The Snapshot Index:
An I/O-Optimal Access Method for Timeslice Queries

Vassilis J. Tsotras† *Nickolas Kangelaris*
Department of Computer Science
Polytechnic University
Brooklyn, NY 11201

ABSTRACT

We present an access method for timeslice queries that reconstructs a past state *s(t)* of a time-evolving collection of objects, in $O(\log_b n + |s(t)|/b)$ I/O's, where $|s(t)|$ denotes the size of the collection at time t, n is the total number of changes in the collection's evolution and b is the size of an I/O transfer. Changes include the addition, deletion or attribute modification of objects; they are assumed to occur in increasing time order and always affect the most current state of the collection (thus our index supports transaction-time.) The space used is $O(n/b)$ while the update processing is constant per change, i.e., independent of n. This is the first I/O-optimal access method for this problem using $O(n/b)$ space and $O(1)$ updating (in the expected amortized sense due to the use of hashing.) This performance is also achieved for interval intersection temporal queries. An advantage of our approach is that its performance can be tuned to match particular application needs (trading space for query time and vice versa). In addition, the Snapshot Index can naturally migrate data on a write-once optical medium while maintaining the same performance bounds.

Index Terms: Access Methods, Transaction-time Databases, Optimal I/O, Data Structures.

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

1. Introduction

There are many applications where the ability to use past information provides additional functionality [33]. For example, by retaining past states, a transaction-time database [11] can more accurately model reality. In contrast, conventional databases have always one logical state; when a transaction commits the database evolves to a next consistent state, while the previous state is discarded. Supporting evolution on the temporal dimension can create an increasing amount of data that has to be stored; it is therefore essential to invent efficient access methods for temporal data. Here we present one such method, the *Snapshot Index.*

Consider a real-world system whose state consists of a collection of objects. For the purposes of this presentation, time is assumed to be discrete and described by a succession of nonnegative integers. As time proceeds, the system's state changes by adding new objects or deleting objects from it. We assume that changes are always applied to the current system state. Associated with every real-world object is a unique, time-invariant object identity (*oid*) and a time interval during which the object exists in the real-world system. The birth and deletion times of an object correspond to a lifespan for this object [4]. Rebirths of objects are allowed, hence an object may have a set of non overlapping lifespans. Together with object additions and deletions the evolution may also incorporate value changes. An object can thus have attribute(s) with a time-variant value. As an example, consider the temporal evolution of a company. The state of the company at a given time instant is the collection of employees working for it at that instant. The state changes over time as employees join or leave this company. A time variant attribute could be an employee's salary.

Suppose that the above temporal evolution is captured in a database. Objects are represented in the database by records and can be identified by some *oid*. For every real world change that occurred at time t, a transaction updates the database about this change timestamped with t. This timestamp is another attribute of the data and can therefore be queried. We assume that transactions are applied to the database following the time order of the real-world changes. Consistent with the widely used temporal terminology [11], the Snapshot Index supports "transaction time" and computes timeslices in a transaction-time database.

We define as the *database state* $s(t)$ the collection of objects that exist at t in the real-world system modeled by the database. The notion of the database state is thus a logical one and should not be confused with the database itself which stores records for both current and past data. For the company example, the database state $s(t)$ is the set of all employee names (and their salaries) working for the company at *t*; however, the database also stores the records of all the employees that were working for the company before and after *t*. A fundamental temporal query is to reconstruct (a timeslice, or, a snapshot of) the database state *s(t)* for a given *t*; we call this query the *timeslice* query. The Snapshot Index has been designed to optimally address the timeslice query. The meaning of "Snapshot" in the "Snapshot Index" is that the index reconstructs snapshots (timeslices) for a transaction-time database. It should not be confused with the term "snapshot database" [11] (i.e., a conventional database that only keeps its current state). To avoid confusion and to agree with the temporal database glossary of [11], in the rest of the paper we will use the term "timeslice" when referring to a database state at some time instant.

Any access method used to address the timeslice query is characterized by the following costs: *space* (the space consumed by the method's data structures in order to keep past and current data), *update processing* (the time needed to process one change occurring in the evolution and update the method's data structures) and *query time* (the time needed to solve the query). All three costs are functions of the number of changes *n* during the real-world system's evolution, where a change is the addition, deletion or value change of a real-world object. The number of changes is a basic parameter of temporal queries in general, as it corresponds to the minimal information needed by any index to perform errorless reconstruction of past data. Another parameter is the size of the answer to the timeslice query, denoted by $|s(t)|$, and corresponds to the number of objects in the requested state *s(t)*.

In a database environment the computational model is based on how many pages are transferred between main and secondary memory. In transaction-time databases in particular, this is very crucial as the bulk of data, due to their volume, will definitely be stored in secondary storage media. It is therefore natural to use an *I/O complexity* cost [13] that measures the number of pages that are either used for storing the data, or transferred during an update or a query. We will assume that every secondary memory access transmits one block of *b* records and this counts as one I/O. A block is a logical unit of transfer and in general it can be a page, a sector, etc. In the rest we will use the terms block and page interchangeably. As an example, a B-tree is an access method that requires $O(\log_b n)$ page accesses for searching or updating. In contrast a binary tree can take $O(\log_2 n)$ page accesses, as each node of the tree can be stored in a different page. The reader should be careful with the notation: $\log_b n$ is an $O(\log_2 n)$ function only if *b* is considered a constant. In the rest of this paper *b* is considered a variable, since the size of a page affects the I/O process. All I/O bounds will be expressed in terms of *n*, $|s(t)|$ and *b* (i.e., all constants are independent of these parameters). For I/O complexity, $\log_b n$ represents a $\log_2 b$ speedup over $\log_2 n$ which is a great improvement, as transferring a page takes about 2-25 msec (in contrast, comparing two keys in main memory takes about 5-10 nsec).

In order to provide an understanding of the cost functions, consider two obvious ways to solve the timeslice query. One could store the whole current state *s(t)* for every time instant when some change took place, or, enter the updates (changes to the state) time-stamped on a "log". In the first case, i.e., the "copy" solution, the I/O for answering a query is minimal: $O(\log_b n + |s(t)|/b)$ assuming that a multilevel paginated index on the time dimension is utilized. However, the update time (time to store every state) and the space can be problematic, i.e. $O(n/b)$ per change and $O(n^2/b)$ blocks respectively, since in the worst case the state could always be increasing by insertions of new keys. For the "log" solution, the space requirement is minimal, just $O(n/b)$ blocks. Similarly, the update cost is minimal ($O(1)$), as only the last block of the log is accessed in order to insert the new change. But the query time of this method suffers, since in the worst case one has to search back the whole log, resulting in $O(n/b)$ I/O.

We refer to a method that solves the timeslice query as *I/O-optimal* if it preserves the following cost characteristics: *O(n/b)* space, $O(1)$ update processing per change, and $O(\log_b n + |s(t)|/b)$ I/O for query time. As it will be shown in section 3, a certain "optimality" property holds for such a method: the timeslice query is reduced to the predecessor problem for which a lower bound can be shown if space is restricted to *O(n/B)*. An access method with query time $O(\log_b n + |s(t)|)$ is not I/O-optimal because in the worst case it may require many more block I/O's (the data is not well clustered as each record of the answer can be stored in a different block). Similarly, a method with $O(\log_2 n + |s(t)|/b)$ query time is not I/O-optimal since its index is not fully paginated. Note that the optimality property involves the query time and the space requirement, i.e., it holds independently of the update processing. However, in our definition of an I/O-optimal method we have the additional requirement that the update processing is $O(1)$. This requirement emanates from the fact that since changes arrive in order, ideally they could have just kept in a "log" resulting in $O(1)$ update processing per change.

One of the contributions of this paper is that the timeslice query can be solved optimally with a method that uses $O(1)$ update processing (in the expected amortized sense, due to the use of hashing). This is a difference with the key-range timeslice query (for example, "find the employees who were working for the company on January 1st, 1990 and whose keys are between k_1 and k_2") where a logarithmic update processing is always required. This difference is due to the fact that for the key-range timeslice query updates arrive out of key order and must thus be ordered.

In Section 2 we summarize previous work on the timeslice and related queries. Section 3 describes the Snapshot Index and its properties. Section 4 contains performance characteristics of our Index based on simulations while section 5 describes how to migrate past data from a magnetic media to a write-once optical disk. Finally, section 6 contains conclusions and open problems.

2. Previous approaches

One of the earliest approaches for temporal access methods proposes the use of "reverse chaining", i.e., all past versions of a given key are linked in reverse order [25]. There is also a separate key index. This method can easily find the history of a given key, however, to answer timeslice queries it must search all the version chains until the time of interest.

A variety of temporal access methods have been proposed in recent years [1, 2, 7, 8, 9, 12, 15, 16, 18, 19, 20, 21, 22, 23, 26, 30, 31, 34, 35, 36, 37]. Reviews of various methods appear in [14, 29]. Most of these approaches assume that changes occur in increasing time order; therefore, they are appropriate for "transaction time" evolution (transaction-time databases). In contrast, the notion of "valid time" evolution has also been proposed in the literature (valid-time databases) [33]. Under valid time, changes can occur in any time order; thus one could also support *corrections* on the data; consider for example changing the lifespan of a given object to a new interval. It should be noted that typically any proposed method (including the Snapshot Index) can be modified to support such corrections, but the update processing will be different for these corrections. In practical situations, the performance will not deteriorate if the number of corrections is limited. It is an open and challenging problem to invent an access method that I/O*-optimally* supports both valid and transaction time dimensions (bi-temporal databases) on the *same* index.

Regarding transaction time support, there have been two main approaches to the problem. In the first approach [2, 15, 16, 19, 21, 22, 23, 26, 34] the time and key dimensions are combined in some form in the index. The advantage of such indexes is that key-range timeslice queries can easily be addressed. However, the update processing per change is bound to be at least logarithmic, as at best any update follows a path on a balanced tree. In the second approach [7, 8, 9, 12, 20, 30, 31, 35, 36, 37] the time dimension is independent from the key dimension; the major effort here is on efficiently addressing (whole) timeslice queries. Ideally, the update processing of this approach is minimal as updates can take constant I/O.

The Time-Split B-Tree [21, 22, 23] is one of the first efficient access methods proposed for key-range timeslice queries. It incorporates the key-space and the time-space into a single two-dimensional B-tree-like structure. Data are clustered into pages according to their time and key coordinates. It uses $O(n/b)$ space, $O(\log_b n)$ update per change and $O(\log_b n + |s(t)|)$ query time. These bounds hold for data representing key insertions and value updates for already existing keys (and assuming a time split is always performed before a key split). Key deletions are supported by a special value assigned to the deleted key however the above query bound is then not guaranteed.

The fully-persistent B+-tree [19] maintains full persistence of a B+-tree. A structure is called "ephemeral" if past states are erased when updates are made; if past states are kept the structure is termed "persistent". A "fully persistent" data structure allows updates to all past states. A "partially persistent" data structure allows updates only to the most recent state ([5] shows how to transform a main memory resident ephemeral structure into a fully or partially persistent structure). The fully-persistent B+-tree can still be used for partial persistence (which is needed for temporal data). In the *fat-node* method, the history of a B+-tree which indexes an evolving collection of objects is maintained. There are extra nodes in the path from the root to the data (the version blocks) and a new leaf page is created for every update. Thus space is *O(n)*, not *O(n/b).* It was assumed that each version block can keep all versions of the corresponding block, however this may not be the case if many changes (versions) occur. Thus the query time can be more than logarithmic (a typical path on this index is of logarithmic size and in each version block encountered on this path by the search process, a logarithmic search may be involved). For better performance, the *fat-field* method was proposed, where updates can fit in the space of non-full pages. Since version blocks are still used, the fat-field approach has the same asymptotic query time performance as the fat-node approach. Pages though are more "clustered". When a leaf page becomes scarce of "alive" records a page consolidation algorithm is used that will create a new page out of the alive records of this page and one sibling page. In a (pathological) worst case scenario, thrashing may occur (were insertions and deletions can cause the same page to be continually split and consolidated) resulting in creating a new page per update.

Recently, a fully paginated solution to key-range timeslice queries was proposed in [2]. The Multiversion B-Tree uses *O(n/b)* space, logarithmic update per change and has $O(\log_b n + |s(t)|/b)$ query time for a key-range query. The update is $O(\log_b m)$ where m is the size of the current state of the structure when the update is performed (*m* will usually be much smaller than *n*). This method presents the optimal solution to the general key-range timeslice query (including key deletions), however the storage utilization is larger than the Time-Split B-Tree (even though the space used by the Multiversion B-Tree is $O(n/b)$, the constant in the bound is about 10).

The Segment R-tree [15, 16] combines aspects of the Segment Tree data structure [3] with the R-tree [10]. An advantage of this method is the ability to simultaneously represent multi-dimensional interval data. This characteristic could possibly allow the SR-Tree to support both valid and transaction time on the same index. However, a logarithmic factor is added for the space requirements (due to the Segment tree property), while in worst case scenarios the update processing may be more than logarithmic (due to the use of the R-tree). In addition, it is assumed that both endpoints of a record's interval are known at insertion time.

The Time-Index [7, 8, 9] maintains a set of linearly ordered indexing points on the time dimension. Repeated timeslices are kept (after a constant number of change instants) together with the changes between them. Assuming that at each time instant at most a constant number of changes can happen, and, a timeslice keeps the full records (and not pointers to them) the query time is $O(\log_b n + |s(t)|/b)$. However, the update processing and the space are $O(n/b)$ and $O(n^2/b)$ respectively (if objects remain alive for a long time, their copies will appear in many timeslices resulting in the quadratic space consumption). Thus this method is conceptually similar to the "copy" solution.

The Append-Only tree [30, 31] is a multi-level index on the "log" of the time instants when a change occurred. While the space is minimal (append at the end), the update and query time can in the worst case be $O(n/b)$ I/O as the whole log may have to be searched in order to find and update a deleted record or to answer a query.

[12] involves an interesting implementation of a database system based on transaction time. Changes that occur for a base relation *r* are stored incrementally and timestamped on the relation's log; this log is itself considered a special relation, called a *backlog*. In addition, timeslices of relations can be stored. A timeslice can be *fixed* (for example: $r(t_1)$) or *time-dependent* ($r(now\text{-}t_1)$). A time-dependent base relation can also be computed from a previous stored timeslice and the set of changes that occurred in between. These changes correspond to a *differential* file (instead of searching the whole backlog). The performance of this approach for the timeslice query depends

on how often timeslices are kept. If the number of changes between timeslices is not fixed, the method behaves asymptotically as the "log" approach, else like the "copy" approach.

The "Checkpoint Index" of [20] indexes timeslices taken periodically (at checkpoints) from the state of an evolving relation, with tuples in between checkpoints stored separately. This method's asymptotic behavior also depends on how often timeslices are stored and can behave like the "log" or the "copy" approaches.

[35, 37] present lower bounds for the timeslice query and algorithms that lead to linear space, constant update processing and $O(\log_b n + |s(t)|)$ query time. They combine the minimal space and updating of the "log" solution with the fast query time of the "copy" solution. However, this solution is not paginated. In this paper we improve on that result by presenting a fully paginated access method for the timeslice query.

It should be noted that none of the above approaches is *I/O-optimal* (according to the definition in section 1) for the timeslice query. The Snapshot Index presented in this paper is I/O -optimal since it optimally clusters data by the time dimension and uses constant update processing per change (in the expected amortized sense, as explained in the next section.) Hence the Snapshot Index can practically "follow" the evolution.

3. The Snapshot Index

We begin this section by describing how the real-world changes are stored in the database. Then we present a controlled *copying* procedure that allows for better clustering of related records in pages. Finally, we introduce the notion of page *usefulness,* which for each page provides the time interval during which this page contains enough records that satisfy timeslice queries. The real-world evolution is then transformed to a *meta-evolution* that instead of object lifespans deals with page lifespans. The initial problem is thus reduced to finding all "useful" pages at a given time. The appropriate data structures (a doubly-linked list and an array) are also presented.

The state of the real-world system evolves from the current time-instant to the next by issuing a number of changes on the most current state. The allowed changes on the real-world system's state are addition and deletion of objects and attribute value modifications. We are only interested in evolving systems which satisfy a certain discrete notion of continuity: there exists a constant K that upper bounds the number of changes than can occur in the system's state at each time instant.

A real-world object is represented in the database by a record (tuple). Each such record, has a time-invariant key, a time-variant (temporal) attribute and a semi-closed interval [*start_time*, *end_-time*) that represents a time interval when the <key, attribute> pair was valid. This is the widely used tuple-versioning temporal model [11, 24, 27, 32]. Consider first only additions and deletions of real-world objects. When a real-world object with identity *oid* and attribute value *val* is added (born) in the real-world system at time instant t_1, a record with key the same *oid* and attribute value *val* is created in the database with its interval being initialized to [t_1, *now*]. The special variable *now* represents the current real-world system time. At a later time t_2, if the real-world object *oid* is deleted, the database will locate the record <*oid*, *v*, [t_1, *now*]> and will update the interval to [t_1, t_2). The actual deletion of the real-world system object is represented in the database by an interval update. For the rest of this discussion "deletion" of a database record will actually mean the update of a database record about a corresponding real-world deletion.

For the database, an attribute value modification of a real-world object *oid* is similarly represented with the "deletion" of the record with key *oid* and the old value, followed by the instantaneous rebirth (at the same time instant) of a new record with key *oid* and the new value. Consequently, the lifespan interval of a real-world object may be stored as a set of database records with the same *oid* and contiguous intervals that overlap only at their start and end points. Therefore for simplicity of presentation, the only changes that we will consider are additions and deletions of real-world objects. Hence, we can call the interval of a database record the *lifespan* of this record. For all time instants in its lifespan, this record is called *alive* in the database.

The timeslice query is defined as following: given time instant *t*, reconstruct state *s(t)* as it was at *t*. To reconstruct *s(t)*, the database must locate all records whose lifespans "include" time *t*; a semi-closed time interval [*x*, *y*) includes all time instants *t* that satisfy $x \le t < y$. Let $|s(t)|$ be the total number of these records (i.e., the cardinality of the answer); we will present an index that solves the timeslice query with $O(\log_b n + |s(t)|/b)$ I/O's. The Snapshot Index keeps "logical" timeslices of the database states, i.e., the timeslices are not explicitly stored but reconstructed; therefore, the space used will still be $O(n/b)$.

For every real-world change, the database input is of the form: *t*, *oid*, *op*, and represents the fact that at time *t*, operation *op* (addition or deletion) was applied on real-world object *oid*. Based on this input the database will create a new record or update an existing one. As new records are created, they are stored sequentially in a block. At any given time, there is only one block that stores arriving records and it is called the *acceptor* block. When an acceptor block is full of records, a new acceptor block is assigned to store new arriving records. Besides storing records about objects, each block stores one, special, constant length record *SR*, that keeps information about the logical position of that block in the Snapshot Index (Figure 1). Since the ratio of the block size to the record size is usually very large, for simplicity of the presentation, instead of saying that each block has space for b-1 object records and one *SR* record, in the rest we use b to denote the total number of object records per block (i.e., we do not “count” the single *SR* record).

Input:

(t,	*oid*,	*op*)
1,	*u*,	+
2,	*b*,	+
4,	*f*,	+
8,	*c*,	+
10,	*b*,	-
15,	*d*,	+
16,	*g*,	+
17,	*d*,	-

Block *A*

SR record
<*u*, [1, *now*]>
<*b*, [2, *now*]>
<*f*, [4, *now*]>

(a)

Block *A*	Block *B*
SR record	*SR* record
<*u*, [1, *now*]>	<*d*, [15, 17)>
<*b*, [2, 10)>	<*g*, [16, *now*]>
<*f*, [4, *now*]>	
<*c*, [8, *now*]>	

(b)

Fig. 1: An example of the basic block structure. Each block contains a header (*SR* record) and has space for 4 data records. (a) At time $t = 7$, the only block used is block *A* which is also the current acceptor block. (b) At time $t = 19$, there are two blocks used, *A* and *B* with *B* being the new acceptor block.

For deletions, the database needs to locate the (object) record to be updated. By keeping the time dimension separate from the key dimension, our approach leads to constant update processing, i.e., independent of the (always increasing) evolution size n. A dynamic perfect hashing scheme is utilized for this purpose [6]. It has $O(1)$ expected amortized insertion and deletion cost, $O(1)$ worst case lookup and linear space. Thus when an update involves inserting or deleting from the hashing function, it will count as $O(1)$ in the expected amortized sense. The amortization means that some insertions/deletions may require more than $O(1)$ update processing to update the hashing function (since it is dynamic) but for any sequence of n updates no more than a total of $O(n)$ time will be required. The expectation is not based on any distribution assumption for the keys used rather it is based on randomized choices of the hashing algorithm (a much stronger condition). For simplicity, in the rest of this paper, constant updating means constant in the <u>expected amortized</u>

sense since the hashing is involved.

A real-world addition of an object causes the insertion of the object's *oid* in the hashing scheme together with a pointer to the block that will store the database record representing this new object. For a deletion, the database record that currently represents the deleted object *oid* is first accessed through the hashing scheme and updated; its *oid* is then removed from the hashing scheme. Hence, hashing provides a representation of the current state of the real-world system (previous states are not kept).

To guarantee worst case update performance, one could provide access to objects by their *oid* using a balanced dynamic index (B^+-tree) on the *oid*s of the alive objects (again next to each *oid* would be a pointer to the page that the corresponding record is stored). This solution leads to $O(\log_b |s(t)|)$ I/O per update on state $s(t)$. While it is still independent of n (rather it depends on the size of the current database state where the update is performed) we feel that in most practical cases a good hashing function should be preferred as it would provide a much faster update (especially if the number of "alive" objects is large.) Finding a good hashing function should not be difficult in a practical implementation considering that the key space does not change dramatically.

Our approach is similar to the "log" approach, however by using the hashing scheme, we can always go in previous blocks and update the lifespan intervals of records that correspond to deleted objects. Note that as real-world changes occur and blocks are filled, the "alive" records at any given time t are dispersed in many blocks; in the worst case we may have each "alive" record in a different block. We now introduce the notion of block "usefulness" as a measure of data clustering.

Definition: A block is defined to be *useful* for the following time instants:

(a) (**I-useful** mode) for all time instants t, such that t is greater or equal to the time of the insertion of the first record in the block and less or equal to the time instant that this block became full (i.e., the time that the last record was inserted in this block).

(b) (**II-useful** mode) for all time instants t, such that t is greater than the time of the insertion of the last record in the block and for which the block contains at least $a \cdot b$ "alive" records from the state $s(t)$. The *usefulness* parameter a ($0 < a \leq 1$) is a constant that can be chosen accordingly, to tune the behavior of our access method.

When a block is not in any of the above modes it is called *non-useful*. The following properties for block usefulness hold:

(1) At any time instant there is exactly one block that is in I-useful mode (by construction since at any time there is only one acceptor block).

(2) The time instants during which a block can be in II-useful mode are contiguous (if a block is II-useful for an instant t_1 and not II-useful for an instant $t_2 > t_1$, it is not II-useful for all $t \geq t_2$ as no new records are added in that block).

(3) A block has exactly one I-useful interval (since it starts as an acceptor block) and at most one II-useful interval. If a block does not have at least $a \cdot b$ "alive" records when it is filled, it will never attain them in the future and therefore this block will not have a II-useful mode. If a block has both I- and II-useful modes, their corresponding intervals are disjoint and consecutive, with the I- interval followed by the II- interval.

The I-useful mode represents the time interval that a given block was an acceptor; during that period, a block may not have $a \cdot b$ "alive" records, but still contains useful information. The II-useful mode is attained by a block only if it continues to have at least $a \cdot b$ "alive" records for a period after being the acceptor block. Figure 2 shows examples of usefulness intervals for different blocks. The I-useful period of blocks B and C are intervals [15, 25] and [30, 42] respectively. At time $t = 51$, the real-world deletion of record g ended the usefulness period of block B. The II-useful interval of block B is (25, 51).

The purpose of introducing the usefulness parameter a is to identify those blocks that were in II-useful mode for the time of the query, i.e., those blocks that have at least $a \cdot b$ records that satisfy the query predicate. As it will be shown later our method will only access such blocks plus at most one more block that was in I-useful mode at the time of the query and may thus contain less than $a \cdot b$ records from the answer.

Just adding and updating the records in a log of blocks equipped with a hashing scheme is not enough since the records are allocated to blocks according to their real-world system birth time. In addition, declaring a block useful is not enough if the records are kept only in the block where they were initially stored. We therefore introduce a *controlled* copying procedure which basically splits the lifespans of long-lived objects into continuous subintervals that are assigned to different blocks. We will prove that the space used by the copying procedure is linear to the number of real-world changes so that the total space used by the Snapshot Index remains $O(n/b)$.

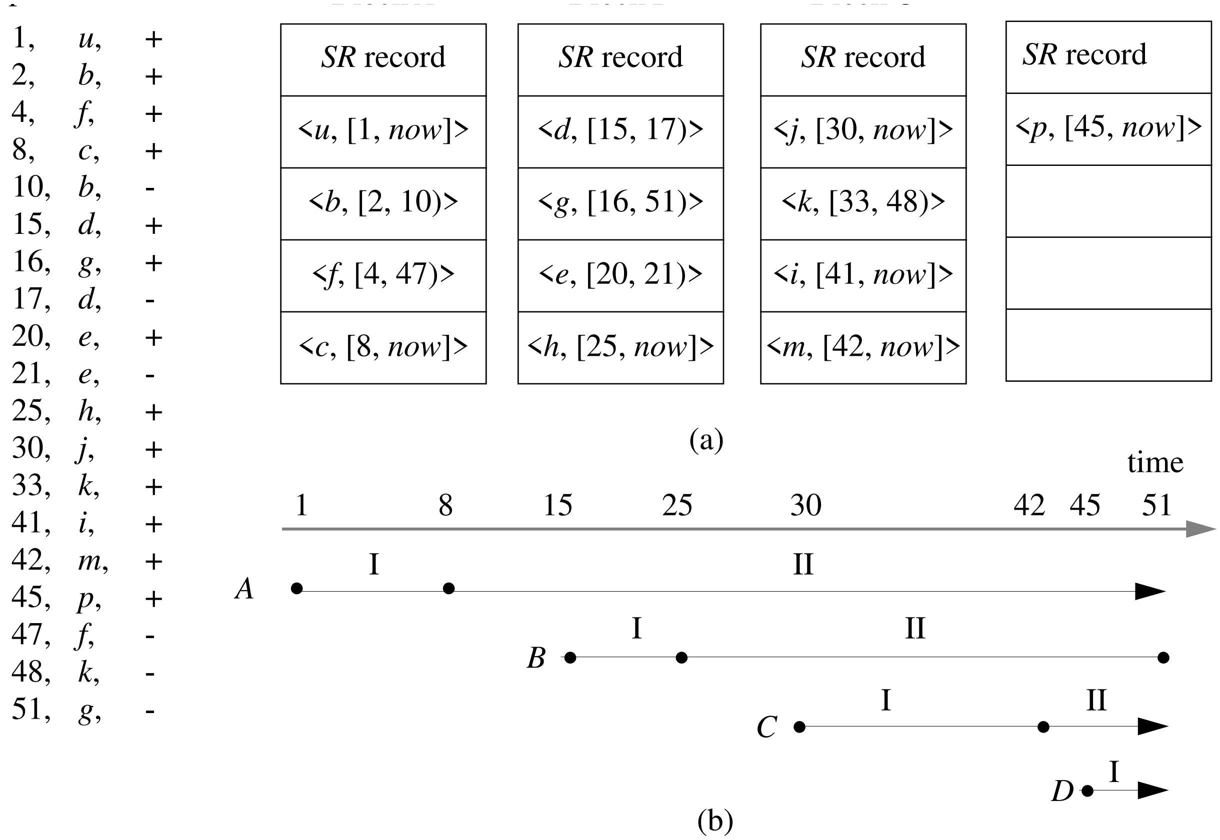

Fig.2: Examples of usefulness periods for the input shown. Blocks have become acceptors in the order *A*, *B*, *C*, etc. The value of *a* is 1/2. (a) The blocks at time *t* = 51. (b) The corresponding usefulness periods per block.

Copying Procedure: When a block *X* ends its whole useful period at time *t* (either at the end of the II-useful period or, if this block does not have a II-useful period, at the end of the I-useful period), the number of its records that are “alive” at *t* (i.e. not yet “deleted”) are copied to the current *acceptor* block. For each “alive” record of *X* its *end_time* is first updated to *t*; then a new record is created with the same *oid* in the acceptor block and with lifespan interval initialized to [*t*, *now*]. The hashing scheme is also updated about the new physical location (block) of the copied record, but this still takes constant time per copied record. The copying process has thus internally created two subsequent lifespan subintervals for a real-world object which did not change at *t*.

Figure 3 shows the evolution with the copying procedure present; when record *g* is “deleted”, block *B* does not have enough “alive” records (with *a* = 1/2) so the lifespan of the “alive” record *(h*, [25, *now*]*)* is “split” into: *(h*, [25, 51*))* and *(h*, [51, *now*]*)* which is stored in block *D*. Similarly, at time t = 53, block A becomes non-useful and a copy of record u is added in D.

Input:

....
30, *j*, +
33, *k*, +
41, *i*, +
42, *m*, +
45, *p*, +
47, *f*, -
48, *k*, -
51, *g*, -
53, *c*, -

Block *A*	Block *B*	Block *C*	Block *D*
SR record	*SR* record	*SR* record	*SR* record
<*u*, [1,53)>	<*d*, [15, 17)>	<*j*, [30, *now*]>	<*p*, [45, *now*]>
<*b*, [2, 10)>	<*g*, [16, 51)>	<*k*, [33, 48)>	<*h*, [51, *now*]>
<*f*, [4, 47)>	<*e*, [20, 21)>	<*i*, [41, *now*]>	<*u*, [53, *now*]>
<*c*, [8, 53)>	<*h*, [25, 51)>	<*m*, [42, *now*]>	

Fig.3: An example of the copying procedure, with $a = 1/2$. At time $t = 51$, the real-world deletion of object g forces block B to end its usefulness period. Similarly, at time $t = 53$, block A becomes non-useful due to the deletion of c. The copying procedure has thus "split" the lifespans of records h from B and u from A and created new copies in acceptor block D.

L e m m a 1: *The total space used by the copying procedure is O(n/b) and the updating cost per change is constant.*

P r o o f: A block degrades from II-usefulness to non-usefulness when a deletion occurs in the real-world system that decreases the number of "alive" records from $a \cdot b$ to $(a \cdot b$-1). In case that a block degrades from I-useful directly to non-useful the number of "alive" records is even less than $a \cdot b$. Consider the first block in the system: it has b real-world records which, independently of how they will be "deleted", can at most create $a \cdot b$ copies; each record is therefore responsible for at most a copies. Thus the $a \cdot b$ copies can in turn create at most $a^2 \cdot b$ copies and so on. In general, if E is the total number of real-world births (E is obviously $O(n)$ as every birth is a change), the number of first time copies would be less than $a \cdot E$. These copies can in turn create second time copies which are at most $a^2 \cdot E$ and consequently, the total number of records ever created in the database is given by the summation: $(E + aE + a^2E + \ldots)$ which is upper bounded by $E/(1-a)$. Hence, the total number of blocks used is $O(E/b(1-a))$ which is $O(n/b)$.

If a change is an addition of a real-world object, there is no additional updating cost due to the copying procedure. For a real-world deletion, the worst case happens if the corresponding "deleted" record causes a block to change from II-useful into non-useful, while the acceptor block is almost full. Then $a \cdot b$-1 records are copied from the non-useful block to the acceptor block, causing the creation of a new acceptor block which can definitely accommodate the excessive copies. In addition, the hashing scheme has to be updated for the new position of each copied record but this takes constant time (in the exp. amort. sense) per copied record. QED

Using the copying procedure and the usefulness concept we have reduced the initial problem of reconstructing the state *s(t)* into the problem of finding the blocks that were in useful mode at *t*. If a query answer is large, it will be dispersed in a number of II-useful blocks (blocks that at least have $a \cdot b$ records which satisfy the query's predicate) plus one I-useful block (which may have less than $a \cdot b$ records from the answer). If the query is small it will be contained in a single I-useful block that has to be accessed. Observe however that in any case at most one block will be accessed with less than $a \cdot b$ records from the answer. The following lemma provides bounds for the total number of (I and II) useful blocks:

L e m m a 2: *For any time t, the number of useful blocks* $|s_B(t)|$ *and the answer size* $|s(t)|$ *satisfy:* $ab(|s_B(t)|-1) \le |s(t)| \le b|s_B(t)|$.

P r o o f: We first consider the left inequality. From property 1 the number of useful blocks consists of exactly one I-useful block and maybe a number of II-useful blocks. If $|s_B(t)| = 1$ then the left inequality holds. If there exist II-useful blocks, let $|s_{\mathrm{II}}(t)|$ denote their number. Observe that at any time instant there is exactly one "alive" record with identity *oid* which represents the real-world object *oid* that exists at *t*. This "alive" record is either the original record with identity *oid*, or one of its copies as produced from the copying procedure (since when a copy of a record is made internally, the previous record is "deleted"). By definition every II-useful block has <u>at least</u> $a \cdot b$ of the records that constitute the answer *s(t)*. The answer *s(t)* is also distributed in the single I-useful block; let x be the number of records that are "alive" at t from this block. Then $ab|s_{\mathrm{II}}(t)| + x \le |s(t)|$ since some of the $|s_{\mathrm{II}}(t)|$ blocks may have more than $a \cdot b$ "alive" records.

In order to show the right inequality it is sufficient to prove that the answer $|s(t)|$ is contained in the useful blocks $|s_B(t)|$. By contradiction, let a record *r* with "lifespan" interval that includes *t* be stored in a block *X* which is non-useful at *t*. Block *X* is thus useful for a time period that ends before or starts after *t*. In the first case, let *X* be useful until a time instant $t_1 < t$, which implies that all records of *X* can be "alive" at most until t_1 since from the copying procedure all "alive" records of *X* would have to be "deleted". Consequently, *r* cannot be "alive" at *t*. Similarly, for the second case, all records of X can be "alive" after t. QED

We proceed by describing the basic data structures of the Snapshot Index and how it can efficiently find all the $|s_B(t)|$ useful blocks. From property 3 we can visualize the whole usefulness period of a block (I and, if applicable, II) as a contiguous "block-lifespan" interval. Therefore, a block is specified by a unique block identity (*bid*) and a semi-closed interval [*s_time*, *e_time*); *s_time* is the birth time of the first record inserted in this block and *e_time* is either the time of the last insertion in this block (if it has only I-useful mode) or the time of the record "deletion" which made a II-useful block to have less than $a \cdot b$ "alive" records. By considering *s_time* and *e_time* the "birth" and "deletion" times of a block we create a *meta-evolution* where *meta-changes* are "births" or "deletions" of blocks. Finding all useful blocks at a given time is then equivalent to finding all "alive" blocks at that time. For this problem we will use ideas from [37].We need to show that translating the real-world evolution problem to a block meta-evolution still uses linear space and constant updating.

The meta-evolution changes are triggered by real-world changes. A real-world change may cause at most two meta-changes. This happens when the deletion of a real-world object causes the "deletion" of a record which may trigger the "deletion" of a block (a "deletion" meta-change) which in turn may trigger the creation of a new acceptor block (a "birth" meta-change). The addition of a real-world object may cause only one "birth" meta-change. Since at any time instant there will be at most K real-world object changes, the number of meta-changes per time instant is also bounded by a constant.

A meta-evolution change is represented by a triplet <*time*, *block-id*, *meta-change*> where *time* is the time instant of the meta-change. In order to realize the meta-change and the block on which it is applied from the database input, note that when a record id is added in the hashing scheme, a pointer to the block that contains this record is also stored. Hence, for a record "deletion" the hashing scheme will also provide the block which contains that record and on which a meta-deletion may occur.

There are three kinds of time instants in the *meta-evolution* specified by the kind of meta-change that occurred at that time.

A time instant with no meta-changes in the meta-evolution is a *no-change* instant. It can correspond to an instant where nothing happened in the real-world. In addition, many real-world changes do not cause meta-changes (like those corresponding to records that are just added on an acceptor block or those corresponding to record "deletions" that do not change the usefulness state of a block).

A time instant during which at least one "birth" meta-change occurred is called a *meta-birth* instant. Note that an instant t corresponding to a record "deletion" which causes the "deletion" of a block and the subsequent and instantaneous creation of a new acceptor because of overflow (due to copying), is a meta-birth instant. A time instant where only "deletion" meta-changes occurred is called a *meta-deletion* instant (i.e., when the "alive" records from "deleted" blocks do not cause the creation of a new acceptor block). Meta-change instants are in general a small subset of the real-world change instants.

We use two main data structures: a doubly-linked list L and an array AT. First, we describe the structure and use of list L. The array AT will be described later. At any time t, list L will contain the "alive" blocks of the *meta-evolution*. The list is implemented as a separate object that has a top element, represented by a special block S, and a pointer *lpr* to its last element. Initially, the list is empty and *lpr* points to S. When a block is "born" it is added at the end of the list and *lpr* points to this new block. When a block is "deleted", it is removed from the list and is added as the next child of the block preceding it in the list. List L is thus characterized by *append_at_the_end* and *delete_-anywhere* operations. Based on the blocks' "lifespan" behavior and through list L a forest structure is created where a node in this forest represents a block. At any time instant the forest is rooted from the blocks that are elements of list L at that time. We call this forest the *access forest* as it will facilitate the location of all "alive" blocks for a given query.

Figure 4 presents an example of the access forest. In this figure we have assumed that the evolution of Figure 3 has continued until time $t = 80$; the block-lifespan intervals are shown. Initially, block B was "deleted" and became the first child of block A ($t = 51$). Before B's "deletion" list L contained elements: S, A, B, C and D. When A was "deleted" ($t = 53$) it became the first child of special block S. At $t = 60$, some real-world addition created a new acceptor block E, that remained as an acceptor until time $t = 65$, when a new acceptor was created, block G (i.e., block E never attained a II-usefulness period). At that time, E was "deleted" and became a child of block D. Similarly, block G was "deleted" at time 70 when a new acceptor H was created. Finally, some real-world deletion at time $t = 80$, caused block D to become non-useful and thus it goes under block C; its alive records have been copied to acceptor block H.

The implementation of the access forest and the list, is accomplished through the constant

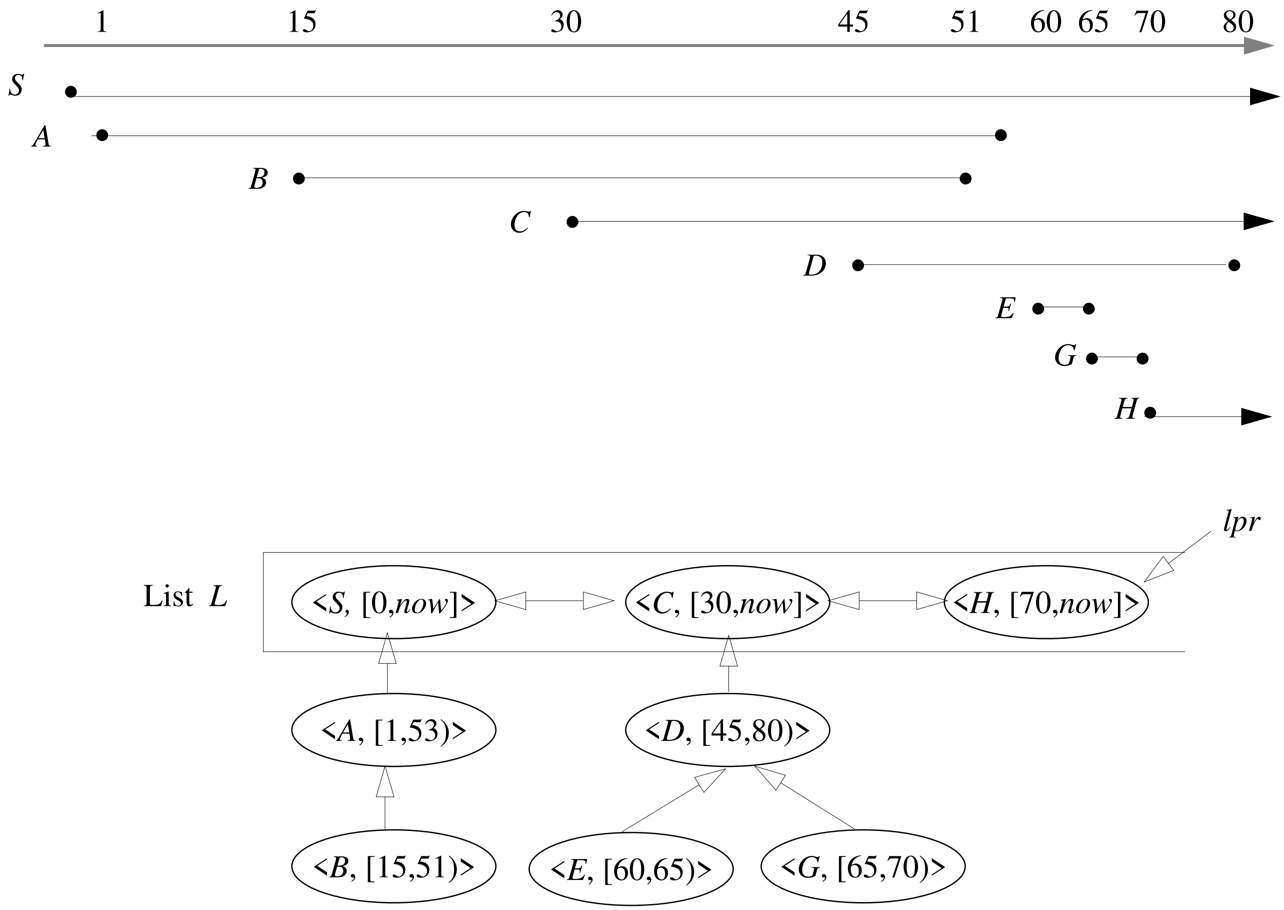

Fig.4: A schematic view of the access forest for a given meta-evolution, as it is at time $t = 80$. A block is represented by a tuple <block-id, block-lifespan>. Block *S* denotes the top of list *L*, while pointer *lpr* points always to the end of *L*.

length *SR* record of each block. More specifically, this record has the following fields: the block id field *bid*, the block-lifespan *s_time* and *e_time* fields, the *parent* field, the *prev* field, the *next* field and the P_{CS} and P_{CE} fields. As a block has a logical position in the access forest, it may have a parent and a number of children blocks. The *parent* field is used for the parent of the block. The P_{CS} (P_{CE}) field is a pointer to the first (last) child block of this block in the access forest. The *prev* and *next* fields are for the implementation of list *L*.

When a block with *bid X* is "born" at time t_1, its *SR* record is filled. The *s_time* field will store time t_1 while the *e_time* field will store a special character *now* to indicate that the block is still "alive". Block *X* is also added at the end of list *L*. The *X.prev* field will store a pointer to the previous last element of the list (using pointer *lpr*). The *X.next* field is initially empty since *X* has no next element in the list. The *X.prev.next* field of the list element preceding *X* is made pointing to block *X*. If *X* is the first block in the list, its *prev* pointer will point to special block *S*. Initially, when a block is created it does not have any parent or children thus the *parent*, P_{CS} and P_{CE} fields are empty.

When block *X* is "deleted" at $t_2 \geq t_1$, its *e_time* field is updated with t_2. From *X.prev* the currently previous block in list *L* is found, say block *F*. Block *X* is then deleted from list *L* (by making the *F.next* pointer point to the block pointed by *X.next* and the *X.next.prev* pointing to *F*). Logically, when a block is "deleted" it acquires a parent; i.e., it is deleted from the "alive" blocks that are kept in list *L* and becomes the next child of its parent block. The parent of block *X* is block *F*. Accordingly, the *parent* field of *X* will store *bid F* when *X* is "deleted", while the P_{CE} field of parent block *F* will point to the "deleted" block *X*. If *X* is the first child it will also be pointed by field P_{CS}. When traversing the access forest our method will encounter children blocks from right to left, hence a child block must have means to access its left sibling under the same parent block; this will also facilitate the migration of blocks to an optical disk. When block *X* is "deleted" it becomes the next sibling of the block pointed from *F*'s P_{CE} field. This left sibling of *X* can be stored inside its *prev* field which is not used after *X*'s "deletion". The *next* field remains unused after *X*'s "deletion". A block's children list is thus characterized by *append_at_the_end* operations.

Note that a block may get children as long as it is "alive"; after it is "deleted" it obtains a parent and no new children are attached under it. A block retains all of its subtree after "deletion". Special element *S* is never deleted from the list, therefore *S* behaves like a "sink" for blocks "deleted" from the top of the list. Since the list *L* and the access forest are implemented inside the blocks using the constant length *SR* records the space used for the implementation of the meta-evolution is $O(n/b)$. The update processing needed to create the access forest is constant per meta-change (as for every block "birth" or "deletion" a constant number of updates are performed by our method).

While list *L* enables the construction of the access forest, array *AT* provides access to the temporal dimension. An entry in array *AT* is of the form: <*time*, *bid*>. New entries are always added at the end of array *AT* in increasing *time* order; no entries are ever deleted from it. For each meta-birth instant *t*, entry <*t*, *bid*> is appended on array *AT*, where *bid* is the block id of the last block added at *t* in list *L*; even if meta-birth *t* contains many block "deletions" and block "births" the *bid* corresponds to the last created acceptor block due to this meta-change. A no-change or a meta-deletion instant is not recorded in array *AT*.

We need array *AT* to assist in the following: for every time *t* find the acceptor block at time *t*. Obviously if *t* is meta-birth instant, searching *AT* will result in entry $<t, bid>$ where *bid* corresponds to the latest acceptor block created at *t*. If *t* is a no-change or a meta-deletion instant, searching array *AT* for *t* will provide some entry $<t', bid>$ where t' is the largest meta-birth that is less than *t*. The interesting case is if *t* is a meta-deletion: observe that the only events that may have happened between t' and t are other meta-deletions. However, by definition, each meta-deletion does not add a new acceptor block (since this would imply that it is a meta-birth); hence the acceptor block changes only at meta-births and *bid* was the acceptor block at *t*.

Figure 5 presents the access forest and array *AT* at $(t = 52)$, and after the "deletion" of block A $(t = 53)$, using the corresponding part of the meta-evolution depicted in Figure 4. Observe that array *AT* is not augmented by the no-change instant $(t = 52)$ or the meta-deletion instant $(t = 53)$.

Since the entries of the array are added in increasing time order, it is easy to maintain a multilevel, paginated index for *AT*. Pages are added only on the right path of this index. The number of entries in the array is clearly bounded by $O(n/b)$; consequently, the multilevel index and the array use $O(n/b)$ space. The total time needed to create the index is $O(n/b)$ which can be amortized on the total number of array entries yielding a constant updating per meta-change. With the multilevel index, access by time to any entry of the array is done in $O(\log_b n)$ I/O's.

A query is answered in two logical steps. For simplicity, assume that the time of interest *t* is a meta-birth instant and let block *Y* be the latest acceptor block created at *t*. In the first step, the given

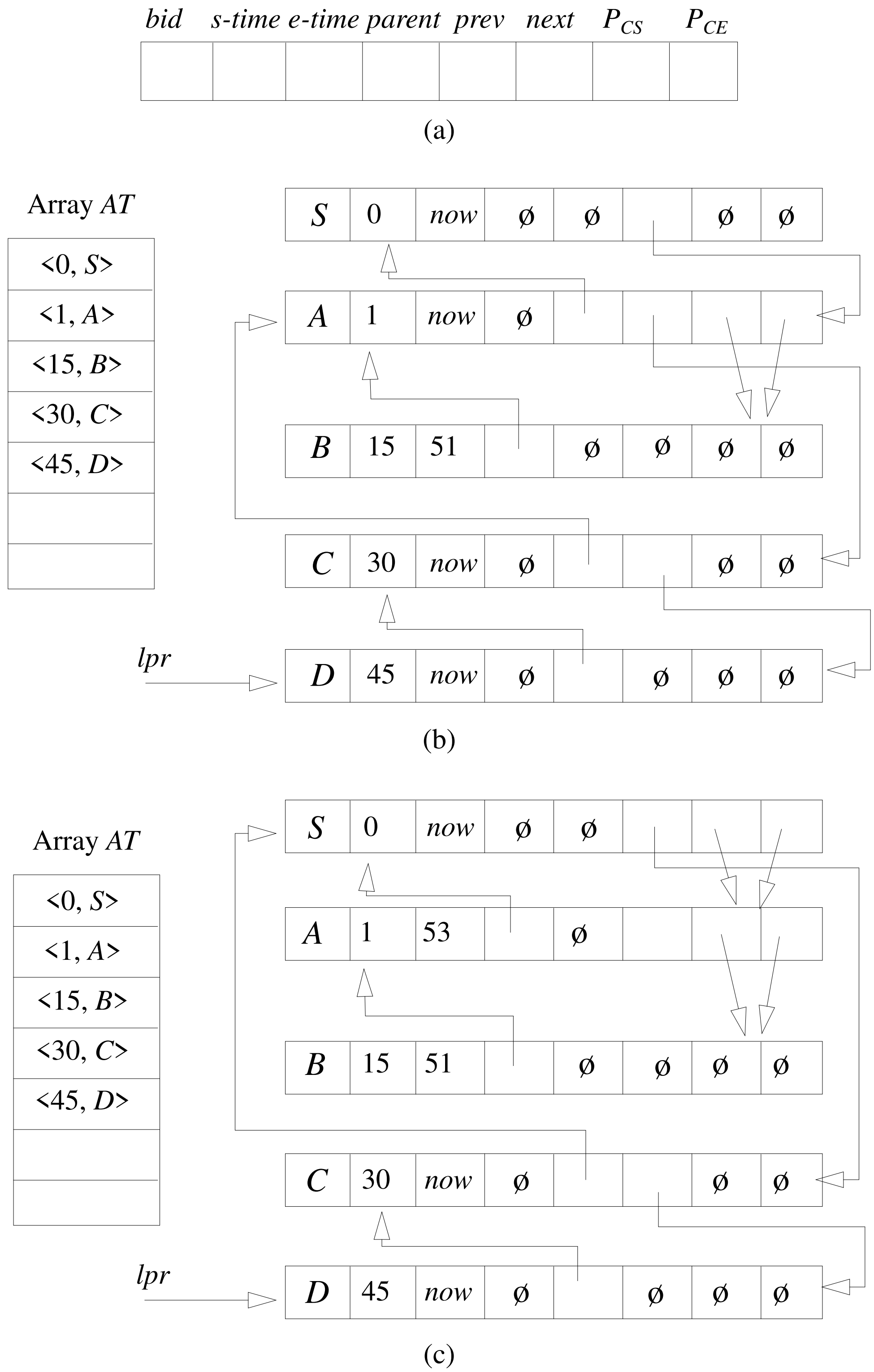

Fig.5: (a) A typical *SR* record. (b) The implementation of the access forest, at time t = 52, and (c) at time t = 53, using the first part of the meta-evolution of Figure 4. A block is now represented by its corresponding *SR* record. The ‘ø’ sign denotes an empty field. The *next* field of record *A* remains unused after *A*’s “deletion”.

time t is located after $O(\log_b n)$ I/O's in array AT using the multilevel index, and thus block Y is found. During the second step all useful blocks are found.

In order to find the useful blocks at a given time t, our method makes use of the following properties of the access forest:

(1) In each tree of the access forest, the *start_time* fields of the blocks in the tree are organized in a *preorder* fashion.

(2) The [*start_time*, *end_time*) interval of a block, includes all the corresponding intervals of the blocks in its subtree.

(3) The intervals $[d_i, e_i)$ and $[d_{i+1}, e_{i+1})$ of two consecutive children under the same parent block may have one of the following orderings: $d_i < e_i < d_{i+1} < e_{i+1}$ or $d_i < d_{i+1} < e_i < e_{i+1}$.

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

In order to complete the search for useful blocks, our method must also find which of the blocks from set-2 were useful at time t. The search starts from block Y. Then the left sibling of Y (if any) is checked to see if it is useful mode for time t. After this, for any block encountered in useful mode the search continues (1) to its subtree starting from its rightmost child, and, (2) to its left sibling. This is repeated at each level. For the blocks that are in path p the subtree search starts from the block that is in the path p. When a block is encountered that was not useful at time t the method does not check its left sibling or its subtree as no useful blocks can be found towards these directions. The search stops when no new useful block is found. Therefore, to find all the $|s_B(t)|$ useful blocks our method checks at most $2|s_B(t)|$ blocks in the access forest. As an example consider finding the useful blocks at time $t = 70$ from figure 4. The correct answer is: C, D, H, while the algorithm checks blocks: H, C, D, G, S and A. Similarly, for time $t = 60$, the correct answer is: C, D, E, and the algorithm checks blocks: E, D, C, S and A. We can now prove the following Theorem:

T h e o r e m 1: *The Snapshot Index solves the timeslice query in* $O(\log_b n + |s(t)|/b)$ *I/O's,* $O(n/b)$ *space and constant update processing.*

P r o o f: Clearly from Lemma 1 and the above discussion the space bound is established. The update processing contains two parts. First, from Lemma 1 the updating due to the copying procedure is constant per change instant (in the expected amortized sense). Second, the updating for creating the access forest and the array *AT* is also constant per meta-change and the number of meta-changes is clearly *O(n)*. For every useful block provided by the Snapshot Index, its records are checked in main memory for being "alive" at *t*; but due to Lemma 2, at least *a%* of the records of each useful block (except probably block *Y*) belongs to the answer *s(t)*. QED

It is easy to see that the constants in the *O()* notation are small: the constant in the space requirement is 1/(1-*a*) while the constant in front of $|s(t)|/b$ in the query time is 2/*a*. A particular optimization is possible for the query time, that reduces the constant to (asymptotically) 1/*a*. First observe that from a given useful block accessed by the searching procedure its left sibling block may also be accessed without this sibling being another useful block. This scenario may cost a lost I/O if the given useful block is not in the list *L* (if it is in the list the left sibling is also "alive" and useful). Such I/O can be avoided if a block keeps internally the *end_time* of its left sibling [28]. Since by construction, the left sibling block has an earlier *start_time*, it suffices to check its *end_-time* to decide if it useful or not. The *end_time* of the left sibling is available when a block is "deleted" in the access forest, since (also by construction) its left sibling has already being "deleted". In addition, a useful block accessed by the searching procedure may also access its rightmost child block without this child being another useful block. Such an access may happen from any block in the access forest, i.e., "deleted" or "alive". To avoid this lost I/O each block needs to keep the lifespan (useful period interval) of its rightmost child. Since when a block is "deleted" it does not accept new children, this information is available at "deletion" time. While a block is "alive" new children blocks can be added to it as they are "deleted". Thus a block in list *L* needs to keep the lifespan of its currently rightmost child. This lifespan is known when a child is added and it is updated when a new child arrives. Using the above techniques, our algorithm will access ONLY the useful blocks for a query plus one more block (the top node *S* of the list). Thus the constant in the query bound behaves like 1/*a*.

The choice of *a* affects the performance of the algorithm. Since supporting of temporal data greatly affects the space requirement, a choice of small *a* would keep the space consumption small.

Having the ability to tune its performance is another advantage of our approach since it can be accustomed to different application needs.

The Snapshot Index maintains a certain notion of *optimality*. We first show that the timeslice query is reduced to the "predecessor" problem for which a lower bound is then established.

The predecessor problem is defined as following: Given an ordered set B of N distinct items, and an item k, find the larger member of set B that is less than or equal to k. For the reduction of the timeslice query assume that set B contains integers $t_1 < t_2 < ... < t_N$ and consider the following real-world system evolution: at time t_1 a single real-world object with name (*oid*) t_1 is created, and lives until just before time t_2, i.e., the lifespan of object t_1 is $[t_1, t_2)$. Then, real-world object t_2 is born at t_2 and lives for the interval $[t_2, t_3)$, and so on. At any time instant t_i the state of the real-world system is a single object with name t_i. Hence the N integers correspond to $n = 2N$ changes in the above evolution. Consequently, finding the timeslice at time t reduces to finding the largest element in set B that is less or equal to t, i.e., the predecessor of t inside B. The Snapshot Index will thus solve the predecessor problem in $O(\log_b N)$ I/O's.

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

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

Observe that the lower bound was shown for the query time, irrespectively of the update processing. If the elements of set B are given in order, one after the other, $O(1)$ time (amortized) per element is needed in order to create an index on the set that would solve the predecessor problem in $O(\log_b N)$ I/O's (more accurately, since no deletions are needed, we only need a fully paginated, multilevel index that increases on one direction). If the integers are given out of order, then $O(\log_b N)$ time is needed per insertion (B-tree index). In the timeslice query however, as time is always increasing, $O(1)$ time for update processing per change is enough and clearly minimal. Using a hashing function our method achieves the constant updating (even in the expected amortized sense), so it is I/O-optimal. In a practical implementation any good hashing function will provide the good updating performance.

For the timeslice query, the Multiversion B-Tree [2] achieves $O(\log_b n + |s(t)|/b)$ query time and $O(n/b)$ space, but logarithmic (worst case) update processing. It is actually the range-timeslice query which requires to order keys by their value (as they arrive in time order but out of key order) that implies this logarithmic update. Thus, while the Snapshot Index is the optimal solution to the timeslice query, the Multiversion B-Tree is the optimal solution to the range-timeslice query. In addition, in order to accommodate enough records from a key range per page, the Multiversion B-tree uses a more complicated copying procedure that needs extra space. This procedure still results in *O(n/b)* space, but the constant is larger than the corresponding constant of the Snapshot Index. Thus for timeslice queries, the Snapshot Index (even if a B-tree is used instead of a hashing function) would use less space than the Multiversion B-tree.

The Snapshot Index can also optimally address "interval intersection" queries, i.e. given an interval $Q = [t_a, t_b]$ find all real-world objects whose lifespan intervals intersect Q. An interval $P = [t_c, t_d]$ intersects Q iff $(t_c \le t_a \le t_d)$ or $(t_a \le t_c \le t_b)$. If i_Q is the number of such real-world objects, the following theorem can be proved:

Theorem 2: *For a given interval Q the Snapshot Index solves the interval intersection query in* $O(\log_b n + i_Q/b)$ I/O*'s using* $O(n/b)$ *space and constant update processing.*

Proof: To solve the interval intersection query we must first reconstruct the real-world state $s(t_a)$ $(t_c \le t_a \le t_d)$ and then search for all real-world objects born after t_a and before or at t_b $(t_a \le t_c \le t_b)$. State $s(t_a)$ is reconstructed from the Snapshot Index after $O(\log_b n + |s(t_a)|/b)$ I/O's. For simplicity assume that t_a and t_b are meta-evolution "birth" instants for blocks *X* and *W*, respectively. Furthermore, assume that each block keeps a pointer to the block created after it (such a pointer is easily kept at no additional space or updating cost). Let $|s(t_b - t_a)|$ denote the real-world births after time t_a and before or at t_b; obviously $i_Q = |s(t_a)| + |s(t_b - t_a)|$.

Note that the only records that can appear in blocks *X* through *W* are the originals or copies of i_Q. Clearly this holds for the $|s(t_b - t_a)|$ real-world births since by construction they have to be placed in blocks between *X* and *W*. Consider the placement of the records of the $s(t_a)$ state at time t_a: in the worst case they may have been placed in $|s(t_a)|/ab$ blocks located before block *X*, including block *X*. This happens if the $|s(t_a)|/ab$ blocks were at a "threshold" condition at time t_a, each of them having exactly $a \cdot b$ "alive" records from state $s(t_a)$. During the evolution between t_a and t_b, real-world deletions may occur that will make these blocks degrade in a non-useful mode, creating copies from their "alive" records at t_a. These copies will be stored in the blocks between *X* and *W*. However, from Lemma 1 we can show that the number of possible blocks resulted from the evolution of the i_Q records can at most be $i_Q/(1-a)b$ and are in the sequence that starts from block *X* until block *W*. Block *X* is found from the logarithmic search needed to reconstruct $s(t_a)$; the rest of the blocks are accessed in sequence using the pointer mentioned above, until the first block that is created after t_b. QED

It follows that the Snapshot Index can <u>optimally</u> address queries of the form: "find all real-world objects that were born after (before) time *t*".

With a minor modification, the different versions of a given key can be linked together so that queries of the form: "find all versions of a given key" can be addressed in $O(\log_b n + v)$ I/O, where *v* represents the number of such versions. Each object record is augmented with one pointer. When an object is deleted, its *oid* is not deleted from the hashing function, rather it is marked as deleted. When a new version of this object is born, the hashing function can locate the previous version, thus it is possible to have a pointer form the record of the new version to the record of the previous version. When the first copy of an object is created due to the copying procedure, its record will point to the first record of the current version. Later copies of this version can thus point directly to the first record of their version. Thus the above query can avoid searching all copies of versions.

Another interesting query is finding the lifespan of a given object version. While the controlled copying procedure "partitions" the lifespan of an object into smaller intervals (one interval per copy of the object record), it is easy to find the whole lifespan of an object. As above, each copy can have a pointer to the first record of this object version; this first record is updated when the object is finally deleted in the real world evolution and its whole lifespan is known.

We conclude this section by discussing how the Snapshot Index can be used to address key-range timeslice queries. As it was mentioned earlier, such queries are answered more efficiently by methods that combine the time axis with the key range. Clearly no method can solve all queries optimally and the trade-off between the two approaches is on the update processing. Accordingly, each method should be used for the appropriate application.

The obvious way to answer a key-range query with a method that clusters data based only on time behavior is to produce the whole timeslice and then eliminate tuples outside the query range. While this solution may be acceptable when the size of the past state is small, it is unsatisfactory when the answer to the range query is much smaller than the timeslice size. Another solution [9, 31], is to pre-divide the key range into consecutive regions where a key region could be a single key or a collection of keys (like a sub-range of the key space or simply a relation). The history of each "region" is kept independently, using an individual temporal access method (like the Time Index, the Append-Only Tree or the Snapshot Index). A separate B-tree index on the regions directs a real-world change to update the appropriate key region history. There are three drawbacks with this solution: (1) as the regions are pre-decided a query may still be a small fraction of a region, (2) all regions in the given query range have to be searched, even if they may contain no "alive" objects at the time of interest t, and, (3) for each such region, a search for time t in its corresponding history "log" is performed. We are currently investigating a method that addresses some of the above drawbacks [17].

4. Performance of the Snapshot Index

A number of history evolutions were created in order to simulate and measure the performance of the Snapshot Index. Each evolution had T contiguous time instants; at a time instant t the number of "real-world" objects created is a uniformly distributed random variable in $[0, K_1]$. The evolution examples are distinguished by the maximum possible lifespan of an entity, LIFEMAX. Deletions were implemented as following: the lifespan of a born object is a random number from [1, LIFEMAX) after the object's birthtime. Different values of LIFEMAX were chosen. Note that in a practical case LIFEMAX can be also have infinite value (i.e., objects that never die), however this does not affect our algorithm, so for the simulation we used finite LIFEMAX values. In order to keep the number of changes constant per time instant, when a deletion time of an object is decided, we check if at that time the number of deletions is less than some constant K_2 (so as to guarantee at most $K = K_1 + K_2$ changes per instant). If that instant has already K_2 deletions we extended the lifespan of the object by one time instant, and so on, until we find the first time instant that can accommodate one more deletion. The choice of uniform distributions was made for simplicity only; our algorithm is not affected by such an assumption. We used $K_1 = K_2 = K/2$ and unless otherwise stated, $K = 10$ and $T = 2^{16}$. Throughout this section, a block is assumed to have enough space to accommodate 50 object records. Based on these evolutions we created our access method and measured its performance.

We first measured the three basic costs of our method as a function of the usefulness parameter a (Figures 6-8). Various evolution histories were created for different a's with LIFEMAX = 500. Figure 6, presents the space requirements of the Snapshot Index as a function of a. More specifically it shows the Space Ratio SR = (*actual space/minimal space*), where *actual* space corresponds to the number of blocks used by the access forest of our index structure, and *minimal* space represents the minimal number of blocks that one algorithm could use to store the changes of the evolution. Such an algorithm would simply store the changes on a log. The Snapshot Index also contains a multilevel index on array AT which does not appear in SR. However, the space used by the multilevel index on AT is clearly bounded by the size of array AT which in turn is much smaller (in number of blocks) than the number of blocks in the access forest (in the worst case the number of blocks of AT is the *actual* space divided by the block size b; in addition, in practice an array entry is much smaller than an object record). In Figure 6, as a increases the ratio SR increases. By increasing a, blocks are "deleted" easier since a larger number of "alive" objects is needed for a block to remain useful. This creates extra copies of records, which increases the space consumption. SR is upper bounded by an $1/(1-a)$ function, as Lemma 1 also implies.

The first part of the Query phase has a guaranteed $O(\log_b n)$ performance due to the multilevel index on array AT. More interesting is the second part where the search through the access forest finds the s "alive" objects at a given time t. The minimal number of pages that any algorithm would access in order to find the s objects is obviously s/b and corresponds to the case that the whole answer s was clustered in pages. Let QR (Query Ratio) be the ratio of the number of blocks accessed (checked) by our algorithm during the second part of the query phase, to the minimal number of blocks s/b that should at least be accessed; this ratio is thus a measure of data clustering. Figure 7

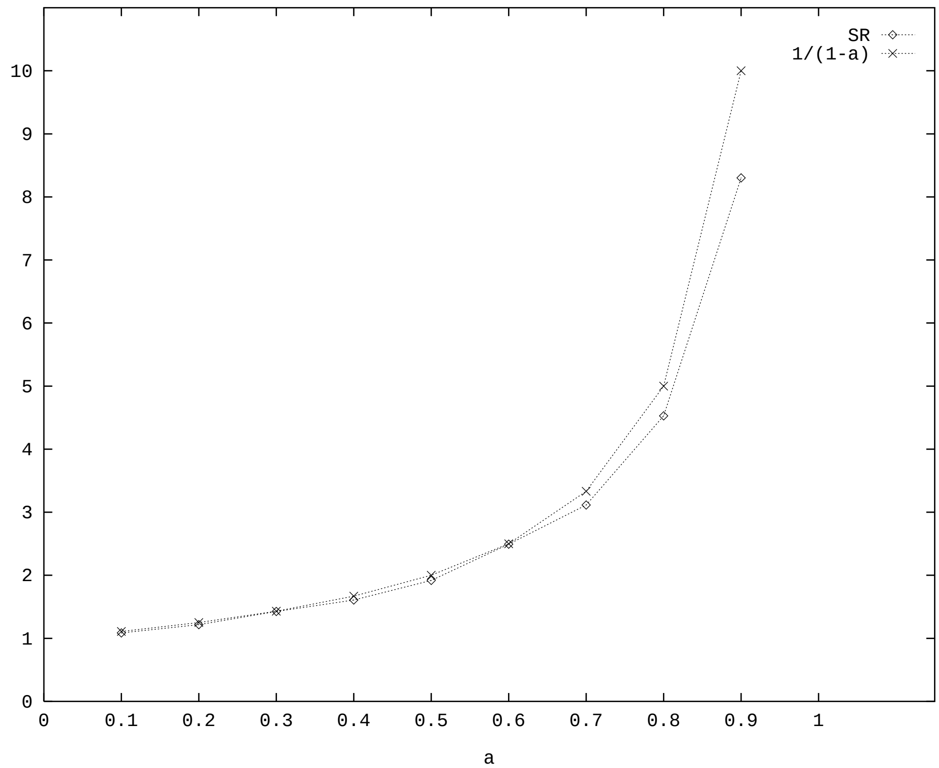

Fig.6: Ratio *SR* (:actual space/minimal space) as a function of the *usefulness* parameter *a* (LIFEMAX = 500, $T = 2^{16}$, $K = 10$). Function 1/(1-*a*) is also depicted.

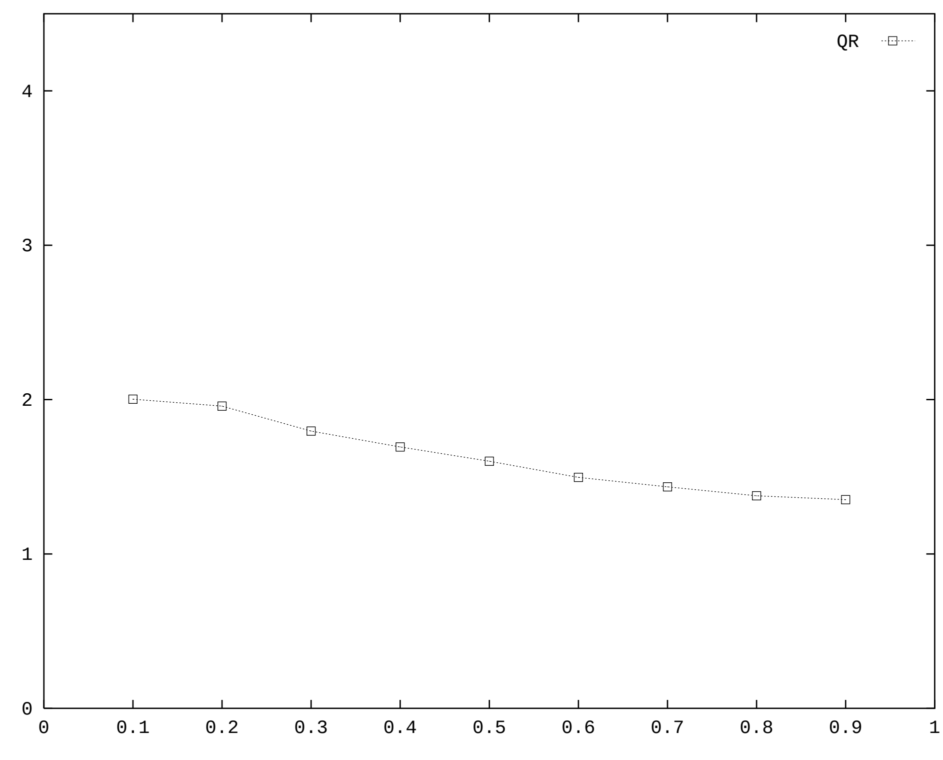

Fig.7: Ratio *QR* (:number of blocks checked/answer size in blocks) as a function of the *usefulness* parameter *a* (LIFEMAX = 500, $T = 2^{16}$, $K = 10$).

shows *QR* as a function of parameter *a*. Ratio *QR* was computed as an average of more than 2000 different timeslice queries. Obviously, as *a* increases we expect more block "deletions" which in turn creates more copying that will cluster the data better and hence will result in less query time. The simulation was performed on the algorithm without the extra page optimization; that is, the ratio is bounded by $2/a$ (instead of $1/a$). In practice, the actual number of blocks checked is much less than that ratio since many useful blocks may have more than just *a*% alive objects.

The update processing per change as a function of *a* is depicted in Figure 8. The update processing was computed as the total number of block accesses for a given evolution and then it was divided by the total number of "real-world" changes (object births or deletions), to give the Update Ratio *UR*. The update processing also includes the blocks accessed due to the hashing scheme. As *a* increases, there will be more blocks that become non-useful and are consequently "deleted", which in turn will create more object copying, i.e. more update processing per object. Ratio *UR* behaves similarly with *SR*, as *SR* represents the extra space per object; but extra space means also more updating.

We also measured the three basic costs against various lifespan lengths. Figure 9 shows all three ratios for different LIFEMAXs. Various evolution histories were created with usefulness parameter $a = 0.5$. The extra space used by the Snapshot Index per object (ratio *SR*) is practically constant, independent of the lifespan distribution. There is a slight decrease of *SR* as LIFEMAX increases since then objects in our evolution examples tend to live longer, implying less object deletions, less block "deletions" and accordingly less copying and space consumption. The same behavior holds for ratio *UR* as an object birth is just added on the end of the current acceptor page, while an object deletion may produce a block "deletion" and object copying. Ratio *QR* is also practically independent from different LIFEMAX lengths. Again there were more than 2000 different queries per each evolution example whose response times were averaged in order to compute *QR*.

The performance of the Snapshot Index for different choices of *K* is depicted in Figure 10 (with LIFEMAX = 500, $a = 0.5$ and $T = 2^{15}$). As expected, *SR* is independent of *K*. Ratio *QR* tends to decrease slightly with *K*. As *K* increases objects are better "clustered", as the newly created objects are placed together in the same block; better clustering means faster query times. Similarly, there is a slight increase of ratio *UR*; this is expected as larger *K* means more changes per time instant and consequently more updating. Figure 11 shows the three ratios as functions of the maximal evolution length *T*. The performance is independent of *T* (with LIFEMAX = 500, $a = 0.5$ and $K = 10$).

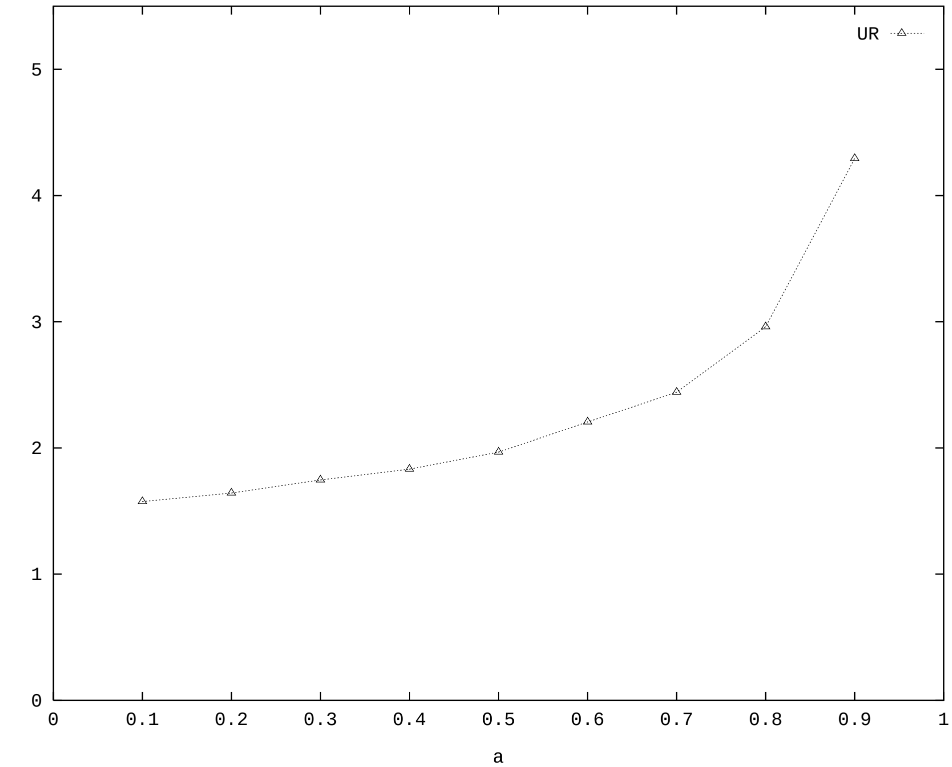

Fig.8: Ratio *UR* (:update processing per change) as a function of the *usefulness* parameter *a* (LIFEMAX = 500, $T = 2^{16}$, $K = 10$).

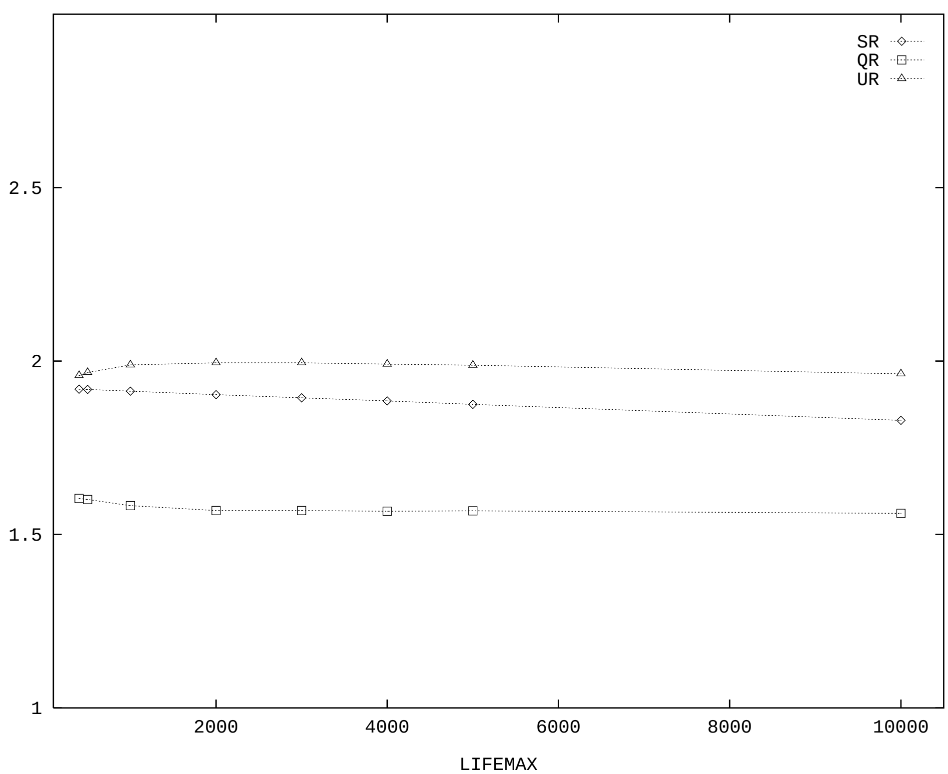

Fig.9: Ratios *SR*, *QR* and *UR* as functions of LIFEMAX ($a = 0.5$, $T = 2^{16}$, $K = 10$).

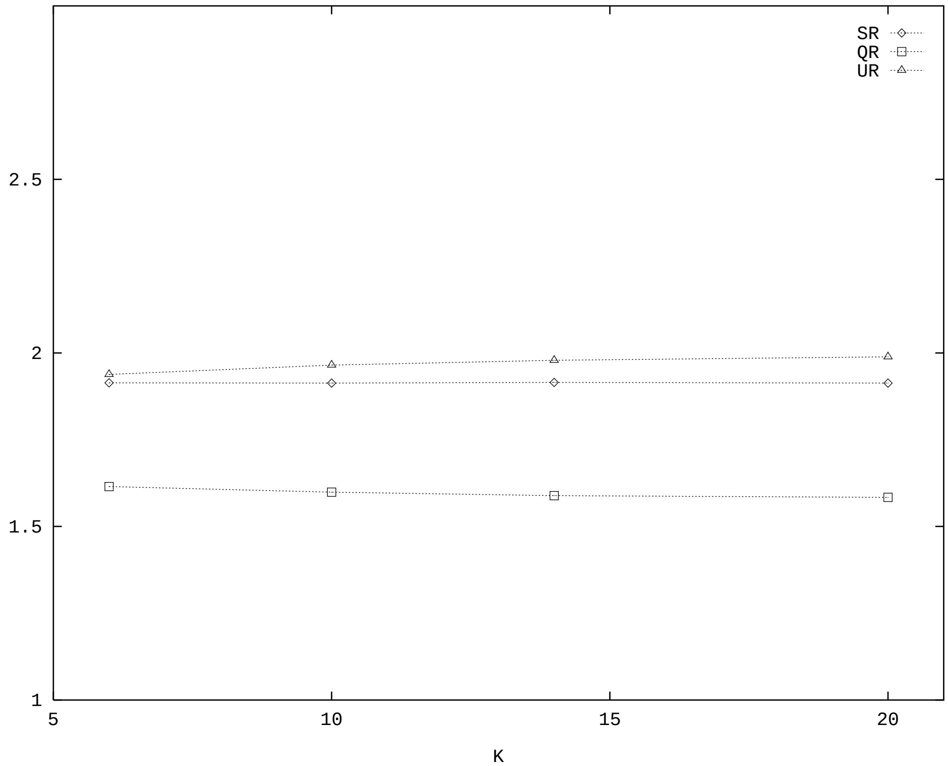

Fig.10: Ratios *SR*, *QR* and *UR* as functions of *K* ($a = 0.5$, $T = 2^{15}$, LIFEMAX=500).

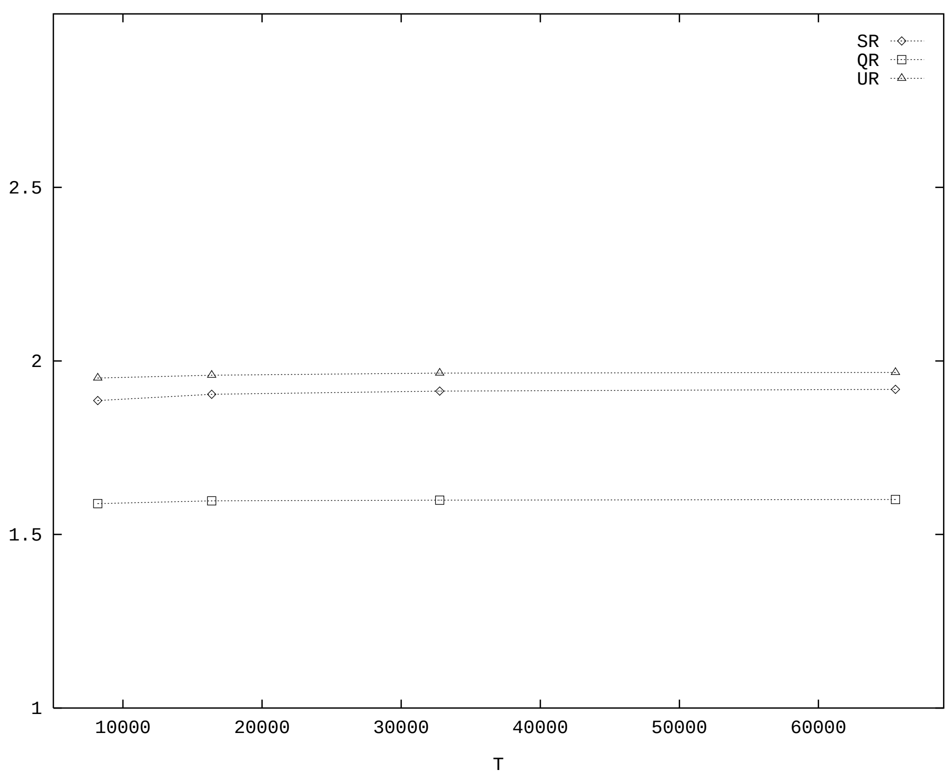

Fig.11: Ratios *SR*, *QR* and *UR* as functions of *T* ($a = 0.5$, LIFEMAX = 500, $K = 10$).

The choice of the usefulness parameter a can tune the behavior of the Snapshot Index to match the requirements of a particular application. Clearly, higher values of a will provide faster query times in the expense of extra space. On the other end, if a becomes very small, only one copy of each record is used (SR tends to 1) and the query time increases. The value of a can be changed dynamically during the evolution to move to different operating points. Existing blocks will retain their organization, however, new blocks will be organized with the new value of a. In this way, important parts of the evolution can have faster access by allocating more space.

5. Migrating data to Write-Once Optical Disks

As the magnetic disk space is limited and the amount of past data increases there is a need to migrate past states from the magnetic disk to a larger capacity medium such as a write-once optical disk. The write-once read-many (WORM) optical disk has the characteristic that after a block is written, its contents cannot change due to the error-correcting code that is written at the same time for the whole block. Thus data can be migrated only if it is not changed in the future. For simplicity we will assume that the block sizes in the two storage media are the same. We will present a migration procedure that results in a “hybrid” index, where past data can be transferred to a WORM disk while present data is on a magnetic disk. This procedure does not affect the performance characteristics of the Snapshot Index (except from the fact that access to an optical disk block is generally slower than access to a magnetic disk block). The migration can therefore be performed in parallel with the evolution input.

In the past, migration of past data has been addressed in the Time-Split-B tree, the Time Index, and the Segment R-tree. As data becomes “old” it can be transferred to the WORM disk. There are two ways to achieve the separation of old and recent: (a) With the “manual” approach, a process will vacuum all records that are “old” at the time of the process invocation; this vacuuming process can be invoked at any time. (b) With the “automated” approach, “deleted” pages are migrated to the optical disk due to a direct cause from the evolution process; thus when a page contains “dead” records it can be stored on the WORM disk. The total I/O and space consumption involved in the automated approach is likely to be smaller than in a vacuuming or manual method, since it is piggybacked on I/O which is necessary for index maintenance in any case.

The Segment R-tree uses the first approach (also called “asynchronous vacuumer” [34]). The other methods are using the second approach. The Time Index, “reserves” space on the WORM disk for the data of the magnetic disk. Since records are organized only according to birth time, if a record remains alive during the whole evolution, its page cannot be transferred to the WORM disk, thus the reserved space may be never used; this situation in the worst case can double the total space used by the method. The Time-Split B-tree migrates a page when it contains only past data, starting from the leaf pages. Since searching on the Time-Split B-tree is performed in a top-down fashion, a child page does not have to keep a pointer to its parent (as in a regular B-tree). This implies that when a page is transferred to the WORM it does not need to keep a pointer to its parent page (which is still on the magnetic disk) and thus no "reserved" space is needed. As a result the TSB migration occupies as much space as it would occupy at the magnetic disk.

Searching in the Snapshot Index may go from a child block to its parent block (in the access forest), thus the parent should be accessible from the child. Consequently, a block should know the address of its parent when moved to a WORM disk. Our approach also "reserves" space on the WORM, however, due to the copying procedure, we show that the unused reserved space can be limited to a controllable small fraction of the total space.

We consider first the migration of blocks from the "access forest". Conceptually, when a block is "deleted" from list *L*, it represents "past" information that could be stored on optical disk space ("past" in the sense that all the records in the block have been updated as "deleted"). Note that the first block *A* ever "deleted" at the <u>top</u> of the list becomes the first child under special block *S* and its position never changes. The special block *S* can be represented in the optical disk as some special address. All fields in the *SR* record of block *A* have then their final value; i.e., the parent of *A* is known, no left sibling exists and no new children will ever be added. This block and its whole subtree could thus be migrated to the WORM disk. After *A* is migrated, its WORM address is known and the whole subtree of *A* can also be migrated in preorder fashion. The same holds for every other block (and its subtree) which will ever become a child of special block *S*. A simple migration policy will thus send to the optical disk whole subtrees as they are attached under *S* in the access forest.

Even though for most practical cases this policy would be enough (assuming that eventually all blocks will be "deleted" and finally move under *S*) it will not work if some block *X* in list *L* remains "useful" for a very long period of time; *X* will be attaining a large subtree of "deleted" blocks, i.e. blocks that were inserted in list *L* after *X* but "lived" less than *X*. Such a subtree cannot move under *S* as long as its root block *X* is still "alive" in list *L*. We propose a more general migrating policy that can migrate blocks from every subtree in the access forest. Our policy takes into advantage the random access characteristic of the optical disk.

For the purposes of the following discussion, the *SR* record of every access forest block is augmented with two more fields: the optical disk address for the block itself (*obid*) and for the block's parent (*oparent*). The policy can be stated as: whenever an "alive" block *X* gets its first child, it "reserves" an optical disk block for itself; this optical block address is written in the *obid* field of the *SR* record of *X*. Any block of the access forest can then be migrated to the optical disk as soon as it is "deleted". We assume that if free optical blocks are requested by the migration procedure (for reservation or for writing) they are provided by the optical disk in increasing optical block-id order. This will ensure that a previously reserved optical block will not be reserved again. If ever block *X* gets deleted from list *L*, it will be written on the "reserved" optical disk block *obid*.

Note that migration can also take place in "batched" form by delaying the migration and writing to the optical disk whole subtrees (in preorder fashion), but for simplicity we assume that when a magnetic block is "deleted" it is directly migrated to the optical disk.

Suppose that during a query, the parent of an optical disk block has to be accessed. The method checks first if the *parent* block is in the same medium, i.e. if the parent block on the optical disk is written. If the parent block is empty, i.e. simply reserved, the search locates the *parent* block on the magnetic medium. The parent's siblings on list *L* are located on the magnetic medium. Since a block's children represent "deleted" blocks, they are searched on the optical disk. Hence it is possible to search through the access forest even though parts of it may already reside on the optical medium and parts on the magnetic one. The query time has the same I/O complexity as when the Snapshot Index uses only a magnetic medium.

At any time *t*, the total number of optical blocks written or reserved by the Snapshot Index for the access forest blocks is the number of "deleted" blocks plus at most the number of the currently "alive" blocks of list *L*, i.e., the optical disk space is $O(n/b)$. More specifically, since an "alive" block reserves optical disk space only when it gets its first child, in the worst case each alive block may have exactly one child block, resulting in reserving 1/2 times the actual space needed. This reservation ratio can be arbitrarily small if migration is "batched" in subtrees of size *w*. Then we reserve one optical disk block for each "alive" block whose subtree contains *w* blocks; in the worst case we would reserve 1/(1+*w*) times the actual space needed. Making an access forest block ready for migration requires a constant amount of work, thus the update processing remains constant per meta-evolution change.

Searching array *AT* is done through its multilevel index in a top-down fashion. Thus no parent pointer is needed per page of this structure. Migration from the *AT* array and its multilevel index is done starting from the pages of *AT*. Recall that the $O(n)$ changes of the whole evolution are kept in $O(n/b)$ pages (blocks) in the access forest. Each page of the access forest is referenced from an entry of array *AT*, thus the size of array *AT* is $O(n/b^2)$ pages. Whenever the entries of a page of array *AT* point fully to access forest data pages which have already been moved on the WORM disk, this page can also be moved on the WORM. (We assume that the corresponding array *AT* entry is updated when the access forest page is moved on the WORM. This implies that each access forest page has a pointer to its corresponding entry in array *AT*. In addition, each entry of array *AT* keeps an additional field which marks if the corresponding page address is on the magnetic disk or on the WORM. Each page of array *AT* keeps also a counter that is updated when a corresponding access forest page is moved on the WORM). Similarly, for the multilevel index a page is moved only if it points to children pages that have already been moved.

6. Conclusions

Given the amount of data that has to be stored, access methods for transaction-time databases ought to be efficient. In this paper we have presented the Snapshot Index, an I/O -optimal access method for timeslice queries. The Snapshot Index is a method that can be easily implemented and uses $O(n/b)$ space and $O(\log_b n + |s(t)|/b)$ I/O for query time, where $|s(t)|$ denotes the size of the answer, n is the total number of changes in the evolution and b is the size of an I/O transfer. Furthermore, it has constant processing per change (in the expected amortized sense) and can thus "follow" the real-world evolution. An advantage of our approach is that its performance can be tuned using the usefulness parameter a to match particular application needs (trading space for query time and vice versa). In addition, it can naturally migrate past data to a write-once optical disk, while preserving the same performance bounds. More research is needed for access methods in bi-temporal databases; while a lot of work has already been done for indexes that support a single time axis, it is still an open problem if there exists a method that optimally support both time notions, i.e., valid and transaction time.

Acknowledgments:

We would like to thank B. Salzberg for helpful comments on an earlier version of this paper and J.P. Schmidt for helpful discussions on lower bounds in a paginated environment. We would also like to thank T. Sellis for kindly providing us with access to the computing facilities of his laboratory at the National Technical University of Athens, where some of the simulations were performed while the second author was in Greece. Finally, we acknowledge the anonymous referees whose valuable remarks improved the presentation of this paper.

R e f e r e n c e s :

[1] I. Ahn, R. Snodgrass. Performance Evaluation of a Temporal Database Management System. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 96-107 (1986).

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

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

[4] J. Clifford, A. Croker. The historical relational data model (HRDM) and algebra based on lifespans. In *Proceedings of the* 3*rd IEEE International Conference on Data Engineering*, pp. 528-537 (1987).

[5] J.R. Driscoll, N. Sarnak, D. Sleator, R.E. Tarjan. Making Data Structures Persistent. *Journal of Comp. and Syst. Sci.*, **38**, pp. 86-124 (1989).

[6] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer, H. Rohnhert and R. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *Proceedings of the* 29*th IEEE FOCS*, pp. 524-531 (1988).

[7] R. Elmasri, Y. Kim, G. Wuu. Efficient Implementation Techniques for the Time Index. In *Proceedings of the* 7*th IEEE International Conference on Data Engineering*, pp. 102-111 (1991).

[8] R. Elmasri, G. Wuu, Y. Kim. The Time Index: An Access Structure for Temporal Data. In *Proceedings of the* 16*th Conference on Very Large Databases*, pp. 1-12 (1990).

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

[10] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOD International Conference on the Management of Data*, pp. 47-57 (1984).

[11] C.S. Jensen, editor et al. A Consensus Glossary of Temporal Database Concepts. In *ACM Sigmod Record*, **23**(1), pp. 52-64 (1994).

[12] C.S. Jensen, L. Mark, N. Roussopoulos. Incremental Implementation Model for Relational Databases with Transaction Time. *IEEE Transactions on Knowledge and Data Engineering*, **3**(4), pp. 461-473 (1991).

[13] P.C. Kanellakis, S. Ramaswamy, D.E. Vengroff, J.S. Vitter. Indexing for Data Models with Constraints and Classes. In *Proceedings ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 233-243 (1993).

[14] N. Kangelaris, V.J. Tsotras. Access Methods for Historical Data. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pp. KK1-KK12, Arlington Texas (1993).

[15] C. Kolovson. Indexing for Historical Databases. In A.Tansel, J. Clifford, S.K. Gadia, S. Jajodia, A. Segev, and R.Snodgrass (eds.). *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings (1993).

[16] C. Kolovson, M. Stonebraker. Segment Indexes: Dynamic Indexing Techniques for Multi-dimensional Interval Data. In *Proceedings of ACM SIGMOD International Conference on the Management of Data*, pp. 138-147 (1991).

[17] A. Kumar, V.J. Tsotras. Synchronization of Modular Histories. In preparation

[18] G.M. Landau, J.P. Schmidt, V.J. Tsotras. On Historical Queries along Multiple Lines of Time Evolution. To appear at the *VLDB Journal* (1995).

[19] S. Lanka, E. Mays. Fully Persistent B^+ Trees. In *Proceedings of ACM SIGMOD International Conference on the Management of Data*, pp. 426-435 (1991).

[20] 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 (1993).

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

[22] D. Lomet, B. Salzberg. The Performance of a Multiversion Access Method. In *Proceedings of ACM SIGMOD International Conference on the Management of Data*, pp. 353-363 (1990).

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

[24] N.A. Lorentzos, R.G. Johnson. Extending Relational Algebra to Manipulate Temporal Data. *Information Systems*, **13**(3), pp. 289-296 (1988).

[25] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, J. Woodfill. Designing DBMS Support for the Temporal Database. In *Proceedings of ACM SIGMOD International Conference on the Management of Data*, pp. 115-130, 1984.

[26] Y.Manolopoulos, G. Kapetanakis. Overlapping B+ Trees for Temporal Data. In *Proceedings. of 5th Jerusalem Conference on Information Technology*, Jerusalem, Israel, pp. 491-498 (1990).

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

[28] B. Salzberg, personal communication.

[29] B. Salzberg, V.J. Tsotras. A Comparison of Access Methods for Temporal Data. Available as a technical report from Polytechnic University and from Northeastern University.

[30] A. Segev, H. Gunadhi. Event-Join Optimization in Temporal Relational Databases. In *Proceedings of the 15th Conference on Very Large Databases*, pp. 205-215 (1989).

[31] A. Segev, H. Gunadhi. Efficient Indexing Methods for Temporal Relations. *IEEE Transactions on Knowledge and Data Engineering*, **5**(3), pp. 496-509 (1993).

[32] R.Snodgrass. The Temporal Query Language TQUEL. *ACM Transactions on Database Systems*, **12**(2), pp. 247-298 (1987).

[33] R. Snodgrass, I. Ahn. A Taxonomy of Time in Databases. In *Proceedings of ACM SIGMOD International Conference on the Management of Data*, pp. 236-246 (1985).

[34] M. Stonebraker. The Design of the Postgres Storage System. In *Proceedings of the 13th Conference on Very Large Databases*, pp. 289-300, 1987.

[35] V.J. Tsotras, B. Gopinath. Managing the History of Evolving Databases. *International Conference on Database Theory*, Paris, 1990. Published in *Lecture Notes in Computer Science*, Vol 470, pp. 141-174, Springer-Verlag (1990).

[36] V.J. Tsotras, B. Gopinath. Optimal Versioning of Objects. In *Proceedings of the 8th IEEE International Conference on Data Engineering*, pp. 358-365 (1992).

[37] V. J. Tsotras, B. Gopinath, G.W. Hart. Efficient Management of Time-Evolving Databases. To appear at the *IEEE Transactions on Knowledge and Data Engineering* (1995).