The Time Index*: An Incremental Access Structure for Temporal Databases

by V. Kouramajian, I. Kamel, R. Elmasri, and S. Waheed

T.R. 94-18
The Time Index+: An Incremental Access Structure for Temporal Databases

Vram Kouramajian* Ibrahim Kamel† Ramez Elmasri‡ Syed Waheed§

Abstract

In this paper, we propose a new indexing structure, called the Time Index+, which extends the incremental structure technique introduced in the Time Index [ElWK90, ElKG93]. The Time Index performs well for data that often overlaps and has a non-uniform distribution. However, it requires huge amounts of storage and suffers from degradation in update performance. The Time Index+ overcomes the deficiencies of the Time Index by proposing an efficient new storage model for partitioning logical buckets and by suggesting a graceful new method for handling object versions with long and very long time intervals.

We validate our claims for the efficiency of our new techniques by analyzing and comparing four indexing structures: the Time Index+, the Time Index, the Packed R-Tree [RoLe85, KaFa93], and the Parameterized R-Tree. Our simulation identifies important parameters, and shows how they affect the performance of the four considered indexing structures. These include mean of version lifespan, block size, query time interval length, and total number of versions.

Our simulation results show that: (1) The Time Index+ requires on average 60% less storage than the the Time Index but 50% more storage than the Packed R-Tree and the Parameterized R-Tree and (2) the Time Index+ provides an improvement in search time of 10% over the Time Index, of an order of magnitude over the Packed R-Tree, and of at least 100% over the Parameterized R-Tree.

*Office Of The Vice President for Research & Information Systems, Rice University, P.O. Box 1892, Houston, Texas 77251, Fax: (713) 523-0259, Telephone: (713) 523-0080, Email: vram@rice.edu.
†Department of Computer Science, University of Maryland, College Park, Maryland 20742, Telephone: (301) 405-2714, Email: kamel@umiacs.umd.edu.
‡Computer Science Engineering Department, The University of Texas at Arlington, Arlington, Texas 76019, Fax: (817) 273-3784, Telephone: (817) 273-3785, Email: elmasri@cse.uta.edu.
§Computer Science Engineering Department, The University of Texas at Arlington, Arlington, Texas 76019, Fax: (817) 273-3784, Telephone: (817) 273-3785, Email: waheed@cse.uta.edu.
1 Introduction

Temporal databases support both valid time, which records a history of changes in the real world, and transaction time, which records a history of updates made to the database. This permits users to query over the complete history of a given Universe of Discourse. However, the incorporation of time in database models has a profound impact on every facet of database implementation. An important facet that requires a complete re-evaluation is indexing techniques and search methods, which is the topic of this paper.

Current trends suggest that database systems will need to manage ever-larger volumes of data. This entails new indexing structures and faster search algorithms for achieving acceptable performance. Since temporal databases maintain larger volumes of data than conventional databases, their increasing use will only exacerbate this situation. One challenge will be to devise techniques that provide reasonable performance for historical data without affecting the performance of the database system in manipulating the current data. Another challenge will be to minimize the excessive storage overhead needed for indexing historical data.

Recent years have witnessed an increase in research on indexing techniques and storage structures for temporal data. These access structures can be broadly classified into three categories:

1. Modifications of regular $B^+-$Tree indexing structures [Come79, ElNa94] such as the Fully Persistent $B^+-$Tree [LaMa91].

2. Extensions of spatial indexing structures [Gutt84, OoMS87] such as the Segment $R-$Tree [KoSt91].

3. Techniques based on incremental structures such as the Time Index [ElWK90, ElKG93] and the Monotonic $B^+-$Tree [ElJK92].

Each of these access structures works well under different specific circumstances and provides efficient storage for a certain class of temporal data. For instance, the $AP-$Tree [SeGu93] and the Monotonic $B^+-$Tree [ElJK92] work only for append-only databases, whereas the Segment $R-$Tree [KoSt91] (based on the $R-$Tree [Gutt84] and the Segment Tree [Bent77]) expects database
support for pages of different sizes and performs well for data that has a uniform distribution and a fixed domain.

There has been considerable discussion concerning the applicability of conventional schemes for indexing interval data. Some researchers argue that $R$-Trees can explicitly accommodate interval data [KoSt91], whereas others insist that mapping intervals to their end points provides an efficient alternative [Lome91]. In this paper, we argue that temporal indices based on incremental structures provide superior solutions to both point and $R$-Tree based schemes.

A well-known incremental structure for handling temporal data is the Time Index [ElWK90, ELKG93]. This structure distinguishes among different types of object version pointers by partitioning them into logical buckets. The Time Index performs well for data that often overlaps and has a non-uniform distribution. However, the main drawback of this structure is that under some circumstances, it requires huge amounts of storage and suffers from degradation in update performance. One such circumstance is databases that include large numbers of object versions with very long time intervals.

In this paper, we propose a new indexing structure, called the Time Index$^+$, which extends the incremental structure technique introduced in the Time Index. The Time Index$^+$ overcomes the deficiencies of the Time Index by proposing an efficient new storage model for partitioning logical buckets and by suggesting a graceful new method for handling object versions with long and very long time intervals.

We validate our claims for the efficiency of our new techniques by analyzing and comparing four indexing structures: the Time Index$^+$, the Time Index, the Packed $R$-Tree [RoLe85, KaFa93], and the Parameterized $R$-Tree.

There are many variants of the $R$-Tree access structure. A recent survey can be found in [Same89]. We chose the Packed $R$-Tree and the Parameterized $R$-Tree in our simulation studies for two main reasons. Firstly, the Packed $R$-Tree uses spatial packing techniques to achieve excellent storage utilization. This allows to compare the Time Index$^+$ with an access structure which provides 100% space utilization. Secondly, the Parameterized $R$-Tree provides a better search performance than other $R$-Tree variants. This provides an indirect way of comparing our structures to previously proposed temporal indexing structures such as the Segment
$R$-Tree [KoSt91]. Therefore, this structure can serve as a good benchmark for comparing the Time Index$^+$ with $R$-Tree variants.

Our simulation identifies important parameters, and shows how they affect the performance of the four considered indexing structures. These include mean of version lifespan, block size, query time interval length, and total number of versions. Our simulation results confirm our claims and show that: (1) The Time Index$^+$ requires on average 60% less storage than the the Time Index but 50% more storage than the Packed $R$-Tree and the Parameterized $R$-Tree, and (2) the Time Index$^+$ provides an improvement in search time of 10% over the Time Index, of an order of magnitude over the Packed $R$-Tree, and of at least 100% over the Parameterized $R$-Tree.

The rest of this paper is organized as follows. Section 2 starts with a discussion of the time representation and the logical storage model used in our work; it then reviews the Time Index, the $R$-Tree, and the temporal operations used in our experimental simulations. Section 3 presents the Time Index$^+$ temporal indexing structure and discusses the main differences between the Time Index$^+$ and the Time Index. Section 4 presents experimental simulation results and observations. Finally, Section 5 concludes the paper.

2 Background

Research in time notation has centered around two basic symbolic models of time. One is the description of the state of the real world as continuous points in time; and the second as intervals of time. In Section 2.1 we discuss the representation of time used in this paper. The time dimension can be associated either with tuples or attributes. Section 2.2 describes our temporal representation model. We then review the Time Index and the $R$-Tree indexing structures in Sections 2.3 and 2.4, respectively. Finally, in Section 2.5 we define the temporal operations used in our experimental simulation studies.
2.1 Time Representation

Let $T = \{t_{-\infty}, \ldots, t_1, t_2, \ldots, t_{\text{now}}, t_{\text{now}+1}, \ldots, t_{+\infty}\}$ be a countably infinite set of totally ordered discrete time points\(^1\) (or chronons), where $t_{\text{now}}$ represents the current time point which is continuously increasing. We define a time interval, denoted by $[t_s, t_e)$, to be a set of consecutive chronons; that is, the totally ordered set $\{t_s, t_{s+1}, \ldots, t_{e-1}\} \subset T$.

2.2 Logical Storage Model

Our access structure is defined over a storage model, called TRDB (Temporal Relational DataBase), which is based on object (tuple) versioning [Snod87, NaAh89]. A TRDB consists of a collection of object versions; that is, $TRDB = \{e_{11}, e_{12}, \ldots, e_k, e_{11}, e_{12}, \ldots, e_{lm}\}$, where $e_{ij}$ is a version of an object $e_i$. Each object version $e_{ij}$ is augmented with two additional attributes $t_s$ (valid start time) and $t_e$ (valid end time), which define the version time interval $[t_s, t_e)$. This represents the fact that the object version $e_{ij}$ is valid from time point $t_s$ till the time point before $t_e$. An object version $e_{ij}$ with $e_{ij}.t_e = \text{now}$ is considered to be the current version of some object $e_i$. An object version $e_{ij}$ is immutable if the value assigned to its $e_{ij}.t_e$ is less than $\text{now}$, and is called a closed version. A version whose $e_{ij}.t_e \geq \text{now}$ is called an open version. (We will assume that there is at most one open version for each object.)

Object versions can be classified based on the length of their time intervals. In this paper, we distinguish among three types of object versions:

1. Short Lived Version (SLV): An object version whose time interval is short.

2. Long Lived Version (LLV): An object version whose time interval is long.

3. Very Long Lived Version (VLLV): An object version whose time interval is very long.

Distinguishing among object versions of type SLV, LLV, and VLLV is important for designing efficient temporal indexing structures. For instance, the Time Index, which treats similarly

\(^1\)A time point is really the smallest possible interval, which is considered atomic or indivisible. This has been called a chronon [JCGS92]. An application cannot distinguish between different points within the chronon. We will use the terms time point and chronon interchangeably.
<table>
<thead>
<tr>
<th>Name</th>
<th>Dept</th>
<th>Valid_Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>emp1</td>
<td>A</td>
<td>(0, 4)</td>
</tr>
<tr>
<td>emp1</td>
<td>B</td>
<td>(4, now)</td>
</tr>
<tr>
<td>emp2</td>
<td>B</td>
<td>(0, 6)</td>
</tr>
<tr>
<td>emp3</td>
<td>C</td>
<td>(0, 8)</td>
</tr>
<tr>
<td>emp3</td>
<td>A</td>
<td>(8, 10)</td>
</tr>
<tr>
<td>emp4</td>
<td>C</td>
<td>(2, 4)</td>
</tr>
<tr>
<td>emp4</td>
<td>A</td>
<td>(8, now)</td>
</tr>
<tr>
<td>emp5</td>
<td>B</td>
<td>(10, now)</td>
</tr>
<tr>
<td>emp6</td>
<td>C</td>
<td>(12, now)</td>
</tr>
<tr>
<td>emp7</td>
<td>C</td>
<td>(11, now)</td>
</tr>
</tbody>
</table>

**Figure 1: A Historical Database – The EMPLOYEE Table**

these three types of object versions, requires huge amount of storage; whereas the Time Index\(^+\), which introduces special techniques to handle object versions of type $LLV$ and $VL\ell V$, requires considerably less storage.

### 2.3 The Time Index

The *Time Index* [ElWK90, ElKG93] is defined over a *TRDB*. The basic idea behind the 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 an object version interval either starts or terminates. The Set of all Indexing Points $SIP$ is formally defined as follows:

$$SIP = \{ t \mid (\exists e_{ij}) \ ( (e_{ij} \in TRDB) \land ( (t = e_{ij}.t_s) \lor (t = e_{ij}.t_e) ) ) \}$$

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

Since all the indexing points $t$ in $SIP$ can be totally ordered, we can use a regular $B^+$-Tree [Come79, ElNa94] to index these time points. In a temporal database, there may be a
large number of version pointers associated with an indexing point, and many of these will be repeated from the previous indexing point. To reduce this redundancy and make the Time Index practical in terms of storage requirements, an incremental scheme is used. We keep pointers to all objects overlapping a time point in $SIP$ only if the time point is the first entry of a leaf node. We call this indexing point the leading entry of a leaf node. At each leading entry $t_i$ we will have three pointers to buckets: the plus bucket $SP(t_i)$, the minus bucket $SM(t_i)$ and the continuous bucket $SC(t_i)$. The bucket $SP(t_i)$ contains pointers to all object versions whose start time is $t_i$, and the bucket $SM(t_i)$ contains pointers to all object versions whose end time is $t_i$. The bucket $SC(t_i)$ contains pointers to all object versions that were valid at the previous indexing point and continue to be valid at the current indexing point $t_i$. Since most object versions will continue to be valid at the next indexing point, we keep only the incremental changes ($SP$ and $SM$ buckets) at the non-leading entries in a leaf node. For instance, in Figure 3 the non-leading entry at time point 10 includes only two pointers\(^2\) $SP(10) = \{+p_{51}\}$ and $SM(10) = \{-p_{32}\}$ in its incremental buckets indicating $e_{51}$ starts at point 10 and $e_{32}$ terminates at 10. Hence, a Logical Bucket $LB(t)$ at time point $t$ can be computed as follows:

\(^2\)We use $p_{ij}$ (called version pointer) to denote a pointer to object version $e_{ij}$.
Figure 3: Storing Incremental Changes in the Time Index Buckets

\[ LB(t) = \left( SC(t_i) \cup \left( \bigcup_{t_i \in SIP, \ t_i \leq t} SP(t_i) \right) \right) - \left( \bigcup_{t_i \in SIP, \ t_i < t} SM(t_i) \right) \]

For a complete and formal description of the Time Index access structures, please see [ElKG93].

2.4 The R–Tree

The R–Tree [Gutt84] is one of the most promising multi-dimensional indexing structures. It is the extension of the B–Tree for multi-dimensional objects. A geometric object is represented by its Minimum Bounding Rectangle (MBR). Non-leaf nodes contain entries of the form \( (\text{ptr}, R) \) where \( \text{ptr} \) is a pointer to a child node in the R–Tree; \( R \) is the MBR that covers all rectangles in the child node. Leaf nodes contain entries of the form \( (\text{objptr}, R) \) where \( \text{objptr} \) is a pointer to the object description, and \( R \) is the MBR of the object. Figure 4 illustrates data rectangles (in black), organized in an R–Tree with fanout 3. Figure 5 shows the file structure for the same R–Tree, where nodes correspond to disk pages.

Subsequent work on R–Trees includes the work by [Gree89], the \( R^+ \)–Tree [SeRF87], and finally, the \( R^* \)–Tree [BKSS90], which seems to have the best performance among the R–Tree
Figure 4: Data (Dark Rectangles) Organized in an $R$-Tree with Fanout=3

Figure 5: The File Structure for the $R$-Tree of the Previous Figure (fanout=3)
variants. Packed R-trees (100% space utilization) is first proposed in [RoLe85]. [KaFa93] proposed Hilbert packed R-tree. For one dimensional data the two methods are almost identical.

In this work, we have used the Packed R-Tree variant for the following two reasons: (1) the space utilization in Packed R-Trees is 100% which helps alleviate the storage requirement in temporal applications, and (2) although Packed R-Trees are used to index static data in spatial applications, they can be used to index dynamic (append-only) temporal data since insertion for (append-only) temporal data occurs in monotonically increasing time order.

In the Packed R-Tree, each object version is represented with a line segment \([t_s, t_e]\) or equivalently \([t_s, t_e - 1]\). As new object versions are created, they are inserted in the rightmost leaf node until the node becomes full; then, a new leaf node is created. Higher level nodes of the tree are created in a similar way. The resulting R-Tree will be fully packed, with possible exception of the last node at each level. Thus; the utilization is \(\approx 100\%\).

Excessive overlapping of temporal data adversely affects the performance of the R-Trees. If the query interval falls inside the intersection area of two nodes, the two subtrees rooted by these nodes should be searched. To avoid the overlap in the R-Tree, we first transform the one-dimensional line segments into points in two-dimensional space using the starting and ending points of the line segments. Then, we use Packed R-Trees to store the two-dimensional points (in the parameter space). For example, Figure 6 shows two one-dimensional line segments representing objects \(A\) and \(B\), which are mapped into points in the two-dimensional space. Note that all the line segments are mapped to the upper triangle (the shaded area). In the following, we will call this method Parameterized R-Tree. This way we can avoid the negative effect of the excessive overlap in data segments. (Similar idea was proposed in [FaRo91] using \(B^+\)-Tree as the underlying index structure.)

### 2.5 Temporal Operations used in Simulation Studies

Temporal and spatial indexing structures can be used to improve the efficiency of temporal operations for various temporal queries, such as those expressed in TQUEL [Snod87], TSQL [NaAh89] and TGORDAS [ElWK93]. Below, we review the operators, DURING and INCLUDE, used in our experimental simulation studies. We considered these two operations
Figure 6: An Example of a Parameterized $R$–Tree

since they require a large number of disk I/O operations and represent a good benchmark for temporal operations [ChSe93]. (For a complete description of temporal search operations, please see [EIKG93].)

- The time interval operator $DURING([t_a, t_b])$ returns all object versions that are included in $[t_a, t_b]$; that is, the set of object versions that start after the start time of a given interval $[t_a, t_b]$ and end before the end time of the time interval. Formally, it is defined as:

$$DURING([t_a, t_b]) = \{ e | (e.t_s > t_a) \land (e.t_e \leq t_b) \}$$

This operation is implemented for the Time Index as follows: Initially, compute the set $LP$ of incremental plus pointers between $t_a$ to $t_b$ and the set $LM$ of incremental minus pointers$^3$ in the same time interval. Then, get the required set of object versions by incrementally calculating the intersection of $LP$ and $LM$.

This operation is implemented for the (Packed) $R$–Tree as follows: (S1) Search nonleaf nodes: Invoke Search for every entry whose MBR intersects the query segment $[t_a, t_b]$, and (S2) Search leaf nodes: Report all the entries completely covered by the query segment $[t_a, t_b]$ as candidate.

$^3$Incremental plus pointers are stored in $SP$ buckets, whereas incremental minus pointers are stored in $SM$ buckets.
In the case of a Parameterized R–Tree, the *DURING* query $q_D$ is mapped to a region query (shaded area) as shown in Figure 7.

- The time interval operator $\text{INCLUDE}([t_a, t_b])$ returns all object versions that contain a given time interval $[t_a, t_b]$. Formally, it is defined as:

$$\text{INCLUDE}([t_a, t_b]) = \{ e \mid (e.t_s < t_a) \land (e.t_c > (t_b + 1)) \}$$

This operation is implemented for the Time Index as follows: Initially, compute the set $LB$ at $t_a$ by using continuous, incremental plus and incremental minus pointers as described in Section 2.3. Then, subtract *incrementally* from $LB$ the set of incremental minus pointers between the time interval $t_a$ and $t_b$. Finally, retrieve the object versions by using the pointers in $LB$.

This operation is implemented for the (Packed) R–Tree as follows: (S1) *Search nonleaf nodes*: Invoke Search for every entry whose MBR completely covers the query segment $[t_a, t_b]$, and (S2) *Search leaf nodes*: Report all the entries that completely cover the query segment $[t_a, t_b]$ as candidate.

In the case of a Parameterized R–Tree, the $\text{INCLUDE}$ query $q_I$ is mapped to a region query (shaded area) as shown in Figure 8.

3 The Time Index$^+$

One of the main criticism of the Time Index is the huge amount of storage needed for $SC$ buckets. This is mainly true for databases that have a large number of object versions with long and very long object versions. The following example illustrates the problem.

Suppose that the average number of object versions in a database is 100,000, which implies that each $SC$ bucket will contain on average 100,000 version pointers. Also assume that every hour on average a new leaf node is added to the Time Index, 1,000 new object versions are created, and 1,000 object versions are closed. After one day, there will be 24 $SC$ buckets each with 100,000 version pointers, 24,000 new open versions, and 24,000 new closed versions.
Figure 7: *DURING* Query for Parameterized *R*-Tree; the shaded area represents the query $q_D$ in the parameter space

Figure 8: *INCLUDE* Query for Parameterized *R*-Tree; the shaded area represents the query $q_I$ in the parameter space
Hence, each day the total storage needed for $SC$ buckets will be 14.4 GBytes ($24 \times 100,000 \times 6 = 14,400,000$ Bytes, where 6 is the size of a version pointer), for $SP$ buckets will be 144 KBytes, and for $SM$ buckets will be 144 KBytes.

Although the above example describes a worst-case scenario, it highlights the fact that the Time Index can be a vulnerable structure and impractical in many industrial strength applications. The situation is similar to materialized views where intermediate results of computations on base relations are kept [BILT86]. Various solutions were proposed for managing the size of intermediate results without greatly affecting the performance of queries over materialized views [BICL89]. Here, the changes in the base relations correspond to $SP$ and $SM$ buckets, and the materialized intermediate views to $SC$ buckets.

The Time Index$^+$ has a similar structure to that of the Time Index. These two access structures distinguish among different types of object version pointers by partitioning them into three logical buckets: $SC$, $SP$, and $SM$. However, the main difference between the Time Index$^+$ and the Time Index is the way intermediate results are maintained; i.e., $SC$ buckets. The Time Index requires a large disk storage to keep intermediate results, whereas the Time Index$^+$ controls the size of intermediate results. The next two subsections present techniques to handle redundancy in $SC$ buckets.

### 3.1 First Technique: Controlling Redundancy in Version Pointers to $LLVs$

This technique allows the elimination of redundancy in version pointers which are stored in $SC$ buckets and point to $LLVs$. The main characteristic of $LLVs$ is that they are referenced from a small number of consecutive $SC$ buckets. The Time Index$^+$ combines these duplicates without affecting the performance of search operations.

Let $SC_i$ ($1 \leq i \leq k$) be the $SC$ bucket associated with the $i$th leaf node, where $k$ is the number of leaf nodes. Adjacent buckets $SC_j$ and $SC_{j+1}$ ($1 \leq j < k$) may include a large number of duplicate copies of version pointers. The Time Index$^+$ combines duplicate copies in $SC_j$ and $SC_{j+1}$ buckets, as we shall discuss below.

Before proceeding to describe formally our index structure, we define some additional no-
tation that will be useful in our discussion. We define the position of a node \( N \), denoted by \( pos(N) \), for each tree level recursively as follows: \( pos(N) = pos(left\_sibling(N)) + 1 \), where the function \( left\_sibling(N) \) returns the sibling to the left of node \( N \). The position of the leftmost node at each level is equal to zero. We say that the position of a node \( N \) is even if \( pos(N) \) is even and the position of a node \( N \) is odd if \( pos(N) \) is odd.

Each leading entry \( t_i \) of a leaf node in the Time Index* has four pointers to buckets: \( SCE(t_i) \) (for \( SC \) Exclusive), \( SCS(t_i) \) (for \( SC \) Shared), \( SP(t_i) \), and \( SM(t_i) \) (see Figure 9). Similar to the Time Index, the bucket \( SP(t_i) \) contains pointers to object versions whose start time is \( t_i \), and the bucket \( SM(t_i) \) contains pointers to object versions whose end time is \( t_i \). The contents of the buckets \( SCE(t_i) \) and \( SCS(t_i) \) depend on whether the position of the leaf node is odd or even. The idea is that consecutive odd–even pairs of leaf nodes will share one \( SCS \) bucket, thus reducing duplication of version pointers.

For odd leaf nodes, the bucket \( SCE(t_i) \) contains pointers to object versions that were valid at the previous indexing point and continue to be valid at the current indexing point \( t_i \); and the bucket \( SCS(t_i) \) contains pointers to object versions that were valid at the previous indexing point and continue to be valid at the next leading entry (i.e. the leading entry of the sibling node to the right). For even leaf nodes, the bucket \( SCE(t_i) \) contains pointers to object versions that were valid at the previous indexing point, continue to be valid at the current indexing point \( t_i \), but were not not valid at the previous leading entry (i.e. the leading entry of the sibling node to the left); and the bucket \( SCS(t_i) \) contains pointers to object versions that were valid at the previous leading entry and continue to be valid at the current indexing point \( t_i \).

Each \( SCE \) bucket is associated with a single leaf node, whereas each \( SCS \) bucket is associated with two adjacent leaf nodes. Thus, two adjacent leading entries point to the same \( SCS \) bucket, which results in substantial reduction in storage requirements. This is illustrated in Figure 9, which shows (partial) structures of leaf nodes, where each leading entry points to two types of \( SC \) buckets: \( SCE \) and \( SCS \).
3.2 Second Technique: Controlling Redundancy in Version Pointers to VLLVs

This technique allows the elimination of redundancy in version pointers which are stored in SC buckets and point to VLLVs. The main characteristic of VLLVs is that they are referenced from a large number of consecutive SC buckets. The Time Index+ combines these duplicate version pointers and stores them in a new type of continuous bucket, called SCI, which is referenced from an internal node. This is similar to the idea of the Segment Tree [Bent77] where large intervals are stored in non-leaf nodes. This technique reduces storage allocated for version pointers that reference VLLVs and improves the performance of interval queries that span over a large number of tree nodes.

A version pointer whose corresponding object version spans a sub-tree is stored in a SCI bucket associated with the root of the sub-tree. Figure 10 shows how continuous buckets are rearranged so that they are referenced from internal (SCI) and leaf nodes (SCE and SCS). In some cases, a version pointer may be stored in more than one continuous bucket where these buckets are referenced from different levels of the tree. This is the case of an object version
which spans some high level node(s) but fails to completely span the following node at the same level. Although, storing the same version pointer on different levels is similar to the concept of cut which creates spanning and remnant portions [KoSt91], The Time Index+ does not suffer from the problem of relating spanning/remnant index records. This is because base version pointers are never cut since they are accessed only from leaf nodes through $SP$ and $SM$ buckets.

Theoretically, the Time Index+ allows continuous buckets to be referenced from any level of the tree. However, in practice, there is no need to associate continuous buckets with more than two levels (i.e. leaf nodes and their parents). This is because continuous buckets referenced
from high level internal nodes store a small number of version pointers. In addition, maintaining these continuous buckets may result in expensive update operations.

SCSI buckets can be maintained in the following three ways:

1. Each time an object version is inserted, updated, or closed, SCSI bucket are updated if necessary.

2. SCSI buckets are updated periodically during idle periods.

3. SCSI buckets are updated while being transferred to optical disks. (For a thorough discussion of migration techniques, please see [ElkK93].)

The update performance and memory requirements of these three methods are different. The second and third methods use tags to indicate only modified continuous buckets and reduce substantially CPU and I/O usage. The first method requires frequent rearrangement of the continuous buckets and results in very expensive update operations. The second method is attractive if transfer to optical disk is very seldom. The third method is the best one since it does not require additional read and write disk operations.

This technique also speeds up interval based search operations since SCSI buckets enable faster retrieval of object versions which span over a large number of tree nodes. However, the search operations defined for the Time Index need to be modified to take advantage of the new structures of continuous buckets (SCE, SCS, and SCI).

Firstly, the computation of logical buckets must take into account continuous buckets that are associated with non-leaf nodes as shown below:

\[
LB(t) = \left( SC(t_i) \cup \left( \bigcup_{t_i \in SIP, t_i \leq t} SP(t_i) \right) \right) - \left( \bigcup_{t_i \in SIP, t_i < t} SM(t_i) \right) \cup SCI(parent(t_i))
\]

where SCI(parent(t_i)) corresponds to the continuous bucket associated with the parent of the leaf node that has t_i as a leading entry. The computation of logical buckets LB for the Time Index + requires less memory buffer size and CPU cycles than that of the Time Index. This is due to the fact that the SCI bucket is not involved in the differential computation, but just
added to the final result. (We haven't included memory buffer size and CPU utilization in our simulation studies.)

Secondly, the temporal search algorithms described in [ElKG93] can take into account the fact that whenever queries span over internal nodes, the SCI buckets corresponding to these nodes can be used to reduce the search space. To examine the benefit of this concept, we have modified the search algorithms of the Time Index and observed in our simulation results an improvement of 10% in response time (see Section 4).

4 Simulation Results

In this section, we report on the experimental results of our simulation. The performance of different indexing structures was evaluated based on two metrics: disk storage requirement and response time. In our experiments, we assumed that CPU time was negligible and response time was measured in terms of disk access.

The main parameters put to study include mean of object version lifespan, mean and percentage of SLVs, mean and percentage of LLVs, mean and percentage of VLLVs, total number of object versions, and time length for an interval query. A temporal database was assumed to be implemented with a record-based storage system [AhSn88] which supports object (tuple) versioning [Snod87, NaAh89]. The temporal database was initialized with 1,000 objects in all simulation runs and the database was populated with 150,000 object versions. Internal and leaf nodes were assumed to be equal to a disk block, where disk block size was assumed to be equal to 1,024 bytes.

The lifespan of an object version was generated using a normal distribution. An exponential distribution was used to generate the close of an object version and insert a new object events. In the following subsections, we present experimental performance results comparing The Time Index+, The Time Index, the Packed R-Tree, and the Parameterized R-Tree. We also, used $R^*$ tree and quadratic R-tree in our experiments. But their performance were consistently inferior to the Packed and Parameterized R-tree and they are not shown in the figures. (For space limitations, we only describe a limited number of experimental results. For a complete
discussion, please see [Kour94]).

4.1 Analyzing Response Time

In this section, we compare the response time of the four access structures. We modified the search algorithms of the Time Index to take advantage of the extensions proposed in this paper and noticed a substantial improvement in search performance. The details are given below.

In Figure 11, the effect of VLLVs was studied on response time of INCLUDE queries by increasing the number of VLLVs from 10% to 50% of the temporal database size. In this run, the query interval size was assumed to be 4,000. Comparing the four graphs, we notice that: (1) on the average, the Time Index+ results in 10% improvement in search time over the Time Index, (2) the Time Index+ always has an order of magnitude of performance advantage over both R-Tree variants, and (3) both R-Tree variants have the same search time. The performance improvement of the Time Index+ over the Time Index is due to the fact that the new search algorithms take advantage of continuous buckets associated with internal nodes, and thus reduce the search space.

In Figure 12, the effect of query interval size was studied on response time of DURING queries. The query interval size was assigned the values 1,000, 2,000, 3,000, 4,000, 5,000, 6,000, 7,000, and 8,000\footnote{The entire search space was 200,000 units.}. In this simulation run, 20% of the database was assumed to be VLLVs, 30% to be LLVs, and 50% to be SLVs. The main observation from the graph is that Time Index+ has the best response time among the four methods. With a closer look at the graph we draw the following observations: (1) the Time Index+ always has an order of magnitude of performance advantage over Packed R-Tree, (2) the savings of the Time Index+ over the Parameterized R-Tree increases with the interval size. (3) the Time Index+ shows less than 10% improvement in search time over the Time Index.

4.2 Analyzing Storage Requirement

In order to compare the storage requirements of the four access structures, we varied two ratios in the temporal database: (1) VLLVs over SLVs and (2) LLVs over SLVs. We consistently
observed a similar behavior in our simulation runs: (1) the storage requirements of Packed R-Trees was less than that of the Time Index+ and the Time Index and (2) the Time Index+ required substantially less storage than the Time Index. We also noticed quite a few important differences among the different runs, which we mention below.

In Figure 13, the effect of VLLVs was studied on storage requirement by increasing the number of VLLVs from 10% to 50% of the temporal database size. Comparing the four graphs, we notice that: (1) the Time Index+ requires around 50% less storage than the Time Index when the VLLVs are 10% of the database but 70% less storage when the VLLVs are 50%, (2) both the Time Index+ and Time Index storage requirements grow with greater percentage of VLLVs, and (3) the Time Index+ requires around 45% more storage than Packed R-Trees when the VLLVs are 10% of the database but 75% more storage when the VLLVs are 50%. The large advantage of the Time Index+ over the Time Index is due the fact that the Time Index+ associates continuous buckets with internal and leaf nodes. Packed R-Trees require
Figure 12: Response Time for During queries versus Interval Query Size

less storage than the Time Index and the Time Index\(^+\) because they don’t maintain results of incremental computation and their storage utilization of tree nodes is 100%.

In Figure 14, the effect of \(LLV\)'s was studied on storage requirement by increasing the number of \(LLV\)'s from 10% to 50% of the temporal database size. Comparing the four graphs, we notice that: (1) the Time Index\(^+\) requires around 60% less storage than the Time Index independent of the number of \(LLV\)'s in the database, (2) both the Time Index\(^+\) and Time Index only experience a small increase in storage requirements with greater percentage of \(LLV\)'s, (3) the impact of \(LLV\)'s is not as severe on storage utilization as compared to \(VLLV\)'s (see Figures 13 and 14), and (4) the Time Index\(^+\) storage requirement is on average 50% more than the \(R\)-Tree. The ratio between the storage utilization of the Time Index and the Time Index\(^+\) remains constant since an increase in the number of \(LLV\)'s is matched by a decrease in the number of \(SLV\)'s. Thus, an increase in the number of version pointers to \(LLV\)'s implies a decrease in the number of version pointers to \(SLV\)'s.
5 Conclusions

In this paper, we proposed a new indexing structure, called the Time Index\(^+\), which extends the incremental structure technique introduced in the Time Index. The Time Index\(^+\) overcomes the deficiencies of the Time Index by proposing an efficient new storage model for partitioning logical buckets and by suggesting a graceful new method for handling object versions with long and very long time intervals.

We then validated our claims for the efficiency of our new techniques by analyzing and comparing four indexing structures: the Time Index\(^+\), the Time Index, the Packed R–Tree, and the Parameterized R–Tree. Our simulation identified important parameters, and showed how they affect the performance of the four considered indexing structures. These include mean of version lifespan, block size, query time interval length, and total number of versions.
Figure 14: Storage Requirement with Varying LLVs

The simulation results showed that:

1. The Time Index\textsuperscript{+} provides an improvement in search time of 10% over the Time Index, of an order of magnitude over the Packed $R$-Tree, and of at least 100% over the Parameterized $R$-Tree.

2. The Time Index\textsuperscript{+} requires on average 60% less storage than the the Time Index but 50% more storage than the Packed $R$-Tree and the Parameterized $R$-Tree.

References


