THE TIME INDEX: AN ACCESS STRUCTURE FOR TEMPORAL DATA

Ramez Elmasri[1,*], Gene T. J. Wuu[2], and Yeong-Joon Kim[1]

[1]Department of Computer Science, University of Houston, Houston, TX 77204
[2]Bell Communications Research, 444 Hoes Lane, Piscataway, NJ 08854

ABSTRACT

In this paper, we describe a new indexing technique, the *time index*, for improving the performance of certain classes of temporal queries. The time index can be used to retrieve versions of objects that are valid during a specific time period. It supports the processing of the temporal WHEN operator and temporal aggregate functions efficiently. The time indexing scheme is also extended to improve the performance of the temporal SELECT operator, which retrieves objects that satisfy a certain condition during a specific time period. We will describe the indexing technique, and its search and insertion algorithms. We also describe an algorithm for processing a commonly used temporal JOIN operation. Some results of a simulation for comparing the performance of the time index with other proposed temporal access structures are presented.

1 Introduction

Research in temporal databases has been mostly concerned with defining data models and operations that incorporate the time dimension. Extensions to the relational data model and its operations for handling temporal data have been presented in [CT85, SA85, Sno87, CC87, Gad88, GY88, NA87]. In addition, some work has been concerned with defining temporal extensions to conceptual data models and query languages [SS87, EW90]. These temporal data models define powerful operations for specifying complex temporal queries. There has been relatively less research in the area of defining efficient storage structures and access paths for temporal data [Lum84, Ahn86, AS88, SG89, GSsu, RS87, KS89, LS89]. These proposals do not discuss indexing schemes for supporting the high-level temporal operators defined in [GY88, EW90]. This paper describes indexing techniques for improving the efficiency of temporal operations, such as *when*, *select*, and *join* [GY88], *temporal selection* and *temporal projection* [EW90], and aggregation functions.

The storage techniques for temporal data proposed in [AS88, Lum84] index or link the versions of each individual object separately. In order to retrieve object versions that are valid during a certain time period, it is necessary to first locate the first (current) version of each object, and then search through the version index (or list) of each object separately. In comparison, our time index will lead directly to the desired versions without having to search the version index of each individual object separately. The method proposed in [RS87] allows a search based on time using a multi-dimensional partitioned file, in which one of the dimensions is the time dimension. In their scheme temporal data items are associated with a time point rather than a time interval, and hence is not useful when time intervals are assumed. Other work ([LS89, KS89]) discusses indexing historical data when optical disks are available, and is mainly concerned with the index behaviour as older data is transferred to optical disk.

We consider our time index to be a basic indexing technique for temporal data. It can be combined with a conventional attribute indexing scheme to efficiently process temporal selections and temporal join operations. Figure 1 shows a simple example of a temporal database consisting of two relations. We will use this example to illustrate the structure of our time index. Section 2 describes the index access structure, and presents the search and insertion algorithms. Section 3 discusses how the basic time index can be ex-

*This work is supported by Bellcore and by Texas ARP grant 003652081-ARP

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

tended to improve the efficiency of additional temporal operations. Section 4 includes some performance simulation results for the time index. Finally, Section 5 presents conclusions and directions for future research.

Name	Dept	Valid_Time
emp1	A	[0, 3]
emp1	B	[4, now]
emp2	B	[0, 5]
emp3	C	[0, 7]
emp3	A	[8, 9]
emp4	C	[2, 3]
emp4	A	[8, now]
emp5	B	[10, now]
emp6	C	[12, now]
emp7	C	[11, now]

The EMPLOYEE table

Dept	Manager	Valid_Time
A	Smith	[0, 3]
A	Thomas	[4, 9]
A	Chang	[10, now]
B	Cannata	[0, 6]
B	Martin	[7, now]
C	Roberto	[0, now]

The DEPARTMENT table

Figure 1: A Temporal Database

2 The Time Index Access Structure

In this section, we first give a storage model for temporal data based on the object versioning approach[1] [SA85]. The time indexing technique can be adapted to other temporal database proposals, such as time normalization [NA87] or attribute versioning [GY88]. We use object versioning because it is a simpler approach for storage management, and allows us to concentrate our presentation on the properties of the time index itself. In Section 2.2, we will describe our time index, and provide search, insertion, and deletion algorithms. Sections 2.3 and 2.4 show how the time index may be used to efficiently process the temporal WHEN operator and aggregate functions.

[1]This approach is called tuple versioning in [SA85]

2.1 The Temporal Storage Model

The time dimension is represented, as in [GY88, CW83, Gad88] and others, using the concepts of discrete time points and time intervals. A *time interval*, denoted by $[t_1, t_2]$, is defined to be a set of consecutive equidistant time instants (points), where t_1 is the first time instant and t_2 is the last time instant of the interval. The *time dimension* is represented as a time interval $[0, now]$, where 0 represents the starting time of our database mini-world application, and *now* is the current time, which is continuously expanding. The distance between two consecutive time instances can be adjusted based on the granularity of the application to be equal to months, days, hours, minutes, seconds, or any other suitable time unit. A single discrete time point t is easily represented as an interval $[t, t]$, or simply $[t]$.

We will assume an underlying record-based storage system which supports *object versioning.* Records are used to store versions of objects. In addition to the regular record attributes, A_i, each record will have an interval attribute, called *valid_time*, consisting of two sub-attributes t_s (valid start time) and t_e (valid end time). The *valid_time* attribute of an object version is a time interval during which the version is valid. In object versioning, a record r with $r.valid\_time.t_e = now$ is considered to be the *current version* of some object. However, numerous *past versions* of the object can also exist. We assume that the versions of an object are linked to the current version using one of the basic storage techniques (chaining, clustering, accession list) proposed in [AS88, Lum84]. In addition, we assume that the current version of an object can be efficiently located from any other version; for example, by using a pointer to a linked list header, which in turn points to the current version.

Whenever an object o is updated with new attribute values, the current version, r, becomes the *most recent* past version, and a new current version r' is created for o. If the valid time of the update is t_u, then the update is executed as follows:

```
$r.valid\_time.t_e \leftarrow (t_u - 1)$ ;
create a new object version $r'$ by setting $r' \leftarrow r$ ;
for each modified regular attribute $A_i$
    set $r'.A_i \leftarrow$ the new attribute value ;
set $r'.valid\_time.t_s \leftarrow t_u$ ;
set $r'.valid\_time.t_e \leftarrow now$ ;
```

Such a database is called *append only* since older object versions are never deleted, so the file of records continually has object versions appended to it. An operation to delete an object o at time t_d is executed as follows:

```
find the current version $r$ of the object $o$;
set $r.valid\_time.t_e \leftarrow t_d$ ;
```

Finally, an operation to insert an object o at time t_i is executed as follows:

```
create the initial version $r$ for $o$ ;
set $r.valid\_time.t_s \leftarrow t_i$ ;
set $r.valid\_time.t_e \leftarrow now$ ;
```

Because the append-only nature of such a temporal database will eventually lead to a very large file, we assume that a *purge(t_p)* operation is available. This operation purges all versions r with $r.valid\_time.t_e < t_p$ by moving those versions to some form of archival storage, such as optical disk or magnetic tape.

2.2 Description of the Time Index

Conventional indexing schemes assume that there is a total ordering on the index search values. The properties of the temporal dimension make it difficult to use traditional indexing techniques for time indexing. First, the index search values, the *valid_time* attribute, are *intervals* rather than points. The *valid_time* intervals of various object versions will overlap in arbitrary ways. Because one cannot define a total ordering on the interval values, a conventional indexing scheme cannot be used. Second, because of the nature of temporal databases, most updates occur in an *append* mode, since past versions are kept in the database. Hence, deletions of object versions do not generally occur, and insertions of new object versions occur mostly in *increasing time value*. In addition, the search condition typically specifies the retrieval of versions that are valid during a particular time interval.

A time index is defined over an object versioning record-based storage system, TDB, which consists of a collection of object versions, TDB = $\{e_1, e_2, ..., e_n\}$, and supports an interval-based search operation. This operation is formally defined as follows.

Given a Search Interval, $I_S = [t_a, t_b]$, find the following set of versions:

$S(I_S) = \{e_j \in TDB \mid (e_j.valid\_time \cap I_S) \neq \emptyset \}$

A simple but inefficient implementation of this search operation is to sequentially access the entire storage system, TDB, using linear search, and to retrieve those records whose *valid_time* intersects with I_S. Such a search will require O(N*M) accesses to the storage system, where N is the number of objects and M is the maximal number of versions per object.

Notice that the interval-based search problem is identical to the k-dimensional spatial search problem, where k = 1. There have been a number of index methods proposed for k-dimensional spatial search [Gut84, OSD87], which are not suitable for the time dimension for the reasons discussed below. These index methods support spatial search for 2-dimensional objects in CAD or geographical database applications. The algorithms proposed in [Gut84, OSD87] use the concept of a region to index spatial objects. A search space is divided into regions which may overlap with each other. A sub-tree in an index tree contains pointers to all spatial objects located in a region. Since spatial objects can overlap with each other, handling the boundary conditions between regions is quite complex in these algorithms. In temporal databases, there can be a very high degree of overlapping between the *valid_time* intervals of object versions. A large number of long or short intervals can exist at a particular time point. Furthermore, the search space is continuously expanding and most spatial indexing techniques assume a fixed search space. In addition, temporal objects are appended mostly in increasing time value, making it difficult to maintain tree balance for traditional indexing trees. Because of these differences between temporal and spatial search, we do not consider the spatial algorithms in [Gut84, OSD87] to be suitable for temporal data if they are directly adapted from 2-dimensions to a single dimension.

The idea behind our time index is to maintain a set of linearly ordered *indexing points* on the time dimension. An indexing point is created at the time points where (a) a new interval is started, or (b) the time point immediately after an interval terminates. The set of all indexing points is formally defined as follows:

(PR1) $BP = \{t_i \mid \exists e_j \in TDB\ ((t_i = e_j.valid\_time.t_s) \vee (t_i = e_j.valid\_time.t_e + 1))\} \cup \{now\}$

The concept of indexing points is illustrated in Figure 2 for the temporal data shown in the EMPLOYEE table of Figure 1. In Figure 2, e_{ij} refers to version j of object e_i. There exist 9 indexing points in BP for all employee versions, $BP = \{0, 2, 4, 6, 8, 10, 11, 12, now\}$. Time point 2 is an index point since the version e_{41} starts at 2. Time point 6 is an index point since e_{21} terminates at 5.

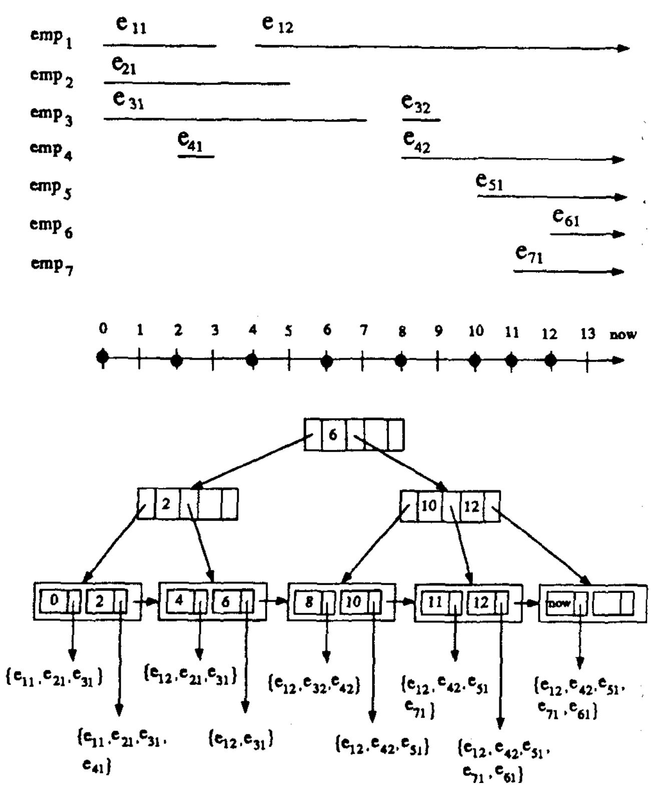


Figure 2: Versions of EMPLOYEE Objects, and a Time Index

Before proceeding to describe our index structure, we define some additional notation that will be useful in our discussion. Let t_j be an arbitrary time point, which may or may not be a point in BP. We define t_j^- (t_j^+) to be the point in BP such that $t_j^- < t_j$ ($t_j < t_j^+$) and there does not exist a point $t_m \in BP$ such that $t_j^- < t_m < t_j$ ($t_j < t_m < t_j^+$). In other words, t_j^- (t_j^+) is the point in BP that is immediately before (after) t_j. We also define $t_j^{-=}$ as follows:

1. If there exists a point $t_k \in BP$ such that $t_j = t_k$, then $t_j^{-=} = t_k$.
2. Otherwise, $t_j^{-=} = t_j^-$

Since all the indexing points t_i in BP can be totally ordered, we can now use a regular B^+-tree [Com79, EN89] to index these time points. Each leaf node entry of the B^+-tree at point t_i is of the form:

$$[t_i, bucket]$$

where *bucket* is a pointer to a bucket containing pointers to object versions. Each bucket $B(t_i)$ in our index scheme is maintained such that it contains pointers to all object versions whose *valid_time* contains the interval $[t_i, t_i^+ - 1]$. Such a property can be formally specified as follows:

(PR2) $B(t_i) = \{e_j \in TDB \mid ([t_i, t_i^+ - 1] \subseteq e_j.valid\_time)\}$

Figure 2 shows a B^+-tree of order 3, which indexes the BP set of points of the EMPLOYEE versions. Each node in the B^+-tree contains at most two search values and three pointers. Consider the leaf entry for search time point 4, for instance; (PR2) indeed holds.

$$\begin{aligned} B(4) &= \{e_{12}, e_{21}, e_{31}\} \\ &= \{e_j \in TDB \mid ([4,5] \subseteq e_j.valid\_time)\} \end{aligned}$$

In a real temporal database, there can be a large number of object versions in each bucket, and many of those may be repeated from the previous bucket. For example, in Figure 2 the object version e_{12} appears in multiple consecutive buckets. To reduce this redundancy and make the time index more practical, an incremental scheme is used. Rather than keeping a full bucket for each time point entry in BP, we only keep a full bucket for the first entry of each leaf node. Since most versions will continue to be valid during the next indexing interval, we only keep the *incremental changes* in the buckets of the subsequent entries in a leaf node. For instance, in Figure 3 the entry at point 10 stores $\{+e_{51}, -e_{32}\}$ in its incremental bucket indicating e_{51} starts at point 10 and e_{32} terminates at the point immediately before point 10. Hence, the incremental bucket $B(t_i)$ for a non-leading entry at time point t_i can be computed as follows:

$$\begin{aligned} B(t_i) = B(t_l) &\cup (\textstyle\bigcup_{t_j \in BP, t_l < t_j < t_i} SA(t_j)) \\ &- (\textstyle\bigcup_{t_j \in BP, t_l < t_j < t_i} SE(t_j)) \end{aligned}$$

where $B(t_l)$ is the bucket for the leading entry in the leaf node where point t_i is located, $SA(t_j)$ is the set of object versions whose start time is t_j and $SE(t_j)$ is the set of object versions whose end time is $t_j - 1$.

We now describe our search algorithm as follows:

1. Suppose the time search interval is $I_S = [t_a, t_b]$. Perform a range search on the B^+-tree to find

   (C1) $PI(I_S) = \{t_i \in BP \mid t_a \le t_i \le t_b\} \cup \{t_a^{-=}\}$

2. Then compute the following set as the result of the algorithm.

   (C2) $T(I_S) = \bigcup_{t_i \in PI} B(t_i)$

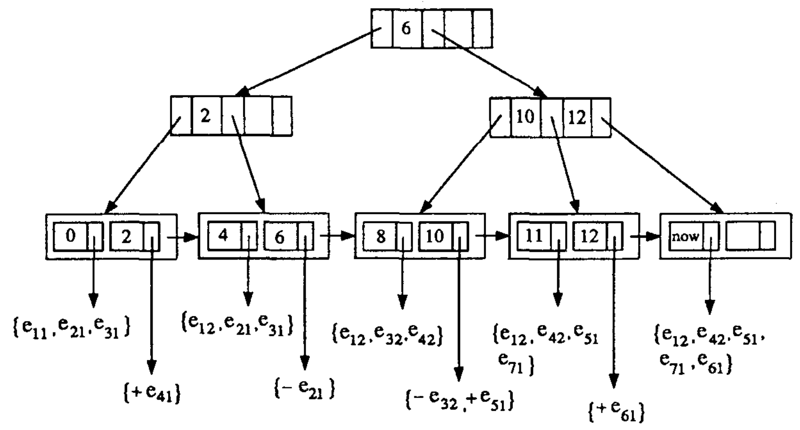


Figure 3: Storing Incremental Changes in the Time Index Buckets

Insertion or deletion of a new object version should maintain the properties (PR1) and (PR2). The algorithms for inserting and deleting an object version e_k are shown in Algorithm A. Note that, in general, *version deletion will not occur* in append-only databases except for an exception such as correction of an error.

It is easy to argue that (PR1) and (PR2) are maintained after each execution of the Insert or Delete operation. We will not show the proof argument here due to the lack of space.

2.3 Using the Time Index for Processing the WHEN Operator

The time index can be used to efficiently process the WHEN operator [GY88] with a constant projection time interval. An example of the type of query is: List the salary history for all employees during the time interval [4, 5]. The result of such a query can be directly retrieved using the time index on the EMPLOYEE object versions shown in Figure 3. We will discuss in Section 3 how an extension to the time index will permit efficient processing of temporal SELECT operations. Notice that a simple query such as the one given above is very expensive to process if there was no index on time.

2.4 Using the Time Index for Processing Aggregate Functions

In this section we will describe how the time index scheme is used to process *aggregate functions* at different time points or intervals. In non-temporal conventional database, the aggregate functions, such as COUNT, EXISTS, SUM, AVERAGE, MIN, and

$\boxed{\text{Algorithm A}}$

```
Insert($e_k$)
begin
    $t_a \leftarrow e_k.valid\_time.t_s$ ;
    $t_b \leftarrow e_k.valid\_time.t_e + 1$ ;
    search the $B^+$-tree for $t_a$;
    if ($\neg$found) then
        insert $t_a$ in the $B^+$-tree;
    if entry at $t_a$ is not a leading entry in a leaf node
        add $e_k$ into $SA(t_a)$;
    search the $B^+$-tree for $t_b$;
    if ($\neg$found) then
        insert $t_b$ in the $B^+$-tree;
    if entry at $t_b$ is not a leading entry in a leaf node
        add $e_k$ into $SE(t_b)$;
    for each leading entry $t_l$ of a leaf node
    where $t_a \leq t_l < t_b$
        add $e_k$ in $B(t_l)$;
end

Delete($e_k$)
begin
    $t_a \leftarrow e_k.valid\_time.t_s$ ;
    $t_b \leftarrow e_k.valid\_time.t_e + 1$ ;
    search the $B^+$-tree for $t_a$;
    if entry at $t_a$ is not a leading entry in a leaf node
      remove $e_k$ from $SA(t_a)$;
    search the $B^+$-tree for $t_b$;
    if entry at $t_b$ is not a leading entry in a leaf node
      remove $e_k$ from $SE(t_b)$;
    for each leading entry $t_l$ of a leaf node
    where $t_a \leq t_l < t_b$
      remove $e_k$ from $B(t_l)$;
end
```

MAX are applied to sets of objects or attribute values of sets of objects. In temporal databases, an aggregate function is applied to a set of temporal entities over an interval. For instance, the query 'GET COUNT EMPLOYEE : [3, 8]' [EW90] should count the number of employees at each time point during the time interval [3, 8]. The result of the temporal COUNT function is a function mapping from each time point in [3, 8] to an integer number that is the number of employees at that time point. For instance, the above query is evaluated to the following result if applied to the database shown in Figure 1:

$$\{ [3] \rightarrow 4, [4, 5] \rightarrow 3, [6, 7] \rightarrow 2, [8] \rightarrow 3 \}$$

Our time index can be easily used to process such aggregate functions. Let I_S be the interval over which the temporal aggregate function is evaluated. The query performs a range search to find $PI(I_S)$. Each point in $PI(I_S)$ represents a point of state change in the database. That is, the database mini-world changes its state at each change point and stays in the same state until the next change point. Therefore the aggregate function only needs to be evaluated for the points in $PI(I_S)$. The query is evaluated by applying the function on the bucket of object versions at each point. If the incremental index shown in Figure 3 is used, the running count from the previous change point is updated at the current change point by adding the number of new versions and subtracting the number of removed versions at the change point. Similar techniques can be used for other aggregate functions that must be computed at various points over a time interval.

3 Extensions of the Time Index for Other Temporal Operators

The basic indexing scheme can be extended to support other important temporal operators, such as *temporal selection* [EW90, GY88]. In a non-temporal database, a common form of a selection condition is to compare an attribute with a constant or with a range; for example, 'EMPLOYEE.Dept = B' or '20K < EMPLOYEE.Salary < 30K'. Such conditions evaluate to a boolean value for each object. In a temporal database, however, a θ comparison condition evaluates to a function which maps from [0, now] to a boolean value. For instance, the condition 'EMPLOYEE.Dept = B' when evaluated on *empl* of Figure 1 will have the following result:

$$\{ [0, 3] \rightarrow \text{FALSE}, [4, \text{now}] \rightarrow \text{TRUE} \}$$

A complete temporal selection should specify not only a condition but also *when* the condition holds. For example, to select employees who had worked in department B during the time period [3, 4], a SELECT condition should be specified as:

$$[\![\text{EMPLOYEE.Dept} = \text{'B'}]\!] \cap [3, 4] \neq \emptyset$$

The notation $[\![c]\!]$, where c is a θ comparison condition, represents the time intervals during which c evaluates to TRUE for each object. A search for objects that satisfy such a temporal condition can be formulated as a spatial search problem, as illustrated in Figure 4. The search space has two dimensions: the attribute dimension and the time dimension. A vertical line represents an object version whose search value consist of a point value on the attribute dimension and an interval value on the time dimension. A two-dimensional range is specified by a rectangle. For instance, the dotted rectangle in Figure 4 specifies the search condition:

$$[\![\text{'B'} <= \text{EMPLOYEE.Dept} <= \text{'C'}]\!] \cap [6, 9] \neq \emptyset$$

A search involves the retrieval of all object versions that intersect with the search rectangle.

These types of operations combine selection based on a time interval with a selection based on conditions involving attribute values. If the condition to be satisfied is based on the value of a single attribute, we can use a traditional B^+-tree index constructed on that attribute, which searches for the *current version* of each object that satisfies the (attribute) search condition. In current proposed storage structures [Lum84, Ahn86], the current version of an object can be used to access past (and possibly future) versions of the object using various techniques such as clustering, reverse chaining, or an accession list. These proposed structures work well only if the index search field is a non-temporal attribute of the objects; that is, an attribute whose value does not change with time. The reason is that for conditions on temporal (time-varying) attributes, the attribute value of an object *may have been changed* so that it is not possible to access a past version that *used to satisfy* the search criterion via the current version. In order to solve this problem, the index must include direct pointers to all *past and present* object versions that have the search attribute value. This can result in a very large number of pointers in the leaf nodes of the index. It is then still necessary to search through all these versions and to check whether each version satisfies the time condition.

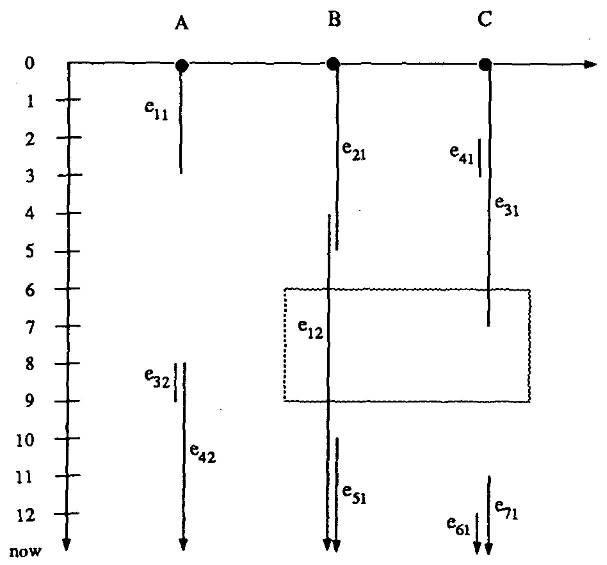
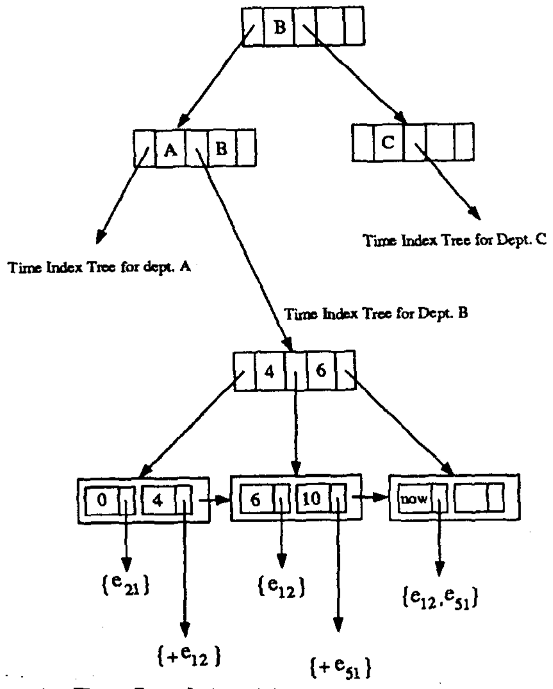


Figure 4: Two-Dimensional Search



B
A
B
C
Time Index Tree for dept. A
Time Index Tree for Dept. C
Time Index Tree for Dept. B
4
6
0
4
6
10
now
{e_{21}}
{e_{12}}
{e_{12},e_{51}}
{$+e_{12}$}
{$+e_{51}$}

Figure 5: Two-Level Combined Attribute/Time Index

3.1 The Two-Level Combined Attribute Time Index

To solve this problem, our approach is to use a two-level indexing scheme. The top-level index is a B^+-tree built on a search attribute; for example, the *Dept* attribute of EMPLOYEE in Figure 1. Each *leaf node entry* of the top-level index tree includes a value of the search attribute and a *pointer to a time index*. Hence, there is a time index tree for each attribute value. The internal structure of each time index tree is similar to the basic time index described in Section 2.

Figure 5 shows the combined index structure for the EMPLOYEE table shown in Figure 1. The top-level index tree is built on the *Dept* attribute. Since there are three departments, A, B, and C, there are three time index trees for them. Figure 5 only shows the time index tree for department B, which indexes versions of EMPLOYEE objects that have a *Dept* attribute value of 'B'.

3.2 Processing the Temporal Select Operator

We now describe how to use the two-level index scheme to process a temporal SELECT condition such as:

$[\![$ Employee.Dept = 'B' $]\!] \cap [3, 4] \neq \emptyset$

This selects all employees who worked for department B during the time interval [3, 4]. The first step is to search the top level (the *Dept* attribute) index for the *Dept* value 'B'. This leads to the time index for department B, which is then searched for the time interval [3, 4]. As a result, the appropriate object versions are retrieved.

Note that each of these retrieved versions records a partial history of a selected object. However, in most temporal data models ([GY88]), the SELECT operator should return the full set of versions (the entire history) for each selected object. Hence, it is necessary to assume that versions of each object will contain back pointers to access the current version as part of the basic temporal access structure. Any one of the traditional *version access structures* for object versions (such as clustering, accession list, or reverse chaining) can then be used to retrieve the entire version history via the current object for the selected objects.

3.3 Using the Time Index to Process Temporal JOIN operations

The time index can also be used to improve the efficiency of certain temporal join operations. There have been several temporal join operations discussed in

[SG89]. However, most of these join operations are defined for joining together a temporal object that is *vertically partitioned* into several relations via time normalization. For example, the attributes of temporal EMPLOYEE objects would be partitioned into several relations, where each relation would hold the primary key and those attributes (usually a single one) that are always modified synchronously. There would be a relation for EMP_SALARY, one for EMP_JOB, and so on. The EVENT JOIN [SG89] is used to build back the temporal objects from the partitioned relations.

A time index can be used to increase the efficiency of join operations. This includes more general types of join operations that correspond to the NATURAL JOIN operation of a non-temporal database. This type of operation joins the tuples of two relations based upon an *equality join condition* on attribute values during a *common time interval*. Hence, the result of the join would include an object version whenever two object versions have the same join attribute value, *and* the intersection of the valid time periods during which the join attributes are equal is not empty. The valid time of the resulting join object would be the intersection of the valid times of the two joined object versions. For example, consider the database shown in Figure 1, where two relations EMPLOYEE and DEPARTMENT are shown. Suppose we want to execute a join operation to retrieve the time history of employees working for each department manager. In this case, we want to join each department object with the appropriate EMPLOYEE objects during the time periods when the employees worked for that department. The result of the join would be as shown in Figure 6.

We can use a two-level time index on the *Dept* attribute of EMPLOYEE to retrieve the employee versions working for each department during specific time periods. The join algorithm outline is shown in Algorithm B.

Name	Dept	Valid_Time	Manager
emp1	A	[0, 3]	Smith
emp1	B	[4, 6]	Cannata
emp1	B	[7, now]	Martin
emp2	B	[0, 5]	Cannata
emp3	C	[0, 7]	Roberto
emp3	A	[8, 9]	Chang
emp4	C	[2, 3]	Roberto
emp4	A	[8, 9]	Thomas
emp4	A	[10, now]	Chang
emp5	B	[10, now]	Martin
emp6	C	[12, now]	Roberto
emp7	C	[11, now]	Roberto

Figure 6: Result of Temporal Join

Using the time index would increase the efficiency of locating the EMPLOYEE object versions based on a particular *Dept* value and time interval combination. Hence, the versions of DEPARTMENT and EMPLOYEE to be joined can be directly located. The intersection of their valid time intervals is then calculated for the result of the join.

4 Performance Evaluation

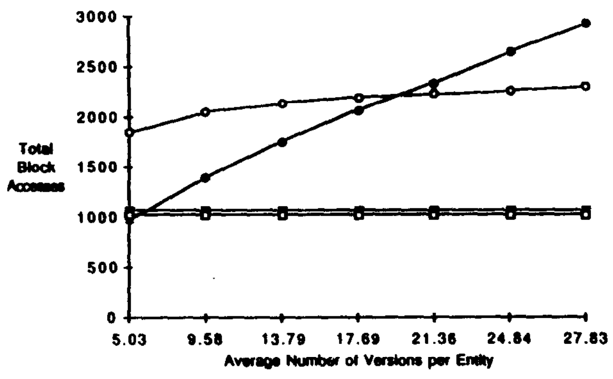
We simulated the performance of the time index in order to compare it with traditional temporal access structures. Some of the results of the simulation are shown in Figures 7 to 12. The database had 1000 objects, and versions where added based on an exponential distribution for interarrival time. New versions were assigned to objects using a uniform distribution. Objects where also inserted and deleted using an exponential distribution with a much larger interarrival time than that for version creation.

Algorithm B

```
for each DEPARTMENT object do
    begin
    for each version of the DEPARTMENT object do
        begin
        retrieve the Dept value, and valid time [t1, t2] of the version;
        use the EMPLOYEE top-level index to locate the time index for the Dept value;
        use the time index to retrieve EMPLOYEE versions whose time interval overlaps [t1, t2];
        join each EMPLOYEE version to the DEPARTMENT version;
        end;
    end;
```

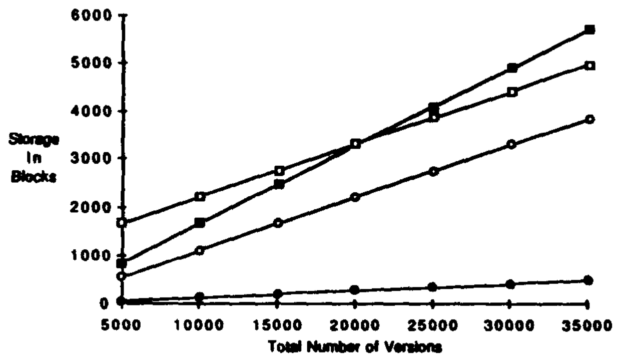
○ Accession List
● Clustering
■ Time Index with Accession List
□ Time Index with Clustering
Legend for Figure 1

Figure 1. Block Accesses for Interval Query



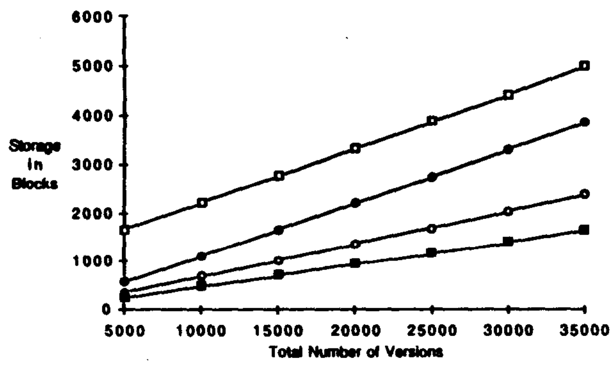
○ Accession List Time Index
● Time Index B+Tree Storage
■ Clustering Time Index
□ Data File Blocks
Legend for Figure 2

Figure 2. Storage for Regular Time Index



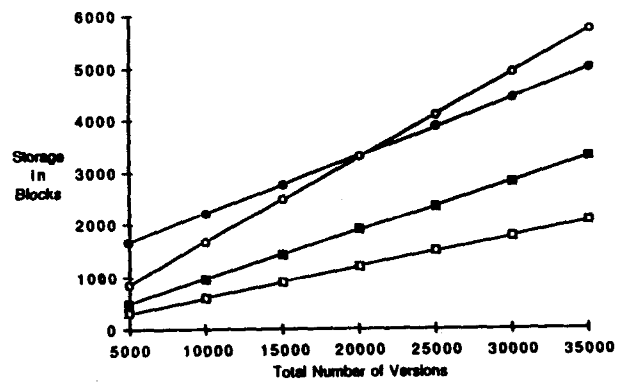
□ Data File Blocks
● Time Index (1-block leaf nodes)
○ Time Index (2-block leaf nodes)
■ Time Index (4-block leaf nodes)
Legend for Figure 3

Figure 3. Time Index Storage (Accession List)



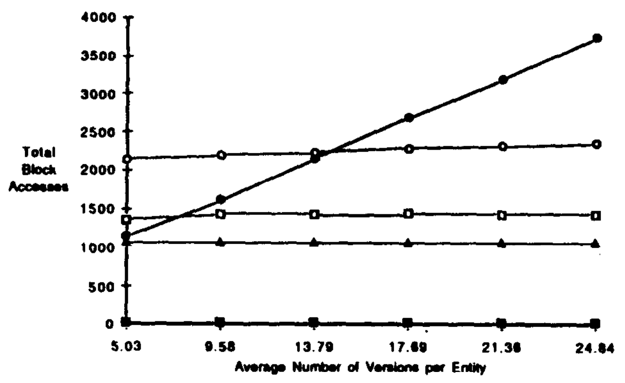
● Data File Blocks
○ Time Index (1-block leaf nodes)
■ Time Index (2-block leaf nodes)
□ Time Index (4-block leaf nodes)
Legend for Figure 4

Figure 4. Time Index Storage (Clustering)



○ Accession List Time Index
● Clustering Time Index
■ Two-Level Time Index
□ Interval Query (Two-Level Index)
▲ Regular Time Index
Legend for Figure 5

Figure 5. Block Accesses for Temporal Selection Query



■ Two-Level Time Index
● Regular Time Index
○ Data File Blocks
Legend for Figure 6

Figure 6. Two-level Time Index Storage

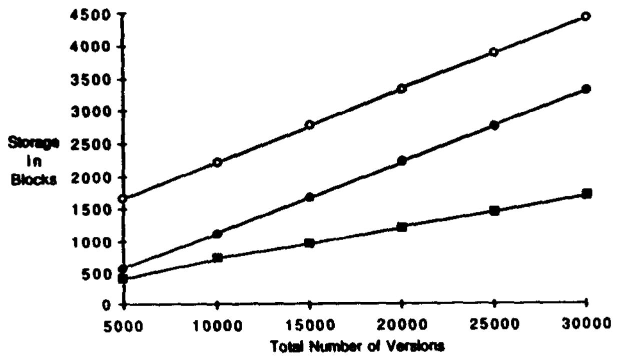


Figure 7 compares the performance of a time index with the traditional access structures of clustering (all versions of an object are clustered on disk blocks) and using an accession list (each object has an accession list to access its versions based on time) [AS88]. The number of block accesses needed for an interval query is calculated (an interval query retrieves all versions valid during a particular time period). Figure 7 shows how performance for clustering and accession list deteriorates as the number of versions per object grows, whereas using a time index maintains a uniform performance.

Figure 8 shows the storage requirements for a basic time index. As we can see, the B^+-tree itself does not require much storage but the buckets for leading entries in each leaf node require too much storage. This led us to simulate the case where each leaf node in the tree has two and four disk blocks in order to reduce the total number of buckets for leading entries. As can be seen in Figures 9 and 10, this led to an appreciable reduction in the storage requirements for the time index. Our simulation also showed that this did not adversely affect the performance of an interval query.

Figure 11 simulates the two-level time index performance for a temporal selection query (select all employees who work in a particular department during a particular time period). This temporal selection shows the most dramatic improvement over traditional access structures, since only 16 block accesses were needed using a two-level index compared to over 1000 block accesses with traditional structures. Because of this promising result, we simulated the performance of an interval query (Figure 11) using a two-level index, and the result was only about 30% higher than when using a regular (single-level) time index. This suggests that it may be sufficient to have only two-level time indexes on the various attributes. The storage requirements for the two-level index are also considerably less than for a regular time index because the versions are distributed over many time trees (Figure 12) leading to smaller buckets for leading entries in the leaf nodes.

5 Conclusions and Future Directions

We described a new indexing technique, the *time index*, for temporal data. The index is different from regular B^+-tree indexes because it is based on objects whose search values are *intervals* rather than points. We create a set of *indexing points* based on the starting and ending points of the object intervals, and use those points to build an indexing structure. At each indexing point, all object versions that are valid during that point can be retrieved via a bucket of pointers. We used incremental buckets to reduce the bucket sizes. Search, insertion, and deletion algorithms are presented.

Our structure can be used to improve the performance of several important operations associated with temporal databases. These include temporal selection, temporal projection, aggregate functions, and certain temporal joins. We showed how our index structure can be used to process each of the above temporal operations. Previous proposals for temporal access structures are mainly concerned with linking together the versions of a particular object, and dc not provide for efficient access strategies for the types of temporal operations discussed above. Results from simulating the behaviour of our access structure, and comparing its performance with the other proposed techniques show that the two-level time index is a very promising access structure for temporal selection queries. The one-level time index is efficient for interval queries, but requires much storage space; the storage space can be reduced by having larger leaf nodes in the $B^+ - tree$ to reduce the number of leading buckets. Our time index is hence a secondary access path that can be used to locate temporal objects efficiently without having to perform a search through the whole database when certain temporal operations are specified.

Appendix A

In this Appendix, we prove the correctness of the search algorithm for the time index.

Theorem: $S(I_S) = T(I_S)$.

Proof: ($\Rightarrow$) Assume $e_j \in S(I_S)$. We will show $e_j \in T(I_S)$. The condition $e_j.valid\_time \cap I_S \neq \emptyset$ implies that one of the following cases is true.

Case 1: $t_a \leq e_j.valid\_time.t_s \leq t_b$.

By (C2), it suffices to show both of the following two conditions hold: $e_j.valid\_time.t_s \in PI(I_S)$ and $e_j \in B(e_j.valid\_time.t_s)$. From the way $PI(I_S)$ is constructed (PR1), the first condition holds. We now show the second condition is also true. By (PR2), it suffices to show $[e_j.valid\_time.t_s, e_j.valid\_time.t_s^+ - 1]$

$\subseteq e_j.valid\_time = [e_j.valid\_time.t_s, e_j.valid\_time.t_e]$. Since these two intervals have the same starting point, it suffices to show $e_j.valid\_time.t_s^+ - 1 \leq e_j.valid\_time.t_e$. This is proved by contradiction. Assume $e_j.valid\_time.t_e < e_j.valid\_time.t_s^+ - 1$, or $e_j.valid\_time.t_e + 1 < e_j.valid\_time.t_s^+$. By (PR2), $e_j.valid\_time.t_e + 1$ is a point in BP. It is a contradiction since there cannot exist a point between $e_j.valid\_time.t_s$ and $e_j.valid\_time.t_s^+$.

Case 2: $e_j.valid\_time.t_s < t_a \leq e_j.valid\_time.t_e \leq t_b$.
It can be easily shown by contradiction that $[t_a^-, t_a - 1] \subseteq e_j.I$. By (PR2) and (C2), $e_j \in B(t_a^-) \subseteq T(I_S)$.
Case 3: $e_j.valid\_time.t_s < t_a \leq t_b < e_j.valid\_time.t_e$.
The argument is similar to Case 2.

($\Leftarrow$) Assume $e_j \in T(I_S)$. By (C2), at least one of the following two cases are true.
Case 1: $\exists t_k \in PI(I_S)(e_j \in B(t_k))$.
By (PR2), $[t_k, t_k^+ - 1] \subseteq e_j.I$. By (C1), t_k must be a point between t_a and t_b; that is, $t_k \in I_S$. Since t_k is contained in both $e_j.J$ and I_S, $e_j.valid\_time \cap I_S \neq \emptyset$ holds. Hence $e_j \in S(I_S)$.
Case 2: $e_j \in B(t_a^{-=})$.
From the way $t_a^{-=}$ is computed, there are two possibilities.
Case 2.1: $\exists t_m \in BP(t_m = t_a)$.
In this case, $t_a^{-=} = p_m$. Since $e_j \in B(p_m)$, by (PR2), $[t_m, t_m^+ - 1] \subseteq e_j.J$ follows. Since $t_m = t_a$, t_a is contained in both $e_j.J$ and I_S. Thus $e_j.J \cap I_S \neq \emptyset$ follows. Therefore $e_j \in S(I_S)$.
Case 2.2: $\nexists t_m \in BP(t_m = t_a)$. In this case, there exists a point t_l in BP such that $t_l = t_a^{-=} = t_a^-$. It can be easily shown that $t_l < t_a < t_l^+$. Since $e_j \in B(t_l)$, by (PR2), $[t_l, t_l^+ - 1] \subseteq e_j.J$ follows. Since $t_a > t_l$, $[t_a, t_l^+ - 1] \subseteq [t_l, t_l^+ - 1] \subseteq e_j.J$ holds. (Note that, by the fact $t_a < t_l^+$, the length of the interval $[t_a, t_l^+ - 1]$ is at least one.) Again, since t_a is contained in both $e_j.J$ and I_S, $e_j.J \cap I_S \neq \emptyset$ follows. Thus $e_j \in S(I_S)$.

References

[Ahn86] I. Ahn. Towards an implementation of database management systems with temporal support. In *IEEE Data Engineering Conference*, February 1986.

[AS88] I. Ahn and R. Snodgrass. Patitioned storage for temporal databases. *Information Systems*, 13(4), 1988.

[CC87] J. Clifford and A. Croker. The historical data model: an algebra based on lifespans. In *IEEE Data Engineering Conference*, February 1987.

[Com79] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(12), June 1979.

[CT85] J. Clifford and A. Tansel. On an algebra for historical relational databases: Two views. In *ACM SIGMOD Conference*, May 1985.

[CW83] J. Clifford and D. Warren. Formal semantics for time in databases. *ACM TODS*, 8(2), June 1983.

[EN89] R. Elmasri and S. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, 1989.

[EW90] R. Elmasri and G. Wuu. A temporal model and language for er databases. In *IEEE Data Engineering Conference*, February 1990.

[Gad88] S. Gadia. A homogeneous relational model and query language for temporal databases. *ACM TODS*, 13(4), December 1988.

[GSsu] S. Gunadhi and A. Segev. Efficient Indexing Methods for Temporal Relations. Submitted to *IEEE Knowledge and Data Engineering.*

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD Conference*, May 1984.

[GY88] S. Gadia and C. Yeung. A generalized model for a temporal relational database. In *ACM SIGMOD Conference*, June 1988.

[KS89] C. Kolovson and M. Stonebraker. Indexing techniques for historical databases. In *IEEE Data Engineering Conference*, February 1989.

[LS89] D. Lomet and B. Salzberg. Access methods for multiversion data. In *ACM SIGMOD Conference*, June 1989.

[Lum84] V. Lum et al. Design dbms support for the temporal dimension. In *ACM SIGMOD Conference*, April 1984.

[NA87] S.B. Navathe and R. Ahmed. A temporal relational model and a query language. In *Information Sciences*, North-Holland, Vol. 49, No. 1, 2, and 3, 1989.

[OSD87] K. Ooi, B. McDonell and R. Sacks-Davis. Spatial kd-tree: Indexing mechanism for spatial database. In *IEEE COMPSAC 87*, 1987.

[RS87] D. Rotem and A. Segev. Physical organization of temporal data. In *IEEE Data Engineering Conference*, 1987.

[SA85] R. Snodgrass and I. Ahn. A taxonomy of time in databases. In *ACM SIGMOD Conference*, May 1985.

[SG89] A. Segev and H. Gunadhi. Event-join optimization in temporal relational databases. In *Very Large Databases Conference*, August 1989.

[Sno87] R. Snodgrass. The temporal query language tquel. *ACM TODS*, 12(2), June 1987.

[SS87] A. Segev and A. Shoshani. Logical modeling of temporal data. In *ACM SIGMOD Conference*, June 1987.