computer*, Vol.19, No.9, pp 35-42, 1986.

[SG89] A. Segev, H. Gunadhi, “Event-Join Optimization in Temporal Relational Databases”, *Proc.* 15*th Conference on Very Large Data Bases*, pp 205-215, 1989.

[SJ96] R. Snodgrass, C.S. Jensen, private communication, 1996

[SK86] A. Shoshani, K. Kawagoe, “Temporal Data Management”, *Proc.* 12*th Conf. on Very Large Data Bases*, pp 79-88, 1986.

[SL95] B. Salzberg, D. Lomet, “Branched and Temporal Index Structures”, College of Computer Science Technical Report, NU-CCS-95-17, Northeastern University.

[SRF87] T. Sellis, N. Roussopoulos, C. Faloutsos, “The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects”, *Proc. VLDB Conf.*, Sept. 1987.

[TG90] V.J. Tsotras, B. Gopinath, “Managing the History of Evolving Databases”, *Proc. ICDT*’90 *Intern. Conference on Database Theory*, in *Lect. Notes in Comp. Science*, Vol 470, pp 141-174, Springer-Verlag,1990.

[TGH95] V. J. Tsotras, B. Gopinath, G.W. Hart, “Efficient Management of Time-Evolving Databases”, *IEEE Trans. on Knowledge and Data Engineering*, Vol. 7, No. 4, pp 591-608, Aug.1995.

[TK95] V.J.Tsotras, N. Kangelaris, “The Snapshot Index, an I/O-Optimal Access Method for Timeslice Queries”, *Information Systems, An International Journal*, Vol. 20, No.3, 1995.

[TK96] V.J. Tsotras, A. Kumar, “Temporal Database Bibliography Update”, *ACM Sigmod Record*, Vol. 25, No. 1, pp 41-51, March 1996.

[V85] J.S. Vitter, “An Efficient I/O Interface for Optical Disks”, *ACM Trans. on Database Systems*, Vol.10, No.2, pp 129-162, 1985.

[VV94] R.M. Verma, P.J. Varman, “Efficient Archivable Time Index: A Dynamic Indexing Scheme for Temporal Data”, *Intern. Conf. on Computer Systems and Education*, pp 59-72, 1994.

[VV95] P.J. Varman, R.M. Verma, “An Efficient Multiversion Access Structure”, Tech. Rep. TR-9518, Dept. of Electr. and Comp. Engineering, Rice Univ. To appear in *IEEE Trans. on Knowledge and Data Engineering*.