
Efficient Archivable Time Index: A Dynamic Indexing Scheme for Temporal Data*

Rakesh M. Verma

Department of Computer Science
University of Houston
Houston, TX 77004
E-mail: rmverma@cs.uh.edu

Peter J. Varman

Dept. of Electrical & Computer Engg.
Rice University
Houston, TX 77005
E-mail: pjv@rice.edu

Abstract

We present a practical and asymptotically optimal indexing structure for a versioned timestamped database with step-wise constant data. Three version operations, insertions, updates, and deletes are allowed for the present version, whereas query operations are allowed for any version, present or past. Snapshot and time-range queries can be answered optimally with this structure. As a two-level index, attribute-search and attribute-history queries can be solved in time proportional to the output size plus an additive logarithmic term. The time index uses linear storage; this improves upon previous work which either had logarithmic query overhead time and quadratic space, or linear space and linear query overhead time. The tradeoff is a small increase in the time for version operations from constant to logarithmic. All measures are worst-case. The index has a natural structure for archiving in write-once storage media like optical disks.

1 Introduction

Several applications in commercial and manufacturing enterprises need access to past versions of data. In a traditional (snapshot) database only the most current version of the data is stored. As records are updated or deleted, previous versions of records are overwritten and lost from the database. In many applications, however, it is necessary to retain and maintain access to the old versions of these records, so that queries about the past states of the database can be answered. A temporal database [3, 23, 22, 20] provides automatic storage and management of old (versioned) data.

For example, consider a departmental store database tracking the transactions associated with different items sold by its various departments. Figure 1(a) shows a hypothetical snapshot relation in this database; only the most recent stock-in-hand for an item is stored in the relation. In contrast, in a temporal database a new version of a record is created with each update as shown in Figure 1(c), following the transactions noted in Figure 1(b). Notice that each record is now timestamped with the time at which the up-

*Research supported in part by NSF grants CCR-9010366, CCR-9006300 and an IBM research award

date took place. Several queries that were not possible in a snapshot database can be made in this context. For instance, a query may request the stock-in-hand of each item on May 1, 1993. Such a query that asks for the state of the database at some time in the past is called a *snapshot* query. A generalization of such a query asks for the state of the database during some time interval; this is called a *time-range* query. For example, a query may require the average (or maximum/minimum) stock-in-hand of each item over the time period January 1, 1993 to May 31, 1993. A snapshot or time-range query can be restricted to satisfy an equality predicate on some other attribute as well. Such queries are called *attribute-search* and *attribute-history* queries respectively. In the special case where the attribute is a surrogate (*i.e.*, an attribute that does not depend on time), these queries will be called *key-search* and *key-history* queries respectively. For instance, finding the stock-in-hand of all items in Department A on May 1, 1993 is an attribute-search query, while tracking the inventory of item number 560 over a time range is a key-history query.

It is necessary to organize the data to answer queries such as these efficiently, while allowing for database-modifying operations that insert new records into the database, and update or delete current records. Such a structure is referred to as a *multiversion access structure* (MVAS) or time index.

In this paper we present an efficient and asymptotically optimal MVAS with *guaranteed worst-case bounds* on storage, and on the time needed for some frequently occurring query operations. Our structure maintains this performance in the presence of `insert`, `update` and `delete` operations on the current database. This work improves upon earlier works as follows. We decrease the worst-case storage requirements over the proposals in [9, 8, 7] by an order of magnitude (from quadratic to linear), without increasing the

time for inserts and updates. We decrease the worst-case time requirements for snapshot and time-range query operations over the proposal in [11] from linear to logarithmic, at a small increase in cost (from constant to logarithmic) for inserts.

Our scheme is inspired by the data structure of McCreight [18] for rectangle intersections. However, McCreight's data structure, like most geometric data structures (e.g. [19, 21]), is a binary tree structure for main memory, and hence the focus is on record-level efficiency. We modify and adapt this scheme for secondary storage and extend it for temporal indexing. The rest of the paper is organized as follows. In Section 1.1 we discuss the model used in the paper. In Section 1.2 we discuss previous work in more detail. We then give an overview of our MVAS in Section 2. In Section 3 we describe how to use this structure to perform query operations. In Section 4 we describe the version operations and dynamic behavior of the MVAS. Pagination of the MVAS is discussed in Section 5. The paper concludes with Section 6.

1.1 Model

In our model a record is the basic unit of storage and retrieval. The records may contain the actual data or may be pointers to the objects of interest. In either case we refer to these as *data records*. Two types of operations may be performed on the database - *version* operations: these are operations that modify the database, and *query* operations that retrieve selected data records. The version operations are *insert*, *update* and *delete*. The term *version number* is used to keep track of the total number of version operations performed on the database [5]; every version operation increases the version number by one.

Each version operation has a timestamp associated with it. Version operations can

be performed on only the latest version of records *i.e.* on the current database. However, queries may be posed about objects present at any time, past or current. In the classification proposed by Snodgrass [24] this model is referred to as a rollback database. It is convenient to consider the timestamp of an operation to be the same as its version number. That is the timestamps of successive version operations are mapped to a set of consecutive integers. We use the terms *time* and *version number* synonymously. Finally, in common with other related work the model assumes that we are dealing with step-wise constant data; *i.e.*, the values of a record with a given key remain unchanged between successive updates to the record with that key.

Each record has a pair of timestamps **start** and **end**. The lifespan of the record is from **start** to **end**. The special time value $\$$ is used to indicate the present time. A record is created at some time t_0 using an **insert** operation and its lifespan is from t_0 to $\$$. The record remains valid until the time at which it is either **updated** or **deleted**. If the record is updated at time t_1 , a new version of the record is created that is valid from t_1 to $\$$, and the **end** time of the previous version becomes t_1 . If the record is deleted at time t_1 , then no further version of the record is created. A record is said to be valid at time i if i is greater than or equal to **start** and less than **end**. A *present version* record is one which is valid at the present time, $\$$.

Figure 2 shows these definitions graphically. Records with keys A, B, D, and C are inserted at times 1, 2, 3, and 4 respectively. At time 5, A gets updated and its new version (denoted as $(A, 2)$) remains valid until time 8, when another update is made to it. Similarly B gets updated at times 7 and 12, and D at times 6 and 10. C gets deleted at time 9 and no new version of it is therefore created. The valid entries at time 3 are $(A, 1)$, $(B, 1)$ and $(D, 1)$. If the current time is 13, then

the present version entries are $(A, 4)$, $(B, 3)$ and $(D, 3)$.

1.2 Previous Work

Many previous proposals in recent years have considered the design of multiversion access structures [1, 2, 6, 10, 11, 13, 14, 15, 16, 17, 25]. The Monotonic B+-Tree of Elmasri et al [10], and the methods, using AP, ST and AT trees, of Segev and Gunadhi [11] are based on keeping the time and key dimensions separate. Snapshot and time-range queries use the AP tree in [11] and the monotonic B+-tree in [10] as the indexing mechanism. An advantage of the AP tree of [11] is that insertion requires only a constant number of bucket accesses. However searching in an AP tree for a snapshot or time-range query involves scanning all records with start time less than the given query time; this results in a worst-case number of bucket accesses that grows linearly with the total number of buckets present at the query time. The proposals in [10, 8, 7] use the notion of indexing points to perform snapshot or time-range queries in logarithmic time plus a term that depends on the output size. However, the tradeoff is that it requires quadratic space in the worst case. In comparison, our method uses only linear storage and requires only logarithmic time overhead for the same queries, thereby achieving the good time performance of [10, 8, 7] and the good storage utilization of [11].

For attribute-history and attribute-search queries, a two-level indexing scheme has been proposed [10, 11]. The first level is an attribute index (which may either be on the surrogate or on a temporal attribute); the second level is a time index that is linked to the leaf level nodes of the attribute index. In [11] the AP tree is used as the time index, while [10, 8, 7] use the Monotonic B+-tree for this purpose. We follow a similar approach to answer attribute-search

and attribute-history queries, by linking our MVAS to the leaves of the attribute index. All of the benefits of our MVAS discussed above still hold when it is used as the second level of a two-level index. Several simplifications of the scheme are possible when the attribute is a surrogate; for instance, the present version entries for a surrogate attribute could be stored directly at the leaf node of the attribute index to which the time index is linked.

Multiversion access structures for main memory databases have been discussed by Tsotras and Gopinath [26]. Since these models do not consider the pagination problems associated with secondary storage, the results are not comparable with our method. Ahn and Snodgrass [1] proposed the use of access lists that permit efficient retrieval of the history of any key. They also discuss various alternative clustering methods associated with this approach. There have been a number of approaches that combine the attribute and time dimensions into a single multi-dimensional access structures. The Write-Once B-Tree (WOBT) of Easton [6] has been the basis of several such multiversion access structures [15, 2]. All of them are quite complicated to maintain and require more space than the corresponding two-level schemes.

A more general model allowing updates in past versions of objects (full persistence rather than partial persistence in the terminology of [5]) was proposed in [14]. As has been observed previously [2], the storage requirements for this structure are excessive even when specialized for partial persistence. Finally multiversion access structures in the POSTGRES system [25] and the schemes in [17, 13] (based on R-trees [12]) are more concerned with indexing historical databases.

2 Time Index

In this section we give a logical view of the MVAS. It consists of two structures: a *current version* access structure (CVAS) and a *past version* access structure (PVAS). CVAS holds the present version records; *i.e.*, those whose `end` time is `$`. CVAS is a conventional B+-tree using the `start` time of a record as the ordering key. The PVAS has the structure of a multiway tree whose nodes correspond to time intervals. We define a *segment* of a record to be the time interval $[a, b]$, where a is the `start` time, and b the `end` time.

2.1 Overview of PVAS

Logically PVAS may be viewed as a dynamically growing binary tree. For convenience we describe the structure as a fixed size complete binary tree, \mathcal{T} of $2^\alpha - 1$, $\alpha \geq 1$, nodes. Its pagination is described in Section 5. Let T denote the current time. At time $T \leq 2^\alpha$ only some of the nodes would actually have been created. A node is denoted by $N_{[i,j]}$, where $i < j$ are integers representing time. The *span* of a node $N_{[i,j]}$ is $(j - i)$. The root is $N_{[0,2^\alpha]}$. The left child of $N_{[i,j]}$ is the node $N_{[i,(i+j)/2]}$ and its right child is $N_{[(i+j)/2,j]}$. Hence the span of a node is the sum of the spans of its two children. The span of a leaf node of the tree is two.

Records with `end` timestamps at most T are stored in the nodes of the binary tree. Each record is stored in *exactly one* node. A record $[a, b]$ is stored in the node $N_{[i,j]}$ having the shortest span, and satisfying $i \leq a < b \leq j$. In other words, a record is stored in the lowest-level node (*i.e.*, closest to the leaves) whose interval contains the record. Figure 6 shows the storage of record $[3, 16]$ in the time index. The record $[3, 16]$ is contained in the intervals of the nodes $N_{[0,64]}$, $N_{[0,32]}$ and $N_{[0,16]}$. Since the last of these has the shortest span, the record is stored

in $N_{[0,16]}$.

The nodes of \mathcal{T} are partitioned into three disjoint sets containing *passive*, *active* and *future* nodes, as defined below. Node $N_{[i,j]}$ is passive if no more records can ever be stored in that node. It is active if it is possible for a record ending at T to be stored in that node. It is a future node, if the only records that can be stored in it must have **end** times greater than T , *i.e.*, sometime in the future.

All nodes begin as future nodes at $T = 0$. Node $N_{[i,j]}$ becomes active at $T = (i + j)/2$ if it is a leaf node and at $T = (i + j)/2 + 1$ otherwise. (See section 4.) At $T = j + 1$ it becomes passive and remains passive thereafter. Future nodes do not store any records and therefore do not need to be physically created till they become active. The information stored in passive nodes will not change in the future; hence their records can be organized and stored using simple *static* access structures that facilitate efficient search. However, since records can be added to active nodes, *dynamic* access structures are needed to maintain the records of these nodes. Finally records must be moved from the dynamic to the static structure as the nodes become passive.

The records in all *passive* nodes are stored in two *sequential* files, **IFILE** and **JFILE**. In both files, if node $N_{[a,b]}$ becomes passive before node $N_{[x,y]}$, then all records of the former node are stored before those of the latter. If several nodes become passive at the same time, the records of active nodes higher in the binary tree will precede the records of active nodes lower in the tree. Hence, maintaining these files is easy. We merely need to append the records of newly passive nodes to the end of **IFILE** and **JFILE**. In **IFILE** (**JFILE**) the records of any node are stored in increasing (respectively decreasing) order of their **start** times (respectively **end** times). Further, a passive node in the binary tree holds two pointers to the beginning of its records in **IFILE** and **JFILE** respectively. This makes

it easy to find all records of a passive node whose **start** (**end**) times are smaller (larger) than a given value t , by an efficient sequential scan of **IFILE** (**JFILE**), using the pointers of the passive node.

Two dynamic structures, **ITREE** and **JLISTS**, are used to store the records of the *active* nodes. An active node in PVAS holds pointers to the beginning of its records in **ITREE** and **JLISTS**. These records are similar to those of **IFILE** and **JFILE** respectively. **JLISTS** is a collection of lists, one for each active node; each list is maintained in sorted order of the **end** times of records stored at that node. Since, records at an active node will always appear in increasing order of their **end** time, maintaining **JLISTS** involves simply appending to the end of the appropriate list. When this node becomes passive, its list is appended in reverse order to the end of **JFILE**. **ITREE** is implemented as a B+-Tree [4] of records using the **start** time of a record as the ordering key. Hence traversing the leaves of the B+-Tree sequentially will give us the records in sorted order of their **start** times. We show in Section 4 that the records in the B+-Tree will also be clustered by nodes. This allows us to efficiently update **IFILE** as well when an active node goes passive.

3 Query Operations

We first describe a time-range query operation using the MVAS. This is the fundamental search operation. Given a query interval $Q_{[x,y]}$, we wish to report all segments that intersect with this interval.

Both CVAS and PVAS must be searched. The search in CVAS is simple since all present version records whose **start** time is less than equal to y must be output. Since CVAS is a B+-tree using **start** as the ordering key, this is easy to do in $R_{curr}/\alpha + \log_{\alpha} W_{curr}$ worst-case bucket accesses, where W_{curr} is the total number of buckets contain-

ing present version records, R_{curr} is the number of such records satisfying the query, and α is the minimum bucket occupancy. Note that α is at least $B/2$. To search the PVAS, begin at the root node. When a node \mathcal{N} of the PVAS is visited, one of the following actions is taken. Either *all* segments in the *subtree* rooted at \mathcal{N} intersect $[x, y]$; in this case these are reported and the tree is not traversed any further. Otherwise, all segments stored in \mathcal{N} that intersect $[x, y]$ are found and reported. In the latter case, the query is then decomposed into either one or two subqueries, depending on whether the query interval intersects one or both of \mathcal{N} 's children's intervals, and the subquery passed to the appropriate child or children. We show below that even though a query may split into two subqueries at a node, the maximum number of binary tree nodes ever visited is bounded by $4d$ in the worst case, where d is the height of the tree.

We introduce some notation to help describe the search procedure. With respect to a node $N_{[i,j]}$, query $Q_{[x,y]}$ is classified as one of the following four types. Let $m = (i+j)/2$ denote the midpoint of the interval $[i, j]$.

1. **Class A:** $i < x \leq y < m$ or $m < x \leq y < j$.

The search interval does not straddle the midpoint of the node interval, and is properly contained within the node interval.

2. **Class B:** $i < x \leq m \leq y < j$.

The search interval straddles the midpoint of the node interval, and is properly contained within the node interval.

3. **Class C:** Either $i = x$ or $j = y$, but not both.

Exactly one endpoint of the search interval coincides with the endpoint of the node interval.

4. **Class D:** $i = x$ and $j = y$.

The search interval coincides with the node interval.

The details of the search are described in procedure Search below. We illustrate its operation with an example. Consider the query $Q_{[30,64]}$ in the tree of Figure 6. The root of the tree spans the interval $[0, 64]$, and hence at the root it is a class C query. Since the query interval straddles the midpoint of the node interval, all segments stored at the root node overlap the query interval. The query is then decomposed into two subqueries and passed to children $N_{[0,32]}$ and $N_{[32,64]}$. The subquery $Q_{[32,64]}$ to the right child $N_{[32,64]}$ is of class D, and hence all the segments stored in its subtree *will* intersect the query interval. The subquery $Q_{[30,32]}$, to the left child $N_{[0,32]}$, is of class C. This will be decomposed into a sequence of class C subqueries and passed successively down the chain of right children, $N_{[16,32]}$, $N_{[24,32]}$ and $N_{[28,32]}$. At each of these nodes, we reports intersections by comparing the **end** times of the records stored at that node with the start of the subquery interval. Finally the class D subquery $Q_{[30,32]}$ is passed to the rightmost leaf of this chain $N_{[30,32]}$, and all segments stored there are reported.

It is easy to verify that the total number of nodes of the tree that are searched with queries of class A, B or C combined is no more than $2d$. This may be seen as follows. A class A query generates exactly one subquery, of class A or B. A class B query generates two class C subqueries. A class C subquery generates at most one class C and one class D subquery. Finally, a class D query does not generate any subqueries. The number of class D subqueries is bounded by the number of class C subqueries, and hence cannot exceed $2d$. Hence, an upper bound on the number of nodes in the binary tree that are visited is $4d$.

Procedure Search($Q_{[x,y]}$, $N_{[i,j]}$)

Let $m = (i + j)/2$, the midpoint of the node interval.

Case class of $Q_{[x,y]}$ with respect to $N_{[i,j]}$:

Class A:

if ($y < m$) /* *left-half query* */

Report all segments $[a, b]$ stored at $N_{[i,j]}$
for which $a \leq y$;

Search ($Q_{[x,y]}$, $N_{[i,m]}$);

else if ($x > m$) /* *right-half query* */

Report all segments $[a, b]$ stored at $N_{[i,j]}$
for which $b \geq x$;

Search ($Q_{[x,y]}$, $N_{[m,j]}$);

Class B:

Report all segments stored in $N_{[i,j]}$;

Search ($Q_{[x,m]}$, $N_{[i,m]}$); /* *Class C* */

Search ($Q_{[m,y]}$, $N_{[m,j]}$); /* *Class C* */

Class C:

if ($i = x$)

if ($y < m$)

Report all segments $[a, b]$ stored at
 $N_{[i,j]}$ for which $a \leq y$.

Search ($Q_{[i,y]}$, $N_{[i,m]}$); /* *Class C* */

else if ($y \geq m$)

Report all segments stored in $N_{[i,j]}$;

Search ($Q_{[i,m]}$, $N_{[i,m]}$); /* *Class D* */

Search ($Q_{[m,y]}$, $N_{[m,j]}$); /* *Class C* */

else if ($j = y$)

/* *Symmetrical actions as above* */

Class D:

Output all segments stored in the
subtree rooted at $N_{[i,j]}$.

End Case

Nodes visited by the search procedure above may be either passive or active. We describe the reporting of intersecting intervals for passive nodes here.

As described in Section 2.1 a passive node contains pointers to its records stored in **IFILE** and **JFILE**, and all segments belonging to a node are clustered together. Hence reporting intersections for class B intervals involves a simple sequential scan in any one of these files. For class A and C intervals we need to sequentially scan the records in either increasing **start** time order or decreasing

end time order until the comparison indicated in the procedure above fails. Depending on the case encountered either **IFILE** or **JFILE** is sequentially scanned until an out-of-place record is encountered. Finally, for a class D interval, we need to output all records stored in the subtree of the node. The records of nodes are stored in **IFILE** (and **JFILE**) in the order in which the nodes become passive. As will be apparent from the discussion in Section 4, this will mean that the records of all nodes in a subtree will be clustered. Immediately following a node's records will be the records of nodes in the chain leading to its rightmost leaf. Immediately, before it will be the records of the nodes in the remainder of the subtree. Hence, by a sequential scan in both directions, all records in the subtree can be obtained.

Let \mathcal{S}_p denotes the set of passive nodes in the binary tree that are visited by the search query. Let n_i be the number of records satisfying the query at node $i \in \mathcal{S}_p$. Let N_p be the total number of records satisfying the query that are stored in passive nodes. Let B denote the number of records in each block of the sequential files. Since the qualifying records at each node are accessed in sequential order, the number of block accesses required at node i is no more than $\lceil \frac{n_i}{B} \rceil + 1$. Hence the total number of block accesses to retrieve qualifying records in passive nodes is upper bounded by $\sum_{i \in \mathcal{S}_p} (\lceil \frac{n_i}{B} \rceil + 1) \leq N_p/B + 2|\mathcal{S}_p|$.

Similarly, for active nodes all records accessed will be made by a sequential scan of either **JLISTS** or the leaves of **ITREE**. Hence, an upper bound for the number of blocks needed to access records in the active nodes is $N_A/B + 2|\mathcal{S}_A|$, where N_A is the total number of records satisfying the query that are stored in active nodes, and \mathcal{S}_A is the number of active nodes visited by the search. Note that $|\mathcal{S}_p| + |\mathcal{S}_A|$ is the total number of nodes visited by the search procedure, which is at

most $4d$ in the worst-case, where d is the height of the binary tree. Thus, the number of block accesses is bounded by the sum of the minimum number of blocks required to hold the output records plus a logarithmic overhead. Note that this is a worst case upper bound and holds for the worst possible distribution of the segments.

For snapshot queries the same procedure is used; note that since the query interval is just one time point, only one path down the tree will be followed in this case. Attribute-search and attribute-history queries are performed exactly as in [11, 9] using the two-level indexing schemes proposed there. The time complexities of the attribute-history and attribute-search operations are just the sum of the complexity for the corresponding search operation using the MVAS (logarithmic plus number of output buckets) and the height of the attribute index.

4 Version Operations

An `insert` operation at time t inserts the record into CVAS with `start` time t . Since CVAS is implemented as a B+-tree ordered by `start` time of the records, and every version operation increments time, this merely involves adding the record to the rightmost (largest time) leaf bucket of the B+-tree. A `delete` at time t requires the record to be deleted from CVAS. Assume the record to be deleted has the `start` time t_0 . The record must be removed from CVAS, its `end` time set to t , and the record added to PVAS. An `update` at time t of a record whose `start` time was t_0 requires three operations: removal of the record with `start` time t_0 from CVAS, changing the `end` time to t and adding it to PVAS, and insertion of a new record with `start` time t and `end` time $\$$ to CVAS. Note that all records inserted into PVAS will be to active nodes.

We will now construct an efficient access scheme for the *active* nodes. The time at

which a node $N_{[a,b]}$ becomes active is given by $(a + b)/2$ if it is a leaf node (span is 2) otherwise it is $(a + b + 2)/2$ (for a non-leaf node this is also the time at which all nodes in its left subtree are passive), and the time at which it becomes passive is $b+1$. Note that an active node's interval always contains the current time, $\$$.

4.1 Active Nodes

For the following discussion, let $\$$ be the current time and let the root node of the index represent the interval $[0, T]$, where $T = 2^{\lceil \log_2 \$ \rceil}$, the least power of two that is no smaller than $\$$. Recall that for time-range queries, we require the records stored at each node to be sorted by `start` time (in `IFILE`) and sorted by `end` time (in `JFILE`). A third requirement is that the records in a subtree be clustered together so that they can be efficiently accessed for class D queries.

The requirement with respect to `end` times is easily handled because records arrive for insertion to PVAS in increasing order of `end` times. Therefore, it is enough to keep a list of the records at each active node in the order of their arrival to PVAS.

To keep the records in increasing order of their `start` times, we can store the segments at each active node in a B-tree, using the `start` time as the ordering key. However, the special properties of this problem permit a more efficient solution. The property that we exploit is the following.

Proposition 1 *If $n_1 = N_{[x_1, y_1]}$ and $n_2 = N_{[x_2, y_2]}$ are any two distinct active nodes at time $\$$ and $x_1 < x_2$, then all segments that must be stored at n_1 must have their starting times before all segments that must be stored at n_2 .*

Note that this is true only for nodes that are active at $\$$ and for segments that arrive for insertion during the period when both nodes are active. In particular for a pair of

nodes at least one of which is no longer active this is clearly not true. For example, the segment $[33, 63]$ is stored at node a representing $[32, 64]$ and the segment $[35, 255]$ is stored at node b representing $[0, 256]$, but clearly node a is no longer active when the segment $[35, 255]$ arrives for insertion into the index.

To prove this property, assume for the sake of contradiction that segment $[s_1, t_1]$ must be stored at $n_1 = N_{[x_1, y_1]}$ and $[s_2, t_2]$ must be stored at $n_2 = N_{[x_2, y_2]}$ and that $s_2 < s_1$. First observe that all active nodes are on a single path from the root to a leaf, i.e., no pair of active nodes can have a lowest common ancestor that is distinct from both. (This is because two nodes in our index that do not have an ancestor-descendant relationship can have at most one point in common, an endpoint.) Therefore, since $x_1 < x_2$ (by assumption), we must have that n_1 is an ancestor of n_2 and this implies that $y_1 \geq y_2$. Now, by our criterion we have that $x_2 < s_2$ so combining it with $s_2 < s_1$, we have $x_2 < s_1$. But, since both segments are closed and both nodes are active at $\$,$ we have $t_1 \leq \$,$ $t_2 \leq \$,$ and $\$ \leq y_2 \leq y_1$. But, then we have a contradiction since n_2 has a smaller span than n_1 and properly contains $[s_1, t_1]$, so $[s_1, t_1]$ should be stored at n_2 or its descendants.

Because of this property, we can use just a *single* B+-tree ordered by the `start` time to hold the records of *all* active nodes. The records belonging to any active node will be clustered together in the leaves of the B+-tree. A pointer to the leaf in the B+-tree that stores the first record belonging to active node \mathcal{N} is kept at \mathcal{N} . We can scan the leaves of the B+-tree from this point to access all the records belonging to \mathcal{N} in increasing `start` time order. Finally, class D queries require that all the records in a subtree be clustered together. Since all the active nodes are on a single path from the root to a leaf, and the active nodes in the subtree of \mathcal{N} hold records with monotonically increasing `start`

times (since they are descendants of \mathcal{N}), they are also clustered together in the leaves of the B+-tree. Therefore, for searches with a class D interval at node \mathcal{N} we just follow \mathcal{N} 's pointer into the B+-tree for the first record, and then retrieve all records following that in the leaves of the B+-tree.

The time-range query for active nodes is similar to that for passive nodes except that the node pointers are to `ITREE` and `JLISTS` (rather than `IFILE` and `JFILE`), and the interval spanned by an active node must be found by examining the information stored in the node. This is not a problem since we mark nodes active or passive and use this to interpret the fields of the node, and the node interval can be stored with the node itself.

4.2 Active Node Creation

We now discuss the creation of an active node, and its insertion into `PVAS`. Node $N[a, b]$ is created at time $(a + b)/2$, i.e., the time at which it becomes active. Since we have the time $\$,$ we need to determine which nodes to create based on $\$.$ This is done by the following easily-understood procedure. A more efficient $O(\log \$)$ -time procedure can be constructed.

```

Procedure Create_active_nodes ($)
/* Assumes $ > 1, otherwise [0, 2] is the only
active node */

T = 2[log2$] ;
If $ is odd create_node([$ - 1, $ + 1]) ;
/*create active leaf node, if needed */
t = 4 ;
s = T/2;
while (t ≤ T/2) do
  while (s + t ≤ T) do
    If $ = (s + t + 2)/2
      create_node([s, t]) ;
    break ;
    s = s + t ;
  s = T/2 ;
  t = 2 * t ;
If (T = 2 * ($ - 1)) create_node([0, T]) ;

```

As stated earlier, all the active nodes are on a single path from the root of PVAS to an active leaf. Since we are creating active nodes from the leaf to the root, they can be linked into PVAS as they are being created. These simple details are omitted. Insertion of records into active nodes of PVAS is done as follows. We insert the record into the B+-tree using the `start` time as the ordering key. We find the (active) node in PVAS that must hold this record by descending down the chain of active nodes until we find the first node, whose span is smaller than the span of the record being inserted. The record must be stored in the parent of this node. If this record has the smallest `start` value among records stored at that node, we update the `ITREE` pointer at that node to point to the B+-tree where the record was inserted.

4.3 Passive Node Creation

A node $N_{[i,j]}$ becomes *passive* when $\$ > j$, since no further segments can be added to that node. At this time its records are moved from `ITREE` to `IFILE` and from `JLISTS` to `JFILE`.

The records of $N_{[i,j]}$ in `JLISTS` are appended in decreasing `end` time order to `JFILE`. The storage can then be reused by a node that is becoming active at that time. The records of $N_{[i,j]}$ in `ITREE` are deleted from `ITREE` and appended in increasing order of their `start` times to `IFILE`. Since the segments to be deleted will all be clustered in adjacent leaves of the B+-Tree, the deletion can be made very efficient. Finally, the pointers in $N_{[i,j]}$ must be updated to indicate the new locations of its segments.

When a number of nodes become passive at the same time, they must be handled in order of their level, with the node closest to the root being handled first and the leaf last. Note that no more than one node at any level is active at any time, and the nodes that become passive at the same time will form a

chain of right children. This will maintain the property used in Section 3 to ensure that all records of a subtree are clustered together in `IFILE` and `JFILE`.

5 Index Blocking

In this section we describe how to organize the binary tree in blocks. The binary tree will be stored as a compressed m -way tree. Assume that m is a power of two, and that the block size, B is less than $4m$. We will first show how to organize a fixed-size binary tree in block oriented storage, and later discuss its growth as the underlying binary tree expands.

Assume that the binary tree has depth $d = \log_2 N$, such that the leaves are at level 1 and the root at level d . For now assume that $d = \alpha \log_2 m$, for some integer $\alpha \geq 1$. Subtrees rooted at levels $\log_2 m, 2 \log_2 m, \dots, \alpha \log_2 m$ are collapsed into a single *supernode* of the m -way tree. Each supernode is assigned a block of storage. The block will store the m pointers to the m children of the supernode, and the information associated with the $m-1$ binary tree nodes that make up the supernode. Each binary tree node stores two pointers (to `IFILE/JFILE` and `ITREE/JLISTS`), and a fixed amount of information identifying the interval represented by that node. Within a block the nodes of the binary tree may be implemented as an array, and accessed by simple indexing. The depth of the m -way tree of supernodes is clearly $d/\log_2 m = \log_m N$, the path length of an m -way tree of N items.

The growth of the m -way tree as time progresses is easy to see. Storage is allocated in units of blocks. Every time a new leaf block is added to the m -way tree, an entry is made in the second-level block. When m leaf blocks have been added, and all entries in the second level block have been filled, a new second-level block is created and linked to the second entry of the root node. When all m blocks in the second level of the tree

are filled up, a new root block is added, and the process continues. At current time T , the total number of block access required to traverse a path from the root to the leaves is $\lceil \log_m T \rceil$.

6 Discussion

The multiversion access structure described has several attractive features. It is space efficient requiring space proportional to the number of records up to time t , independent of their actual distribution. In contrast [10, 8, 7] may require quadratic space in the worst case. The number of blocks accesses is at most a logarithmic number more than the size of the output. The index has a natural structure for archiving in write-once storage media like optical disks. As time increases, nodes age and become passive; once a node becomes passive it can be archived. No special techniques are needed to handle persistent objects which remain current for long periods of time. A node can be moved to archival storage any time after it becomes passive. The major archived data structures consists of sequential files which only need to be appended.

References

- [1] I. Ahn and R. Snodgrass. Partitioned storage for temporal databases. *Information Systems*, 13:369–391, 1988.
- [2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. On optimal multiversion access structures. In *Workshop on Advances in Spatial Databases*, pages 123–141, 1993.
- [3] J. Clifford and A. Tansel. On an algebra for historical relational databases: Two views. In *Proc. of the ACM SIGMOD Conference*, 1985.
- [4] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2), 1979.
- [5] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.
- [6] M. Easton. Key-sequence data sets on indelible storage. *IBM J. of Research and Development*, 30:230–241, 1986.
- [7] R. Elmasri, M. Jaseemuddin, and V. Kouramajian. Partitioning of time index for optical disks. In *Proc. of the IEEE Conf. on Data Engineering*, pages 574–583, 1992.
- [8] R. Elmasri, Y.-J. Kim, and G.T.J. Wu. Efficient implementation techniques for the time index. In *Proc. of the IEEE Conf. on Data Engineering*, pages 102–111, 1991.
- [9] R. Elmasri and G.T.J. Wu. A temporal model and language for er databases. In *Proc. of the IEEE Conf. on Data Engineering*, 1990.
- [10] R. Elmasri, G.T.J. Wu, and Y.-J. Kim. The time index: An access structure for temporal data. In *Proc. of the VLDB Conference*, pages 1–12, 1990.
- [11] H. Gunadhi and A. Segev. Efficient indexing methods for temporal relations. *IEEE Trans. on Knowledge and Data Engineering*, 5:496–509, 1993.
- [12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD Conference*, 1984.
- [13] C. Kolovson and M. Stonebraker. Indexing techniques for historical databases. In *Proc. of the IEEE Conf. on Data Engineering*, pages 127–137, 1989.

- [14] S. Lanka and E. Mays. Fully persistent B+ trees. In *Proc. of the ACM SIGMOD Conference*, pages 426–435, 1991.
- [15] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proc. of the ACM SIGMOD Conference*, pages 315–324, 1989.
- [16] D. Lomet and B. Salzberg. The performance of a multiversion access method. In *Proc. of the ACM SIGMOD Conference*, pages 353–363, 1990.
- [17] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill. Designing DBMS support for the temporal dimension. In *Proc. of the ACM SIGMOD Conference*, pages 115–130, 1984.
- [18] E. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Technical Report CSL-80-9, Xerox PARC, 1980.
- [19] E. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2), 1985.
- [20] S.B. Navathe and R. Ahmed. A temporal relational model and a query language. *Information Sciences*, 49(1,2, and 3), 1989.
- [21] N. Sarnak and R. Tarjan. Planar point location using persistent search trees. *Comm. ACM*, 29(7), 1989.
- [22] A. Segev and A. Shoshani. Logical modeling of temporal data. In *Proc. of the ACM SIGMOD Conference*, 1987.
- [23] R. Snodgrass. The temporal query language tqel. *ACM Trans. on Database Systems*, 12(2), 1987.
- [24] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 10(9), 1986.
- [25] M. Stonebraker. The design of the POSTGRES storage system. In *Proc. of the VLDB Conference*, pages 289–300, 1987.
- [26] V.J. Tsotras and B. Gopinath. Efficient algorithms for managing the history of evolving databases. In *Proc. of the Int'l Conf. on Database Theory*, 1990.

Item No.	Stock	Dept
315	150	A
325	350	A
560	50	B
⋮	⋮	

Time	Item No.	Transaction
5/1/93	315	Sold 75
5/1/93	560	Recd. 100
5/2/93	315	Sold 50
⋮	⋮	⋮

Time	Item No.	Stock	Dept
4/30/93	315	150	A
5/1/93	315	75	A
5/2/93	315	25	A
4/30/93	325	350	A
4/30/93	560	50	B
5/1/93	560	150	B

Figure 1: Example Snapshot and Temporal Database Relation

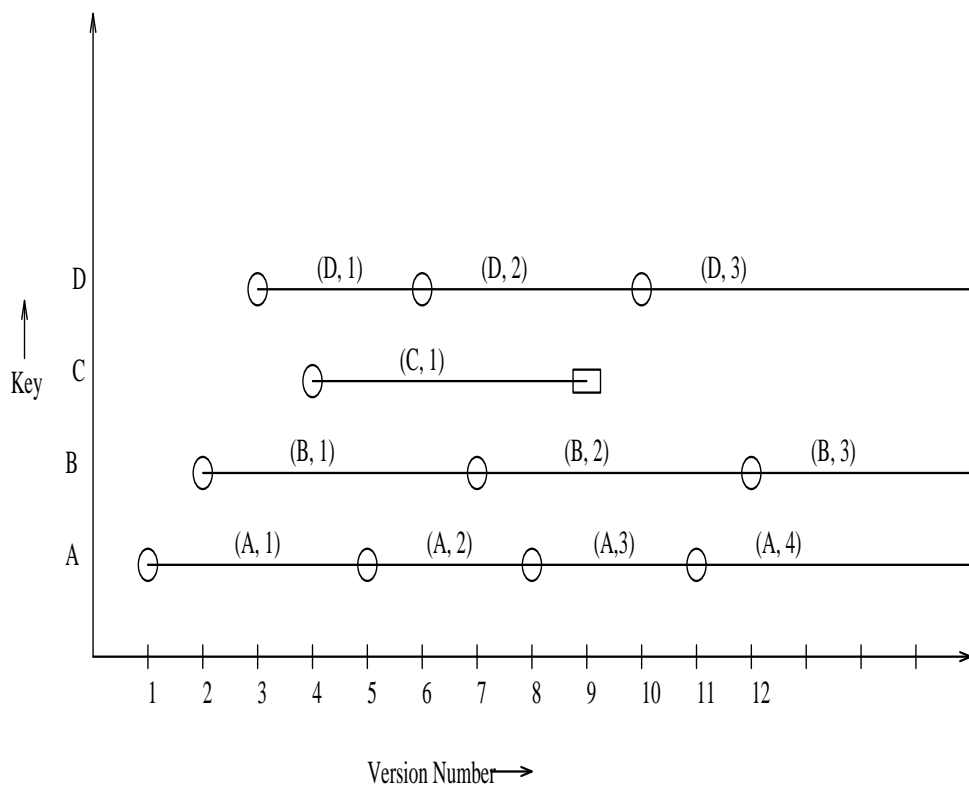


Figure 2: Figure Illustrating the Model

