Designing Access Methods for Bitemporal Databases

Anil Kumar *Vassilis J. Tsotras*†
Dept. of Electrical Engineering & Computer Science
Polytechnic University
Brooklyn, NY 11201
{akumar, tsotras}@photon.poly.edu

*Christos Faloutsos**
Department of Computer Science
University of Maryland
College Park, MD 20742
christos@cs.umd.edu

A B S T R A C T

By supporting the valid and transaction time dimensions, bitemporal databases represent reality more accurately than conventional databases. In this paper we examine the issues involved in designing efficient access methods for bitemporal databases and propose the *partial-persistence* and the *double-tree* methodologies. The partial-persistence methodology reduces bitemporal queries to partial persistence problems for which an efficient access method is then designed. The double-tree methodology "sees" each bitemporal data object as consisting of two intervals (a valid-time and a transaction-time interval), and divides objects into two categories according to whether the right endpoint of the transaction time interval is already known. A common characteristic of both methodologies is that they take into account the properties of each time dimension. Their performance is compared with a straightforward approach that "sees" the intervals associated with a bitemporal object as composing one rectangle which is stored in a single multidimensional access method. Given that some limited additional space is available, our experimental results show that the partial-persistence methodology provides the best overall performance, especially for transaction timeslice queries. For those applications that require ready, off-the-shelf, access methods the double-tree methodology is a good alternative.

Index Terms: Bitemporal Databases, Access Methods, Transaction-time, Valid-time.

†This work was partially supported under ARPA Contract No. DAAH01-94-C-R116, by NSF under grants No. IRI-9111271 and IRI-9509527, and by the New York State Science and Technology Foundation as part of its Center for Advanced Technology program.
*This work was performed while the author was visiting AT&T Bell Laboratories, Murray Hill, NJ 07974. This work was partially supported by the Institute of Systems Research and by NSF under Grants No. EEC-94-02384, IRI-8958546 and IRI-9205273, with matching funds from Empress Software Inc. and Thinking Machines Inc.

1. Introduction

Conventional database systems capture only a single snapshot of the modeled reality. While serving many applications well, they are not sufficient for applications that require the support of time-varying information (past and/or future data). Instead, *temporal* database systems have been proposed as they can store and query temporal data through the support of two orthogonal time dimensions: the *valid* and *transaction* times [OS95].

According to [J+94], "the valid time of a fact is the time when the fact is true in the modeled reality". Transaction time on the other hand refers to the time when a new value is posted to the database by a transaction. A temporal database is categorized as transaction-time, valid-time or bitemporal, according to which temporal dimension(s) it supports.

The transaction time dimension represents the history of a database activity rather than real world history. Transaction times are system generated and monotonically increasing[1]. Since it is impossible to change the past, transaction times cannot be changed and there is no way to correct errors in past tuples. A valid-time database maintains the entire history of an enterprise as best known now, i.e., it stores the current knowledge about the past and future. Any errors discovered in this history, are corrected by modifying the database. When a correction is applied, previous values are not retained; therefore it is not possible to view the database as it was before the correction.

Clearly both time dimensions are needed in order to accurately model reality. In a bitemporal database one can query tuples that are valid at some (valid) time, as known at some other (transaction) time. A variety of applications can benefit from the support of valid and transaction time [OS95]:

(a) accounting, marketing, tax-related, billing applications [W82]. The retroactive/postactive changes that occur in such applications require the support of the valid-time dimension. For auditing purposes transaction time is also needed (so that no one can "alter" recorded data). In a tax application, we may want to find what tax laws were valid when a tax return was filed. Similarly, a billing system should be able to issue corrections on past records and keep track of when this correction was made effective (retroactive salary increases etc.)

(b) social/medical applications [F72]. A physician makes decisions on a patient based on the patient's history as best known when the decision is made (by looking at the hospital database). Patient information though can be recorded at various times. The validity of a decision can be tested only against the information on which it was made.

[1] Here we concentrate on *linear* transaction time (as opposed to *branching* transaction time [OS95]).

(c) financial/stock-market applications [A92]. A broker makes recommendations to clients based on the information available at the time of the recommendation. However, due to delays, not all data is presented at real time to the broker. The correctness of a given recommendation can be tested only against the knowledge available to the broker when the recommendation was made.

Bitemporal data tends to increase in size as transaction time proceeds, making the need for efficient indexing more crucial than in conventional databases. While much work has been done recently on access methods that support a single time axis, not much has been done for bitemporal indexes, i.e., methods that support both transaction and valid time dimensions on the same index.

In this paper we examine the issues involved in designing efficient bitemporal access methods and propose two methodologies for constructing such methods. The partial-persistence methodology starts with a data structure that can efficiently address the valid-time dimension of a bitemporal query and makes it partially persistent [DSST89] so as to address the transaction-time dimension, too. We discuss two examples based on this methodology: the *Bitemporal Interval Tree* and the *Bitemporal R-Tree*. The double-tree methodology assigns bitemporal objects in two categories according to their transaction-time behavior and uses a separate (tree-based) access method to store objects in each category.

The Bitemporal Interval Tree was introduced in a workshop paper [KTF95] where we presented initial performance results as related to a small class of bitemporal queries. Here we present a more thorough coverage of the general bitemporal index design problem. For completeness, we also provide an outline of the Bitemporal Interval Tree. The additional contributions of this paper are summarized as:

(1) the introduction of the Bitemporal R-Tree, whose implementation provides the best overall performance for bitemporal queries in our experiments;

(2) the optimization of the Bitemporal R-Tree based on various merging and splitting policies and two representations (intervals and points);

(3) in addition to the "timeslice" query of [KFT95], this paper examines various other bitemporal query classes and uses a larger collection of test experiments.

A by-product of this research is a method to maintain a disk-based partially persistent ordered list (Appendix A). The rest of the paper is organized as following: section 2 discusses the bitemporal environment and section 3 presents previous work. Section 4 describes the principles of the proposed methodologies. Sections 5 and 6 elaborate on access methods derived from the partial persistence methodology, namely the Bitemporal Interval Tree and the Bitemporal R-Tree. Section 7 presents the experimental results. Conclusions and problems for further research are in section 8.

2. The Bitemporal Environment and its Queries

We start with a description of temporal databases with a single time dimension (transaction or valid) and continue with a bitemporal database (for a thorough coverage we refer to [SA86]). We also introduce a classification for bitemporal queries which is used throughout the paper [SJ96].

Consider an initially empty set of objects that evolves over time according to the following rules: time is always increasing and at each time instant one (or more) changes may happen. A change is an object addition or deletion. An object is called *alive* from the time it is added in the set until (if ever) it is deleted from the set. The state of the evolving set at t, namely $s(t)$, consists of all the alive objects at t. Changes are always applied on the most current state $s(t)$. This evolution model can also support object attribute modifications, since a modification can be represented by the artificial deletion of the alive object followed by the simultaneous re-insertion of this object having modified attribute(s). For simplicity we concentrate only on object additions/deletions.

Assume that the above evolution is to be stored in a database system. Since time is always increasing and the past is not changed, a transaction-time database system can be utilized, with the implicit assumption that when an object is added in, or, deleted from the evolving set at time t, a transaction with the same timestamp t updates the database system about this change. This process associates with each object a transaction-time interval $[t_1, t_2)$, where t_1 is the transaction time when this object was entered in the database (transaction *insertion* time), and t_2 is the time the object was deleted (transaction *deletion* time). As the future is unknown, when an object is added its deletion time t_2 is yet not known. So typically, an object is inserted in the database with a transaction interval of the form: $[t_1, now)$ where *now* is a variable representing the current time. If this object is later deleted, its transaction-time interval is updated with the appropriate deletion time.

A valid-time database has a different abstraction. Consider a dynamic collection of objects where associated with each object is a valid-time interval. This interval represents the validity period of some object property. The only time dimension defined is that of the valid-time intervals. Changes may happen to this collection, where a change is the addition/deletion or modification of an object. Changes are not timestamped and when a deletion or modification occurs, the previous object is <u>not</u> retained in the database. Hence a valid-time database keeps only the most current collection of objects. If we consider the order implied by the relative position of the valid-time intervals on the valid-time axis as a (valid-time) history, then a valid-time database represents this history as "best known now".

In a bitemporal environment both time dimensions exist. To better clarify their differences, consider an application that keeps track of a company's contracts. A contract is of the form: (c, I), where c is some contract identifier and I is the contract duration interval. Interval I corresponds to an interval on the valid time axis and the contract is an example of an object. The contract information is recorded in the database at some transaction time t that is orthogonal to interval I. For example at time t we may record past, current or even a future contract I. (According to [S+94] this example will create a *bitemporal-state* table). A *history timeslice* (denoted by *ht(t)*) contains the history (in the valid-time domain) of the company's contracts as best known at time t (Figure 1).

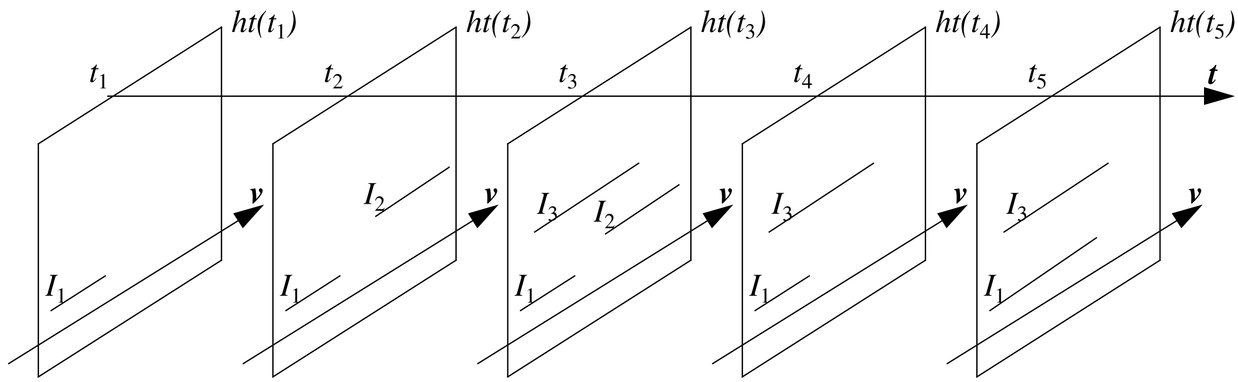


Figure 1: A conceptual view of a bitemporal database. Each interval represents a contract; for simplicity no other contract attributes are shown. The t-axis (v-axis) corresponds to transaction (valid) times. At transaction time t_1 interval I_1 is added on $ht(t_1)$; then at t_2 and t_3 intervals I_2 and I_3 are added in history timeslices $ht(t_2)$ and $ht(t_3)$ respectively. At t_4 interval I_2 is deleted, while at t_5 interval I_1 is modified.

It is possible that at a later transaction time $t' > t$, some previously recorded contract interval is found to be erroneous (for example, a contract has shorter duration in reality than what was recorded, or should have never been recorded) or a proactive/retroactive change modifies the interval of some recorded contract to a new one. As a result a new history timeslice $ht(t')$ is created to reflect these changes. The evolution of the company is best represented if all *ht*'s are retained and can be queried. A bitemporal database can therefore be "visualized" as a series of history timeslices *ht(t)*, each marked by some transaction time t and consists of a collection of valid-time intervals. In general the two time axes are orthogonal: the transaction time at which a new historical timeslice is created is irrelevant to the intervals reported on the timeslice, which can be earlier, current or later than this time (especially since future contracts can also be recorded).

Since access methods are closely related to the queries they are designed for, we conclude this section with a notation scheme (proposed in [SJ96]) used to classify bitemporal queries. For our purposes an object is characterized by three *entries*: a non-temporal key (e.g., the contract identifier c), a valid-time interval and a transaction-time interval. In general an object may have many non-temporal attributes but for simplicity we assume only one.

A query is classified using the notation: *Key/Valid/Transaction*. This notation specifies which entries are involved in the query and in what way. Each entry can be described as a "point", "range", "*," or "-". A "point" for the *Key* entry means that the user has specified a single value to match the key attribute, while "point" for the *Valid* or *Transaction* entry implies a single time instant is specified for this entry. Similarly, "range" indicates a specified range of 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 this entry is not applicable.

A simple bitemporal query is: "find all the company contracts that were valid on v = January 1, 1994, as recorded on t = November 1, 1993" (a "*/point/point" query). The history timeslice on Nov. 1, 1993 contains all contracts known as of that time; this includes past, current or future contracts with respect to Nov. 1, 1993. From all these, the query retrieves only the contracts that would be valid on Jan. 1, 1994. One of the most general bitemporal queries is: "for a key range K, a transaction time interval P and a valid time interval E, find all contracts with identifiers in range K whose valid-time interval intersected E during transaction times in P"; this is an example of a "range/range/range" query. Note that in general a query may ask for objects with intervals before or after a given interval; here we only deal with intersection, i.e., the query answer includes all objects whose intervals intersect the query interval.

3. Previous Work

We begin this section with a short presentation of the performance costs involved in a temporal access method and continue with a discussion of previous work.

Any access method used to organize time-evolving data is characterized by the following costs: *space* (the space consumed by the method's data structures in order to keep such data), *update time* (the time needed to update the method's data structures for data changes) and *query time* (the time needed to compute a temporal query). All three costs are functions of three basic parameters: the answer size a, the number of changes n and the page (block) size b. The answer size a is the number of objects satisfying the query predicate. The number of changes n corresponds to the total number of valid-time changes that occur in a bitemporal evolution (thus n is also an upper bound to the number of transaction-time updates since in general, at a given transaction a number of valid-time updates is processed). Parameter n represents the minimal information needed by any index to perform errorless reconstruction of time-varying data. A valid-time change corresponds to either the insertion, deletion or modification of a valid-time interval. Regarding the page size, we assume that every secondary memory access transmits one page of b records and this counts as one I/O.

There are two desirable properties for an efficient access method: *index pagination* and *query* *clustering*. Index pagination deals with the issue of how well index nodes of a method are paginated. Query clustering is achieved 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.

A variety of temporal access methods have been proposed in recent years. All previous approaches directly support a single time axis; most methods assume that time is always increasing and/or updates are always applied on the latest state (i.e., the past is not changed). These are characteristics of transaction-time. Assuming a <u>transaction-time</u> database, a common query is the "*/-/point" or *pure-timeslice* query [ST94]. For example: "find *all* employees recorded as working on January 1st, 1990". More general is the "range/-/point" or *range-timeslice* query, where the predicate adds a condition on the objects' attribute space: "find all employees recorded as working on January 1st, 1990 and whose id's are in the range *(X,Y)*".

Various methods have been proposed to solve the "*/-/point" query [EWK90, JMR91, LM93, SG89, TGH95, TK95]. These methods keep the time evolution separate from the key space. To answer "range/-/point" queries, such methods compute first the whole past state and then eliminate objects outside the requested key-range. In another approach, the whole key space is divided into *predefined* key-ranges and the requested "range/-/point" query is computed by solving smaller "*/-/point" queries on the predefined key-ranges that cover the query key-range.

To better address "range/-/point" queries, a method must combine the time and key spaces [BGO+93, KS91, LM91, LS89, MK90, S87, VV95]. The optimal solution is provided by the Multi-Version B-tree (MVB) [BGO+93] and the Multiversion Access method (MVAS) [VV95]. Both use $O(n/b)$ space, $O(\log_b m)$ update per change (in the amortized[2] sense [CLR90]) and $O(\log_b n + a/b)$ query time. Using the transaction-time database abstraction, n represents the total number of objects ever added or deleted, m denotes the number of "alive" objects when an update takes place and a represents the answer size to a "range/-/point" query. Using improved merge/split policies, the MVAS [VV95] has a smaller constant in the space bound which in practice results in lower space. The Time-Split B-tree (TSB) [LS89] is more space efficient, but it can guarantee worst case query performance only when the evolution contains no deletions.

Obviously, a method that can efficiently address a "range/-/point" query can also address a "*/-/point" query with the same efficiency. However, by combining the time and key spaces such a method requires logarithmic update time which is not needed for the "*/-/point" query. The Snapshot Index [TK95] optimally solves the "*/-/point" query using constant instead of logarithmic updating (in an expected amortized sense).

[2] This means that some changes could require more than $O(\log_b m)$ update processing, but for any sequence of k changes no more than a total of $O(k \log_b m)$ time will be required.

Two common queries in a valid-time database are the "*/point/-" and the "*/range/-" (find all objects whose valid interval contains a time instant v, or respectively, intersects a given interval E). The first query has been termed the *point enclosure* and the second the *interval intersection* query. The combination of dynamic interval insertions and (physical) deletions with the above queries is known in the computational geometry literature as the *dynamic interval management* problem. The best main-memory solution for the dynamic interval management problem is achieved using the *priority search tree* [Mc85] or the *interval tree* [E83], yielding $O(k)$ space, $O(\log_2 k)$ update processing per change (addition or deletion of a valid interval) and $O(\log_2 k + a)$ query time. Here k corresponds to the number of intervals in the structure when the query is asked and a is the size of the answer. Depending on the query predicate, a corresponds to the number of intervals containing the query instant or the number of intervals intersecting the query interval. It has been proved that this is the optimal solution in a main memory environment.

Until very recently finding an I/O optimal solution even for the simplest of the valid-time queries ("*/point/-") was an open problem [KRVV93]. In [AV96] such optimal solution is presented but it is rather complex to be practical (it is I/O optimal since it uses $O(k/b)$ space, $O(\log_b k)$ update time and $O(\log_b k + a/b)$ query time). Note that for I/O's $O(\log_b k + a/b)$ is different than $O(\log_2 k + a)$ since the page size b is not a constant but another problem parameter. The problem becomes more difficult if an object key range is included in the predicate (creating a "range/point/-" or a "range/range/-" query). One could use an R-tree [G84] to dynamically store valid intervals. While such approach may be practical for many applications, recall that R-trees use $O(k/b)$ space and $O(\log_b k)$ time for interval insertion, but interval deletion and search can in the worst case be $O(k/b)$. Searching for an interval implies following all R-tree index nodes that overlap this interval. At worst the whole tree may have to be searched.

Among the single time axis approaches, the work in [KS91] falls between transaction and valid time databases. This method associates with each temporal object an interval whose both endpoints are known and uses the SR-tree (a variation of an R-tree [G84]) to store and query such intervals. The method is optimized towards insertions of intervals and searches. Interval deletions may be problematic since intervals are split in many segments, making their update more difficult. As mentioned in [KS91] such interval deletions would correspond to "revising" the history, hence they are not critical for an index for historical data. This is the case for transaction time databases but not for valid-time ones. However, when data objects are added in the SR-tree both of their interval endpoints are known (which is not the case for transaction-time).

If we view the SR-tree as a transaction-time structure, new objects can be inserted with interval

$[t_1$, *now)* where *now* is some very large number. The performance would degrade for two reasons: excessive overlapping and frequent deletions (which would be needed when an $[t_1$, *now)* interval is updated to $[t_1, t_2)$). Alternatively, newly inserted objects whose intervals have "unknown" right endpoints would have to be kept in a separate structure (such structure is not described but it could be some variation of an R-tree). The SR-tree could be used for valid time databases if the interval associated with each object corresponds to the object's valid time interval. However, in this environment deletions of intervals are possible, hence intervals would have to be physically deleted from the structure frequently.

In our double-tree methodology, bitemporal objects are transferred from one access method to another when the right endpoints of their transaction time intervals become known. This object migration is reminiscent of the Dual-Root Mixed Media R-tree proposed in [KS89]; however, as it is explained in section 4.1, these approaches address different problems. We proceed by describing our methodologies for designing bitemporal access methods.

4. The Proposed Methodologies

In designing methods for bitemporal queries there exist some obviously inefficient approaches. At one extreme, one could explicitly store the whole *ht(t)* at each transaction time *t* (Fig. 1). The disadvantage of this solution is the space and update requirements. At the other extreme, one could store all updates on a sequential log, but this has prohibitive query performance. A hybrid solution stores whole *ht*'s every *l-th* transaction and the update sequence between subsequent timeslices. If the distance between timeslices is fixed the hybrid method would behave like one of the two extremes depending on the choice of *l*. Note that in this paper we assume <u>no</u> query locality[3] which implies that *l* cannot change according to the queries asked. Another solution would be to index bitemporal objects only on transaction time and use a single time access method. Then a bitemporal query is answered in two steps: first all objects existing at transaction time *t* are found and then the valid time interval of each such object is checked whether it includes valid time *v*.

In another straightforward approach, a bitemporal object is represented by a "bounding rectangle" created by the object's valid and transaction-time intervals, that is stored in a <u>single</u> multidimensional index (Figure 2). Due to the characteristics of transaction time (unknown future), a bitemporal object with valid-time interval *I* which is inserted in the database at transaction time *t*, is represented by a rectangle with a transaction-time interval of the form [*t*, *now*). Here *now* is a variable that represents the current transaction time and extends to "infinity" or "forever". If this

[3] This assumption distinguishes this work from incremental computation [JMR91] where query locality is shown to provide better query performance than indexing.

object is deleted or modified at some t' ($t' > t$) its corresponding rectangle is removed from the access method and a new rectangle with transaction-time interval $[t, t')$ is inserted. As an example, a “*/point/point” query translates into finding all rectangles that include the query point (t_i, v_j).

For comparison purposes we have implemented this approach and tested it against our proposed methodologies. Since in the implementation we use a single R*-tree [BKKS90] as the multidimensional access method, we refer to this straightforward approach as the 1-R approach. While the 1-R approach has the advantage of using a ready, off-the-shelf access method, it has a major disadvantage due to the large overlapping caused by rectangles whose right-end extends to (transaction-time) *now*. This overlapping affects both updating and querying.

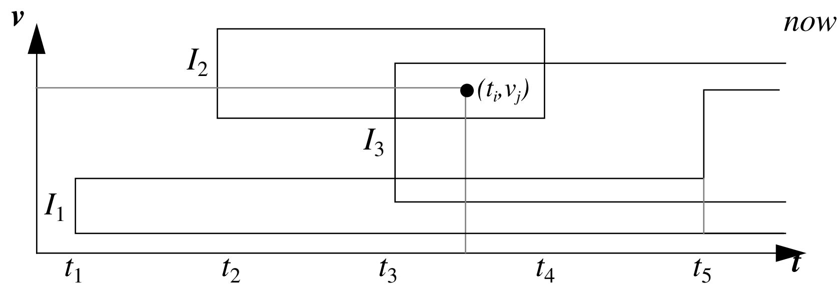


Figure 2: The bounding-rectangle (1-R) approach; only the valid and transaction axis are shown. The evolution of Fig.1 is depicted, as of (transaction) time $t > t_5$. The modification of interval I_1 at t_5 ends the initial rectangle for I_1 and inserts a new rectangle from t_5 to *now*.

Some applications require the support of valid time *now*, i.e. valid intervals whose right endpoint is the always moving current time. One solution would be to treat such special intervals as ending to a special *valid-now* variable and keep them together with the regular valid intervals. Internally *valid-now* is stored as the largest value in the valid time domain. To evaluate *valid-now*, a reference to current time is needed which we assume is given as another query parameter r. Consider a query about transaction time t, valid time v and parameter r. All valid intervals in $ht(t)$ that contain v will be accessed, including those that start before v and end in *valid-now*. Despite performance degradation due to extended overlapping, this approach is problematic if $r < v$ because none of the accessed special intervals qualifies for the answer. Thus, if an application has many intervals ending to valid *now*, a better solution is to keep such intervals in a separate structure which will be accessed only if $r \geq v$. Such structure is a simple variation of the structures presented here since if special intervals are stored separately, only their start points are needed. We thus proceed with the discussion of the proposed methodologies as if all valid-time intervals are regular ones.

4.1 The Double-Tree Methodology

Our double-tree methodology avoids the above overlapping problem while retaining the advantage of using off-the-shelf access methods. For its implementation we use two R*-trees and in the rest we refer to it as the 2-R methodology (in general various other multidimensional access methods could be facilitated).

When a bitemporal object with valid-time interval I is inserted in the database at transaction-time t it is inserted at the *front* R-tree. The front R-tree keeps bitemporal objects for which the right transaction endpoint is unknown. If a bitemporal object is later "deleted" at some transaction 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' on the *back* R-tree. The back R-tree keeps bitemporal objects with known transaction-time interval (Figure 3). 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.

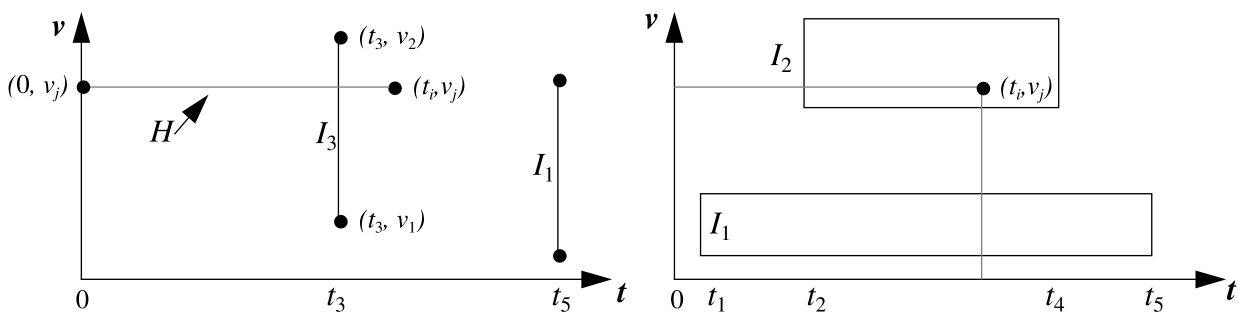


Figure 3: In the 2-R methodology, bitemporal data is divided according to whether their right transaction endpoint is known. The scenario of Fig. 2 is presented here (i.e., after time t_5 has elapsed). The left 2-dimensional space is stored in the *front* R-tree while the right in the *back* R-tree. The query is then translated into an interval intersection and a point enclosure problem, respectively.

As an example, "*/point/point" query about (t_i, v_j) is answered with two searches. The back R-tree is searched for all rectangles that contain point (t_i, v_j). The front R-tree is searched for all vertical intervals which intersect a horizontal interval H. Interval H starts from the beginning of transaction time and extends until point t_i at height v_j (Figure 3). 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 reminds 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. The first R-tree is contained on magnetic disk while the second R-tree is on the optical disk except for its root which is always stored on the magnetic disk. Insertions are made to the first R-tree. When it reaches a threshold, a *vacuuming* process completely vacuums all its nodes (except the root) and inserts them to the optical disk using the second R-tree. Note however that there is no notion of transaction and valid time. Instead, each object is associated with an object key and a single time interval. Both endpoints of the interval are known at the insertion of a new object. If intervals represent transaction times then the right endpoints are not known at insertion. If intervals represent valid time then interval deletions should be supported (however it is not clear how this can be achieved if an interval to be deleted has already been migrated to a write-once read-many optical disk). Object migration in [KS89] is batched and depends on the size of the first R-tree. In our approach an interval-object is migrated to the back R-tree automatically when it is deleted on the transaction-time axis. Finally, in the Dual-Root Mixed Media R-tree approach both R-trees store similar kind of objects. In our 2-R approach the *front* R-tree stores a valid interval and a transaction time per object while the *back* R-tree stores a rectangle (transaction/valid intervals) per object.

4.2 The Partial-Persistence Methodology

This methodology emanates from the abstraction of a bitemporal database as a sequence of history-timeslices *ht* (Figure 1). It reduces bitemporal queries to problems of partial persistence. A data structure is called *persistent* [DSST89] if an update creates a new version of the data structure while the previous version is still retained and can be accessed. Otherwise, if old versions are discarded the structure is termed *ephemeral*. Partial persistence implies that only the newest version can be modified (i.e., changes are applied only to the newest version), while in full persistence every version can be modified.

Partial persistence "suits" nicely with linear transaction time since changes are always applied on the latest *ht*. Our methodology has two steps. First, a good ephemeral structure is chosen to represent each *ht*. This structure must support dynamic addition/deletion of (valid-time) interval-objects; the supported queries depend on what bitemporal queries need to be answered. Second, this structure is made partially persistent. [DSST89] shows how to make any linked *main-memory* ephemeral data structure partially (or fully) persistent. Since in temporal databases the major portion of data will be stored on a disk-based environment, issues like I/O, pagination and query-clustering have to be efficiently addressed when designing the bitemporal access method. In addition, the chosen ephemeral structure should be space efficient since partial persistence can only add to the space requirements of the structure.

By "viewing" a bitemporal query as a partial persistence problem, we obtain a <u>double</u> advantage. First we disassociate the valid-time requirements from the transaction-time ones. More specifically, the valid time support is provided from the properties of the ephemeral structure while the transaction time support is achieved by making this structure partially persistent. Conceptually, this methodology provides fast access to the *ht* of interest on which the valid-time query is then performed. Second, changes are always applied on the most current state of the structure and last until updated (if ever) at a later transaction time.

We use the partial-persistence methodology to design two bitemporal access methods, namely the Bitemporal Interval Tree and the Bitemporal R-Tree.

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/-" and "*/range/-" queries, respectively. As mentioned earlier constructing a method with good worst case behavior even for the simpler of the valid-time queries ("*/point/-") is a difficult problem [KRVV93, AV96]. Ideally we need a practical external ephemeral structure that provides the I/O optimal solution for these problems. In the absence of such method we use a <u>main-memory</u> data structure with good <u>worst-case</u> performance, and make it partially persistent and well paginated. Among three possible ephemeral candidates, i.e., the Interval Tree [E83], the Segment Tree [B77] and the Priority Search Tree [Mc85], we use the Interval Tree. We did not use the Segment Tree since it needs more than linear space (for k intervals the space is $O(k\log_2 k)$) and would increase the overall space. The Priority Search Tree could be another choice but it has a disadvantage over the Interval Tree. Partial persistence keeps copies of all structural updates in an evolving ephemeral structure. Each update to the Interval Tree involves some logarithmic searching and two structural updates. In contrast, each update to the Priority Search Tree involves a logarithmic number of structural updates which would increase the space of the partially persistent structure.

The Bitemporal R-Tree is 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, we use the R-tree [G84], an access method that has good <u>average-case</u> performance and is well-paginated. Making an R-tree partially persistent resulted to the Bitemporal R-Tree.

By its nature, partial persistence provides efficient access to the appropriate history timeslice *ht(t)*. Thus the Bitemporal Interval Tree and the Bitemporal R-Tree are optimized towards bitemporal queries that involve transaction-time *instants* instead of transaction-time *intervals*. While not all *ht*'s are explicitly stored, partial persistence relies on some limited copying of bitemporal objects. To answer queries that involve transaction-time intervals with the Bitemporal Interval Tree or the Bitemporal R-Tree, special care is needed during searching (due to the above copying process). In particular, the worst case query performance of the Bitemporal Interval Tree is only guaranteed for "*/point/<u>point</u>" and "*/range/<u>point</u>" queries. This is not an issue for the Bitemporal R-Tree since it only provides good average case performance.

We conclude with an interesting observation. Partially persistent *multidimensional* access methods (like the Bitemporal R-tree) have yet another application. They can be used to efficiently index data cubes [HRU96] whose one dimension is time (i.e., time-evolving data cubes).

5. The Bitemporal Interval Tree

There are various main-memory implementations of an Interval Tree [E83]. The semi-dynamic implementation [M84] presumes that interval endpoints take values from a known universe U and uses a full binary tree as the basic (backbone) data structure. In the fully-dynamic implementation [M84] no universe knowledge is assumed, but a data structure that supports rotations is used (red-black binary tree [CLR90]). We used the semi-dynamic implementation because it is easier to make partially persistent and well-paginated. Hence, the Bitemporal Interval Tree (BIT) as presented here is more efficient for bitemporal queries whose predicate valid time v (or valid interval E) satisfies $v \in U$ (respectively $E \subseteq U$). For simplicity assume $U = \{1, 2, \ldots, V\}$. Queries on valid time instants outside U are still answered, but their performance is not as efficient.

Let S be a set of n intervals with endpoints from U. An Interval Tree for set S with respect to U, consists of a (backbone) full binary tree T with V leaves and a number of lists (Figure 4). Each leaf is labeled with one element from U. Each non-leaf node u is assigned a value $val(u)$ that serves in directing the search from node u to its subtrees. Every interval $[l, r)$ from S is associated with a single non-leaf node u of T, where u is the node that contains l and r in its left and right subtrees respectively [S89]. Intervals associated with some node u are kept in two doubly-linked lists: $L(u)$ and $R(u)$. $L(u)$ (respectively, $R(u)$) keeps the intervals in increasing (decreasing) order of their left (right) endpoint. For fast insertion/deletion, each list is implemented using a balanced binary tree (not shown in Fig.4). Inserting an interval $[l, r)$ in the Interval Tree is easy: starting from the root of T, find the first node u such that $l < val(u) < r$. Then l is inserted in $L(u)$ and r in $R(u)$. Searching for u would at most need to go down a path of the interval tree, thus it takes $O(\log_2 V)$. Inserting in each list takes at most $O(\log_2 n)$ time. Deleting an interval is done in a similar way. Since every interval from S is kept in a single node u the space used by all the lists is $O(n)$. In addition, $O(V)$ space is used for the backbone binary search tree.

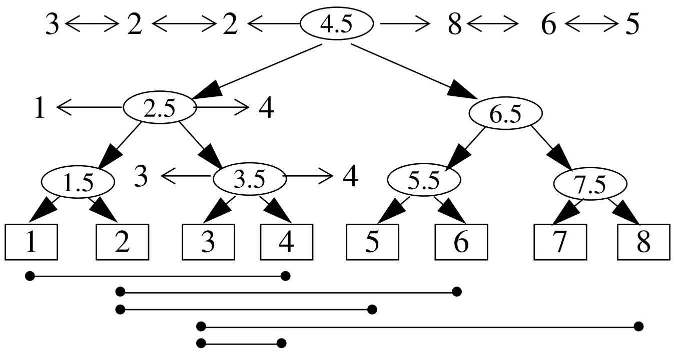


Figure 4: An Interval Tree with $U = \{1,..., 8\}$ and $n = 5$ intervals. The backbone binary tree T and the doubly-linked lists are shown. The *value* of each node appears inside the node. The left/right lists for the root node contain the endpoints of (2,6), (2,5) and (3,8).

For simplicity we discuss the point-enclosure query. Intervals containing an instant v are found in $O(\log_2 V + a)$ comparisons, where a is the number of such intervals. Let p be the path in T from the root to the leaf labeled v. For every node u_i in p the algorithm checks whether $v < val(u_i)$ or $v > val(u_i)$. If $v < val(u_i)$ then v emanates from the left subtree of u_i and all intervals assigned to u_i have right endpoints r that extend v ($r>v$). Each such interval would contain query v, if and only if, its left endpoint l is before v ($l<v$). These left endpoints already exist in increasing order in list $L(u_i)$ and hence the algorithm simply has to traverse list $L(u_i)$ starting from the first endpoint and until the endpoint greater than v is found. No other endpoints need to be read from this list since they correspond to intervals starting after v. A similar argument holds when $v > val(u_i)$. The $\log_2 V$ lists of p, traversed by the query are the *qualifying* lists for this query.

The key property of the Interval Tree is that it transforms interval insertions/deletions to insertions/deletions of their endpoints in ordered lists. If every *ht* of Figure 1 is stored using an Interval Tree, the evolution of valid-time intervals as transaction time proceeds is transformed to evolutions of ordered lists. The first step in designing the Bitemporal Interval Tree is to make all node lists partially persistent. To answer a "*/point/point" query for transaction time t and valid-time v, the backbone tree T of Figure 4 is first searched for the leaf with value v. For each node visited the left or right list is "rollback" to the state it had at transaction time t. Then v is searched on the past state of such lists. While this is a correct high-level description of how our method works, it is a simplification. There are additional issues that have to be addressed, mainly how to paginate the backbone tree when combined with the partially persistent lists.

Let X be an ephemeral ordered list, similar to a node list of the Interval Tree. As (transaction) time proceeds, $X(t)$ denotes the ordered sequence of elements the list had at t. The inclusion of time in the problem creates a transaction interval $[t_1, t_2)$ for each list element, where t_1 is the time the element was added in the current list and t_2 is the time (if ever) this element was deleted from the current list. For all t in $[t_1, t_2)$ this element is called "alive" in $X(t)$. We need a well paginated and easily updated structure to keep $X(t)$ over t, and efficiently address the query: given (transaction) time t, find the elements from $X(t)$ that are less than v (which corresponds to some valid time).

This problem is a *special* case of the "range/-/point" query. An obvious solution would be to use the MVB [BGO+93](or MVAS [VV95] or TSB [LS89]) tree and store $X(t)$ on its leaf pages, but this would create an extra overhead in the Bitemporal Interval Tree query time. Searching a list must always start at the root of the MVB structure, for a $O(\log_b n)$ query time overhead per qualifying list. Note that we only need access to the leaf pages which contain the elements of list $X(t)$ in order, starting from the beginning of the list. The MVB (MVAS, TSB) structure cannot explicitly achieve this since its leaf pages are not connected. For a given leaf page there is not a single next sibling page. Rather, as $X(t)$ evolves the next sibling page varies over time.

In Appendix *A* we provide a solution that searches a given $X(t)$ starting from the first page that the list had at t. With this solution the history of every node list in the Interval Tree is abstracted by an ordered array *HT* which provides access to the first page of the list at each time; the other pages of the list are accessed by appropriate pointers. The first list page at some t is found by a logarithmic search that locates the entry of *HT* that is closest to t. This search still takes $O(\log_b n)$ query time, but this overhead can be avoided by "synchronizing" all the lists of the Interval Tree.

The main idea in "synchronization" lies on the fact that a query is answered by following a path on tree T (Figure 4). Consider for example a query for valid time v=3 and transaction time t. This query uses the path leading to v=3, i.e., the path created by nodes 4.5-2.5-3.5. Answering this bitemporal query means that the left list of node 4.5 is first rollback (paying a cost of $O(\log_b n)$ to search t in the list's *HT* array) then the right list of node 25 and finally the left list of node 3.5. Our aim is <u>not</u> to "pay" additional $O(\log_b n)$ query costs for locating t in the *HT* arrays of the lists in 2.5 and 3.5. Instead the search on the first *HT* should provide enough information (in the form of pointers) as where t is located in the lists below. The way to achieve this is by a variation of the "backward" updating technique we use in Appendix *A*, for making a list partially persistent. This technique is now used "vertically", on tree paths. When the *HT* array of a node list is very active creating many entries (if this list is very active or if the first page of the list changes often) the *HT* arrays on the parent node lists are informed (every some constant number of new entries).

Synchronization is combined with the overall pagination (for details we refer the reader to [KTF95]). The Bitemporal Interval Tree 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 m is the number of intervals contained in the current timeslice $ht()$ when the update is performed. The stated query performance is for queries with valid-time predicates included in set U. To represent the whole universe of valid times we add special elements $-\infty$ and ∞ (the "from-ever" and "for-ever" notions on valid time) and sets U_+ and U_-, that represent the finite elements which are greater/less than the maximum/minimum element of U respectively ([KTF95]).

For queries where v belongs to U_+ (or U_-) the query time becomes $O(\log_b n + e/b)$; e corresponds to the number of elements that the list of node U_+ (U_-) had at query time t. The $O(\log_b n)$ component is for rolling back this list to transaction time t and the $O(e/b)$ component is for searching it. At worst all e elements have to be checked but not all contribute to the answer.

6. The Bitemporal R-Tree

The Bitemporal Interval Tree guarantees good worst case query and update performance but has various limitations. Inherently from the interval tree, each object is kept in two places (the left and right node lists) thus doubling the space. The current implementation provides the good performance for "*/point/point" queries that are inside a known valid time universe U; e.g., the method is not completely dynamic since its performance is "focused" on a particular area of valid time queries. Directly associated with the universe size is the size of the backbone tree T; a large U will make the space overhead due to tree T significant as related to the total number of changes n. Hence it is primarily for applications with small valid-time focus of interest as related to the size on the transaction time (which is related to the number of changes).

The above limitations emanate from our choice to represent each *ht* with an Interval Tree. To overcome this problem we *ease* the requirement for good worst case performance and apply the partial persistence methodology to a more robust structure that supports interval intersection queries with good average performance.

An excellent choice for such structure is the R-tree [G84] since it is also well-paginated and uses linear space. While its worst case update and query performance are large, the ephemeral R-tree has been shown to work rather satisfactorily in practice: on average, $O(\log_b k)$ I/O's per interval update and $O(\log_b k + a/b)$ I/Os for interval intersection queries (as before k corresponds to the number of intervals in the tree and a is the size of the answer to an interval intersection query). Multidimensionality is another advantage of the R-tree. Adding a separate dimension for the key attribute of each interval-object, allows the R-tree to efficiently address queries of the form: "find all intervals intersecting a given interval E and whose keys are in a given range K".

Partial persistence coupled with the R-tree's multidimensionality enable the Bitemporal R-Tree to answer efficiently "range/range/point" queries. Using partial persistence the R-tree representing the history timeslice *ht(t)* is first conceptually accessed. This tree is then searched top-down to answer the "range/range/-" part of the bitemporal query. This top-down search is conceptually equivalent to searching an ephemeral R-tree that stores the intervals of *ht(t)*.

Another possible multidimensional structure which handles intervals is the Segment R-tree (SR-tree) [KS91]. We did not use the SR-tree since its ephemeral form uses more than linear space to the number of intervals stored in it. This would only increase the overall space of a partially persistent SR-tree. In addition, the ephemeral SR-tree has been designed to support mainly interval additions. Interval deletions were assumed to be infrequent [KS91] and are not supported as efficiently (they require finding all possible segments of the deleted interval in the SR-tree). This is not the case in bitemporal applications where intervals can be updated frequently.

An R-tree [G84] is an external, dynamic, balanced multiway tree that stores multidimensional objects. Each node corresponds to a disk page. For simplicity let each data object being specified as a *degenerated* 2-dimensional rectangle that has a key attribute (which may not be unique) in one dimension and an interval on the other (corresponding to a valid-time interval). Such a data object is stored in a leaf page as a *data* record with three fields: its key and the two interval endpoints. Each leaf page is associated with a bounding, 2-dimensional rectangle that whose area contains all rectangles in the page. Non-leaf pages contain *index* records of the form *(r*, *child_pointer)* where child_pointer is the address of a lower tree page and *r* is the covering rectangle of the lower page. If more dimensions are used to represent data objects the dimensionality of *r* increases analogously. As with a B+-tree an R-tree page is at least half and at most completely full of records.

In making the ephemeral R-tree partially persistent we can use ideas from previous work on partially persistent B+-trees, in particular MVB-tree [BGO+93] (and its improvement the MVAS [VV95]) and the Time-Split B-tree [TSB89]. This is because both B+ and R-trees are multiway-balanced structures that evolve through page splits and merges. There are however differences that affect the implementation of the Bitemporal R-tree. In contrast with a B+-tree, the R-tree does not keep a linear order among the objects it stores thus creating various possible merging policies. Furthermore, it is likely to have object insertions or deletions on an R-tree page that do not cause a page overflow or underflow but may still change the bounding rectangle associated with this page and provoke further changes on its ancestors. Finally, the multidimensionality of the ephemeral R-tree and the specifics of valid-time allow for various optimizations on the performance of the Bitemporal R-tree that should be examined.

The design of the Bitemporal R-Tree was influenced by the MVB and MVAS trees. We did not use the Time-Split B-tree since it is geared towards applications where the most frequent changes are object insertions and updates. Object deletions are less frequent (*step-wise constant* data [LS89]). Each version of an R-tree data record would then be associated with only one timestamp. For a newly inserted record the timestamp stores the record's insertion-time; subsequent record updates create new versions of the record, each timestamped with the time of the update. Deletions are implicitly represented by some special value. Adapting the [LS89] policies for the Bitemporal R-tree would use less overall space as compared with the MVB and MVAS approaches (one reason is that pure key-splits are allowed); however some of the queries may not be as efficient especially if bitemporal object deletions are frequent. We did not follow this approach but it is an interesting avenue for further research (in particular the effect of pure key-splits on the R-tree updating).

Subsection 6.1 provides an overview of the Bitemporal R-Tree and subsection 6.2 presents the optimizations we performed for tuning its performance.

6.1 Method description

The structure of the Bitemporal R-Tree is a directed acyclic graph of pages. Conceptually it stores the various states that the initial ephemeral R-tree assumes through its transaction-time evolution. As a result, the graph embeds many R-trees and has a number of root pages. Each root is responsible for providing access to a subsequent part of the ephemeral R-tree's evolution.

Data records in the Bitemporal R-Tree leaf pages maintain the transaction-time evolution of the corresponding R-tree data records. Each record is thus extended to include two additional fields: *insertion-time* and *deletion-time*, representing the transaction-time that the corresponding R-tree record was inserted and logically deleted in the bitemporal database. Similarly, index records in the non-leaf pages of the Bitemporal R-tree maintain the evolution of the corresponding index records of the ephemeral R-tree and are also augmented with insertion-time and deletion-time fields. Therefore each record has a transaction interval during which it is called *alive*.

Assume that each page in the Bitemporal R-tree has a capacity of holding b records. Similarly with [BGO+93, VV95], a page is called *alive* if it has not been *time-split* (see below). With the exception of root pages, for all transaction-times t that a page is alive it must have at least q records that are alive at t ($q < b$). This requirement enables clustering of the alive objects at a given time in a small number of pages, which in turn will minimize the query I/O.

The first step of an update (insertion or deletion) at the transaction time t locates the target leaf page in a way similar to the corresponding operations in an ephemeral R-tree. Note that this step is carried out by taking into account the transaction-time intervals of the index and data records visited, i.e., only the latest state of the ephemeral R-tree is traversed. An update leads to a *structural* change if at least one new page is created. *Non-structural* are those updates which are handled within an existing page. After locating the target leaf page, an insert operation at the current transaction time t adds a data record with a transaction interval of [t, *now*) to the target leaf page. This may trigger a structural change in the Bitemporal R-tree, if the target leaf page already has b records. Similarly, a delete operation at transaction time t finds the target data record and changes the record's interval to [insertion-time, t). This may trigger a structural change if the resulting page ends up having less than q alive records at the current transaction time as a result of the deletion. The former structural change is a *page overflow*; the latter is a *weak version underflow* [BGO+93].

Page overflow and weak version underflow need special handling: a *time-split* is performed on the target leaf-page. This is similar to the time-split of [LS89] or the page copying of [TK95]. The time-split on a page x at time t, is performed by copying to a new page y the records alive in page x at t. Page x is considered *dead* after time t. (We can assume that the deletion-time field of all of x's alive records is changed to t even though this is not needed in practice). Then the resulting new page has to be incorporated in the structure.

Briefly, there are three cases for handling the new page y [BGO+93][4]. First, if the number of records in y are within a certain specified range, y is directly inserted in the Bitemporal R-Tree structure. (This specified range is known apriori. In short, the number of records should be between $q+e$ and $b-e$ where e is a predetermined constant. Constant e works as a buffer that guarantees that a new structural change to the new page y can happen only after at least e new changes). The page insertion is carried out by accessing the parent page of x, marking the index record to x as deleted at the current time t, and inserting a new index record pointing to page y. (Conceptually this implies that page x is dead, i.e., not accessed for times larger than t). Even though these changes occur in an internal page, they are similar to insertion and deletion of data records in a leaf-page and are handled identically. Similarly, these insertions and deletions can create new changes up the tree to the ancestors and so on. The second case is if the resulting page y has more records than the specified range; this is called a *strong version overflow* condition and is handled by splitting y into two pages and then accommodating these pages in the structure in a manner similar to the one described above. The third case is if page y has less records than the specified range; this condition is called a *strong version underflow* and is handled by merging y with another "sibling" page and then accommodating the new page(s) in the tree.

There are two basic differences in the way the Bitemporal R-Tree is updated as compared to

[4] Note that the improved policies of [VV95] are somewhat different in handling page y.

the partially persistent B+-tree [BGO+93, VV95]. These differences are:

(1) The single order among the stored elements in a B+-tree creates an order among the tree's pages, too. Hence, a B+-tree page has at most two sibling pages and these are the only possible candidates for this page to merge with, if needed. In comparison, the ephemeral R-tree stores spatial objects and hence the notion of a sibling has to be redefined. Note however that merging in an ephemeral R-tree is not handled explicitly. If a page goes below the lower number of records due to deletions this page is not merged with another page. Instead, the records of the underutilized page are reinserted in the R-tree structure [G84, BKKS90].

The reinsertion method is not feasible in the Bitemporal R-tree, since a persistent structure "records" all changes that happen in its state. An underutilized page of an R-tree is half full and thus it can cause $O(b)$ record reinsertions. Each record reinsertion could at worst modify the whole path in the R-tree (i.e., logarithmic number of changes). Recording all these changes in the Bitemporal R-tree will require excessive space. To avoid this problem, the Bitemporal R-tree performs merging explicitly. Merging with a sibling may still change the whole path but this will happen once for the under-utilized page. It is an interesting optimization problem to chose with which sibling a page is merged; the various policies we examined are discussed in section 6.2.

(2) The second difference is with the way insertions and deletions are handled when they do not lead to structural changes. In an ephemeral B+-tree, an insertion to a page that has enough empty space is simply performed by adding the new key in the page; no parent page is updated. In an ephemeral R-tree a similar insertion may increase the geometric area covered by the data page. Then the parent page must also be changed, in particular the rectangle of the index record that points to the data page (so that the information about previous data page area is not lost). As this may propagate to the root, an insertion can cause a logarithmic number of updates even though no new page is added on the ephemeral R-tree.

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

A high level algorithmic description of the Bitemporal R-tree updating process appears in Appendix B. We proceed with a discussion on query processing. As with the ephemeral R-tree, query processing follows a top-down search, starting from some Bitemporal R-tree root. Since updates can propagate to ancestors, a Bitemporal R-tree root may become full and time-split. This creates a new root page which in turn may be split at a later transaction time to create another root and so on. By construction, each root of the Bitemporal R-tree is alive for a subsequent, non-intersecting transaction-time interval. Efficient access to the root which was alive at time t is possible by keeping an index on the roots, indexed by their time-split times. Since time-split times are in order this root index is well-paginated.

Answering a bitemporal query on transaction time t has two parts. First, using the root index, the root alive at t is found. This part is conceptually equivalent to accessing timeslice $ht(t)$ or, more explicitly, accessing the ephemeral R-tree representing the intervals of $h(t)$. Second, the answer is found by searching this tree in a top-down fashion as in a regular R-tree. This search takes into account the record transaction interval. The transaction interval of every record returned or traversed should include the transaction time t, while its valid time interval and its key attribute should satisfy the valid and key query predicates respectively. Answering a bitemporal query on a transaction time interval $P = [t,\ t')$ is similar. First all roots with transaction interval intersecting P are found. Starting from the first alive root, Bitemporal R-tree pages are recursively accessed provided that they have intervals intersecting P. Since the Bitemporal R-tree is a graph, some pages are accessible by multiple roots. Re-accessing pages can be avoided by keeping a list of accessed pages. This search method finds all records with transaction intervals that intersect P. Hence it may include many copies of the same record; the query answer is found by eliminating such copies.

6.2 Performance Tuning

The performance of the Bitemporal R-tree is described by three interrelated parameters: space, update and query-time. The space of the Bitemporal R-tree is $O(n/b)$. Intuitively, this is because our proposed updating modifications keep the space per update bounded.

Partial persistence implies that the update and query performance of the Bitemporal R-tree are bound by the performance of the underlying ephemeral R-tree. Intuitively this holds because updates to the Bitemporal R-tree are always applied to the latest ht, or conceptually to the most current R-tree. Similarly, queries are applied to the appropriate *ht(t)*, i.e., as if the ephemeral R-tree implementing *ht(t)* is traversed. While the worst case query and update performance of an R-tree are large, R-trees can have good average case behavior and thus adhere to possible optimizations. Since the ephemeral R-tree stores the valid-time intervals, we perform three optimizations as related to the way valid-time intervals are updated or stored. The first two optimizations are based on choosing merging and splitting strategies. The third optimization is based on the storage representation of intervals in an R-tree.

Merging policies. The selection of the sibling page with which an underutilized page will merge is associated with defining a "closeness" metric. We have applied five heuristic criteria.

The first criterion (called *overlap*) chooses as a sibling the currently alive page under the same parent which has the most bounding rectangle area intersection with the underutilized page. If multiple pages have the same area intersection, the one needing the least area expansion is chosen.

In the second criterion (*min_area*), the sibling of a page is chosen to be the currently alive page under the same parent whose bounding rectangle area needs the least geometric <u>expansion</u> to incorporate (include) the data records of the underutilized page.

The third involves choosing the page which when merged with the underutilized page has the least *margin* (or *perimeter*), which is the sum of the lengths of all sides of the bounding rectangle.

The fourth criterion *(combined)* chooses the page that if merged with the underutilized page will provide a page with the least "$area + w{*}margin$"; here w is a small constant ($w \geq 0$), *area* is the rectangle area of the resulting page and *margin* its perimeter.

The last criterion *(random)* randomly picks one of the currently alive siblings of the underutilized page for merging, i.e., no real "closeness" metric is used except that the sibling is also alive.

Splitting policies. Splitting is needed in two cases after a time-split. The first case is when a resulting page after a time-split satisfies the strong-version overflow condition (i.e., it has too many alive records) and hence needs to be split into two new pages. The second case is when the resulting page after a time-split satisfies a strong version underflow condition (i.e., it does not have enough alive records) but after merging with a sibling, the resulting page satisfies the strong-version overflow condition (because the combined number of alive records in the merged pages is large). Since splitting in these two cases deals only with the key and valid-time dimensions of the records (all records in question are alive in the transaction-time sense) we implement three heuristic splitting policies from ephemeral R-trees that attempt to minimize the total area of the page covering rectangles that result after the split. This covering is related to key and valid-time dimensions, only.

The first two policies are the *quadratic* and *linear* splitting of the basic ephemeral R-tree [G84]. They both assign records in two groups (each group corresponding to a new page), but differ on how groups are initialized and records are assigned. The quadratic policy initializes the two groups by picking the pair of records that would waste the most area if put in the same group (the area of the rectangle that covers both records minus the covering areas of the records themselves would be the greatest). Remaining records are assigned to groups in steps. At each step the area expansion required to add each remaining record to each group is calculated, and the record assigned is the one showing the greatest difference in the area expansion needed between the groups.

The linear splitting policy picks records based on their "normalized separation" [G84], defined along each dimension. For example, among the records to be split, the separation along the valid-time dimension is the distance between the highest interval left-endpoint and the lowest interval right-endpoint. This separation is divided by the width on this dimension (highest right-endpoint minus the lowest left-endpoint) to create the normalized separation along this dimension. A similar computation takes place along the key dimension. The linear splitting algorithm initializes the two groups by picking the pair of records with the greatest normalized separation along any of the two dimensions. The linear policy also differs on placing the remaining records: a record is randomly picked and placed in the group that needs the lesser area expansion to accommodate it.

The third splitting policy, *r_star*, comes from the ephemeral R* tree [BKKS90]. To outline the policy we first define a few relevant terms. The margin value of two rectangles is defined to be the sum of their margins. The overlap value between two rectangles is the amount of the common area shared between them. Records are referred to as being ordered in a given dimension if they are sorted according to their values in that dimension. If intervals are stored in a given dimension, the ordering is made according to the starting values of the intervals; ties are resolved using the interval ending values. The *r_star* splitting policy is based on determining various distributions of a page's records after ordering them in each dimension. The *k*-th distribution in a given dimension keeps the initial *k* records in the first group and the remaining in the second group. Distributions which place less than a predefined number of records in any of the groups are not considered since these distributions may lead to bad query performance. The splitting policy starts by first choosing the dimension on which to split. This is performed by ordering the records in each dimension and picking the dimension which has the distribution that leads to the overall minimum *margin* value of the two rectangles that cover the two groups created. The actual split follows: records are ordered on the chosen dimension and distributed in two groups using the distribution which minimizes the *overlap* value between the two rectangles which cover the two groups created in the distribution.

We also considered splitting policies from the ephemeral R+-tree [SRF87]. However those policies are not directly applicable since an R+-tree keeps multiple copies of the records. This is costly in terms of space for the Bitemporal R-tree, since copies of records from all the previous versions should be retained.

Interval transformation. Large intervals tend to increase page area overlapping which in turn affects the performance of the ephemeral R-tree and hence the performance of the Bitemporal R-Tree. To deal with this problem we transformed one-dimensional valid-time intervals into points in a two-dimensional space. A valid-time interval I with v_s, v_e starting and ending values corresponds to point (v_s, v_e) in the two-dimensional space (for simplicity we denote the beginning of valid times as time zero). Such points are located in the upper diagonal area, since the starting value is always less than the ending value. Storing intervals as points would allow for better pagination which should on the average improve both the update and query performance.

A similar problem with interval storing appears more severely in the 2-R methodology. In particular, the front R-tree (Figure 3) will try to accommodate the vertical valid-time intervals I into pages, favoring the creation of vertical page areas that store near-by (in the transaction sense) intervals. The problem is intensified since vertical intervals are placed at given transaction times. The horizontal query interval H of Figure 3 will intersect many of these vertical page areas even though some of these pages may not contribute to the answer and thus affecting the query performance. To avoid this problem we also implemented the 2-R methodology using the above translation of intervals to points. A query for all intervals intersecting H is translated into finding all three-dimensional points in a semi-infinite *cuboid* of height t.

7. Performance Analysis

We implemented the Bitemporal R-Tree (BRT) and the Bitemporal Interval Tree (BIT) and compared them with the 2-R and the straightforward 1-R approach. To avoid overpopulated graphs we first compare the performance of the various Bitemporal R-Tree implementations. We then proceed with comparisons among the optimized Bitemporal R-Tree and the other structures (the BIT, the 2-R and the 1-R). The comparisons begin with the "*/point/point" query and continue with the "range/point/point", "range/range/point" and finally "range/range/range" queries.

7.1 Experimental Setup

For the first set of our experiments we selected fourteen data files (evolutions), each containing 60,000 updates. An update is the addition or deletion of a valid-time interval. To examine the effect of the "mix" of updates on performance, seven of the files had 35,000 insertions and 25,000 deletions (the 35/25 ratio) and seven had 43,000 insertions and 17,000 deletions (the 43/17 ratio). The 43/17 group of data files has many long-lived bitemporal objects (in the transaction-time sense) since only 17,000 of the inserted objects are deleted. In comparison, the 35/25 group has many short-lived bitemporal objects.

Each data file was created by first choosing the valid time intervals. The starting point of a valid interval was selected randomly (with a uniform distribution) anywhere in $U = \{1,..., V\}$. For the above data files we used $V = 1024$. Then the ending point was chosen uniformly within a distance of K from the starting point. Hence $K/2$ is a good estimate for the average size of the valid-time intervals in a data file. To study the effects of the average valid-time interval size data files were created with the following $K/2$ values: 50, 100, 200, 250, 350, 500 and 600.

The transaction time evolution was created next. For simplicity we have assumed that at each transaction time exactly one update occurs. The number of updates per transaction does not affect the update processing which is proportional to the total number of updates over all transactions. Hence in each data file there are 60,000 distinct transaction times (equivalently, the size of the transaction-time universe is 60,000). All evolutions were setup to start with 4,000 inserts, i.e., in each of the first 4,000 transaction times a valid time interval was picked randomly for insertion (from the valid time intervals created above). This guarantees a structure with reasonable number of intervals before carrying out the deletes. At each of the later transaction times, a new interval was randomly inserted with probability p while with probability $1-p$ one of the already inserted intervals was randomly deleted. The value of p was chosen accordingly so as to provide the two groups of insertions/deletions.

7.2 Performance Tuning of the BRT

We used various implementations of the Bitemporal R-Tree based on: (a) two (transaction) time-split policies (i.e., the *basic* [BGO+93] and the *improved* [VV95]), (b) five merging policies (the *overlap, min_area*, *margin, combined* and *random*), (c) three (key/valid-time) splitting policies (the *quadratic*, *linear* and *r_star*) and (d) two valid-time interval representations (*interval* or *point*). Instead of presenting all possible combinations, we proceed with a step-by-step optimization by incrementally fixing the various parameters. All BRT optimizations are shown using the 35/25 datasets; the 43/17 datasets behaved similarly. In all experiments the page size is 1K and each page has a total capacity of about 50 bitemporal objects.

Figures 5-7 present the effect of the *basic/improved* time-split and the *interval/point* representation on the BRT performance. Each BRT was implemented using the *margin* merging policy and the *r_star* for valid-time/key splitting. Figure 5 shows the space used in number of pages. As with B+-trees, the *improved* time-split implementations use less overall space than the *basic* time-split. The *point* representation uses less space than the corresponding *interval* representation since intervals introduce more restrictions in their placing into pages that result to less page utilization. In addition, the space used by the BRTs with the *interval* representation tends to increase with the average size of the valid-time intervals because larger intervals are more difficult to paginate.

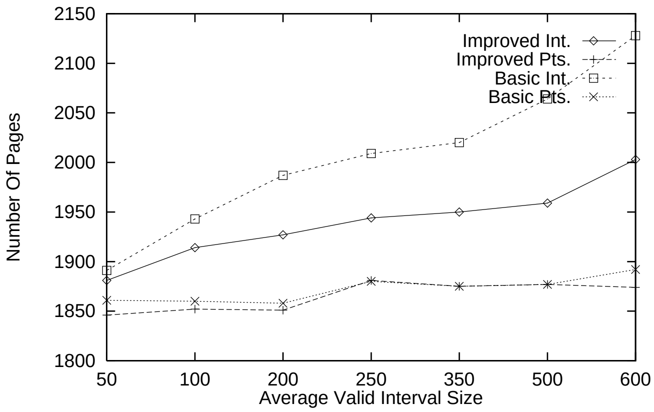


Fig. 5: Number of pages (space) used by the BRT under *basic/improved* and *interval/point*. The 35/25 files are shown with *V*=1024 and varying avg. valid-time interval sizes.

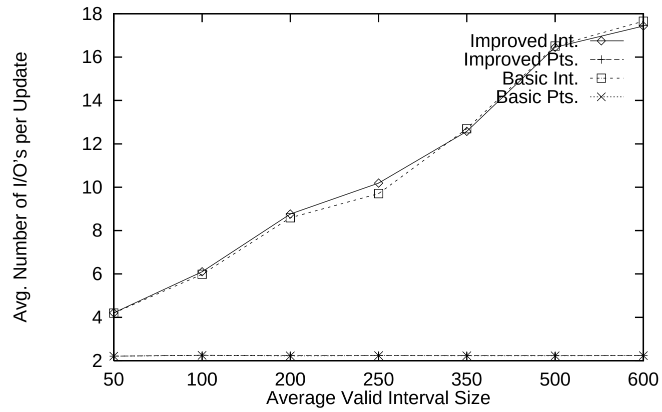


Fig. 6: Avg. number of page I/O's per update for the BRT under *basic/improved* and *interval/point*. The 35/25 files are shown with *V*=1024 and varying avg. valid-time interval sizes.

Figure 6 depicts the average number of I/Os per update. During the update process, bitemporal objects in the current *h(t)* are normally accessed using the combination of their key, valid and transaction-time attributes. To clearly show the effect of the valid-time interval representation, we present the average update per change without using the *key* attribute. Depending on its selectivity, the key attribute can assist in identifying the updated object. A key attribute with low selectivity (for example, *salary*) implies that many objects have the same value on this attribute thus limiting its usage in identifying the updated object. Ignoring the key attribute for updating behaves as an extreme of the low selectivity cases or equivalently, as if all objects have the same value on the key attribute. (We examine the full effect of key selectivity on updates later). Figure 6 indicates that the BRTs with point representation have less updating that is also independent of the average interval size. The interval based approaches have updating that increases with the average interval size. This is again due to the difficulty of efficiently placing larger intervals into pages.

To compare the query performance we computed 10,000 "*/point/point" queries for each data file. Each query is selected by choosing the valid time *v* randomly with a uniform distribution within the set of valid times *U* and the transaction time *t* randomly with a uniform distribution over all

60,000 transaction times. Figure 7 shows the average number of pages accessed per query. The *point* representation has less query time than the interval one and within the same representation the basic/improved time-split policies have basically identical behavior. For all implementations the query I/O increases with the valid interval size because the answer size also increases.

Based on the outcome of the above experiments we fix the BRT implementation to the *improved* time-split (less space) and the *point* representation (less update/query), and proceed with the effect of the merging and splitting policies. We actually run experiments that combined all merging with all splitting policies. In general, we got the best performance when using the BRT with the *r_star* splitting policy. Hence we only discuss the effect of merging policies when combined with the *r_star* splitting policy. The merging policies did not have a significant effect on query time and space (not shown). Query times for all BRTs were similar, with the *margin* policy to provide a slightly better query performance. All policies used an average space of 1860 pages with random variations less than 1.6%. The effect of the merging policies on updating appears in Figure 8. Among the five policies *random* and *overlap* had the worse updating. The rest behaved almost identically, with a slight advantage for the *min_area* and *margin*.

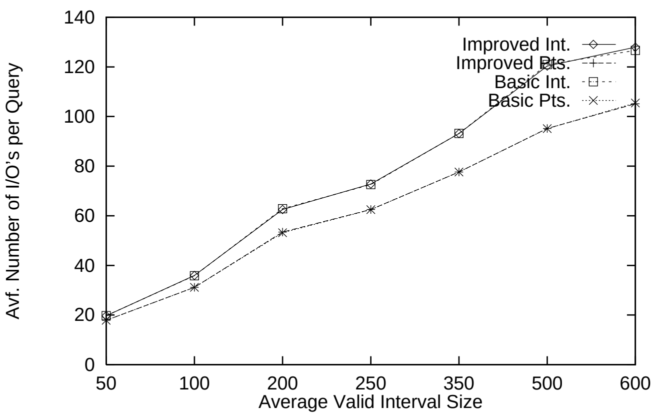


Fig. 7: Avg. number of I/O's per "*/point/point" query for the BRT under *basic/improved* and *interval/point*. The 35/25 files are shown with *V*=1024 and varying valid interval sizes.

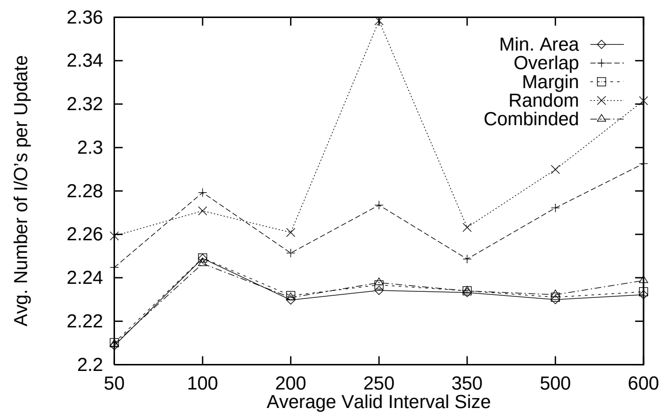


Fig. 8: Avg. number of page I/O's per update for the BRT under the various merging policies. The 35/25 files are shown with *V*=1024 and varying avg. valid-time interval sizes.

There are various reasons why merging policies have limited effect on the BRT performance. For all policies, the page to merge with is chosen among the currently alive siblings of the underutilized page. A page is alive as long as it has at least q alive records. Most pages operate between q+e and b-e alive records (section 6.1). In our implementation we had q = e =10 and b = 50. Hence for most pages the merging policies choose from a moderate number of 20-40 alive siblings. As the experimental results show, with the possible exception of the *random* policy, all merging policies avoided bad sibling choices. Even if some choices were worse than others, there are two more amortizing factors. First, the number of merges in an evolution is small as compared to the number of splits. Merging happens when (due to object deletions) a page underflows, i.e., when the number of alive records becomes lower than threshold q. In contrast, splits happen because a page overflows and are due to either real object insertions or record copies due to persistence. Since deletions are less than the summation of insertions and copies, merges are not as frequent. Second, the number of alive siblings at a given time t is a small part of $ht(t)$ which in turn is a small portion of the whole persistent structure that stores all ht's. (Note that merging in an ephemeral structure has more drastic effects since it deals with a larger portion of the structure.) Based on the query and update performance we fix the BRT merging policy to *margin*.

The effects of the valid-time/key splitting policies appear in Figures 9 and 10. The *r_star* policy uses comparatively less space (Fig. 9), update (not shown) and query (Fig. 10) than *linear* and *quadratic*. This is expected since the valid-time/key splitting policies deal with the page splitting of the most current $ht(t)$ as if it was an ephemeral structure. For an ephemeral R-tree it has been observed that the *r_star* policy performs better than the *linear* and *quadratic* policies [BKKS90].

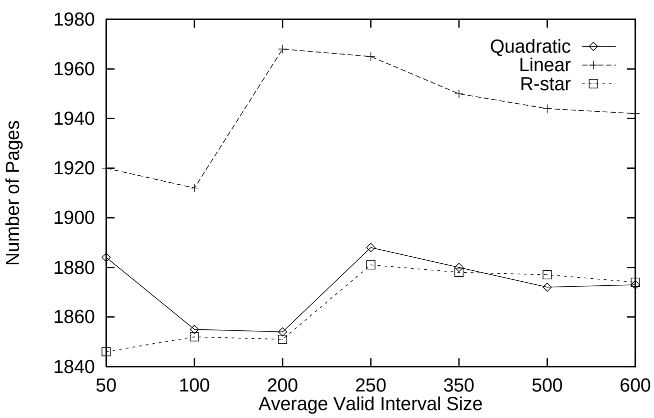


Fig. 9: Number of pages (space) used by the BRT under the various valid-time/key splitting policies. The 35/25 files are shown with V=1024 and varying avg. valid-time interval sizes.

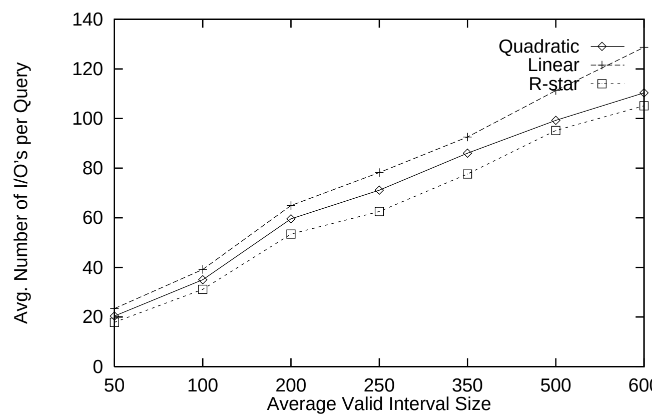


Fig. 10: Avg. number of I/O's per "*/point/point" query for the BRT under the valid-time/key splitting policies. The 35/25 files are shown with V=1024 and varying valid interval sizes.

As a result of the BRT optimization, in the rest when referred to BRT we imply the "optimized" implementation that uses the *improved*, *margin*, *r_star* policies and the *points* representation.

7.3 Performance Comparison of the Various Approaches

We proceed by comparing BRT with the BIT, the 1-R and 2-R approaches. Each of the 1-R and 2-R was implemented using the R* tree [BKKS90] (which uses the *r_star* splitting policy and reinsertions instead of page merging) and two implementations based on intervals and points.

Figure 11 depicts the space consumption for the 35/25 group. The 1-R with interval representation (1-Ri) has the best overall space utilization since a single record per bitemporal object is used. The 2-R methods also use one copy per object but the page utilization is slightly lower. The BRT uses more space (due to record copying) but only about 57% more than the 1-Ri method (on average 1850 versus 1180 pages). The Bitemporal Interval Tree had the largest space requirements. In addition to copying due to partial persistence, each interval in the backbone Interval Tree is kept twice (in some left and right lists). Note that the space requirements for the BIT tend to decrease as the average interval size increases; large intervals will be stored in node lists near the root, leaving the rest of the backbone structure empty (hence less lists will be created). The BIT space is also affected by the way updates are performed on each partially persistent list (see Appendix A). Its overall space can be reduced if instead of the [BGO+93, VV95] policies we use the TSB [LS89] time splits. We did not implement a BIT with the TSB policies but we estimated its space requirement based on the average number of copies per object used in TSB.

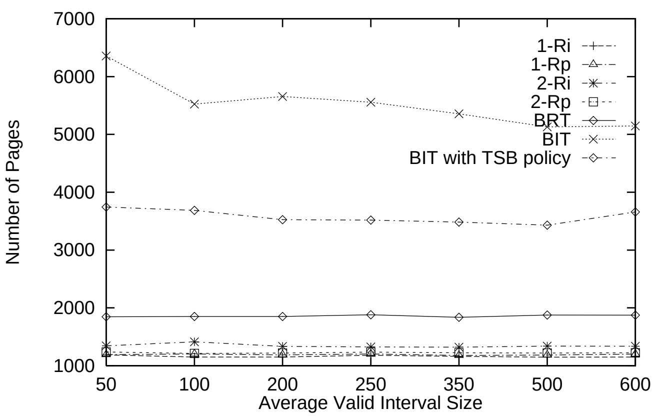


Fig. 11: Number of pages (space) used by the BRT, BIT, 1-R and 2-R methods for the 35/25 files with V=1024 and varying avg. valid-time interval sizes.

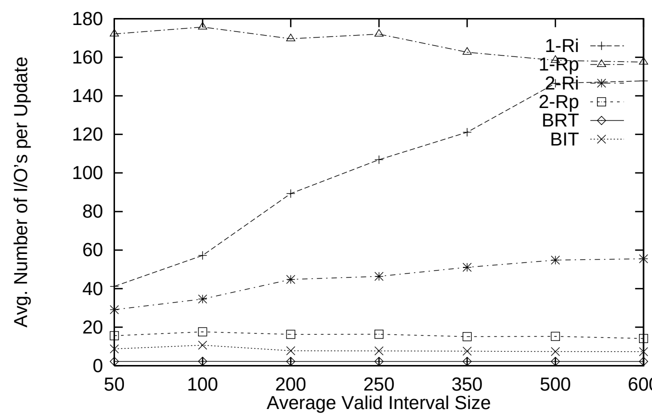


Fig. 12: Avg. number of page I/O's per update for the BRT, BIT, 1-R and 2-R methods for the 35/25 files with V=1024 and varying avg. valid-time interval sizes.

Figure 12 presents the average number of pages accessed per update in all compared approaches, for the 35/25 data files. Again the key attribute was not used during updating (low selectivity). The BRT has the lowest average I/O per update followed by the BIT update. The BRT, BIT and 2-R with points (2-Rp) have updating that is basically independent of the valid-time interval size. The update of the 2-R with intervals (2-Ri) and the 1-Ri increases with the valid-time interval length (larger intervals are more difficult to paginate). The 1-Ri has higher update than the 2-Ri because of the additional burden of transaction endpoints extending to *now*. The 1-R with points (1-Rp) has the worst update processing since in the point formulation, the extension to *now* problem is intensified (*now* becomes a common endpoint to many bitemporal objects).

To compare the query performance we computed 10,000 bitemporal “*/point/point” queries for each data file (Figure 13). The partial persistence approaches have the best performance, with the BIT being faster especially as the interval size increases (the performance of the BIT and the BRT methods is virtually the same for the smaller interval sizes). The 1-Rp method has the worst query performance, followed by the 2-Ri method. The 1-Ri and 2-Rp methods are comparable: 1-Ri is better than the 2-Rp method for smaller valid interval sizes but the 2-Rp prevails for larger interval.

To validate the query experimental results we need a measure of the query answer size. As the interval size increases the average number of answers also increases. For the 35/25 data files the average query size was 305, 595, 1110, 1320, 1710, 2150 and 2335 objects/query for the 50, 100, 200, 250, 350, 500 and 600 valid interval sizes, respectively. A good access method should find the answer with as few data page accesses as possible (accessing directory pages corresponds to a very small part of the query I/O, about 5 pages). In our experiments, the 1-Ri method uses an average of 30 objects per page (there are about 35,000 objects and the space used is about 1180 pages). This is because each page is utilized only about 60% to allow for free space. As a result, any method that uses this minimal R-tree utilization, would need at least 10, 20, 37, 44, 57, 72 and 78 data pages respectively, to store the objects in the average answer for each data file in the 35/25 group. To compute these answers and after excluding directory page accesses, the BRT makes 13, 26, 48, 57, 72, 90 and 100 page accesses on the average, respectively. This constitutes an 30% increase on the average, over the “best” possible (R-tree utilization) solution. The BRT has smaller page utilization since the copying due to partial persistence occupies some page space. In comparison, the 1-R and 2-R tree methods search a much larger number of pages.

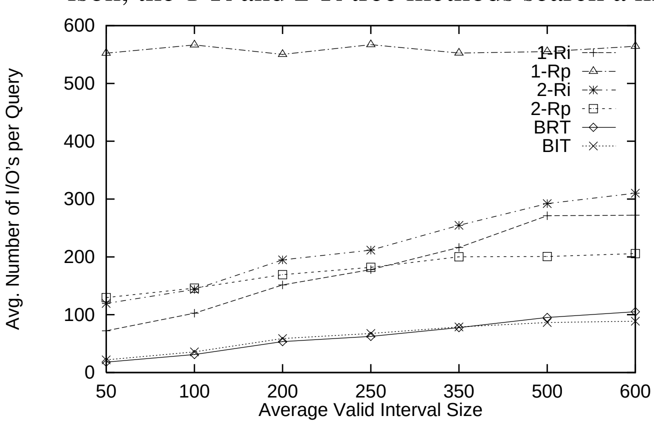


Fig. 13: Avg. number of I/O’s per “*/point/point” query for the BRT,BIT, 1-R and 2-R methods for the 35/25 files with V=1024 and varying avg. valid-time interval sizes.

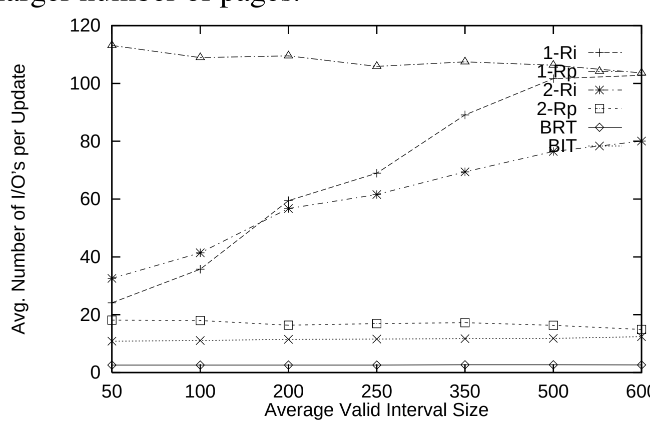


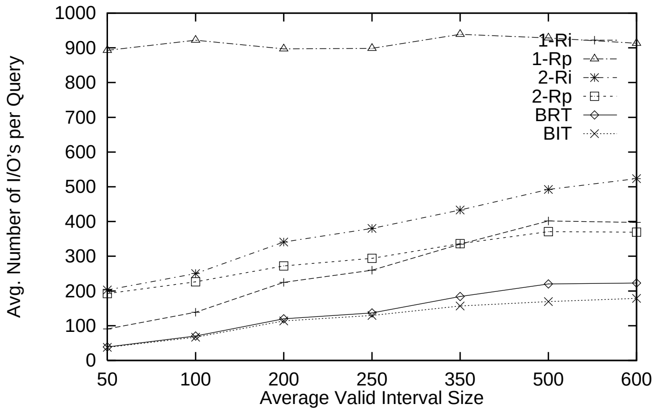
Fig. 14: Avg. number of page I/O’s per update for the BRT, BIT, 1-R and 2-R methods for the 43/17 files with V=1024 and varying avg. valid-time interval sizes.

The performance comparison of all methods for the 43/17 group follows. The space performance (not shown) is similar as for the 35/25 case. For all methods the space is more than the corresponding 35/25 case since there are now more objects created (less deletions). The BRT used on average 65% more pages than the 1-Ri approach (2440 pages versus 1480 pages); the slight increase in BRT space overhead against 35/25 is because the number of insertions has increased and consequently the number of copies due to time-splits. The increased number of insertions makes the update (Figure 14) of the BRT, BIT and 2-R methods to increase, too. In contrast the update of the 1-R methods decreases because the number of deletions has decreased and deletions are a real burden in the 1-R approach. The query time performance for the "*/point/point" queries is depicted in Figure 15; all methods behave similarly as for the 35/25 case. Based on their behavior for the 35/25 and 43/17 datasets we choose the 2-Rp and 1-Ri to be the representatives of the 2-R and 1-R approaches for the rest of our experiments.

Effect of the Valid-Time Universe Size. The performance of BIT is affected by the size of the valid-time universe, since the backbone structure T has size $O(V)$. Large universe size implies more space overhead. To study the effect of valid-time universe size we used a new set of five data files with V equal to: 1024, 2048, 4096, 8192 and 16384. The average valid-time interval size was chosen to be about $V/2$ (i.e., 500, 1000, 2000, 4000 and 8000 respectively). There were again 60,000 transactions with the 35/25 ratio of insertions/deletions. The results appear in Figures 16-18.

As expected the BIT space increases gradually with the valid universe but remains constant for the other methods. The update of BIT and BRT is independent of the valid universe size. Despite that the actual number of changes remains the same, the update of the 1-R and 2-R decreases as the universe increases. This is because the same number of intervals is spread into a larger space and can thus be accommodated easier. The query time (for "*/point/point" queries) of the BIT remains smaller than the rest and independent of the valid universe size, followed by the BRT. Query times for the 1-R and 2-R tend to decrease as the valid universe increases because of better pagination.

While BIT provides good query and update time, its dependence on the valid-time universe size makes it impractical for applications with large such universe (as compared to the number of transactions, i.e., the transaction-time universe). In these cases the other methods should be used. We thus proceed to study the behavior of the BRT, 1-R and 2-R methods under a large valid-time universe. We experimented with nine new data files having V = 16384 and the following average valid-time interval sizes: 50, 250, 500, 2500, 5000, 8000 and 10000. Each data file had again 60,000 transactions using the 35/25 ratio of insertions/deletions. The space (not shown) of all methods behaves as before, i.e., independently of the interval size (the BRT uses about 60% more pages



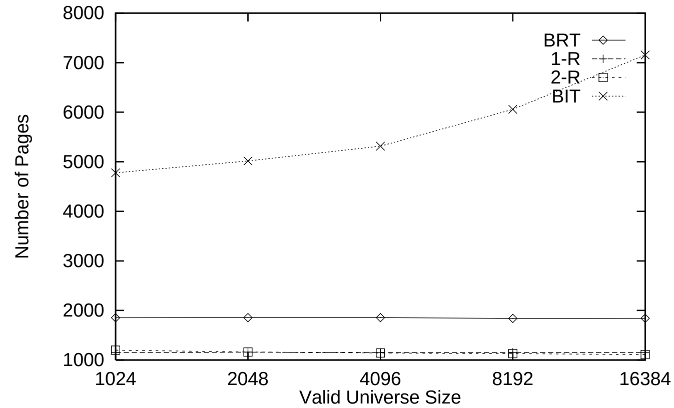
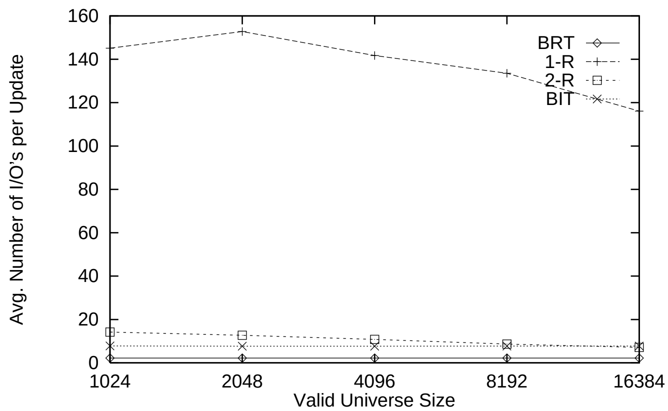


Fig. 15: Avg. number of I/O's per "*/point/point" query for the BRT, BIT, 1-R and 2-R methods for the 43/17 files with *V*=1024 and varying avg. valid-time interval sizes.

Fig. 16: Number of pages (space) used by the BRT, BIT, 1-R and 2-R methods for 35/25 files with varying *V*. Each file had *V*/2 average valid-time interval size.



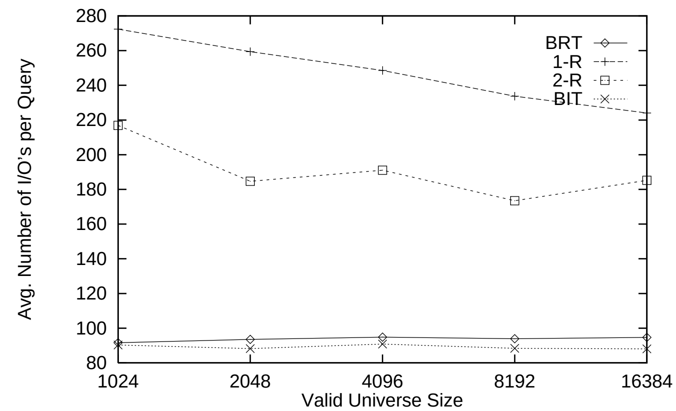
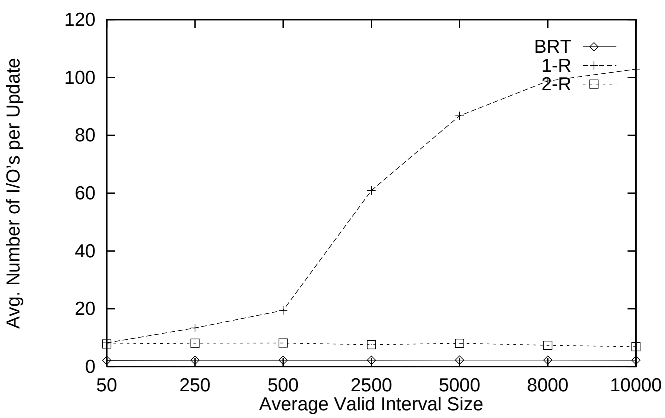


Fig. 17: Avg. number of page I/O's per update for the BRT, BIT, 1-R and 2-R methods for 35/25 files with varying *V*. Each file had *V*/2 average valid-time interval size.

Fig. 18: Avg. number of I/O's per "*/point/point" query for the BRT, BIT, 1-R and 2-R methods for 35/25 files with varying *V*. Each file had *V*/2 average valid-time interval size.



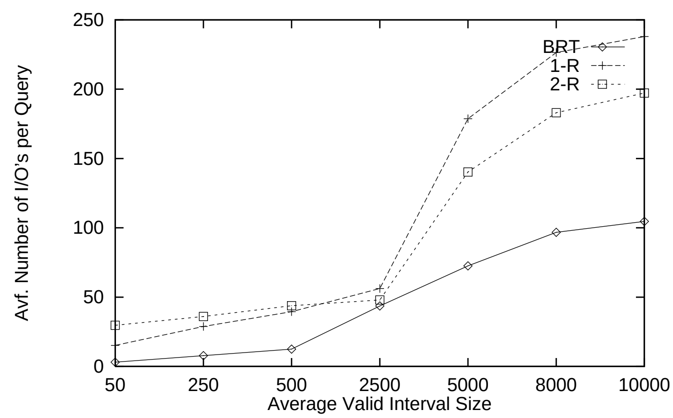


Fig. 19: Avg. number of page I/O's per update for the BRT, 1-R and 2-R methods for 35/25 files with *V*=16384 and varying avg. valid-time interval sizes.

Fig. 20: Avg. number of I/O's per "*/point/point" query for the BRT, 1-R and 2-R methods for 35/25 files with *V*=16384 and varying avg. valid-time interval sizes.

than the 1-R). The update (Figure 19) of the BRT and the 2-R are independent of interval size with the BRT having the least update. The update of the 1-R increases drastically with the valid interval size. The query times (for "*/point/point") were as expected (Figure 20), with the BRT being the fastest followed by 2-R and 1-R.

Effect of Key Selectivity. We proceed to examine the effect of key selectivity on updating (Figure 21). For this problem we experimented with the BRT, the 2-R and 1-R methods. The BIT was not considered since it only indexes the valid and transaction time attributes and not the key attribute. Each data file had 60,000 transactions with the 35/25 ratio, $V = 1024$ and average valid-time interval size equal to 250. Since there are 35,000 insertions per data file, we first created a "pool" of 35,000 keys. These keys were integers chosen randomly from 1 to 1,000,000. Since the number of random choices is much smaller than the key universe there were very few duplicates in the "pool" keys. Data files with different selectivities were created by "filtering" out the lower digits of the "pool" keys. The first data file with selectivity ~1/35000 used the exact "pool" keys. Assuming that most "pool" keys are distinct this data file corresponds to very good/high key selectivity (approaching the case of unique keys). Data for the file with selectivity ~1/1000 were created by making the last three digits of the "pool" keys equal to "000". Assuming that there were no duplicates in the 35,000 "pool" keys, the second file had an average of 35 duplicates per distinct "filtered" key value, i.e. selectivity around 1/1000. Similarly, the other data files were created by filtering out the last 4 and 5 digits of the "pool" keys. The BRT has the lowest update among all methods. This is because BRT focuses the update on the most current *ht* which has relatively small number of objects. For high selectivities the 1-R has less update than 2-R. High selectivity means that objects are accessed directly from the key attribute, but 2-R has still to search two trees. However, as selectivity decreases the key attribute becomes less important as compared with the effect of temporal attributes and the update of 2-R becomes better than 1-R. Figure 21 should be compared with Figure 12 where 2-Rp had also better update than 1-Ri. This is expected since Fig. 12 depicts updating if no keys are used which corresponds to even lower selectivity that the ~1/10 file.

Answering more general queries. The bitemporal "range/point/point" query was examined next. For this query we experimented with the BRT, the 2-R and 1-R methods. The BIT was not considered for this query since it would need to find all alive objects and then disregard the ones outside the query key range. Figure 22 shows the average query time results for various key selectivities using the same data files as Fig. 21. For each data file 10,000 queries were computed. A query was created by randomly choosing the query key range K (picking two random keys in the key uni-

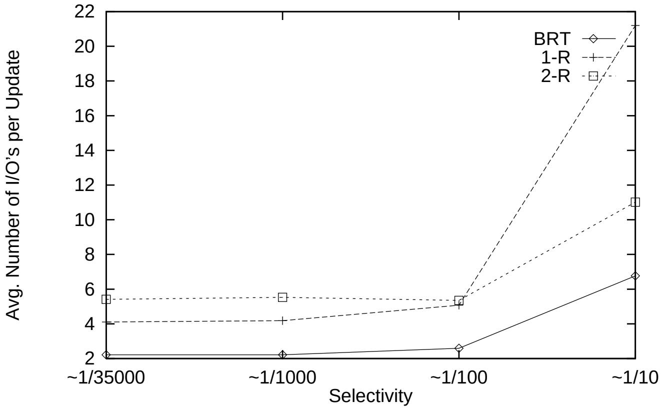


Fig. 21: Avg. number of page I/O's per update of the BRT, 1-R and 2-R methods for 35/25 files with V = 1024, avg. valid interval size 250 and varying selectivities.

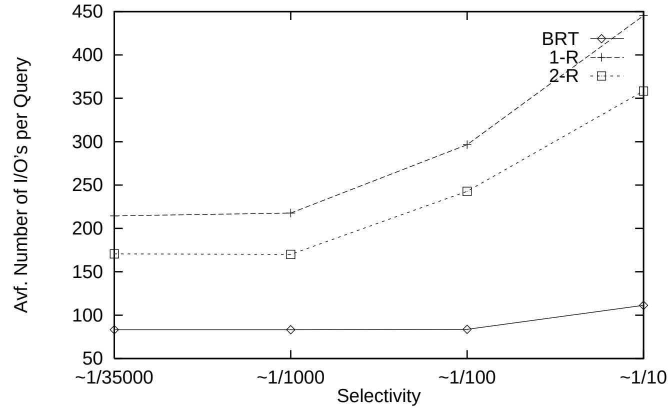


Fig. 22: Avg. number of I/O's per "range/point/point" query for the BRT, 1-R and 2-R methods for 35/25 files with V = 1024, avg. valid interval size 250 and varying selectivities.

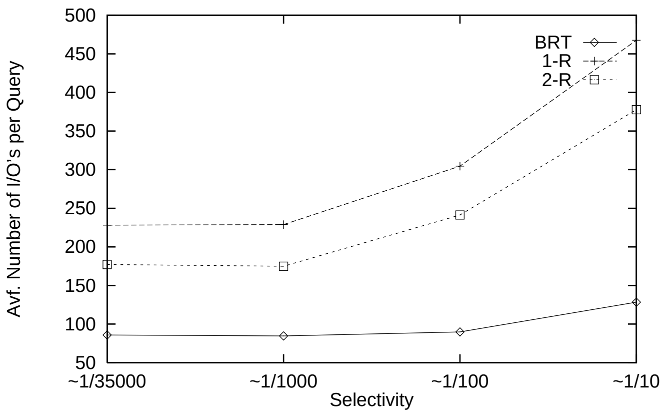


Fig. 23: Avg. number of I/O's per "range/range/point" query for the BRT, 1-R and 2-R methods for 35/25 files with V = 1024, avg. valid interval size 250 and varying selectivities.

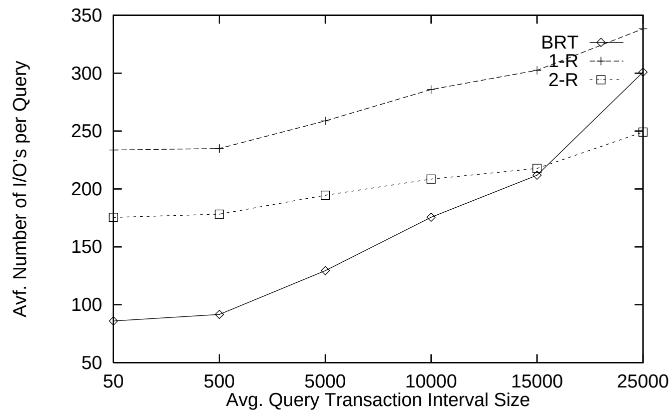


Fig. 24: Avg. number of I/O's per "range/range/range" query for the BRT, 1-R and 2-R methods for the 35/25 files with V=1024, avg. valid interval size 250 and varying query transaction-time interval.

verse), valid instant v and transaction time t. The BRT provides the best query performance followed by the 2-R which is consistently better than 1-R.

Figure 23 shows the average query time results for the "range/range/point" query and various selectivities. The same data files were used, with 10,000 queries per data file, choosing randomly the query key range K, valid-time interval E (by picking two instants in the valid universe) and transaction time t. The query performance remains the same, with the BRT method prevailing.

The last query examined was the "range/range/range" query. By its nature, partial persistence provides fast access to a particular $ht(t)$ by transaction time t. The performance of BRT depends on the size of the transaction-time interval P specified by the query. Figure 24 shows the "range/range/range" query time results for various values of P. The 35/25 data file with V = 1024 and average valid-time interval length of 250 was used. As P increases, the BRT query time increases since the BRT has to search more subtrees and may encounter more record copies. For small through medium size (as compared to the total transaction-time universe size of 60,000) query intervals *P*, the BRT has still the best query performance. The 2-R method becomes faster for large *P*'s. It is actually an interesting open problem to find a partially persistent structure which can address queries on transaction-time intervals with the same efficiency as for transaction-time instant.

8. Conclusions and Further Research

We have addressed the problem of designing efficient access methods for bitemporal queries and introduced two methodologies. The first methodology translates a bitemporal query into a partial persistence problem for which a method is then designed. This approach led to the Bitemporal Interval Tree and the Bitemporal R-Tree. The second methodology divides bitemporal objects in two categories, according to their transaction time behavior. In our implementations a separate R*-tree was used to store the data in each category (2-R method). We compared our methodologies with a straightforward approach that keeps bitemporal objects in a single R*-tree (1-R method).

In general, the partial persistent methods have better update performance than 1-R and 2-R. They also had better query performance except for "range/range/range" queries that specify very large transaction time intervals. The persistent methods should be preferred if some extra space can be tolerated. In particular, the Bitemporal R-Tree is a rather robust method that can address general bitemporal queries with good average performance, using only minimal extra space (about 60-65% more space than the minimal space of the 1-R method). Given the current cost of secondary storage, this seems a very comfortable price to pay for the performance provided by the BRT. Its advantage is that is processes queries as if an ephemeral R-Tree for the queried *ht(t)* is present.

The Bitemporal Interval Tree uses more space (still linear to the number of changes) because in addition to persistence, intervals are kept in two places. The BIT addresses queries that do not specify a key range. Its space is affected by the size of the valid-time universe thus it should be used for applications with small valid-time universe as compared to the transaction-time universe. The importance of BIT is more theoretical in nature since it guarantees good worst case performance for the above bitemporal queries (it is of special interest if we note the difficulty in designing practical access methods with good worst case behavior for the simpler, valid-time queries). The BIT actually provided the best overall average query performance for the "*/point/point" query. However for most applications the Bitemporal R-Tree is a more robust and practical choice.

The 2-R methodology is a good alternative to the partially persistent methods. It has the advantage of using off-the-shelf methods and consumes almost minimal space. Nevertheless, its update performance is worse than the persistent methods and similarly its query time, except for "range/range/range" queries on very large transaction time intervals.

It remains an open problem to find the theoretically I/O optimal solutions for the various bitemporal queries. An interesting problem is to find a partially persistent method that provides the same query efficiency for bitemporal queries on transaction-time intervals as for queries on transaction-time instants. Bitemporal joins are also of great interest; various possible optimizations using the presented methods must be examined. Another open area of research is to find bitemporal access methods to support *branching* transaction time evolution. As we mentioned, this implies making the chosen ephemeral data structure fully persistent. A good starting point is the work in [LM91] where B-trees are made fully persistent, and [LST95] where branching transaction-time issues for transaction-time databases are addressed.

Acknowledgments

We would like to thank B. Seeger for kindly providing us with the R* and MVB-tree code, and R. Snodgrass and C. Jensen for their suggestions for a 3-part bitemporal query terminology. We also thank the anonymous referees whose very insightful comments improved this paper.

References:

[A92] G. Ariav, "Information Systems for Managerial Planning and Control: a Conceptual Examination of their Temporal Structure", *Journal of MIS*, Vol.9, No.2, pp.77-98, 1992.

[AV96] L. Arge, J.S. Vitter, "Optimal Dynamic Interval Management in External Memory", in *Proc. 37th 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.

[BGO+93] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, "On Optimal Multiversion Access Structures", in *Proceedings of Symposium on Large Spatial Databases,* 1993. Published in *Lecture Notes in Computer Science*, Vol 692, pp. 123-141, Springer-Verlag (1993).

[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.

[C86] R. Cole, "Searching and Storing Similar Lists", *J. of Algorithms*, Vol 7, pp. 202-220, 1986.

[CLR90] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[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.

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

[F72] J.F.Fries, "Time-oriented Patient Records and a Computer Databank", *Journal of Am.Med. Assoc.*, Vol. 222, No. 12, 1972.

[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.

[HRU96] V. Harinarayan, A. Rajaraman, J. D. Ullman, "Implementing Data Cubes Efficiently", *Proc. ACM SIG-*

MOD, Conf. on the Management of Data*, pp 205-216, 1996.

[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.

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

[KRVV93] P.C. Kanellakis, S. Ramaswamy, D.E. Vengroff, J.S. Vitter, "Indexing for Data Models with Constraints and Classes", *Proc. 12th ACM Symp. on Principles of Database Systems (PODS)*, pp 233-243, 1993.

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

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

[KTF95] 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. An extended version appears as AT&T TR:#112530-950926-11-TM.

[LM91] S. Lanka, E. Mays, "Fully Persistent B^+ Trees", *Proc. ACM SIGMOD*, pp 426-435, 1991.

[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*, pp 315-324, 1989.

[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, No. 4, pp. 703-726, Oct. 1995.

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

[Mc85] 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 5th JCIT Conf.*, Jerusalem, Israel, Oct.22-25, pp 491-498, 1990.

[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.

[S87] M. Stonebraker, "The Design of the Postgres Storage System", *Proc. VLDB Conf.*, pp 289-300, 1987.

[S89] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1989.

[S+94] R. Snodgrass et. al., "TSQL92 Language Specification", *ACM SIGMOD Record*, 23 (1), pp. 65-86, 1994.

[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. VLDB Conf.*, pp 205-215, Aug. 1989.

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

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

[ST94] B. Salzberg, V.J. Tsotras, "A Comparison of Access Methods for Time-Evolving Data", to appear at *ACM Computing Surveys*; also available as a technical report from Polytechnic University (CATT-TR-94-81), or, Northeastern University (NU-CCS-94-21), 1994.

[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 Snapshot Queries", *Information Systems, An International Journal*, Vol. 20, No. 3, pp 237-260, 1995.

[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*.

[W82] R. Weber, “Audit Trail System Support in Advance Computer-based Accounting Systems”, *The Accounting Review*, VOl. LVII, No. 2, pp 311-325, 1982.

APPENDIX A: Making an ordered list partially persistent and fully paginated

Consider the initial state $X(0)$ of an ordered, time evolving list $X(t)$. We need a representation of $X(0)$ that satisfies the following performance requirements:

(i) *good pagination*. If $X(0)$ has i elements the representation should occupy $O(i/b)$ pages.

(ii) *fast updating*. Fast additions/deletions should be supported anywhere in $X(0)$.

(iii) *fast querying*. The k first elements of $X(0)$ should be accessed in $O(k/b)$ page I/O’s.

Satisfying the first requirement is easy: we simply store $X(0)$ elements sequentially in pages. Each page keeps a *next* page pointer. This creates the list of pages (or *paginated* list) $PX(0)$. Each element inside a $PX(0)$ page has two timestamps. The first timestamp corresponds to the time this element is added in X (elements of $X(0)$ have t=0); the second timestamp is initially empty and is used if this element is deleted from list X. For the second requirement we facilitate an ephemeral B+ tree on top of $PX(0)$, such that: (a) copies of all $X(0)$ elements are stored in the leaf pages of the tree and, (b) each such element copy points to the page of $PX(0)$ that stores the corresponding element of $X(0)$. For the third requirement (searching) we simply keep a pointer to the first page of list $PX(0)$ and then follow next page pointers.

As time evolves, so does $X(t)$ and its representation $PX(t)$. We need to guarantee the same three requirements for every t. For good pagination it suffices to show that at each t every page in $PX(t)$ has enough number of “alive” elements (i.e. enough elements from $X(t)$). This is guaranteed if updates on $PX(t)$ are performed using the merging/splitting policies of [BGO+93] or [VV95]. As before, a PX page is called “alive” as long as it contains at least q alive elements ($q<b$), otherwise it is a “dead” page. Additions of elements in $X(t)$ are represented by a physical addition of the new element in $PX(t)$. Deletions of elements from $X(t)$ are represented as logical deletions in $PX(t)$. The deleted element is found in $PX(t)$ and its second timestamp is updated with the element’s deletion time. If a deletion at some t causes the number of alive elements in a page to go below the q threshold, the page is *time-split*: a new page is created that carries (copies) the remaining alive elements of the previous page. The new page is considered as inserted at t and is added in the current $PX(t)$. The previous page is considered “dead” and is taken off the $PX(t)$ list. Hence at each time t, $PX(t)$ is the list of the currently alive pages.

Since updating is always performed on the elements of the most recent $X(t)$, we need access to these elements in $PX(t)$. This is performed through the ephemeral B+ tree. When an element addition/deletion occurs on $X(t)$, the B+ tree will locate the page of $PX(t)$ where this update is to be applied. After $PX(t)$ is updated, the B+ tree is updated.

The use of $PX(t)$ enables a crucial problem reduction: instead of dealing with element insertions/deletions in $X(t)$, we deal with page insertions/deletions in $PX(t)$. It remains to show how the fast query time is provided, i.e., how to access the pages of a given $PX(t)$ in element order. We present a solution where each page insertion/deletion takes $O(1)$ *amortized* time, the space used is $O(k/b)$ (k is the total number of page insertions/deletions in PX’s evolution and is bounded by $O(n/b)$) and the first a pages of $PX(t)$ are accessed in $O(\log_b k + a)$ page I/Os. At each t the size of the ephemeral B^+ tree is $O(m_X/b)$ where m_X is the number of alive elements in the current $X(t)$. Updating the tree takes $O(\log_b m_X)$ I/O’s and thus the overall update processing (PX list and B+ tree) is logarithmic (in the amortized sense).

The description of the solution follows. First, there is an array *HT* that indicates the first (top) page of *PX(t)* as time proceeds. An entry in *HT* has the form: $<t, pid>$ and specifies that at time *t*, the page with page id (address) *pid* became the top page of *PX*. Entries are incrementally added in *HT* when the top of *PX* changes. If at some *t*, *PX(t)* ceases having alive pages (i.e., there is no element in *X(t)*) an entry $<t, ->$ is added in *HT*. In addition, for a given (alive) page in *PX(t)* we need to keep which is the next (alive) page in *PX(t)*. We thus associate with each page ever added in *PX* a table called Next Table *NT*. The structure is similar with *HT*: an entry in the Next Table of some page *A* is of the form $<t, pid>$ where *pid* is the address of the page that at *t* became the next page after *A* in *PX(t)*. If a page *B* is deleted from *PX* at some t', the B^+ tree provides the previous and next pages of *B* at t', say *A, C*, and an entry $<t', C>$ is added on *NT(A)*. The *NT* tables are not explicitly implemented but are embedded in their corresponding list pages.

Using the *HT* and the *NT*'s we have an obvious way to search through the history of *PX*. A search is first performed on *HT* for the entry which contains the largest time t_S that is less or equal to *t*. Then the page whose *pid* is recorded next to t_S is accessed. A new search is performed on the *NT* array associated with that page and so on, until all pages of *PX(t)* are accessed. Each table search takes logarithmic time on the number of table entries (as table entries are time ordered). While simple, this approach is not efficient. If *k* is the total number of page insertions/deletions in *PX*'s evolution, a query that asks for the first *a* pages from *PX(t)* is answered in $O(a \log_b k)$ I/O's.

To avoid searching in each table encountered, we use a variation of the *backward updating* technique [TGH95]. This technique was used in [C86] for solving computational geometry problems in main memory. The idea is to *synchronize* the next tables among pages in each *PX(t)*. Here we adapt this idea in a paginated environment. For example consider the *NT*'s of two sibling pages *A* and *B*, where *A* proceeds *B* in *PX(t)*. After a number of C_0 entries (where C_0 is a constant greater or equal to 2) are added on *NT(B)*, a "ghost" entry is added in *NT(A)* (the sibling on the left) with the time of the latest of the changes and a pointer to the currently last entry of *NT(B)* (instead of the address of *B* itself). Such backward updating may continue until the *HT* is reached (for more details we refer to the TR version of [KTF95]).

The total number of the extra ghost entries is still linear to the total number of regular array entries. The number of regular array entries is clearly *O(n/b)* since there is an entry when a page is added or deleted. There are at most $1/(C_0 - 1)$ extra ghost entries per regular entry. The technique guarantees that during query processing, the entry with the largest time that is less or equal to *t* inside a table provides a pointer to an entry of the right sibling's table that is at most C_0 away from *t*. Since C_0 is a constant, this means that no new search is needed for locating *t* in the sibling's Next Table. In a paginated environment, a natural choice of C_0 is a page or a good portion of it. Larger C_0 means less extra space but slower search.

The auxiliary structure and the backward updating technique are embedded inside the *PX* pages. A special area of size $C_0 = pb$ table entries is reserved inside each list page, where $2/b < p < 1$. When the reserved space of a *PX* page gets full (because the next sibling list page changes too often) this list page is logically deleted from *PX(t)* and a new copy of it (with all the alive elements of the deleted page and an empty reserved space) is created. Such a change is termed an "artificial" deletion since it is due to backward updating. An artificial deletion is still triggered after *O(b)* element additions/deletions.

At worst, if the reserved areas of all pages of *PX(t)* are full, an element change in the last *PX(t)* page may cause the artificial deletion of all pages currently in *PX(t)*. This happens because the change on the last page overflows the reserved area of the last page which in turn (due to backward updating) overflows the previous page in *PX(t)* and so on. However, this will not happen often (if ever). It can be proved that the update processing per entry is $O(1)$ in the *amortized* sense.

APPENDIX B: Updating in the Bitemporal R-tree

Algorithm *Insert(r,t)*
(Insert a new record *r* at transaction time *t*).
1. Find leaf page *A* for inserting *r*;
   (this search is similar as with an ephemeral R-tree but it follows pages that are alive at *t*).
2. *InsertRecord(r,A,t)*.

Algorithm *InsertRecord(r,A,t)*
(Insert the new record *r* at transaction time *t* in page *A*).
1. Insert the new record into page *A*.
2. If page-overflow of *A* then
   copy the records of page *A* that are alive at *t* into a new page *B*;
   if strong-version-overflow of *B* (i.e., number of alive records greater than *b-e*) then
   split the page *B* into two pages *B* and *C* using one of the splitting algorithms (see section 6.2);
   elseif strong-version-underflow of *B* (i.e., number of alive records is less than *q+e*) then
   Merge(B);
3. Update the parent *P* of *A* (which is also the parent of any new *B* or *C*).
   If no new page is created by the insertion but the bounding rectangle of *A* is changed, adjust the index record *i* of *P* that points to *A*, to the new rectangle (see section 6.1). If a new page is created by the insertion, update the deletion time of index record *i* from *now* to *t* and insert a new index record *j* in *F* for *B* (and possibly *C*). Note that this could lead to further structural changes in the tree but all these changes are at levels above the level of *A* (or *B*, *C*).

Algorithm *Merge(B)*
(Merge page *B* with a sibling. This result to one or two pages depending on the number of records in *B* and its sibling).
1. Find a sibling *D* of *B* to be merged. Use one of the sibling finding methods (see section 6.2);
2. Copy the entries in *D* which are alive at time *t* into *B*.
3. If strong-version-overflow of *B* then
   split page *B* into pages *B* and *C* using one of the splitting algorithms (see section 6.2).

Algorithm *Delete(r,t)*
(At transaction time *t* delete record *r*).
1. Find leaf page *A* that contains record *r* that is alive at *t*;
   (this search is similar as with an ephemeral R-tree but it follows pages that are alive at *t*).
2. *DeleteRecord(r,A,t)*.

Algorithm *DeleteRecord(r,A,t)*
(Delete alive record *r* from page *A* at transaction time *t*).
1. Update the deletion-time of record *r* from *now* to *t*;
2. If weak-version-underflow of *A* (i.e., number of alive records is less than *q*) then
   copy the entries of page *A* that are alive at *t* into a new page *B*;
   Merge(B);
3. Update the parent page *P* of *A* (which is also the parent of *B*).
   If no new page was created no update is needed (see section 6.1). If a new page is created update the index record *i* of *P* that points to *A* by changing its deletion time from *now* to *t*; insert a new index record *j* in *P* pointing to the new page *B*. This may trigger further structural changes in the tree in a manner similar to *InsertRecord*.