VLDB Journal, 4, 703-726 (1995), Stanley Y.W. Su, Editor

Historical Queries Along Multiple Lines of Time Evolution

Gad M. Landau, Jeanette P. Schmidt, and Vassilis J. Tsotras

Received August 5, 1992; revised version received, May 27, 1994; accepted October 6, 1994.

Abstract. Traditional approaches to addressing historical queries assume a *single* line of time evolution; that is, a system (database, relation) evolves over time through a sequence of transactions. Each transaction always applies to the unique, current state of the system, resulting in a new current state. There are, however, complex applications where the system's state evolves into *multiple* lines of evolution. In general, this creates a tree (hierarchy) of evolution lines, where each tree node represents the time evolution of a particular subsystem. Multiple lines create novel historical queries, such as *vertical* or *horizontal* historical queries. The key characteristic of these problems is that portions of the history are shared; answering historical queries should not necessitate duplication of shared histories as this could increase the storage requirements dramatically. Both the vertical and horizontal historical queries have two parts: a "search" part, where the time of interest is located together with the appropriate subsystem, and a reconstruction part, where the subsystem's state is reconstructed for that time. This article focuses on the search part; several reconstruction methods, designed for single evolution lines can be applied once the appropriate time of interest is located. For both the vertical and the horizontal historical queries, we present algorithms that work without duplicating shared histories. Combinations of the vertical and horizontal queries are possible, and enable searching in both dimensions of the tree of evolutions.

Key Words. Rollback databases, CAD databases, access methods, data-structures.

1. Introduction

Conventional databases deal with one evolving logical state; the evolution from one consistent state to the next is achieved using transactions and, when a transaction

Gad M. Landau, Ph.D., is Associate Professor, *landau@pucs2.poly.edu*, Jeanette P. Schmidt, Ph.D., is Associate Professor, *jps@pucs4.poly.edu*, and Vassilis J. Tsotras, Ph.D., is Assistant Professor, *tsotras@aegean.poly.edu*, Department of Computer Science, Polytechnic University, Six Metrotech Center, Brooklyn, NY 11201.

commits, the previous state is discarded. There are many database applications, however, where it is important to capture the history of the database's evolution over time. A *rollback* or *transaction-time* database (Snodgrass and Ahn, 1986), for example, can "rollback" the database state to some past time of interest, thus providing the ability to make historical queries.

Previous approaches to historical queries assume a *single* line of time evolution. As an example, consider an evolving system such as a company. The company's state at time t is the set of employees that are working in the company at time t. This "real-world" system state evolves over time by the application of a set of operations; these operations represent the company's hiring policy, and include additions of new employees, deletions of existing ones, or modifications to attributes of existing employees (e.g., salary increases). The addition, deletion, or attribute modification of an employee is considered one *change* in the state of the company. Assume that a rollback database is used to capture this evolution. It is implicitly assumed that whenever a change occurs in the "real-world," a transaction will update the rollback database at the same time;[1] thus, this change will be timestamped with the commit time of the transaction. In the rest of this article, the (commit) timestamp of a transaction that updates the database concerning some change will be used interchangeably with the time that the change actually occurred. In addition, we assume that transactions are always applied to the most current database state. Using transaction timestamps, a rollback database can support historical queries about past database states, such as: "find all employees working in the company on January 1st, 1990."

For the purposes of this article, time is assumed to be discrete, described by a succession of nonnegative integers. A "log" of an evolution is defined to be the sequence of the evolution changes indexed by the time instant at which they occurred. The log contains the minimal information needed to reconstruct any past state of an evolving system.

There are complex applications, however, where the system's state evolves in a way that results in multiple lines of evolution. In our example, consider the case where, at some time instant, the company itself is split into a number of subsidiary companies. The employees working in the initial company are divided among the subsidiaries, and may even work in a number of them. The important issue is that, after the split, each subsidiary is evolving on its own (i.e., changes occur independently in each subsidiary). A subsidiary may later split into a number of new subsidiaries, and so on. Such an approach creates parallel, independent lines of time evolution that actually resemble a *tree of evolutions* in which each subsidiary has a number of ancestor subsidiaries.

There are two novel categories of historical queries that we address in this

1. In this article, we concentrate only on "transaction" time. An additional time axis ("valid" time) has been proposed in literature to represent reality more accurately (Snodgrass and Ahn, 1986).

article: the *Vertical Query* and the *Horizontal Query*. An example of a Vertical Query is: "find the employees that were working for a given subsidiary C or one of its ancestors, on January 1st, 1990." Suppose that C is a currently existing subsidiary, and consider the path in the tree of evolutions that leads to C; the above query finds which subsidiary in this path existed on January 1st, 1990, and reconstructs its state (i.e., its employees at that time). For example, if C did not yet exist at that time, but the ancestor A of C did, the answer to the query contains employees from A.

The Vertical Query is an extension of the single-line historical queries to the case of multiple lines of evolution. The state of the evolving system and the notion of a state change can be defined in various ways, depending on the application. In the example above, the state is defined as a collection of objects (employees) satisfying a predicate (work in the company). Instead one could ask for the (stored) value of a variable that changes over time, such as the number of full-time employees in a subsidiary; the state in this application is the value of the variable. Thus, another example of a Vertical Query is: "find the number of full-time employees for a given subsidiary C or one of its ancestors, on January 1st, 1990." The number of full-time employees may change many times inside a subsidiary and between subsidiaries. For the variable's evolution, the only change is a new variable value that is recorded, along with the time the change occurred. Again, the subsidiary in C's path that existed on January 1st, 1990 has to be located, but the answer to the vertical query is now a single value.

One could incorporate the changes in these two evolutions in a single data structure. However, for simplicity, we will consider the evolution of a general system, whose state is described either as a set of objects or as the value of a single variable. We have distinguished among these two ways of representing an evolving system because, in the case of a set evolution, after the time of interest has been located, some state reconstruction process is also required.

For vertical queries, it may well be that the subsidiary that existed at the time specified in the query currently does not exist; the recorded information about this subsidiary still might be useful to successor subsidiaries for different reasons. Some of the employees of past subsidiaries still may be employed in new subsidiaries created after splits and past information could be used for employee benefits, salary evolution statistics, retirement plans, tax purposes, etc.

The multiple-line evolution also creates a second category of historical queries, the Horizontal Queries, which did not apply to a single line of evolution. Consider the following example: given a subsidiary D that existed on January 1st, 1990, and an ancestor subsidiary B of D, "find all the employees which were employed on January 1st, 1990, by subsidiaries which were descendents of B." This query finds the state as of January 1st, 1990 for "relevant" subsidiaries (i.e., subsidiaries that were created under some common subsidiary company). It can be visualized as a horizontal "snapshot" of relevant subsidiaries (hence, the term horizontal query). As before, provided that the evolution of a variable indicating the number of employees of each

subsidiary has been kept, one could ask a simplified Horizontal Query, returning the number of employees of "relevant" subsidiaries as of some time of interest.

Both query categories addressed in this article are considered *basic* in the sense that, with their combination, we can search histories efficiently along both dimensions of multiple lines of evolutions. For example, first one could use the vertical query to access a past state of a given subsidiary, and then use the horizontal query to access concurrent (past) states of relevant subsidiaries. Conversely, one could start with the horizontal query at a current subsidiary, and apply some vertical query to each relevant subsidiary found.

In general, each query category has two distinct parts: a *search* part that locates the time of interest in a past subsidiary and a *reconstruction* part that reconstructs the answer. If the whole collection of employees is required, the reconstruction part involves running an algorithm inside the "log" of the subsidiary returned by the query to get the active employees at the time of interest. Standard approaches for the state reconstruction part can be used (e.g., Elmasri et al., 1990; Tsotras et al., 1995). If the query simply requests a past value of an evolving variable, the reconstruction part is simplified to accessing the appropriate entry in the variable's "log." This article concentrates on the search part and provides efficient algorithms for locating a time of interest, with respect to both queries. We return to the reconstruction part in Section 3.

Even though we refer to the "company" example throughout this article for its simplicity, the algorithms presented can be used in other applications. The basic problem of multiple lines of evolution appears in the context of editors for document databases (Xerox Corp.; Marshall, 1991). In general, in large software development projects, parts of programs are written by different programmers, and access to previous designs is needed to allow the designs to be modified in different ways for different purposes. Each part is timestamped and queries made are with respect to these timestamps. Another application appears in design environments (CAD systems).

The problems we address here differ from traditional versioning (see Katz, 1990, who defined the [similar] notion of a "version tree"). The main difference is that, in traditional versioning, each version is identified by a unique version identifier: to access a version, its identifier must be known. Thus, there is no explicit notion of time. In our setting, we do not use version numbers, rather we deal with "parallel" time, where a time instant could be applicable in a number of branches, hence a timestamp is not unique. This allows us to relate events that happened at the same time in different lines of evolution. For example, consider the design of a project that is split among different teams, each team working on a part of the project independently from the others with all changes being timestamped. As the evolution proceeds each sub-design can be split among more teams and so on, creating a *tree of evolutions* that needs to be accessed by *time* both vertically and horizontally (i.e., sub-designs are related vertically and horizontally by time).

The rest of this article is organized as follows: Section 2 presents previous

approaches to historical queries; such approaches could be used to organize the history of a particular subsidiary. The tree of evolutions leads to a basic underlying data-structure that is described in Section 3; in the same section we also discuss the reconstruction part of both query categories. The solution for the search part of the vertical query problem appears in Section 4; in Section 5 we provide the solution to the search part of the horizontal query problem; Section 6 contains conclusions and open problems.

2. Previous Work on Historical Queries

Any access method that addresses a history reconstruction problem is characterized by the following costs: the *space* used by the method's data structure to keep the historical data, the *update processing* to process the changes occurring in the evolution and update the method's data structure, and the *query time* to reconstruct the required past state. We regard the number of changes n as a good measure of the space requirements for a historical access method since it contains the minimal information needed to reconstruct any past state. Similarly, the update processing is measured as the processing per change. Another choice for measuring the performance could be the number of transactions. In practice, however, a transaction may result in a number of updates (changes), all of which share the same transaction commit timestamp. The query time is a function of the number of changes and the answer size.

Since the historical information to be stored keeps increasing as time proceeds, a good solution to a history management problem should use space efficiently. On the other hand, it should also facilitate fast reconstruction of a past state, so that using the stored history is efficient.

All previous approaches to answering historical queries address the problem on a single line of time evolution; changes are always applied on the current state of the evolving system. Various methods have been proposed in recent years (Ahn and Snodgrass, 1986; Kolovson and Stonebraker, 1989, 1991; Lomet and Salzberg, 1989, 1990, 1993; Segev and Gunadhi, 1989, 1993; Elmasri et al., 1990; Manolopoulos and Kapetanakis, 1990; Tsotras and Gopinath, 1990, 1992; Jensen et al., 1991; Becker et al., 1993; Kolovson, 1993; Leung and Muntz, 1993; Tsotras and Kangelaris, 1995). A detailed review of these methods appears in Salzberg and Tsotras (1994).

Most methods can be categorized according to the single-line historical query that they were designed to address more efficiently. Early methods (Lum et al., 1984; Ahn and Snodgrass, 1986) propose the use of "reverse chaining" (i.e., all past values of a given key are clustered together and linked in reverse time order). Such approaches can efficiently address queries of the form: "find the salary of employee A on January 1st, 1990." Pure-snapshot methods (Segev and Gunadhi, 1989, 1993; Elmasri et al., 1990; Tsotras and Gopinath, 1990, 1992; Tsotras et al., 1995) efficiently address snapshot queries of the form: "find the employees of the company at January 1st, 1990." In such methods, some form of a "log" usually keeps

the (timestamped) changes; this approach provides for constant update processing per change, as a new change is appended at the end of the log. Range-snapshot methods (Kolovson and Stonebraker, 1989, 1991; Lomet and Salzberg, 1989, 1990, 1993; Manolopoulos and Kapetanakis, 1990; Becker et al., 1993; Kolovson, 1993) efficiently answer queries of the form: "find the employees of the company on January 1st, 1990 whose names are in range [X, Y]." Since keys have to be kept in order, these methods use some form of a balanced-tree (*B*-trees, *R*-trees) as their main data structure, which leads to logarithmic update processing per change.

Efficient methods dealing with versioned data structures appeared in Driscoll et al., 1989, Lanka and Mays, 1991). A new version can be created from any previous version (i.e., full persistence) and gets a new, unique, version identifier. Driscoll et al. (1989) examined the problem of persistence of linked data structures (i.e., when the state of the evolving system is represented by a linked data structure), while Lanka and Mays (1991) (which is based on ideas from Driscoll et al.) provided efficient ways to store and access versioned *B*+-trees. The notion of a tree of evolutions used here is similar to the version tree defined in Driscoll et al. A version tree differs in that it has no explicit notion of time; rather every node in the tree uses a version number as a unique identifier. This does not allow users to relate parallel concurrent versions by giving them the same time stamp. Indeed, the structure of the version tree was not intended to model concurrent time evolution of a system into several subsystems.

3. Basic Data-Structure

In this section, we present the underlying data structure of our algorithms, and show how it relates to the problem of keeping multiple lines of history. We address this more general history problem under the constraint that the space used should remain linear at all times in the number of changes in the time evolution.

After a subsidiary is created and before it is first split, the changes occurring in its evolution are stored in an array; each entry of this array is in general of the form <*time, change*>, where *time* is the (commit) time of the transaction that updated this subsidiary about a *change.* Inside a given array, transaction times are stored in increasing order. We say that an array A *overlaps* time t on path p if $t \in [t_F, t'_F]$ where t_F denotes the first timestamp recorded in array A and t'_F denotes the first timestamp recorded in the array following A on path p, if any. Arrays are created at splits: after a split, a new array is created for each new subsidiary. Hence, the tree of evolutions is transformed into an equivalent *tree of arrays,* where each node of that tree is an array. In the simplest case, a subsidiary ceases to exist at the first split time, and its evolution is thus recorded in a single array. However, a parent subsidiary could continue its existence after a split (i.e., it "survives" the split); all new changes occurring to this subsidiary are then kept in a new array that is linked to its previous array. Each additional subsidiary created from the parent subsidiary gets a new array. The whole evolution of a given subsidiary is, in general, stored

in a list of arrays. This list of arrays implements the subsidiary's "log" of changes and, therefore, contains all the information needed to reconstruct any past state of the subsidiary; in this sense a subsidiary is called "historically autonomous."

Note that timestamps are increasing within the list of arrays of any subsidiary; this order is also preserved among the arrays on any given path in the tree of arrays. If a transaction results in a number of changes for a given subsidiary, we assume that each change is recorded in a separate entry of the subsidiary's current array, together with this transaction's timestamp. Thus, the number of entries can also serve as the basic performance measure on our algorithms.

Since each array stores (a part of) the evolution of a separate, independently evolving subsidiary, the same timestamp can be recorded in various arrays, written by different transactions that committed at the same time. A timestamp is therefore not unique in our structure.

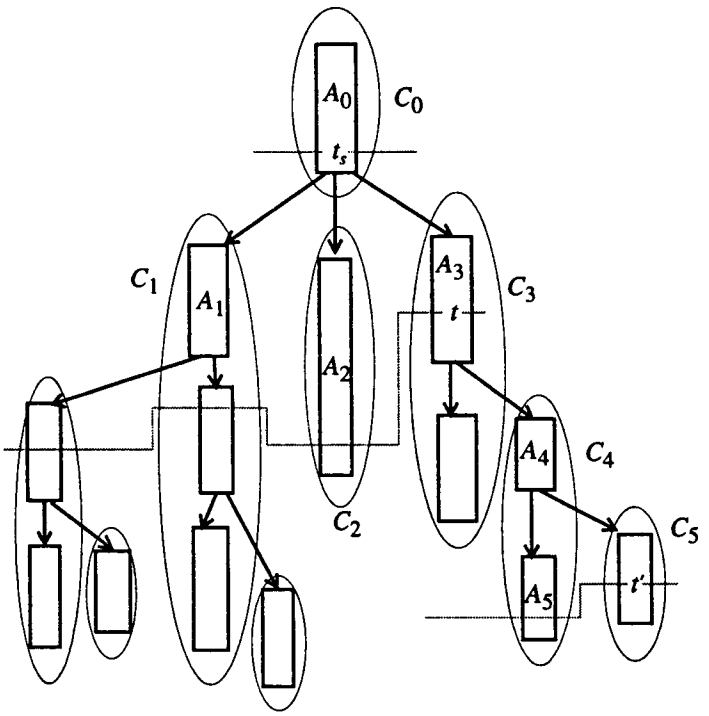
An example of a tree of arrays is shown in Figure 1: C_0 represents a company that was split at time t_s into three new subsidiaries, namely, C_1, C_2, and C3. The changes that occurred in the evolution of C_0 until time t_s are stored in array A_0, in increasing order of (transaction) timestamps. If the database keeps the history of employees, a typical entry in array A_0 is a tuple of the form: $<t_i,(op, emp)>$ that represents the fact that change (op, emp) occurred at the time instant t_i (i.e., operation *op* was applied on employee *emp* of company C_0). If the database stores the evolution of some variable, such as the number of employees, a typical entry would be of the form: $<t_i,val>$ representing the fact that at some time t_i the variable changed its value to *val* (for simplicity we assume that a subsidiary records its state and variable histories separately).

According to Figure 1, company C_0 ceases to exist after the split instance t_s, recording its evolution in a single array. Company C_4, however, "survived" a split; thus, its evolution has been recorded in arrays A_4 and A_5. In the same figure, the list of arrays of a given subsidiary is shown surrounded schematically by a cluster. A path in the tree of arrays will normally contain nodes from various subsidiaries. At a macro level, the tree of evolutions can be extracted from the tree of arrays if each cluster of nodes is viewed as a "macro node" for the tree of evolutions.

Formally, a tree of arrays T is a tree in which each node v_i contains an array A_i with u_i entries. Let m denote the number of nodes in the tree of arrays (i=1,...,m), let u be the maximum number of entries in any array ($u_i \leq u$), and let $s(p)$ denote the number of nodes on path p. For simplicity of notation, the argument p will be omitted if it is clear from the context. No assumption is made on the maximum outdegree of a node. In addition, let n denote the total number of entries (changes) in the tree; obviously n is bounded by mu while $s(p)$ is at most m.

A split on a given subsidiary is described by: (1) the time t_s at which it occurred, (2) the names of the new subsidiaries, and (3) the distribution of the employees existing at t_s to the parent subsidiary among the newly created subsidiaries (and maybe itself, if this subsidiary "survives" the split). It remains to describe how the information in (3) is represented in accordance with our measure, the number of

Figure 1. Example of tree of arrays



Rectangles represent arrays, while clusters represent the list of arrays of a given subsidiary.

changes. Suppose that a subsidiary C with x employees is split into two subsidiaries with x_1 and x_2 employees, where $x_1 < x_2$. The distribution of the employees among the two subsidiaries is equivalent to $O(x_1)$ changes if $x_1 + x_2 = x$ ("seen" by the smaller subsidiary as a batch addition of the x_1 employees that constitute its initial state, and by the larger subsidiary as a batch deletion of the same employees). Usually, $x_1 + x_2 = x$, however, at split time the same employee could also end up in more than one subsidiary, in which case $x_1 + x_2 > x$, or some employee could not appear in either, resulting in $x_1 + x_2 < x$. This latter case requires $max(x_1, x - x_2)$ changes. Since each subsidiary evolves independently from the others, this employee may later be deleted from some subsidiaries, while still working in others.

One of the basic queries that our data structure supports is the vertical query: "find the employees that were working for a given subsidiary C or one of its ancestors, on January 1st, 1990." Let $C_0, ..., C_i, ..., C_{s-1} = C$ be the subsidiaries in the path leading to subsidiary C. To answer the above query, we must find the first subsidiary C_i on the path to the root which was in existence on "January 1st, 1990." We

identify this subsidiary C_i by locating the (unique) array A^* (if any) on the path to the root that overlaps "January 1st, 1990." If the "log" of C_i is implemented as a list of subsequent arrays $A_{i1},..., A_{il}$, then $A^* \in \{A_{i1},..., A_{il}\}$. The "log" of C_i contains the initial state of C_i and all the changes in this subsidiary's evolution. Obviously, if "January 1st, 1990" is overlapped by array A_{ij} $(1 \leq j \leq l)$, then entries with timestamps higher than "January 1st, 1990" are of no interest to the query; therefore, within array A_{ij} one must find the entry with the largest timestamp that is less than or equal to "January 1st, 1990." We call that entry the "starting point" of the vertical query. We present an algorithm that finds the overlapping array in time $O(\log s)$. An additional binary search inside this array will find the starting point in $O(\log u)$ time.

The second type of basic query is the horizontal query: given a subsidiary D which existed on January 1st, 1990, and an ancestor subsidiary B of D, "find all the employees which were employed on January 1st, 1990, by subsidiaries which were descendents of B." We first need to locate the time "January 1st, 1990" in the "logs" of all the relevant subsidiaries (i.e., those subsidiaries that overlap "January 1st, 1990" and are descendents of B). These starting points create the *concurrency line* of subsidiary D on "January 1st, 1990" with respect to B. Figure 1 shows the concurrency line of time t in C_3, with respect to the ancestor C_0 of C_3. All the arrays that are "cut" by this line are descendants of C_0, as required. The starting points of a concurrency line t provide the latest transactions recorded at all relevant subsidiaries on or before time t. In general, such a line need not "cut" all relevant subsidiaries; for example, the concurrency line of time t' in subsidiary C_5, with respect to ancestor subsidiary C_3, does not go through any array of C_3's "log" since subsidiary C_3 ceased to exist before t'. Our horizontal query algorithm avoids checking such logically non-existing relevant subsidiaries. If k is the number of relevant subsidiaries existing at some time t, we present an algorithm that locates the starting points in all these relevant subsidiaries in time $O(k)$.

We now turn to the "reconstruction part" of a multiple-evolution historical query. The "search part" of the query will return some entry in the "log" of a subsidiary that represents the largest transaction (commit) time recorded in this subsidiary that is less than or equal to the time specified by the query.

In the simplest case, where the historical query asks for the value of a predefined variable, such as the number of full-time employees at a time of interest, the "reconstruction part" simply reads the value recorded in the entry provided by the "search part."

If the historical query requests the reconstruction of a subsidiary's set of employees at a given past time, various state reconstruction algorithms may be used after the "starting point" has been found in some subsidiary C_i. A simple but obviously inefficient algorithm could just follow upwards all the changes in the path from A_{ij} until A_{i1} (as any subsidiary is historically autonomous). Another algorithm could use the Snapshot Index (Tsotras and Kangelaris, 1995), which has an optimal reconstruction time; indeed, if a employees satisfy the query predicate, it will find

them in only $2a$ steps. This approach uses more complex data structures in addition to the "log" of changes, but the total space is still linear in the number of changes in subsidiary C_i, and the update processing is constant per change. Hence, the combination of the Snapshot Index and the search algorithms presented in this article can solve the Vertical Query in time $O(\log n + a)$ and the Horizontal Query in time $O(k + a)$, where a denotes the answer size (number of employees satisfying the predicate) of each query. The whole space remains linear in the number of changes. The update processing is amortized constant per change, that is, the total time to perform an arbitrary sequence of n changes is $O(n)$; the time for a single change, however, might not be constant. Other approaches, such as a modification of the method of Elmasri et al. (1990) are also possible. By concentrating on the problem of finding the "starting points" for the vertical and horizontal queries, our results can be used in a number of reconstruction strategies. Obviously, the time for the reconstruction must be accounted appropriately, depending on the scheme used.

From the above discussion, it should be clear that it is not efficient simply to use an existing single-line of evolution temporal access method (e.g., the Snapshot Index; the Time Index, Elmasri et al., 1990; the SR-tree, Kolovson and Stonebraker, 1991; and the ST-Index, Segev and Gunadhi, 1993) for multi-line evolution temporal queries. Consider first the Vertical Query. Conceptually, the changes of a single subsidiary could have been indexed by some form of a traditional temporal index. However, in a multi-line evolution environment, parts of history are shared. If a separate access method was used on each path of the tree of evolutions, a large space and update overhead would result. The Horizontal Query is novel; traditional temporal access methods do not address this problem. Here, we propose data structures that support both multi-line queries using linear space and constant updating per change (in the amortized sense). In addition, our approach provides logarithmic query time. Thus, histories over multiple lines of evolution are supported with performance that is similar to the single-line evolution history.

4. The Vertical Search Problem

In this section, we present an efficient algorithm, *Algorithm VS,* for finding the starting point of a Vertical Query, which we call the Vertical Search problem. To facilitate the search, a number of pointers are installed in the tree of arrays from a node (array) to some of its ancestor nodes, skipping a varying number of ancestors at a time. These pointers will enable us to mimic a binary search on the path to the root during the processing of the query; the pointers are installed as the tree grows in a way to make them readily accessible at the query time. Roughly, pointers point to ancestors at distances that form a geometrical progression (powers of 2) until the root is reached. We will show that our scheme leads to efficient performance by using $O(m)$ additional space for the pointer structure.

The parameters of the vertical query are: (1) a tree of arrays T, (2) a node v in T, and (3) a time t. Node v corresponds to the array of the subsidiary where the search is initiated (subsidiary C in the example of the vertical query). The vertical search problem consists of finding the array on the path of v that overlaps t and inside this array the LAST entry e whose timestamp E is less than or equal to t. Timestamp E corresponds to the commit time of the latest transaction recorded before or at t.

Algorithm VS has two stages. In the first stage, the array w is identified on the path from v to the root that overlaps t. In the second stage, the entry e is found by performing a binary search on array w. The second stage is straightforward. We elaborate below on the first stage, which requires adding additional information to the tree of arrays.

A *modified binary search* will be performed on a given branch of the tree. Each array in the tree has a *representative* corresponding to the first timestamp recorded in this array. Since the next array on the path was created by a split (i.e., by a later transaction, the representatives in any given path of the tree are strictly increasing). Thus, a path with s arrays can be viewed as a (backward) linked list $L = l_1,..., l_s$ of elements sorted in increasing order where a list element corresponds to a node's representative (l_1 is the representative of the root and so on; l_s is the representative of node v).

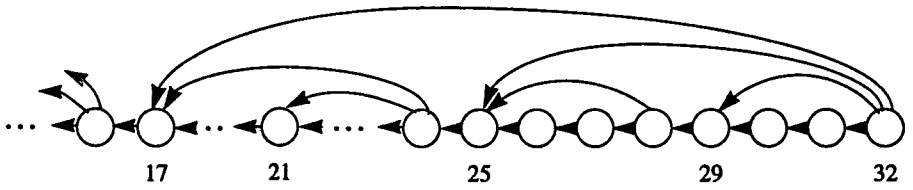
We wish to maintain a data structure that will enable us to answer queries of the form "find the largest element l_i in L for which $l_i \leq t$" in logarithmic time in the length of the list. Our data structure supports the addition of new elements at the end of the list, but no element may be inserted in the middle. This requirement emanates from the time evolution: new tree nodes would be added only at the end of a path (after some split) but no node can be inserted in the middle of a path, as history is unchangeable. Hence, we may assume that each element l_i knows its index i in the list, which does not change.

We first indicate how to perform a modified binary search on a *static* linked list with s elements, which was preprocessed to contain s additional pointers (Algorithm 1). (The added pointers are similar to those used by van Emde Boas et al., 1977.)

There are two differences between the problem solved by our preliminary Algorithm 1 and the vertical search problem: (1) in a tree, each node is typically contained in many paths; thus, its representative is contained in many linked lists, and (2) leaves are added to the trees, and the data structure has to be updated on-line, while maintaining linear space.

The first difference does not cause any difficulty, since a node has the same representative and the same index in all the lists. We show that our data structure can be updated on-line in $O(1)$ time per added element, without compromising the space restrictions and while still allowing the answer to queries in logarithmic time in the length of the path.

Figure 2. Example of pointers added in a static list with $s = 32$ elements.



Numbers indicate list indexes.

4.1 Algorithm 1: Search on Static Linked List With Exactly s Elements

Algorithm 1 has two steps. Suppose that the binary representation of s (the index of the representative l_s of node v) has 1's in positions $b_1,..., b_k$, that is, $s = \sum_{j=1}^{k} 2^{b_j}$. In the first step to locate l_i, the sublist $\overline{L} = l_{j_1} ... l_{j_2}$ of length 2^{b_r} ($b_r \in \{b_1,..., b_k\}$) is identified for which $\sum_{j=1}^{r-1} 2^{b_j} + 1 \leq i \leq \sum_{j=1}^{r} 2^{b_j}$. In the second step, the desired element, l_i is found by a binary search on this sublist. To answer all queries efficiently, $O(s)$ pointers are added in a preprocessing step to a list of length s.

Algorithm 1: Preprocessing—Computing O(s) Pointers

Let $\lambda = \lfloor \log s \rfloor$. An element l_j, for which $j = q2^d$ (for some integers q, $1 < q$ and d, $1 \leq d < \lambda$) has a pointer to the element l_h, $h = (q - 1)2^d + 1$. In addition, an element l_j, for which j is odd, has its original list pointer to l_{j-1}. Note that: (1) for $j = q2^1$ (i.e., even for j), the original list pointer to l_{j-1}, satisfies $j - 1 = (q - 1)2^1 + 1$, and (2) every element has at least one pointer (its original) and at most λ pointers. Since there are $2^{\lambda-d}$ elements whose list index is a multiple of 2^d, the list contains a total of: $S + \sum_{d=2}^{\lambda-1} 2^{\lambda-d} < 3s/2$ pointers (Figure 2).

The computation of the pointers is done in less than λ iterations. In the first iteration, all elements point to their previous element in the list. In the dth ($d \geq 2$) iteration, all elements whose index in the list is a non-trivial multiple of 2^d compute (conceptually in parallel) the pointer to the element which is $2^d - 1$ elements away. Consider any element l_j.

1. Compute the maximum $\overline{d}$ such that $j = q2^{\overline{d}}$ for some integer $q > 1$.
2. Add the following $\overline{d}$ pointers,

for $d = 2$ to $\bar{d}$ do: add a pointer from l_j to l_{j-2^d+1}.

If j is a multiple of 2^d, then both j and $j - 2^{d-1}$ are multiples of 2^{d-1}. In the (d-1)st iteration, pointers were added from l_j to l_{j-2^d+1} and from $l_{j-2^{d-1}}$ to l_{j-2^d+1}; therefore, l_{j-2^d+1} is found in three pointer accesses (via $l_{j-2^{d-1}+1}$ to $l_{j-2^{d-1}}$ to l_{j-2^d+1}).

Algorithm 1, Step 1

Recall that the index of the representative l_s of v, satisfies $s = \sum_{j=1}^{k} 2^{b_j}$. The sublist $\bar{L} = l_{j_1} \ldots l_{j_2}$ of length 2^{b_r}, which contains the desired element and for which $j_1 = (q-1)2^{b_r} + 1$ and $j_2 = q2^{b_r}$ (for some b_r) is identified. Formally, the list is identified by the following procedure:

$j_2 := s$
$j_1 := 1$
While $j_2 - j_1 \neq 2^d - 1$, for some integer d do
  Let l_{j_3} be the endpoint of the "longest" pointer from l_{j_2}
  {Hence $j_2 = q2^{\bar{d}}$ for some odd $q > 1$, (since $\bar{d}$ is maximal), and $j_2 - j_3 = 2^{\bar{d}} - 1$.}
  if $t < l_{j_3}$ then $l_{j_2} = l_{j_3-1}$
  else $\{t \geq l_{j_3}\}$ $l_{j_1} = l_{j_3}$
od
$\bar{L} = l_{j_1} \ldots l_{j_2}$

It is easy to verify that step 1 guarantees that: (1) $l_{j_1} \leq t$, and $t < l_{j_2+1}$, and (2) $\bar{L}$ contains 2^{b_r} elements and l_{j_2} is the $q2^{b_r}$th element in the original list L, (for some $q \geq 1$). To see that Step 1 takes at most $\log s$ iterations, we observe that the values $\bar{d}$ computed in "while loop" form an increasing sequence of integers, (and clearly $\bar{d} < \lfloor \log s \rfloor$). Indeed, it is not hard to verify that the successive values of $\bar{d}$ correspond to the successive values $b_1, b_2, \ldots$, in the binary representation of s. When the sought for value b_r is identified, the "else" statement is executed and the loop terminates.

Algorithm 1, Step 2

The sublist identified in Step 1 has the property that all pointers to all (recursive) midpoints of sublists are available in $O(1)$ accesses. Hence, we essentially mimic the basic recursive step of a binary search on a random access data structure (cf. array), which is to return the first element that is less than or equal to the value sought.

Basic Binary Search: Given two indexes j_1 and j_2, and a key $t \geq l_{j_1}$, if $j_1 = j_2$ then return j_1, else let $j_3 = \lceil (j_1 + j_2)/2 \rceil$. If $l_{j_3} \leq t$ perform the basic recursive step on the pair (j_3, j_2), otherwise, $l_{j_3} > t$ perform the basic recursive step on the pair $(j_1, j_3 - 1)$.

On an array with 2^λ elements, our basic binary search satisfies the invariant: $j_2 - j_3 = 2^d - 1$, for some integer d, and $j_2 = q2^d$ for some integer $q > 1$. Our linked list has a pointer from each such l_{j_2} to the corresponding l_{j_3}, which will provide the required access.

Complexity of Algorithm 1

`Space Complexity`: For a list with s elements (nodes), we add $O(s)$ pointers for a total of $O(s)$ space.

`Time for computing the pointers`: Each pointer computation requires $O(1)$ time for a total of $O(s)$ operations for preprocessing.

`Time Complexity`: Both step 1 and step 2 run in $O(\log s)$ time.

We now return to the description of the two stages of Algorithm VS.

4.2 Algorithm VS—Stage One

Since each path in the tree T is viewed as a linked list, every node in T can be part of many linked lists but has the same list index in all of them (as all indexes start from the root of the tree). Algorithm VS uses Algorithm 1. As described in Algorithm 1, pointers are added in a static list to allow for efficient search. There are two difficulties that have to be addressed when going from the static list to the tree T.

1. New leaves can be added to the tree.
2. The pointers as defined in Algorithm 1 may result in $O(m \log m)$ space in the tree (recall that m is the number of nodes in T). Consider, for example, the following scenario: the tree is essentially one long branch with $m/2$ nodes, where the last node has $m/2$ children. Each of these children may require $O(\log m)$ pointers, which would result in a structure with m nodes and at least $O(m \log m)$ pointers.

To solve these problems, we are obliged to sometimes delay adding pointers to newly added leaves. Each node will keep a special pointer, denoted by I-pointer, to the nearest node on its path to the root, whose pointer structure is not complete. If there is more than one incomplete node on the path, this node will have an I-pointer to the next node with incomplete pointer structure, resulting in a linked list of I-pointers. When a new leaf is added to the tree, it will contribute up to 2 pointers to the pointer structure. All nodes with indexes j, which are not divisible by 4, have only one (list) pointer. Those nodes access their I-pointer and complete the pointer structure of a previous node on their path. If the pointer structure of the accessed node is completed the node is deleted from the I-pointer list and the new leaf is pointing to the next incomplete node (if any). Note that the pointer structure of nodes is completed in a last-come first-served fashion. The following claim shows that the pointer structures of all nodes are completed in a timely fashion, nevertheless.

Claim: At any time, the maximum number of nodes with incomplete pointer structure on a given path of length s is at most $2\log s$.

Proof: Consider a path L in the tree T. Suppose that v, with index $q2^i$, for some odd q, is the node with least index on L, which has an incomplete pointer structure. Consider the node w with index $(q+1)2^i$ on L. Let V be the list of nodes between v and w (V:<$v = v_0, v_1, ..., v_{2^i} = w$>). All the pointers of nodes in $v_1, v_2, ..., v_{2^i-1}$ are directed towards nodes in V (i.e., they don't go beyond $v = v_0$). Hence, for $1 \leq \mu < 2^i$, the total number of pointers that will have to be added to nodes $v_1, v_2, ..., v_\mu$ is $\mu + \lfloor \frac{\mu}{4} \rfloor + \lfloor \frac{\mu}{8} \rfloor + ... < (3/2)\,\mu$. The number of pointers that have to be added to v is at most i. If every node computes 2 pointers, after μ nodes have been added to the path, 2μ pointers can be computed. The pointer structures of v therefore will be complete by the time $(1/2)\,\mu \geq i$ and, hence, when μ reaches $2i$. It also clearly follows from our procedure that when the pointer structure of v is completed, no other node on this path has an incomplete pointer structure. Since $2i < 2^i$, for $i \geq 1$, the pointer structures, in any case, will be complete before node w is added. Since i is bounded by $\log s$ we conclude that: (1) the nodes with incomplete pointer structures are among the last $2\log s$ nodes on the path, and (2) the total number of "missing" pointers, at any given time on a path of length s, is also bounded by $O(\log s)$. □

We note that, since linked lists overlap, an I-pointer of a node may point to a node whose pointer structure has been completed in the meantime. In this case, the I-pointer is simply updated to point to the next node. In any case, this can only speed up the computation.

Since the number of nodes with incomplete pointer structure on any given path of length s is at most $O(\log s)$, we may search all the nodes with incomplete pointer structure sequentially. If the desired node is not among those nodes we proceed as in Algorithm 1.

Complexity of Algorithm VS Stage One

`Space Complexity`: Since each node computes at most 2 pointers, the number of pointers added in the entire tree is $O(m)$.

`Update Processing`: Each pointer is computed in $O(1)$ time; thus, the update processing is constant per new node added to the tree (and, therefore, is constant per change).

`Time complexity`: For a path with s nodes, the time complexity of the algorithm is $O(\log s)$, since both steps 1 and 2 of run in $O(\log s)$ time.

Recall that Algorithm VS consists of 2 stages. The first stage, described above, identifies an array A that overlaps t. In the second stage, the entry e whose timestamp E is less than or equal to t is found by a binary search of array A; for an array

with u entries this takes $O(\log u)$ time. The size of a tree path is bounded by the number of tree nodes m which, in turn, is bounded by the total number of changes n. Similarly, u is bounded by n. Hence, algorithm VS uses linear space in the number of changes, constant update processing per change, and finds the starting point for the vertical query in $O(\log n)$ time.

In the case of a single-line evolution, all n changes would be stored in timestamp order on a single array structure, resulting in $O(n)$ space and constant update processing per change (by appending each change at the end of the array). Under these constraints, the starting point of a vertical search is found in $O(\log n)$ time (with a binary search on an ordered array) and this is asymptotically optimal (in a comparison based model). The VS algorithm provides the *same* optimal performance in the more complicated case of the multiple-lines of evolution.

Implementation Concerns: The actual complexity of the vertical search obviously depends on the length of the path we are searching. In a tree with m nodes a typical path will usually be of size $O(\log m)$, with some paths being much longer. If we were to divide T into pages in a straightforward manner with each page of size B, the number of page faults incurred when searching along a path of length s would be about $s/\log B$, (assuming that a page holds a subtree of height $\log B$). One could argue that a single, very long path of length $s = O(m)$ in the best case could be paginated using indexing to reduce the number of page faults to about $\log m/\log B$, however, it is not clear how to do that when several such paths have considerable overlap. On the other hand, a typical path of length $\log m$, in a balanced part of the tree, would again incur about $\log m/\log B$ page faults using straightforward (subtree) pagination. Therefore, we compare the number of page faults incurred by our algorithm to $\log m/\log B$, even though it is not clear that this bound actually can be achieved using simple paging schemes.

Our algorithm does not offer much improvement for searching a path of size $O(m)$. It is likely, though, that many paths will have $O(\log m)$ nodes. In this case, our algorithm accesses $O(\text{loglog}\, m)$ nodes so that, even if each of these nodes were on a different page, this would still result in considerably fewer page faults, than $\log m/\log B$.

5. The Horizontal Search Problem

This section presents an efficient algorithm (*Algorithm HS*) for the search part of the Horizontal Query, which we call the Horizontal Search problem. In the context of the horizontal search, we also use a pointer structure, but pointers are now between entries of arrays (and not between arrays as in the vertical query). Starting from a given entry with timestamp t, these pointers will enable the fast location of t (or the largest $t_0 \leq t$) in other horizontally "related" arrays. The space required by these additional pointers is still linear in the number of changes. We also show that the update processing cost per change remains $O(1)$ in the *amortized* sense. This

means that the addition of some entries (changes) could require more than $O(1)$ update processing to update the horizontal pointer structure but, for any sequence of n changes, no more than a total of $O(n)$ time will be required. Our approach uses the creation of "ghost" entries and a threshold-based "backward updating" technique.

The Timestamp Properties. Before we proceed, we establish the following properties of timestamps. Let v be an arbitrary node in the tree of arrays, and denote the children of v by $v_1,...,v_k$. (1) All of the first entries of the arrays of $v_1,...,v_k$ have the *same* timestamp (as they are created by the same split operation). (2) We assume one general clock that provides timestamps. Hence, given two time stamps t_1,t_2 such that $t_1 < t_2$, a node with time stamp t_2 was added after a node with time stamp t_1.

The input to algorithm *HS* consists of (1) a tree of arrays T with the timestamp properties, (2) an arbitrary entry e with timestamp t (within some node v of the tree T), and (3) an ancestor node w of v. The horizontal search consists of finding all entries e_j that belong to nodes in the subtree of w, and whose timestamp E_j is the last one (in their subtree) for which $E_j \leq t$. The horizontal search problem produces as an answer the *concurrency* line of e_j with respect to w. In the example of the Horizontal historical query, node v corresponds to some array of subsidiary D, while node w is the first array of the ancestor subsidiary B.

There is one obvious solution to the horizontal search problem. One could start from the ancestor node and travel in a depth-first search manner through its subtree, in an attempt to locate the required entry in each relevant branch. Suppose it was determined that the required entries are located in R arrays: $A_{i_1},...,A_{i_R}$ (within nodes $v_{i_1},...,v_{i_R}$). Locating the correct entries in these arrays requires $\sum_{j=1}^{R} \log u_{i_j}$, which could be as much as $O(R \log u)$.

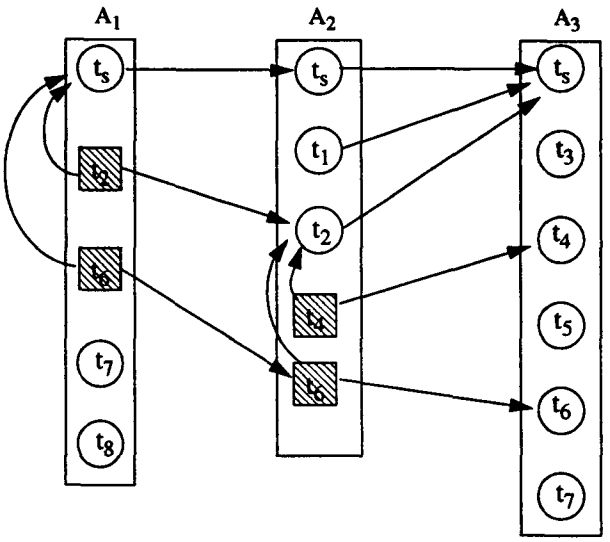
5.1 Algorithm HS

We proceed by first describing the data structure used by the Historical Search algorithm. We then introduce the use of "ghost" entries and the "backward updating" technique, and discuss their effect in the algorithm's performance.

The Basic Data-Structure. To speed up the search, we maintain two horizontal linked lists, a right list (to be traversed from right to left), and a left list (to be traversed from left to right), which connect related entries. Since the maintenance and construction of the two lists is completely symmetric, we only describe the right horizontal lists. The main problem in maintaining horizontal lists that enable us to answer our query is demonstrated by the following example.

Imagine a tree whose root has three children v_1, v_2, v_3, (in left to right order) with arrays A_1, A_2, and A_3, respectively. A_1 and A_3 are very active, and a new

Figure 3. Example of backward updating technique.



Arrays A_1, A_2, and A_3 form the list after split time t_s and before A_1 is split. Circles represent real entries, and rectangles represent artificial ones (ghosts). For this example $\delta = 2$.

entry is added to each of them at every timestamp. Array A_2 is very inactive and entries are rarely added to it. A horizontal query to an entry with timestamp t in A_1 should return the entry with the largest timestamp $t_0 \leq t$ in A_3. Many entries in A_1 would return THE SAME entry t_0 in A_2. On the other hand, to avoid an explosion of space it is desirable to have entries in A_1 point only to entries in A_2.

To overcome the above difficulty, we use a variation of "backward updating" (Cole, 1986), which adds "ghost" entries to neighboring nodes to make up for such discrepancies.

Summary of Backward Updating. For every δ entry added to an array ($\delta \geq 2$), an artificial entry is added to its left neighboring array, (with current timestamp). Any newly added entry (real or artificial) sets its horizontal (right) pointer to point to the last entry of its right neighbor (Figure 3). Any artificial entry also keeps an (upward) pointer to the currently last real entry within its array. (If the timestamp of an artificial entry coincides with the one of a real entry recorded at the same array, no artificial entry is added.)

Space Requirement of Horizontal Lists. Although backward updating obviously can result in a cascade of updates (in the worst case a single update may cause as many updates as the current size of the horizontal list), this happens very infrequently. More precisely, for every δ entry one ghost entry is added to the left neighbor, and every δ^2 entry will add a ghost to a neighbor of the neighbor. Basically every entry is responsible for adding $1/\delta^k$ ghost entries to its kth neighbor to the left, at a total average cost of $1/\delta + 1/\delta^2 + \ldots = 1/(\delta - 1)$ per entry. Thus, the total number of ghosts added is $\frac{n}{\delta-1}$. The additional space required by ghost entries and by the (two) pointers per entry is linear in the number of real entries (changes).

Cost of Maintaining Horizontal Lists. Each added real entry creates a number of ghosts. The total number of ghost entries added during the first n entries is $O(n)$.

Given the horizontal lists described above, an entry e_j with timestamp t, within node v and an ancestor w of v, we wish to answer the horizontal search for e_j with respect to w. We start traversing the horizontal pointer, which emanates from e_j and leads to (say) e_1. Entry e_i is the last entry added to this array prior to time t. If e_1 is an artificial entry, the last real entry preceding it is reported, otherwise e_1 itself is the desired entry. In general, the time stamp of e_1 will be smaller than that of e_j, so caution must be exerted when proceeding. The horizontal search from e_1 leads to entry e_2' within node v_2. Up to δ entries may have been added to v_2 between the time e_1 was added and our actual target timestamp E. Therefore, we examine up to δ entries below e_2', to find e_2, the last one whose timestamp is below E. We then proceed in exactly the same manner as before.

The only remaining problem is to identify where the horizontal traversal is to stop (i.e., the last node in the subtree of w). There are many solutions to this problem. One solution is given by the following labeling scheme.

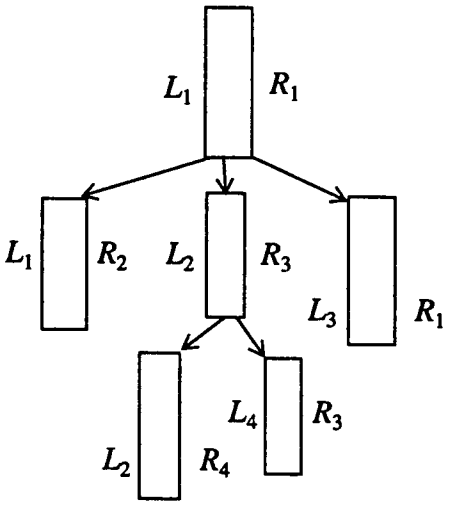
Labeling Scheme. Each node (array) in the tree has a right and a left label. When children nodes are added to a node, the rightmost child inherits the right label of the parent and the leftmost child inherits the left label of the parent. All other labels are new and are taken from a pool of unused labels (Figure 4). The key property of this scheme is that the right labels along any extreme right path (or left labels along any extreme left path) are identical. The traversal of a horizontal list (from right to left) is to stop at the entry, whose node (array) has a left label that is the same as w's left label.

Complexity of Algorithm HS

`Space Complexity`: Since the only extra structures are the horizontal lists, the total space used remains $O(n)$.

`Update Processing`: The extra cost for maintaining the horizontal lists is amortized $O(1)$ per change. The labeling scheme is $O(1)$ per new node in the tree (and, hence, $O(1)$ per change).

Figure 4. Example of labeling scheme in tree of arrays.



`Time Complexity`: Each step in the traversal has a cost of $\delta + O(1)$. If the traversal outputs k relevant entries the complexity of answering a horizontal search query is $O(k(\delta + O(1)))$. Note the trade-off in the choice of δ. The space requirement decreases with δ, while the horizontal search time increases with δ.

5.2 Generalization of Algorithm HS

We now address a generalization of the above scheme, in which we allow nodes to logically die at a time that is different than a split time (consider the case where, in our company example, a subsidiary can close without spliting into new subsidiaries). If a node (array) dies at time t, the last entry in it is labeled "dead" and has timestamp t. In this case, we would like to skip over that node in a horizontal query with time target greater than t.

Maintaining adequate horizontal lists when nodes can die does not cause any difficulty. When a node v, with last entry e, dies at time t, an artificial entry e' (with same timestamp t) is created at its right neighbor v_1. The horizontal left pointer of e' simply skips v and points to the last entry of its left neighbor v_2, which is henceforth considered the left neighbor of v_1. The only difficulty that arises is to determine the rightmost live descendent of a node w and, hence, the node at which a horizontal traversal with respect to w is to stop. This problem is the generic descendent query: Given two nodes w and v in the tree determine whether v is a descendent of w. The obvious solution: use a vertical search algorithm to answer the question in $O(\log s)$ time, where node v is in a path of length s. The descendent query takes $O(\log m)$ since, in the worst case, a path is $O(m)$.

Given an entry e in a descendent v of w, to determine the last node on the horizontal list of e, which is a descendent of w, we must perform a vertical search for each node accessed in the horizontal list of e. Thus, the obvious solution results in $O(Qk)$, where k is the number of nodes on the horizontal list, which are live descendants of w, and Q is the time to answer the generic descendent query ($Q \leq O(\log m)$).

The problem can be solved, however, in time $O(Q \log\log m + k)$ as following: Instead of performing a vertical search at each step of the horizontal traversal, we will do so every $\log m$-th step, and stop at the first node that is outside our scope. The cost so far is at most $O(k + \log m)$. We are left with $\log m$ nodes and need to determine the first one that is outside the scope. We may place these nodes in a temporary array, and determine the target by doing a binary search on these nodes. The required time is $O(Q \log\log m)$ as predicted.

Since $Q \leq O(\log m)$, we have shown that, even in the case where nodes die, we can answer horizontal queries quickly. The obvious question is whether the time complexity of the horizontal search can be reduced to $O(k)$, which clearly would be optimal. It turns out that by maintaining an auxiliary data structure we are able to achieve this.

Using the labels created by our label list, we maintain two label lists, (one for right and one for left labels). The left (right) label list consists of a linked list of all the left (right) labels that are in use. The linked lists will be kept "in the natural order" (e.g., when 3 children are added to a node with left label L, the two new left labels created for the second and third child are added to the linked list immediately preceding L). The label list has the following nice properties:

1. Given a sequence of left labels $L_1, L_2, \ldots, L_q$ on the nodes traversed on any partial horizontal list, $L_1, L_2, \ldots, L_q$ occur in the same order in the left label list (although usually not consecutively).
2. Given nodes v and w with left labels L_v and L_w and right labels R_v and R_w, v is a descendent of w iff L_v precedes L_w on the left label list, and R_v precedes R_w on the right label list.

We may now rephrase the generic descendent query as follows: Given two nodes w and v in the tree with left label L_w and L_v determine whether L_v precedes L_w in the left label list.

A solution to this problem was given by Dietz and Sleator (1987). It is based on assigning integer labels to the elements of the linked list, which increase from left to right. When a new element is inserted, it gets an integer label, which is between the two values. When no integer is available the elements in the neighborhood are relabeled. Dietz and Sleator (1987) exhibited such a relabeling scheme, which takes $O(1)$ amortized time. Order queries can be answered in $O(1)$ time per query. Inserting a new node into the auxiliary data structure takes $O(1)$ amortized time.

As a result, the general case of the horizontal search, where nodes are allowed to die, can also be answered in $O(k)$ time (where k is the cardinality of the answer,

i.e., the number of live nodes satisfying the horizontal query). The space used by the label lists is constant per node and the time needed to maintain the label lists is amortized $O(1)$ per change; thus the general *HS* algorithm uses $O(n)$ space and amortized $O(1)$ update processing.

Implementation Concerns: When implementing the horizontal query, we note that the auxiliary data structure becomes quite important. In the most straightforward partition of our tree into pages, it is likely that most nodes on the horizontal path will be on one page. Their common ancestor, though, probably is not. Therefore, avoiding the ancestor query when deciding where a horizontal traversal is to stop, using the labeling scheme, is very important.

6. Conclusions

Historical queries are an important part of many database systems (e.g., rollback databases and design databases). Previous approaches to history reconstruction problems deal with only a single line of time evolution. There are, however, complex applications where the database's state evolves in a way that results in *multiple* lines of evolution. The key characteristic of these multiple-lines of evolution problems is that portions of the history are shared; answering historical queries should not necessitate duplication of shared histories, as this could increase the storage requirements dramatically. In this article, we provide a general framework for solving multiple-line history queries. We address two novel historical queries in this environment: the vertical query and the horizontal query. The vertical query enables searching by time on the shared path of an evolution, while the horizontal query facilitates searching at concurrent times in the past of evolutions that share the same ancestor. Combinations of these basic queries enable searching both dimensions of the multiple lines of evolution. We are currently extending our results for the case where evolutions can "merge." The problem becomes more complicated as the tree of evolution lines becomes a graph of evolutions.

Acknowledgments

This research was partially supported by NSF grants CCR-8908286, CCR-9110255, IRI-9111271, and by the New York State Science and Technology Foundation as part of its Center for Advanced Technology program. The authors would like to thank R. Snodgrass and I. Munro for many helpful discussions; B. Salzberg for comments on an early version of this article, and S. Marshall from Xerox Labs for bringing the problem of multiple lines of time evolution to our attention.

References

Ahn, I. and Snodgrass, R. Performance evaluation of a temporal database management system. *Proceedings of the ACM SIGMOD Conference on the Management of Data,* Washington, DC, 1986.

Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. On optimal multiversion access structures. *Workshop on Advances in Spatial Databases,* Singapore, 1993.

Cole, R. Searching and storing similar lists. *Journal of Algorithms,* 7:202-220, 1986.

Dietz, P. and Sleator, D.D. Two algorithms for maintaining order in a list. *Proceedings of the ACM Symposium on the Theory of Computers,* New York, NY, 1987.

Driscoll, J.R., Sarnak, N., Sleator, D.D., and Tarjan, R.E. Making data structures persistent. *Journal of Computer and System Sciences,* 38:86-124, 1989.

van Emde Boas, P., Kaas, R., and Zijlstra, E. Design and implementation of an efficient priority queue. *Mathematical Systems Theory,* 10:99-127, 1977.

Elmasri, R., Wuu, G., and Kim, Y. The time index: An access structure for temporal data. *Proceedings of the Sixteenth Conference on Very Large Data Bases,* Brisbane, Australia, 1990.

Jensen, C.S., Mark, L., and Roussopoulos, N. Incremental implementation model for relational databases with transaction time. *IEEE Transactions on Knowledge and Data Engineering,* 3(4):461-473, 1991.

Katz, R.H. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys,* 22(4):375-408, 1990.

Kolovson, C. Indexing for historical databases. In: Tansel, A., Clifford, J., Gadia, S.K., Jajodia, S., Segev, A., and Snodgrass, R., eds. *Temporal Databases: Theory, Design, and Implementation,* Redwood City, CA: Benjamin/Cummings, 1993, pp 418-432.

Kolovson, C. and Stonebraker, M. Indexing techniques for historical databases. *Proceedings of the Fifth IEEE International Conference on Data Engineering,* Los Angeles, CA, 1989.

Kolovson, C. and Stonebraker, M. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. *Proceedings of the ACM SIGMOD Conference on the Management of Data,* Denver, CO, 1991.

Lanka, S. and Mays, E. Fully persistent $B+$ trees. *Proceedings of the ACM SIGMOD Conference on the Management of Data,* Denver, CO, 1991.

Leung, T.Y.C. and Muntz, R.R. Stream processing: Temporal query processing and optimization. In: Tansel, A., Clifford, J., Gadia, S.K., Jajodia, S., Segev, A., and Snodgrass, R., eds. *Temporal Databases: Theory, Design, and Implementation,* Redwood City, CA: Benjamin/Cummings, 1993, pp. 329-355.

Lomet, D. and Salzberg, B. Access methods for multiversion data. *Proceedings of the ACM SIGMOD Conference on the Management of Data,* Portland, OR, 1989.

Lomet, D. and Salzberg, B. The performance of a multiversion access method. *Proceedings of the ACM SIGMOD Conference on the Management of Data,* Portland, OR, 1990.

Lomet, D. and Salzberg, B. Transaction-time databases. In: Tansel, A., Clifford, J., Gadia, S.K., Jajodia, S., Segev, A., and Snodgrass, R., eds. *Temporal Databases: Theory, Design, and Implementation,* Redwood City, CA: Benjamin/Cummings, 1993, pp. 388-417.

Lum, V., Dadam, P., Erbe, R., Guenauer, J., Pistor, P., Walch, G., Werner, H., and Woodfill, J. Designing DBMS support for the temporal database. *Proceedings of the ACM SIGMOD Conference on the Management of Data,* Boston, MA, 1984.

Manolopoulos, Y. and Kapetanakis, G. Overlapping $B+$ trees for temporal data. *Proceedings of the Fifth JCIT Conference,* Jerusalem, 1990.

Marshall, S. Xerox Webster Research Center, private communication, 1991.

Salzberg, B. and Tsotras, V.J. A comparison of access methods for time-evolving data. Technical report CATT-TR-94-81, Polytechnic University, or technical report NU-CCS-94-21, Northeastern University, 1994.

Segev, A. and Gunadhi, H. Event-join optimization in temporal relational databases. *Proceedings of the Fifteenth Conference on Very Large Data Bases,* 1989.

Segev, A. and Gunadhi, H. Efficient indexing methods for temporal relations. *IEEE Transactions on Knowledge and Data Engineering,* 5(3):496-509, 1993.

Snodgrass, R. and Ahn, I. Temporal databases. *IEEE Computer,* 19(9):35-42, 1986.

Tsotras, V.J. and Gopinath, B. Efficient algorithms managing the history of evolving databases. *Proceedings of the Third International Conference on Database Theory,* Paris, 1990.

Tsotras, V.J. and Gopinath, B. Optimal versioning of objects. *Proceedings of the Eighth IEEE International Conference on Data Engineering,* Phoenix, AZ, 1992.

Tsotras, V.J., Gopinath, B., and Hart, G.W. Efficient management of time-evolving databases. *IEEE Transactions on Knowledge and Data Engineering,* 7(4), 1994.

Tsotras, V.J. and Kangelaris, N. The snapshot index, an I/O-optimal access method for snapshot queries. CATT-Tech. *Information Systems,* 20(3):237-260, 1994.