LBL-28798
UC-405

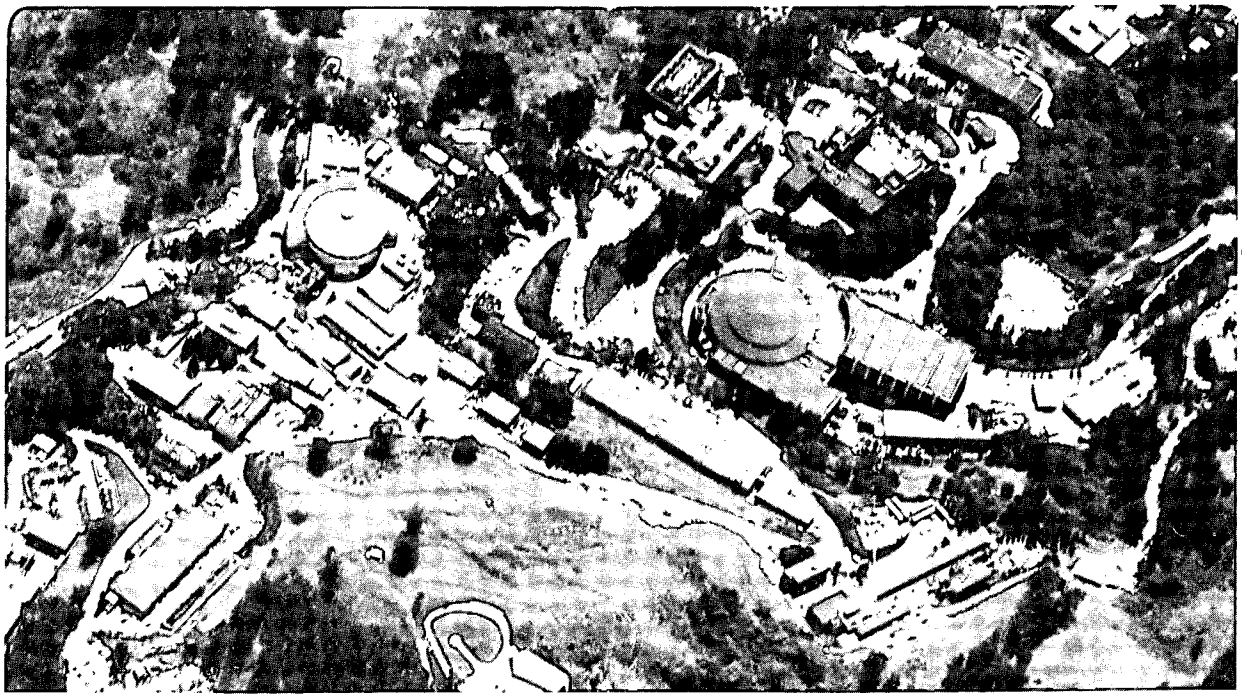
Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Information and Computing Sciences Division

Efficient Indexing Methods for Temporal Relations

H. Gunadhi and A. Segev

March 1990



Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

DISCLAIMER

LBL-28798

EFFICIENT INDEXING METHODS FOR TEMPORAL RELATIONS

Himawan Gunadhi and Arie Segev

Computing Science Research & Development
Information & Computing Sciences Division
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, California 94720

and

Walter A. Haas School of Business
The University of California, Berkeley
Berkeley, California 94720

March 1990

This work was supported by the Director, Office of Energy Research, Applied Mathematical Sciences Research Program, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

LBL-28798
March 1990

EFFICIENT INDEXING METHODS FOR TEMPORAL RELATIONS

Himawan Gunadhi and Arie Segev

Walter A. Haas School of Business
University of California and
Computing Sciences Research and Development Department
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Abstract

The size of temporal databases and the semantics of temporal queries pose challenges for the design of efficient indexing methods. We look at the primary issues that affect the design of indexing methods, and propose several structures and algorithms for specific cases. We develop indexing methods for time based queries, queries on the surrogate or time-invariant key and time, and temporal attribute and time. In the latter case, we present several methods by which to partition the time-line, in order to balance the distribution of tuple-pointers within the index. We analyze our methods against alternatives, and present appropriate empirical results.

Index Terms -- Temporal databases, indexing, physical organization, query processing.

This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

1. INTRODUCTION

The main focus of research in temporal databases has been on the modeling and representation of temporal data [4-6], [13], [16-20], [22]. Until recently, concerns about the performance of temporal databases have, to a large extent, been ignored. There are two major issues that separate the physical design of temporal databases from conventional ones: the size of data, which may be several orders of magnitude larger; and the extended semantics of temporal queries, e.g., [5], [17], [8], [15], which increase the complexity of query processing. Previous studies in physical temporal database design tended to focus on narrowly defined objectives. Methods to index the surrogate (time-invariant) key with sequential access to history were explored by [12], [23], [10]. Partitioning and indexing of static databases was studied in [7], [14]. The performance of traditional indexing methods for temporal queries was investigated by [2], [21], and methods to organize current and historical versions proposed in [1], [3].

In this paper, we investigate methods of indexing time-dependent data, within the context of a a first normal form (1NF) relational representation of temporal data. A framework by which to develop and evaluate physical design architectures for temporal databases is given. We subdivide a temporal database into two or possibly three segments, based on the time-related view or version of the data, i.e., current, a moving window (if it is defined), and the archived history. The major factors that influence design are (1) the physical organization of each portion, (2) the index construction on each portion, and (3) functionality of queries on the database. Several interesting cases are subsequently investigated: (1) Dynamic structures for surrogate and time indexing (ST); (2) Static and dynamic partitioning algorithms for the time-line in the context of temporal attribute and time indexing; and (3) Time-indexing for append-only database. In all the designs, the focus is on the role of the time attribute.

The paper is organized as follows. In Section 2, we discuss the relational representation of data in the temporal context, followed by a framework for analyzing the physical design of a temporal database. In Section 3, we introduce the AP-tree, which is designed for time-based query operations on an append-only database, and is subsequently incorporated into our surrogate-time index of Section 4. In Section 5, we discuss the issue of indexing time and one or more temporal attributes, and delve into techniques for efficiently partitioning the time line. In Section 6 conclusions and future directions are outlined.

2. FUNDAMENTAL CONCEPTS AND DESIGN CONSIDERATIONS

In this section, we look at the fundamental approach undertaken for the paper. We adopt a tuple versioning approach with interval time representation, and look at the possible architectures of a temporal database.

2.1. Relational Representation of Temporal Data

A convenient way to look at temporal data is through the concepts of *Time Sequences* (*TS*) and *Time Sequence Collection* (*TSC*) [16]. A *TS* represents a history of a temporal attribute(s) associated with a particular instance of an entity or a relationship. The entity or the relationship are identified by a surrogate (or equivalently, the *time–invariant key* [13]). For example, in the MANAGER relation of Fig.1, the manager history of employee #1 is a *TS*. A *TS* is characterized by several properties, such as the time granularity, lifespan, type, and interpolation rule to derive data values for non-stored time points. In this paper, we are concerned with two types – *stepwise constant* and *discrete*. Stepwise constant (*SWC*) data represents a state variable whose values are determined by events and remain the same between events; the salary attribute represents *SWC* data. Discrete data represents an attribute of the event itself, e.g., number of items sold. Time sequences of the same surrogate and attribute types can be grouped into a time sequence collection (*TSC*), e.g. the manager history of all employees forms a *TSC*.

There are various ways to represent temporal data in the relational model; detailed discussion can be found in [17]. In this paper we assume first normal form relations (1NF). Fig. 1 shows two ways of representing *SWC* data.

MANAGER	E#	MGR	T_S	T_E
	E1	TOM	1	5
	E1	MARK	9	12
	E1	JAY	13	20
	E2	RON	1	18
	E3	RON	1	20

COMMISSION	E#	C_RATE	T_S	T_E
	E1	10%	2	7
	E1	12%	8	20
	E2	8%	2	7
	E2	10%	8	20

(a) time-interval representation

MANAGER	E#	MGR	T
	E1	TOM	1
	E1	∅	6
	E1	MARK	9
	E1	JAY	13
	E2	RON	1
	E2	∅	19
	E3	RON	1

COMMISSION	E#	C_RATE	T
	E1	∅	1
	E1	10%	2
	E1	12%	8
	E2	∅	1
	E2	8%	2
	E2	10%	8

(b) time-point representation

Figure 1: Representing Step-Wise Constant Data with Lifespan = [1, 20]

The representations can be different at each level (external, conceptual, physical), but we are concerned with the tuple representation at the physical level. Fig. 1(a) shows time-interval representation of the MANAGER and COMMISSION relations. On the other hand, the representation in Fig. 1(b) stores data only for event points and requires explicit storage of *null* values to indicate the transition of the state variable into a non-existence state.

Also, the tuples should be ordered by time in order to determine the values between two consecutive event points. Both representations require the use of the lifespan metadata; it is required for the time-interval representation since we do not store non-existence nulls explicitly, e.g., the lifespan is needed in order to correctly answer the query 'what was the commission rate of E2 at time 12?'. We select time-interval representation for the purpose of generalization, although the indexes can be adapted to event-point representation.

We use the terms *surrogate* (S), *temporal attribute* (A), and *time attribute* (T) when referring to attributes of a relation. For example, in Fig. 1, the surrogate of the MANAGER relation † is E#, MGR is a temporal attribute, and T_S and T_E are time attributes. We assume that all relations are in first temporal normal form (1TNF) [17]. 1TNF requires that for each combination of surrogate instance, time point in the lifespan and temporal attribute (or attributes) there is at most one temporal value (or a unique combination of temporal values). Note that 1NF does not imply 1TNF, e.g., the relation COMMISSION in Fig. 1(a) would not be in 1TNF if for any surrogate instance there were two tuples with the same commission rate value and intersecting time intervals. The temporal normal form (TNF) definition given in [13] is deemed too restrictive, since enforcing it would mean that most relations will contain only a single temporal attribute.

2.2. Lifespan and Organization of a Relation

The physical design requirements of a temporal relation may be viewed according to its lifespan relative to current time. Let the lifespan of relation r be identified by a pair of start-end time-points $LS_r.START$ and $LS_r.END$. Current data, i.e., those tuples with $T_E = NOW$, form the current snapshot; all others make up the history of r. In many instances, a *moving time window* (*MTW*) may be defined on the relation, defined by the closed interval $[max\{NOW-INT+1, LS_r.START\}, NOW]$, where INT is the length of the window. A relation may thus be subdivided into these three segments, or versions -- current snapshot, *MTW* history, and *archived* versions.

Several design issues result from this dichotomy. First is the physical organization of each version, i.e., whether they should be organized jointly or separately. We adopt the position that the data that is covered by the *MTW* should be physically separated from the archived history, which is presumed to be less frequently needed to answer queries. The indexes should also be separated, so as to improve efficiency by reducing the size of each. There are other associated design issues related to this approach, including the media selection and indexing, e.g., the adoption of WORM optical disks for archival and design of suitable indexes for such media; and the issue of migration strategies for moving tuples from one level of the hierarchy to the next, e.g.

† We refer to the data construct as a 'relation', but we mean a 'temporal relation'. It is different from a standard relation because of the associated meta-data.

'vacuuming' techniques in [23], [24]. In this paper, our focus is on indexing only.

For the time window, current data may be separated or stored together with the historical portion. There are several tradeoffs to either approach. If they are separated, overheads will always be incurred in order to move them from one level to another in storage. On the other hand, all current snapshot queries should retain the efficiency of conventional database management systems. Conversely, if the tuples are stored together, the efficiency of snapshot queries can only be preserved if data has been clustered according to the time dimension, and the indexes are specifically designed to allow rapid retrieval of current tuples.

2.3. Functionality of Queries

Various operators for temporal databases are discussed in [16],[17] within the context of the *TSC*, in [13], [19] with respect to an extended relational model, and in [8], [15] for joins involving time. The functionality of queries is our primary interest, i.e., how data should be organized and retrieved at minimum cost. Cost is measured in both storage and disk access time. The following are the main types of temporal queries:

(1). *ST* Queries: What would have been primary key queries in a conventional relation, is now a query on a conjunction of S and T. Functionally, there are four distinct time specifications, i.e. the current time *NOW*, an arbitrary point, an interval $[t_s, t_e]$, or the whole lifespan of the entity. A specific time point can also be further qualified as a T_S or T_E time attribute in the relation, which is semantically different than the specification of an arbitrary time or event point.

(2). *AT* Queries: These are queries based on the temporal attribute value at some point or interval in time.

(3). T Queries: These apply to queries which are primarily qualified on time, i.e., for such queries as aggregates, time ordering or where initial restriction on T is more selective for a conjunctive query with S or A.

(4). Multidimensional Queries: These queries can have arbitrary conjunctions on relational attributes.

2.4. General Notations

Table 1 below summarizes notations which are used throughout the paper.

3. THE APPEND-ONLY TREE

In this section we present a multiway tree to index time values, that is a modification of the standard B^+-tree; it was first introduced to optimize *event−join* operations [15]. The primary reason that it is most suitable

Variable(s)	Description
$LS_r.START$, $LS_r.END$	Start, end times of r's lifespan
$d(X)$	Order of index X
v, V	Search values for an index
$h_X(Y)$	Height of index type X for query type Y
N_v	Total number of keys in a given index
N_{dv}	Total number of keys being deleted from an index
N_r, N_S, $\bar{N}_{T\|s}$	Size of relation (tuples), surrogate domain and time-sequence (tuples)
$l(r)$, $l(S)$, $l(T)$, $l(PTR)$	Byte size of a tuple, surrogate, time attributes and tuple pointer

Table 1: Summary of Notations

for append-only databases is the fact that it is designed to increase node loading and ease insertions at the expense of complicating deletions. This structure is also useful in the case of the ST nested index which will be introduced in the following section, where the values of time arrive in order.

3.1. Definition

The *Append–only* Tree, which we call AP-tree, is a multiway search tree that is a hybrid of an *ISAM* index and a B^+-tree. The leaves of the tree contain all the T_S values in the relation; for each T_S value, the leaf points to the last (towards the end of the file) tuple with the specific T_S value. Each non-leaf node indexes nodes at the next level. Fig. 2 gives an example of an AP-tree of order 4.

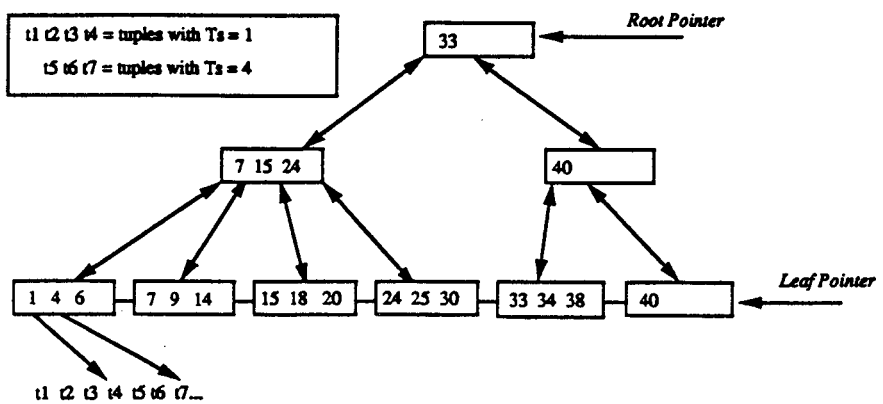


Figure 2: Example of AP-tree of Order 4

Note that the pointer associated with a non-leaf key value, with the exception of the first pointer, points to a

node at the next level having this key value as the smallest node value. The significance of this decision is explained later on. Access to the tree is either through the root or through the right-most leaf. The *AP*-tree is different than the B^+-tree in several respects. First, if the tree is of degree d, there is no constraint that a node must have at least $\lceil d/2 \rceil$ children. Second, there is no node splitting when a node gets full. Third, the online maintenance of the tree is done by accessing the right-most leaf.

Given the premise that deletions are treated as offline † storage management, only the right-hand side of the tree can be affected. The only online transactions that affects the T_S values in an append-only database is appending a new tuple. In most cases, just the right-most leaf is affected, either a pointer is updated or a new key-pointer pair is added, but if it is full a new leaf has to be created to its right, and in the worst case either a new node along the path from the root to the new leaf, or a new root have to be created. These conditions lead to the following formal definition.

DEFINITION. An *AP-tree of order d* is a d-way tree in which

1. All internal nodes except the root have at most d (non-empty) children, and at least 2 (non-empty) children.
2. The number of keys in each internal node is one less than the number of its children, and these keys partition the keys in the children in the fashion of a search tree.
3. The root has at most d children, and at least 2 if it is not a leaf, or none if the tree consists of the root alone.
4. For a tree with n children ($n > 0$), and a height of h, each of the first $n-1$ children is the root of a subtree where
   (i) all leaves in each subtree are on the same level,
   (ii) all subtrees have a height of $h-1$,
   (ii) each subtree's internal nodes have $d-1$ keys.
   For the rightmost subtree rooted at the *nth* child:
   (i) it has a height of at least 1, and no more than $h-1$,
   (iii) when its height reaches $h-1$, and each internal node has $d-1$ keys, the next key insertion into the *AP*-tree creates a new right subtree.

The fourth point in the definition ensures that the *AP*-tree is balanced for all but the right subtree, and that this subtree will continue to grow until it reaches the same height and maximal node loading as its siblings,

† Reorganizing the tree to reflect deletions can be done during idle periods or low load periods. All the procedures function correctly regardless of the timing; the only issue is performance.

before a new right subtree is created. There are various implementation details of the AP-tree that differ from B-trees and $ISAM$ indexes, and these will be described as we proceed.

3.2. Searching an AP-tree

The AP-tree's search procedures is similar to that of a standard B^+-tree for secondary key indexing, except for two primary differences. First, direct access to the rightmost leaf is available, so that insertions can be made rapidly; since access to the tree for insertions is always made through the rightmost leaf, two-way pointers are used to link a node with its parent †. Second, the semantics of time-based queries allow modifications aimed at improving the efficiency of retrieval. A search through the tree may be based on the T_E value or a given time point, which requires a backward scan of tuples starting from those with a key value $v^+ = \max\{v \mid v \leq V\}$. By maintaining metadata about the first T_S value and the largest T_E value indexed by the tree, the fact that all but the first child in an internal node indexes lower level nodes based on the smallest rather than the largest key values, ensures that only one leaf node is visited. In Fig. 3, an AP tree for the data in Fig. 2 is shown with a search organization based on the largest key value;

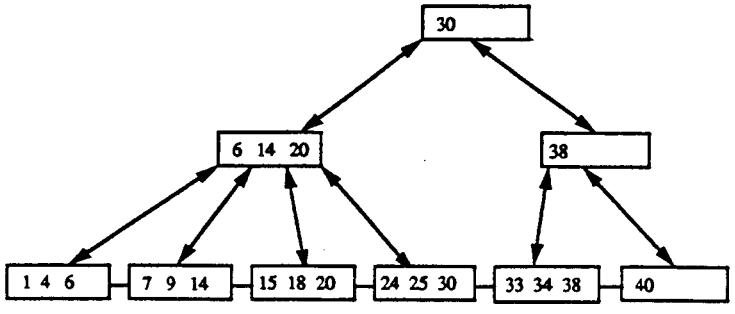


Figure 3: Example of AP-tree with B-tree Key Organization

for a query that requests tuples that have $V = 32$, v^+ will be found in the visited leaf of the tree in Fig. 2, but not for the tree in Fig. 3, where the key resides in the left sibling of the visited node.

In order to optimize searching time, an AP-tree's root node contains the following metadata: T_S^- which is the minimum of all the T_S values indexed by the tree; $LS_r.END$ ‡, which is the end point of the lifespan of the relation r_i being indexed; the leftmost leaf pointer; and the rightmost leaf pointer. We assume that the beginning of the data file pointer for r is resident in main memory or is easily accessible, thus not requiring its inclusion as metadata. Further, due to the need for backward as well as forward sequential scans through the data file, the

† Two adjacent leaves of the tree are also linked by two-way pointers although this detail is not clearly shown in the figures.

data blocks are linked by two-way pointers. In the following procedures, we ignore certain details such as pointer maintenance for brevity. When a query is based explicitly on T_S, using the index is beneficial, but when the query is based on some arbitrary point in time, then it is not possible to determine *a priori* whether an index search is more useful than a simple forward scan. On the other hand, further qualification, such as the specification of a surrogate value(s) in the query, or additional knowledge about the data, may allow a choice of either forward scan or index-based search for optimal response time. We provide only a general search algorithm on the basis of time alone.

SearchTimeStart

[For $V = T_S$]

1. If $V < T_S^-$ or $V > LS_r.END$, search fails;
   else if $V = T_S^-$ perform task.
2. If $V = LS_r.END$, go to *RighMostLeaf*;
   if V is not found, search fails;
   else perform task.
3. Starting from *Root*, follow pointer corresponding to $v^+ = \max\{v \mid v \leq V\}$ until leaf is reached.
   if V is not found in leaf, search fails;
   else perform task.

SearchArbitraryTime

[For arbitrary V]

1. If $V < T_S^-$ or $V > LS_r.END$, search fails;
   else if $V = LS_r.END$ perform step 3;
   else perform step 2 or 3.
2. [Index search]
   Starting from *Root*, follow pointer corresponding to $v^+ = \max\{v \mid v \leq V\}$ until leaf is reached.
   Carry out backward scan, starting from tuple accessed by leaf pointer, to the beginning of data file.
   For each tuple x, if $V \in [x(T_S), x(T_E)]$ perform task.
3. [Non-index search]
   Scan tuples from beginning of file until a tuple with $T_S > V$ is found. For each tuple x, if $V \in [x(T_S), x(T_E)]$ perform task.

‡ T_S^- is not the same as the beginning of the lifespan of r, since the tree may index a moving time window.

3.3. Insertion into the AP-tree

Insertions into the tree are made by first accessing the rightmost leaf. One of the advantages of the *AP*-tree is that it requires no splitting of filled nodes, and when new nodes have to be created, it is done so only on two levels-- the leaf and its parent; no other level will be affected. The only nodes of the tree that are relevant during an insertion procedure are the root, the rightmost child of the root, the rightmost leaf, and the parent of the rightmost leaf and its parent. If the root and rightmost leaf are not stored in main memory, at most five disk blocks have to be retrieved during an insertion; unlike the *B*-tree, recursive procedures are not required. The insertion algorithm below consists of two subroutines-- **InsertLeaf** and **AdjustTree.** We ignore the housekeeping details related to the updating of *RootPtr* and *RightMostPtr*.

InsertLeaf

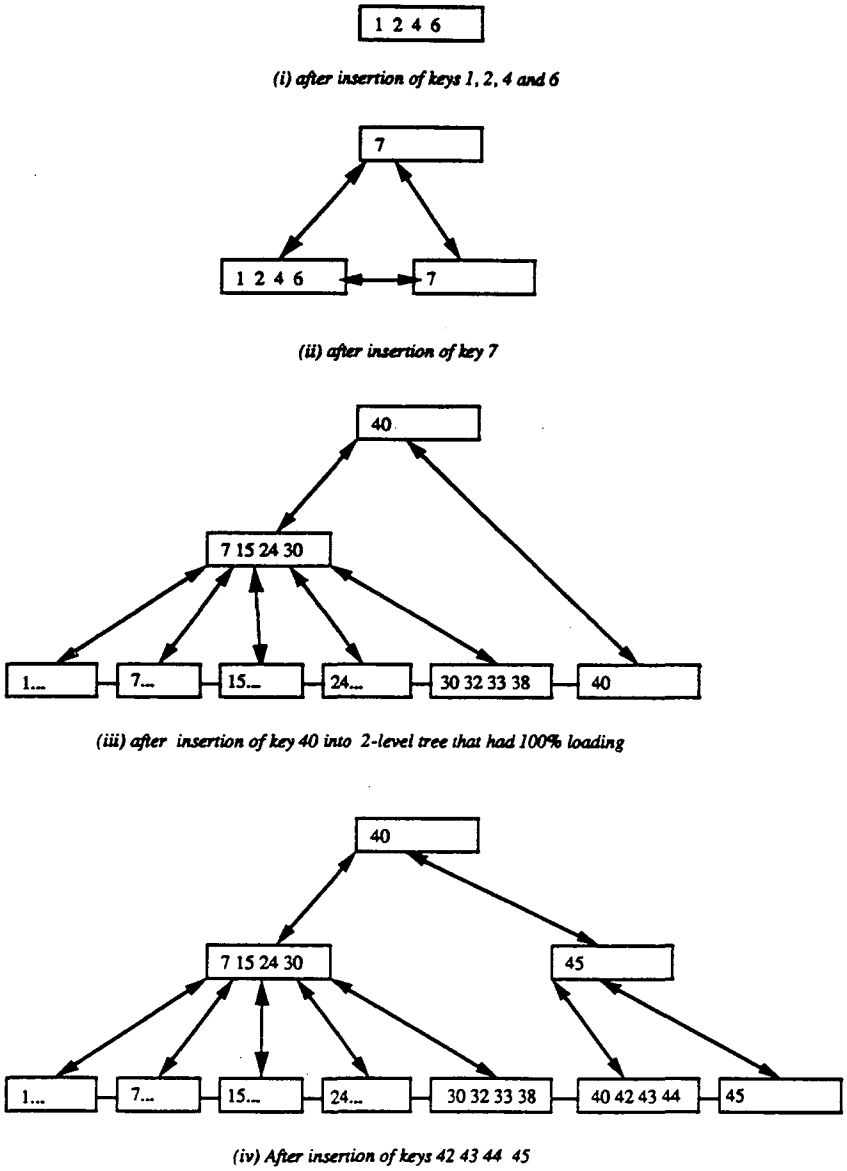
1. If *RootPtr* = *nil*, create *Root* and insert *NewKey*;
   else retrieve *RightMostLeaf*.
2. If *NewKey* is found in *RightMostLeaf* update tuple pointer if necessary.
3. If *RightMostLeaf* is not full, insert *NewKey* into *RightMostLeaf*.
4. [*RightMostLeaf* is full]
   Create *NewLeaf* and insert *NewKey*;
   Set the parent of *NewLeaf* to the parent of *RightMostLeaf*;
   call **AdjustTree.**

AdjustTree

1. Name the parent of *NewLeaf* *CurrentNode*.
2. If *CurrentNode* = *nil*,
   create *NewRoot* and insert *NewKey* into it;
3. If *CurrentNode* is not full, insert *NewKey* into it.
4. [*CurrentNode* is full]
   If the rightmost child of *Root* is full
   if *Root* is also full, create *NewRoot*, insert
   NewKey into *NewRoot*, and link *NewLeaf* and *Root* to *NewRoot*;
   else insert *NewKey* into *Root* and link *NewLeaf* to *Root*.
5. [*CurrentNode* is full but rightmost child of *Root* is not full]
   Create *NewNode* and insert *NewKey*;
   link *NewNode* to the parent of *CurrentNode*;
   link *NewLeaf* and *CurrentNode* to *NewNode*.

In **InsertLeaf**, the appropriate location for the new key is determined; when the rightmost leaf is full, a new rightmost leaf is created and the key inserted into it. The parent pointer of this new leaf is set to the same

parent of the former rightmost leaf. The adjustments are then accomplished by **AdjustTree**; the examples given in Fig. 4 help to illustrate some of the cases.



(i) after insertion of keys 1, 2, 4 and 6

(ii) after insertion of key 7

(iii) after insertion of key 40 into 2-level tree that had 100% loading

(iv) After insertion of keys 42 43 44 45

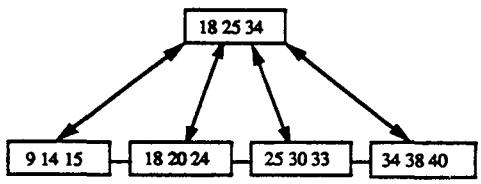
Figure 4: Examples of Insertions into the AP-tree

Step two is illustrated by Fig. 4 (ii), where the parent of the new leaf, *CurrentNode*, is nil, therefore requiring a new root to be created. Where the *CurrentNode* is not yet filled up, *NewKey* has only to be inserted into it. On the other hand, if the *CurrentNode* is full, there are three subcases to consider: (1) the whole right subtree of the *AP*-tree is already filled to capacity but the root is not yet full; (2) both the root and the right subtree of

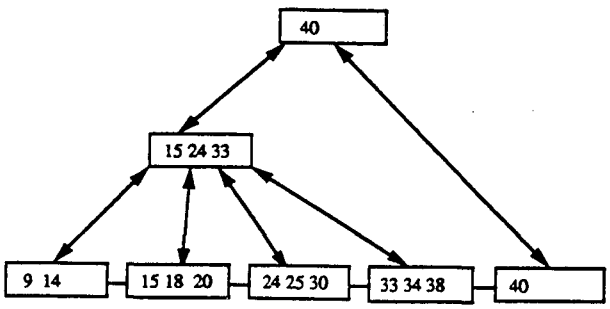
the *AP*-tree are full; and (3) the right subtree has not fully grown. In the first case, a new entry is made into the root, creating a new rightmost subtree consisting only of the new leaf. In the second case, a new root has to be created, with the new leaf again being the sole node in its rightmost subtree and the rest of the old tree as its left subtree; Fig. 4(iii) provides an illustration. For the third case, since the rightmost subtree can still grow, a new internal node is created which becomes the parent of the subtree rooted at *CurrentNode* and the new leaf; Fig. 4(iv) shows such an example.

3.4. Deletions from the AP-tree

Deletions are made in order to reduce the *current* time-window of the database. There are two relevant types of deletions: one based on change points, i.e., T_S values; and on event points, i.e., some given point along the time dimension. Unlike the *B*-tree, deletions from the *AP*-tree require complete reconstruction of the tree in order to maintain balance; in other words, the complexity is proportional to N_v, the total number of keys. If we were to allow the leftmost leaf to be less than 100% full after a delete operation, the complexity is reduced; Fig. 5 provides an illustration of the differences between the two approaches.



(a) Maintaining 100% fill factor for leftmost leaf



(b) Allowing less than 100% fill factor for leftmost leaf

Figure 5: Deletion of Keys 1,4,6 and 7 from AP-tree of Fig. 1

The second method is useful for the deletion of a small number of keys from the leftmost node, since in such an instance, the internal nodes remain unaltered. Otherwise, this technique presents difficulties due to (1) in the worst case, the parent-pointer of most leaves have to be changed, so they need to be read, and (2) modification of the internal nodes means that they have to be chained together for efficient retrieval. In the following algorithms, we adopt the first approach. We call the new cutoff key $Cutoff$: if $Cutoff$ is not a T_S, then T_S^- is set to the leftmost key along the leaf level after deletion is completed, else $T_S^- = Cutoff$. For simplicity, we ignore below the treatment of the data tuples and any index search needed, since that would use the search procedures previously given. Further, housekeeping of pointers and metadata are ignored unless explicit treatment is important.

DeleteTimeStart

[For $Cutoff = T_S$]

1. For *CurrentNode*, delete all $v < Cutoff$ in it and any left siblings.
   if *CurrentNode* becomes empty, make right sibling the new *CurrentNode* and delete empty node;
   else shift remaining keys (and their associated pointers) leftward.
2. Let $T_S^- = Cutoff$.
3. If *CurrentNode* has no siblings, set *Root* to *CurrentNode*;
   else call **RebuildTree**.

DeleteArbitraryTime

[For arbitrary $Cutoff$ point]

1. From leftmost leaf until $v \geq Cutoff$, delete a key v if all tuples with $T_S = v$ have $T_E < Cutoff$;
   if a node becomes empty, delete it.
2. Consolidate non-empty nodes leftwards;
   delete resulting empty nodes.
3. Let T_S^- = smallest key of leftmost leaf.
4. If leftmost node has no siblings, set it as *Root*;
   else call **RebuildTree**.

RebuildTree

1. Create *NewNode* for every d children and enter appropriate keys.
   if the rightmost node has only one child, do not create it, and treat that child as if it were a node on this level.
2. If only a single node is created, set it to *Root*;
   else call **RebuildTree**.

While the **DeleteTimeStart** routine is self explanatory, the **DeleteArbitraryTime** routine requires elaboration: since a key may be associated with tuples of varying T_E values, the best approach is to do a forward scan and eliminate a key only if all associated tuples have been eliminated. After the $Cutoff$ point is reached, the

remaining keys have to be consolidated and this is accomplished by shifting the keys left to right. The **Rebuild-Tree** procedure recursively rebuilds the tree towards the root; it also has to make provision for an unbalanced rightmost subtree. Since two-way pointers between a key and its child are necessary, to avoid having to read a node twice, the above procedures could be rewritten as depth-first recursion, as opposed to breadth-first recursion.

3.5. Analysis and Comparison with B-trees

In analyzing the performance characteristics of the AP-tree, we will compare it to a B^+-tree constructed for time indexing. This B^+-tree incorporates some of the properties of the AP-tree which can be easily included without major changes to the general algorithms. These modifications are: two-way pointers between leaves, so that forward and backward scans can be performed; and metadata on the minimum T_S and maximum T_E values. We do not consider the case of two-way pointers between parent-child, because many modifications would be needed in the deletion and insertion procedures.

Height. The AP-tree, with the exception of the rightmost subtree will have a fill factor of 100%. In contrast, a B^+-tree will almost always be very close to its minimum loading factor of 50%; the reason is that due to the progressive arrival of key values, each leaf node is split just once, and subsequently is never needed for further insertions. Thus the height of the AP-tree is

$$h_{AP}(T) = \left\lfloor \log_{d+1}\left[N_v + 1\right] \right\rfloor + 1, \tag{1}$$

which is equivalent to the lower bound height of a general B^+-tree. The B^+-tree's height is

$$h_B(T) = \left\lfloor \log_{\lceil d/2 \rceil + 1}\left[N_v + 1\right] \right\rfloor + 1 \tag{2}$$

Comparing the two heights, $h_B(T)$ can be represented in terms of a factor that is a function of d times $h_{AP}(T)$, i.e.,

$$h_B(T) = \frac{1}{1 - \log_d 2} \cdot h_{AP}(T). \tag{3}$$

As an example, if $d = 70$, then $h_B(T) \cong 1.2 h_{AP}(T)$.

Searching. Searching cost difference is a function of the height difference, if index search is used.

Insertion. Let us ignore the I/Os needed to insert tuples into the data file; we are concerned only with the I/Os for the index. The lower bounds for both the AP-tree and B^+-tree are equal to two disk I/Os: one read and one write. The upper bound for an insertion into the B^+-tree is $3h_B(T)$: i.e., h is the cost of traversing the tree to the current leaf, and for each level of the tree, a split is required, which necessitates an additional read and write per level. In the case of the AP-tree, the upper bound for insertion is 10 I/Os: this can be derived from step 4 of **InsertLeaf** and step 5 of **AdjustTree** †. As long as $h_B(T) > 3$, the AP-tree performs better for insertions.

Deletion. In considering deletions from the tree, we ignore the cost of tuple deletions from the data file, and assume only T_S deletions in order to simplify comparisons. The lower bounds for both trees are again two disk I/Os. For the B^+-tree, the worst case performance for a deletion of a set of keys with cardinality N_{dv} consists of the following sequence of actions: (1) traverse tree down to first leaf, (2) keep deleting keys until balancing is required, (3) carry out worst case balancing from leaf to root, and return to step (2) until no more keys need to be deleted. The total cost is

$$\frac{N_{dv}}{\lceil d/2 \rceil} (3h_B(T) - 2) \tag{4}$$

For the AP-tree, all but the leaves left of the cutoff point have to be read and written once; as for higher levels, only writes of new internal nodes are needed. The total cost comes to

$$2 \left\lceil \frac{N_v - N_{dv}}{d} \right\rceil + \frac{d^{h_{AP}(T)-1} - 1}{d - 1} \tag{5}$$

If deletions are based on arbitrary time points, the upper bound costs are of the same order as above. In general, deletions for the AP-tree are more costly, since it is dependent on the number of nodes in the tree, as opposed to just the height of the B^+-tree. Fig. 6 shows the behavior of the deletion costs for the two indexes as a function of the percentage of keys deleted, where N_v is 10,000 and d is 40. At very high percentages of key deletions, the cost for the B^+-tree becomes worse than the AP-tree.

† For step 4 of **InsertLeaf** the root, rightmost leaf and parent of the rightmost leaf are read; in step 5 of **AdjustTree** , the rightmost child of the root and parent of the current node are read, while the newleaf, new node, current node, rightmost leaf and rightmost child of the root are written.

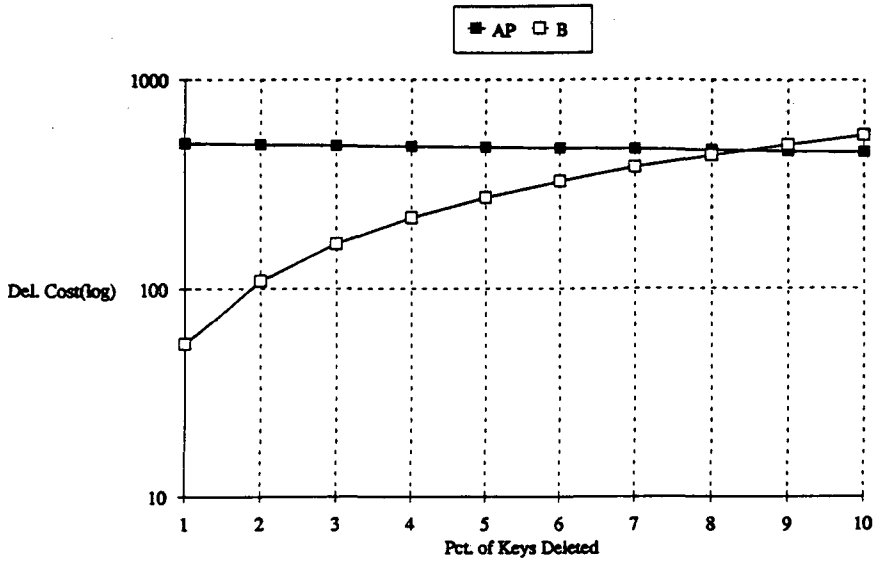


Figure 6: Deletion Cost Comparison

4. INDEXING SURROGATES AND TIME

ST indexes have to take into consideration the efficiency of answering current as well as historical queries, and the need to answer combinations of range and point specifications. It is more likely that range qualification is given for the time attribute. In order to provide a higher degree of selectivity, we present a structure that is based on nested trees, where the first level index is based on the B^+-tree, while the second level index is based on the *AP*-tree. In order to evaluate the performance of this method, we compare it with two alternative methods, both of which are also based on variants of the *B*-tree.

4.1. Nested *ST*-tree

The nested *ST*-tree was introduced for the static database in [7], which is similar to the concept of the $K-b$ tree of [11]. We introduce a modified structure which is designed for (1) a dynamic database, (2) takes advantage of the natural ordering of time stamps for each given S thus enabling the use of an *AP*-tree, and (3) is independent of the physical ordering of data. This approach is designed to answer queries where the primary qualification is on S. Fig. 7 illustrates the basic constructs of the tree.

***S*-superindex.** The first level of the hierarchy is a B^+-tree, with the following modifications: Each leaf entry has two pointers associated with it -- a direct pointer to the current tuple, and one that points to the root node of the *T*-subindex.

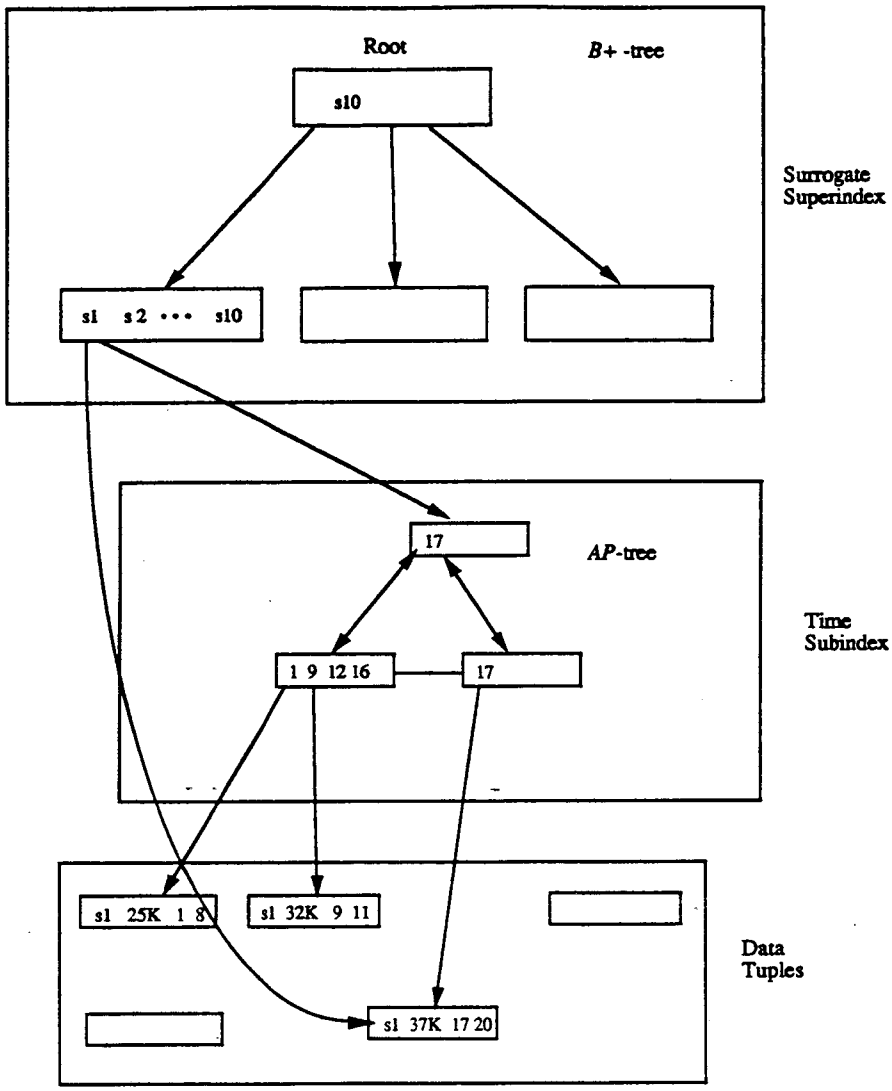


Figure 7: Nested ST Indexing

***T*-subindex.** The *T*-subindex is structured as an *AP*-tree. Unlike a solo *T*-index to the relation, the subindex always maintains a single time entry per *S*-value. In order to compress further the height of the tree, the subindex can be constructed sparsely, i.e., each leaf entry will lead to a tuple-pointer block, rather than to the data block itself. The adoption of the *AP*-tree for this level of indexing is not dependent on an append-only database, since a dynamic database that allows delete operations would delete tuples only on the basis of maintaining some current time-window.

4.2. Nested Tree Procedures

The insertion procedure for the Nested ST-tree is two-stage:

InsertSuperindex

1. If S is found in leaf, **InsertSubindex.**
2. Create new leaf entry;
   Create new root node for T-subindex.

InsertSubindex

Same as in AP-tree.

The search procedures consists of traversing the B^+-tree superindex, and then using the appropriate procedures for the AP-tree subindex. Deletions can be of two types, one to delete some tuples of an S-entry, and the other to delete all history of the entry.

DeleteTuple

1. If T range = ALL, delete S entry in superindex.
2. Delete appropriate T entries in subindex.

In insert and delete operations, any reconstruction is bounded by the complexity of the sum of B^+-tree and AP-tree balancing.

4.3. Alternative Structures

We introduce two structures for comparison purposes: the first is based on a conventional B^+-tree for the composite key S and T, while the second is similar to the idea introduced by [12].

Composite Index. The key is made up of the concatenation of the two attributes, employing a B^+-tree. There are many ways in which the height of the tree can be reduced, including the use of $prefix$–B-trees, compression, and allowing sparse indexing on the T portion of the key.

S-Index and Accession List. In this case, only the S values is indexed, and the associated T values are stored in a list accessible from the appropriate leaf entry. The list is made up of the concatenation of disk pages, each containing pairs of time-value and tuple-pointer. The list can be reduced in length by compression.

4.4. Analytical Comparison of Approaches

We will look at the performance of the three indexes analytically, in terms of heights and complexity with respect to balancing. We assume for simplicity, that the tuples are uniformly distributed across the surrogate instances. We denote the nested structure as 'Nested' or I_N, the composite key index as 'Composite' or I_C, and the S-index and accession list as 'Sparse' or I_{Sp}.

Heights. Let $h_N(ST)$, $h_C(ST)$, and $h_{Sp}(ST)$ denote the heights of I_N, I_C, and I_{Sp} respectively. Further, let N_r and N_S denote the cardinality of the relation and surrogate domain respectively; $\bar{N}_{T|s}$ is the mean number of time values for each surrogate instance, i.e., the average length of a time chain; B_{Sp} represents the blocking factor for the accession list of I_{Sp}; and $d(\cdot)$ is the order of the tree for the specified key.

$$h_N(ST) = h_B(S) + h_{AP}(\bar{N}_{T|s}) \tag{6}$$

$$= \left\lfloor \log_{\lceil d(S)/2 \rceil + 1}(N_S + 1) \right\rfloor + \left\lfloor \log_{d(T)+1}(\bar{N}_{T|s} + 1) \right\rfloor + 2 \quad \text{(UB)}$$

$$= \left\lfloor \log_{d(S)+1}(N_S + 1) \right\rfloor + \left\lfloor \log_{d(T)+1}(\bar{N}_{T|s} + 1) \right\rfloor + 2 \quad \text{(LB)}$$

$$h_C(ST) = h_B(ST) \tag{7}$$

$$= \left\lfloor \log_{\lceil d(S+T)/2 \rceil + 1}(N_r + 1) \right\rfloor + 1 \quad \text{(UB)}$$

$$= \left\lfloor \log_{d(S+T)+1}(N_r + 1) \right\rfloor + 1 \quad \text{(LB)}$$

$$h_{Sp}(ST) = h_B(S) + \left\lceil \frac{\bar{N}_{T|s}}{B_{Sp}} \right\rceil \tag{8}$$

The difference between the lower and upper bounds of the first two techniques is in the fanout of the B^+-trees. There are two types of height comparisons -- the first relates to current time queries, in which case $h_N(ST)$ is never worse than the other two alternatives.

The second type of comparison relates to the maximum height of the three, i.e., in retrieving an arbitrary ST query. In comparing lower bounds of the nested tree versus the other two,

$$h_N(ST) = h_{Sp}(ST) \tag{9}$$

and

$$h_N(ST) \geq h_C(ST) + 1 \tag{10}$$

The height difference is due to the fact that an additional level of indirection is needed to find the T value for the nested tree, given that the loading factor of all trees involved is the maximum possible.

In terms of upper bound differences,

$$h_N(ST) \leq h_{Sp} - \left\lfloor h_{AP}(T) - \left\lceil \frac{\bar{N}_{T|s}}{B_{Sp}} \right\rceil \right\rfloor \tag{11}$$

i.e., the difference is due to the logarithmic versus linear complexity of the two time-indexes. On the other hand, for the nested versus composite approaches, assume that $d(S) = d(T) = \alpha$,

$$h_N(ST) = \frac{\left\lfloor \log_{\alpha/2}(N_S + 1) \right\rfloor + \left\lfloor \log_{\alpha} \bar{N}_{T|s} + 1 \right\rfloor + 2}{\left\lfloor \log_{\alpha/4}(N_r + 1) \right\rfloor + 1} h_C(ST) \tag{12}$$

In Fig. 8, the upperbound comparisons are made for the three methods, letting $\alpha = 50$, $N_r = 1$ million tuples, $B_{Sp} = 100$, and N_S varying between 1,000 to 10,000.

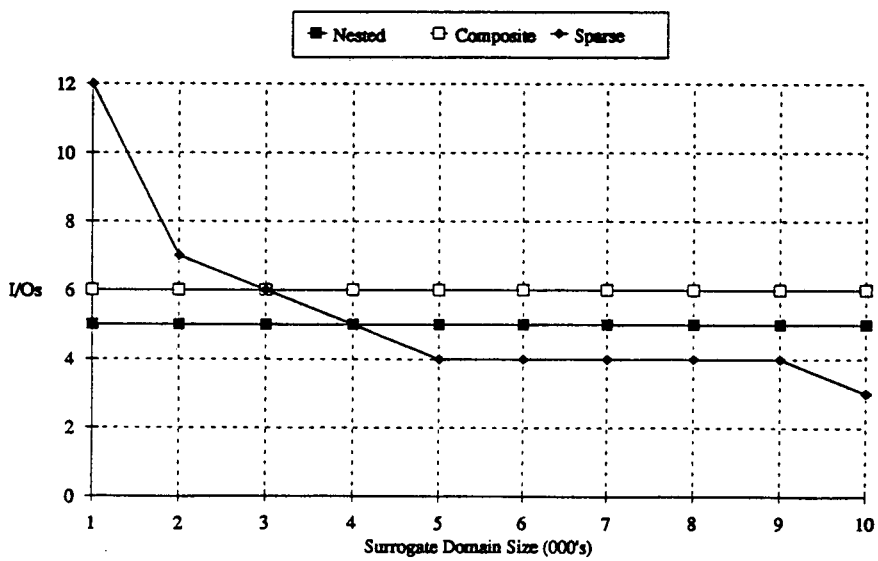


Figure 8: Upper Bound Heights, N_r = 1M tuples

We label the three methods 'Nested', 'Composite' and 'Sparse' in the graphs that follow. As the graph shows, for the given parameter values, I_N is always one level lower than I_C, while I_{Sp} fluctuates, depending on the size

of the history chain. Since the relation size is fixed, the distance of index traversal carried out by I_{Sp} is inversely proportional to the size of the S domain.

Insertions. For an insertion that requires only the insertion of a new T value, the lower bounds is 3 I/Os for I_N and I_{Sp}, and 2 for the I_C; the worst case costs on the other hand, are $h_B(S) + 3$, $h_B(S) + 10$, and $3h_B(ST)$ respectively. On the other hand, when a new S value is inserted, the lower bounds remain the same as before; for the the worst case, while the cost for I_C also remains unchanged, it increases for the other two methods, i.e., $3h_B(S) + 3$ for I_{Sp}, and $3h_B(S) + 10$ for I_N.

Deletions. The case for deletion costs mirror those presented above for the best case scenarios, while for the worst case, they are a function of the worst case costs of the B^+-tree and AP-tree, as presented previously in Section 4.

4.5. Empirical Test of Alternative Structures

In this subsection, we provide results of some tests of the tree indexing techniques. In executing the tests, it was decided that sparse indexing and compression of index entries should be ignored, since they can be applied to each approach. Furthermore, the idea of using *prefix* B-trees was rejected, since it can again be applied to varying degrees to each index, and this method of compacting the index's storage requirement is known not to be very effective [26].

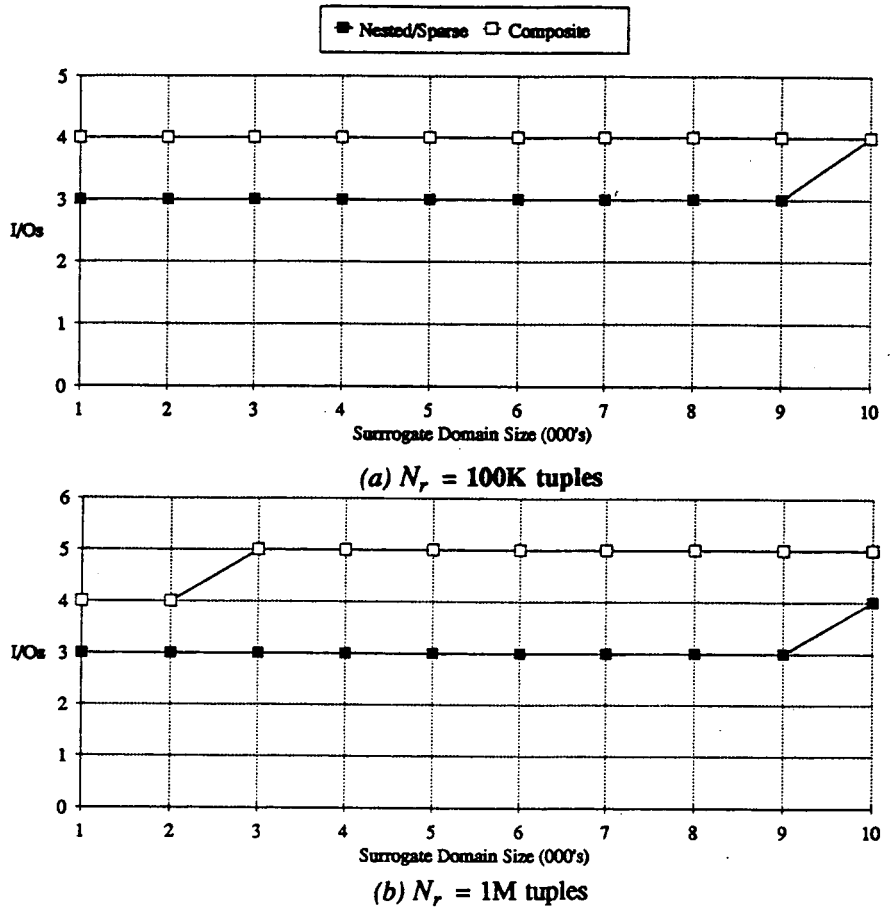
The parameter settings used are shown below, where the relation size is set at 100K and 1M tuples, and the S domain varied in increments of 1K, from 1K to 10K.

Variable	Value
N_r	100K and 1M tuples
N_S	1K to 10K, in 1K increments
$l(r), l(S), l(T), l(PTR)$	40B tuple, 8B S & T, 4B pointer
FILL_INDEX	100% for AP-tree, 75% for B^+-tree
FILL_DATA	75%
CR	50%

Table 2: Table of Values for Tests

$l(\cdot)$ represents the byte size of the argument, and *FILL_INDEX* and *FILL_DATA* are the fill factors for the index and data pages respectively. *CR* is the compression ratio, and is set to 50%, which is about the best most existing algorithms can achieve.

Fig. 9 to 11 exhibit some of the results of the tests. In Fig. 9, the current query cost in terms of disk I/Os is graphed for the three alternatives -- since I_N and I_{Sp} have identical superindexes, they have identical costs. The graphs show that the average cost of accessing the current tuple is cheaper by 1 or 2 disk accesses for these two methods when compared to I_C. Only in Fig. 9(a), with N_S = 10,000 was



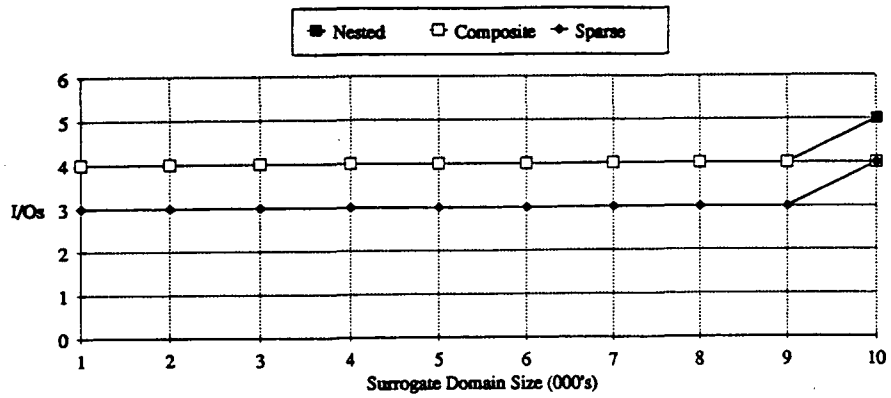
(a) N_r = **100K tuples**

(b) N_r = 1M tuples

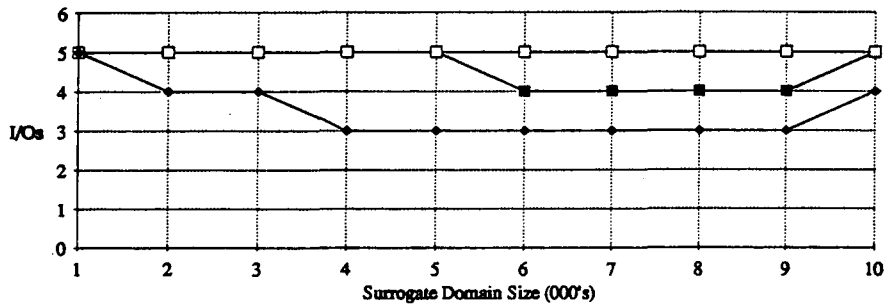
Figure 9: Query on Current Tuple Cost Comparison

the cost identical; the reason is that at this point, the time chain is only 10 tuples long on average, and there is little benefit in maintaining two-level indexing.

Fig. 10 shows the I/O costs associated with arbitrary queries involving historical data. When the relation size is 100K tuples (Fig. 10 (a)), I_{Sp} is more efficient than the other two techniques, which share the same cost for all values except at the right extreme, when the short time chain yields higher overheads for the nested index. When the relation size is fixed at 1M tuples, the results fluctuate much more -- the composite index performs worst. The reason that the sparse index yielded better performance in these two tests, is due to the assumption of a high compression ration of 50%, which allows each block to hold around 300 tuple-pointers. Since our tests are limited to history chains of 1,000 in maximum length, and we assumed uniform distribution



(a) N_r = 100K tuples



(b) N_r = 1M tuples

Figure 10: Arbitrary Query Cost Comparison

of tuples among surrogate instances, the use of chained lists appear to be the most efficient on average.

In Fig. 11, the index storage costs for the three methods, for the

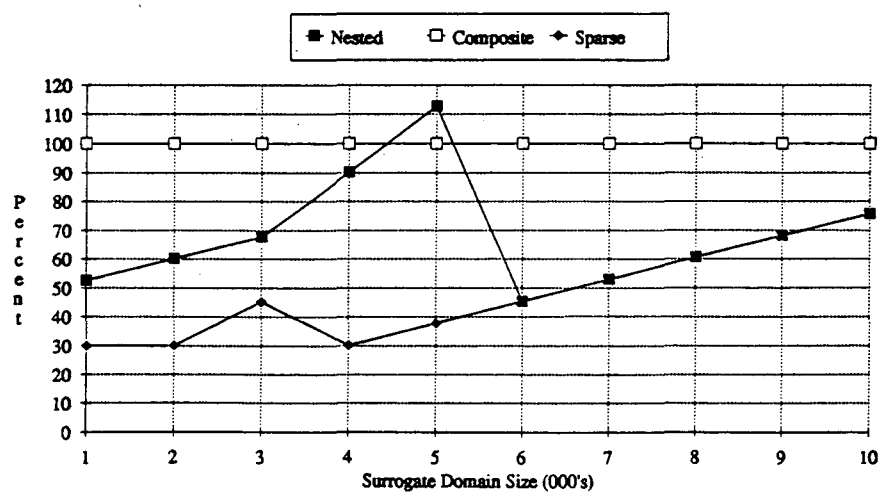


Figure 11: Index Storage Cost, N_r **= 1M tuples**

case of a 1 million tuple relation is examined. The composite approach on the average is the most expensive to maintain -- this is due to the fact that it is a single tree. Other tests undertaken, but for which results are not displayed, included sequential query retrieval costs under conditions of sorted and unsorted data (with S as the primary sort key). The results show little differences between the three methods, since the cost of retrieving data blocks overwhelms the index search costs.

To conclude, the nested approach is an efficient method of retrieving data for ST queries. The insertions are easily carried out, and balancing of one level can be separated from the other; Deletions on the AP-tree level is not a primary concern, thus the weaknesses of the tree is not exposed often. Further, the nested index is much more efficient in answering current queries, and is efficient storage-wise. Compared to the sparse indexing, although the average case analysis for random queries is either worse or equal, it has a lower upper bound.

5. AT-INDEXING AND TIME PARTITIONING

In this section, we look at the problems of indexing a relation on the temporal attribute and time, and the related issue of partitioning the time-line into segments.

5.1. AT Indexing Using Nested Trees

Fig. 12 illustrates our approach to indexing for AT queries. The basic concepts are similar to that of the ST-index, whereby the fist level indexes the non-time attribute, and the second level indexes time. The difference relates to the multiplicity of qualifying tuples for a given A value. Since the temporal attribute is not a unique key, tuples that qualify on it are likely to overlap over their associated time intervals. A straightforward way of indexing the time line in such a case, would be to use a conventional index, such as the B-tree, with one of the two time-attributes, i.e., T_S or T_E as the search key. A major limitation to this is that many queries are not based on the start or end time, rather on an arbitrary point in time or time interval. Using the above method of indexing the time line, multiple overlaps among tuples will occur, meaning that the index will not be highly selective. The greater the degree of overlap, the more time will be spent traversing sibling nodes of the index tree in search of overlapping tuples. We will thus look at methods by which the tuples associated with a given temporal attribute value, or even a range of such values, can be partitioned along the time dimension, such that minimal overlap is achieved.

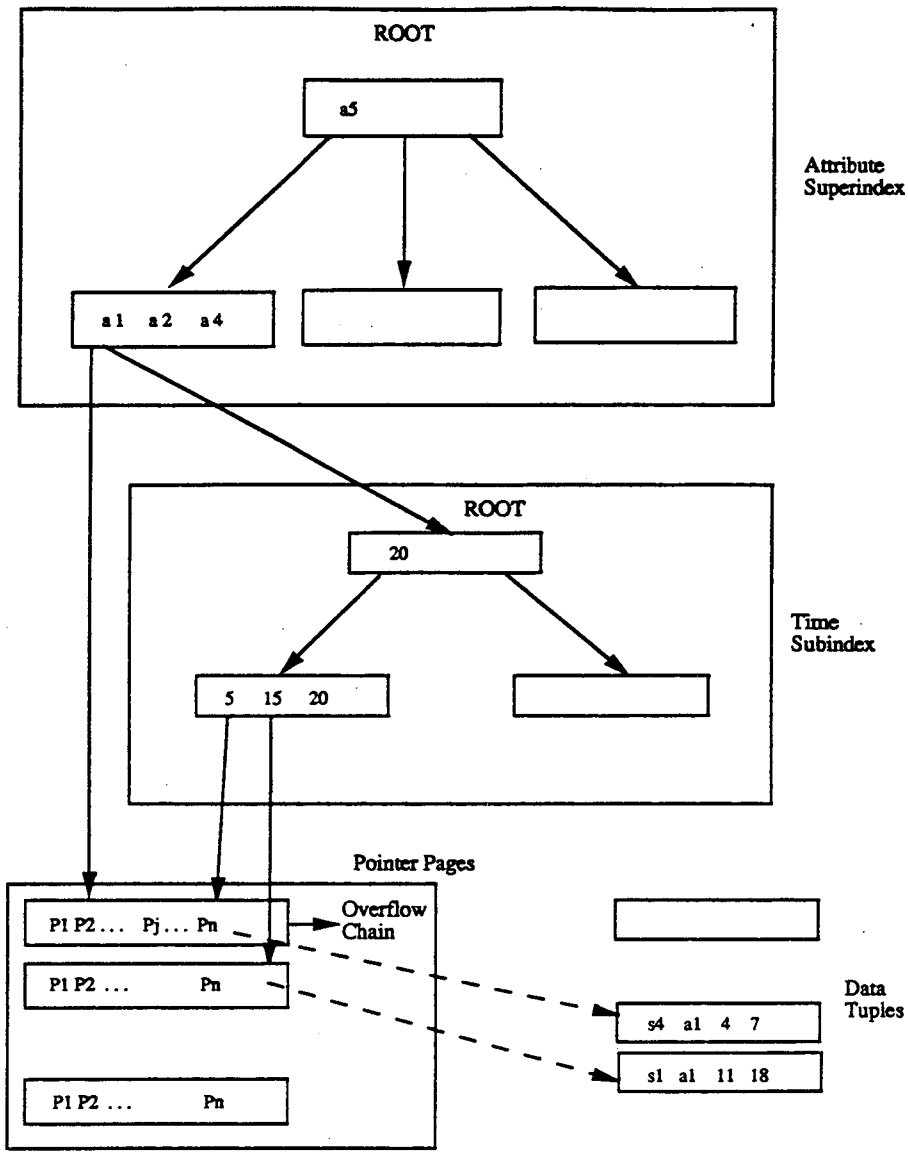


Figure 12: AT Indexing

5.2. Basic Approach and Objectives of Partitioning

The time-line is partitioned into several segments, each segment being represented by a leaf entry in the index. The associated pointer leads to a bucket of tuple pointers, for all tuples where $[T_S, T_E]$ intersects with the segment delimiters, $[V_i, V_{i+1})$. Each bucket consists of one or more disk pages, and in the event of an overflow, additional pages are chained to it, thus increasing the search time along that segment. The objective is thus to minimize overflow resulting from the partitioning, which requires that duplication of tuple pointers across the relevant buckets be minimized. We develop the algorithms from an initial point where the number of tuples are known. These are then extended to the dynamic case.

5.3. Simple Allocation Approach

In this algorithm, divide the bucket capacity into the total pointer count, to derive the starting number of buckets. We then assign pointers to buckets, which will likely cause overflows. A balancing routine is then executed, which attempts a reallocation of overflow pointers between neighboring buckets in an iterative manner. At each step, the bucket with the worst overflow is selected, and the length of its associated segment is reduced by setting V_{max} higher, V_{max+1} lower, or both, if doing so reduces its tuple count without increasing that of its neighbors. The balancing is completed when no more improvement is possible.

Algorithm Simple

1. [*Initial Allocation*]
   Divide total count of tuple pointers by desired bucket capacity to derive number of buckets and segments.
2. Allocate pointers to buckets.
3. [*Balancing*]
   Find bucket with highest overflow. Compute the changes in allocations, if the left boundary is moved to the right, the right boundary to the left, or both. Execute change if feasible and until no more adjustment is possible.
   Eliminate any redundant buckets.
4. Repeat Step 3 until there is no overflow or all buckets have been iterated through.

5.4. Optimal Allocation Approach

There are two major drawbacks that the first algorithm possesses. First, it does not consider the probability distribution of pointers as a function of time. Secondly, the duplication of pointers, which takes place as the partitioning is carried out is ignored. An optimal solution approach is to use a dynamic programming algorithm to solve the problem. We recursively allocate pointers to buckets, such that each receives its optimal allocation, given that its immediate predecessor has also received an optimum assignment of the tuple pointers. The recursive equation is as follows.

$$g_i^* = \min_{x_i} \{ |N(B_i) - \gamma| + g_{i-1}^* \}$$

where

x_i = right boundary point for B_i

$i = 0,..., MAX$

γ = set capacity for buckets

B_i = ith bucket

$$= B_{i-1} + \sum_{i=x_{i-1}+1}^{x_i} \sum_{j=1}^{MAX} c_{ij} + \sum_{i=0}^{x_{i-1}} \left\{ \sum_{j=\begin{cases} i & \text{if } l=1 \\ x_{l-2} & \text{otherwise} \end{cases}}^{x_{l-1}} c_{ij} \right\}$$

This algorithm will require exponential time in order to solve, with a worst case of $N_T!$. We develop two heuristics to implement the formulation, taking into consideration the repetition of pointers as the lifespan is partitioned into intervals. The heuristics differ in the decision to terminate assignment to the current bucket: in the first version, assignment is terminated when either the capacity is reached or exceeded; in the second version, termination occurs if the next assignment will cause an overflow, unless that assignment will be the only one for the bucket. These heuristics also require balancing, since there is a good probability that the procedures will be less effective in assigning tuples as the end of the lifespan is reached. Thus the balancing algorithm is carried out right to left, under the assumption that tuple pointers have to redistributed from the last bucket towards the first.

Algorithm Dynamic-Overflow

1. [*Allocation*]
   Starting from $LS_r.START$, allocate to bucket one until its capacity is reached or exceeded. Determine the rightmost boundary of the associated segment.
2. Continue for bucket two to MAX, until all tuples have been assigned. For each subsequent bucket, start with a count of the predecessor bucket's overlapping intervals.
3. [*Balancing*]
   Perform balancing algorithm similar to that given in **Algorithm Simple**, but starting from bucket MAX to 0, execute a single scan.

Algorithm Dynamic-Underflow

1. Starting from $LS_r.START$, make an initial allocation of pointers.
2. If the bucket is not full, continue allocating the next batch of pointers *iff* they do not cause an overflow;
   else determine the boundary of the associated segment of current bucket.
3. Carry out Steps 2 and 3 of Algorithm **Dynamic-Overflow**.

5.5. Non-redundant Allocation of Pointers' Time Intervals

Our objective of reducing redundancy of pointer assignment was motivated by the need to minimize search time in order to retrieve the relevant tuples associated with a query. A major problem that arises from this approach is that a given time segment does not tightly bind *all* tuples that it is indexing. In other words, a

query that is directed at a particular segment needs to scan all tuples within that segment in order to determine if any of them is relevant in answering the query. To avoid actual physical retrieval of data, each data pointer can be stored together with its time-stamps; but this will dramatically reduce the fanout factor of any indexing tree employed.

A method which eliminates the need to maintain time-stamps for each tuple pointer is one in which each bucket contains only intervals or subintervals that will span the bucket. This requires the pointer intervals to be divided into as many buckets as there are unique starting or ending points along the lifespan. Another advantage with this approach is that a variety of aggregate operations can be facilitated more easily, since each bucket has an identical set of intervals/subintervals. The drawback is that there will be a much larger number of buckets and therefore the indexing tree will be taller. This algorithm does not consider the capacity of the buckets, but it is unlikely that many of them require more than one data page, unless the distribution of the *event* points are not uniform.

Algorithm Tight-Bound

1 Create the first bucket, with starting point equal to the first starting time found among the tuples. Take note of the ending time for this first tuple.

2 Allocate tuples into this bucket, until (i) another tuple is found with the same starting time as the first tuple, but has an earlier ending time, or (ii) another tuple is found with a starting time earlier than the first tuple's ending time.

3 Determine the segment of the first bucket, and create the next bucket.

4. Repeat Steps 1 to 3 until the end of the lifespan is reached.

5.6. Empirical Testing of Partitioning Algorithms

There are several factors involved in the performance of the algorithms. We retained the assumption of uniformity in the occurrence of events. The following parameters were used:

The relation size is fixed between 200k to 1M tuples in increments of 200k. N_t is the number of time points in the lifespan of the relation. We generate time sequences, where once an event is generated, ρ_s indicates the probability that for the next time point, the event remains valid, i.e., there has been no *change*. By varying this probability between 0.70 and 0.90, we allow for varying mean intervals for events, and thus affect the count of actual tuples, i.e., the *change points* associated with each A value. Fixing the A and T domains at arbitrary values should not affect the general results, since we vary the mean length of events and thus

Variable	Value
N_r	200k to 1000k
N_t	1000
N_A	100
ρ_s	0.70 and 0.90
KPTR	page pointer capacity = 70

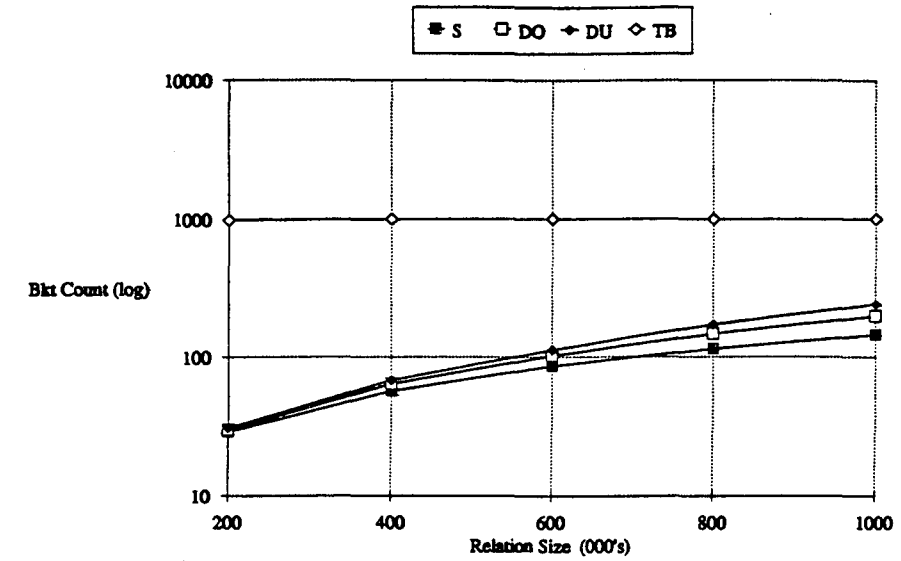
Table 3: Range of Values for Empirical Test

relation size.

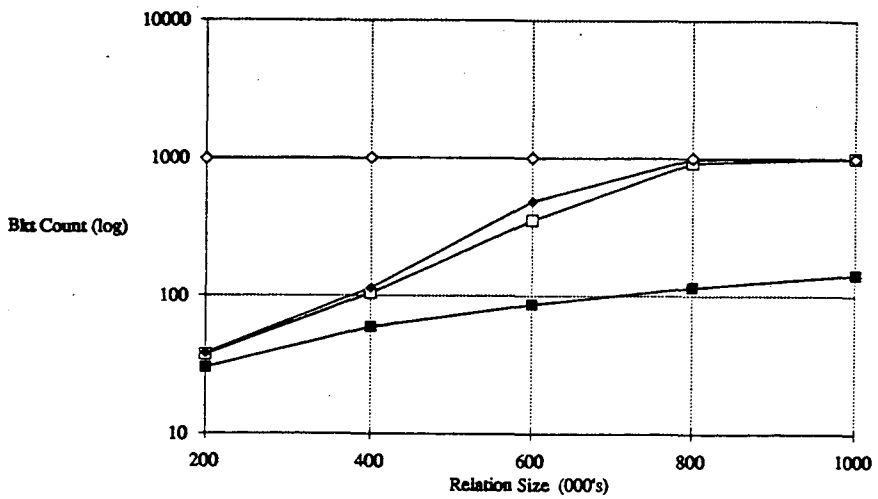
Fig. 13 and 14 display two sets of results from the experiments, where an arbitrary value of A is selected. In the graphs, the four algorithms are labeled as S, DO, DU and TB respectively. Fig. 13 shows the number of buckets that result from each partitioning algorithm, where in (a) ρ_s is 0.7, while in (b) it is 0.9. Regardless of ρ_s or relation size, Alg. TB ,which is the R^+ [25] equivalent in terms of partitioning technique, utilizes almost the same number of buckets. The reason is the high degree of overlaps, as opposed to identical intervals, that exist for any attribute value. Thus, for almost every point in the time line, there is a single bucket associated with it. This is why the number of buckets for the algorithm peaks at 1000, which is the size of the T domain.

On the other hand, the other three algorithms explicitly allocate pointers as a function of the capacity of buckets. Thus, for a given ρ_s the number of buckets rise in proportion to the size of the relation. For all cases, Alg. S has the lowest number of buckets, followed by Alg. DO and Alg. DU. This is due to the linear relationship between Alg. S's bucket generation and the number of tuples. But, the two dynamic algorithms assign tuples in an incremental way, leading to nonlinear growth, since at higher values of ρ_s and N_r, the increasing number of overlaps cause suboptimal use of many buckets. As expected, Alg. DU is inferior to Alg. DO. In Fig. 13 (b), we see that Alg. DO, DU and TB converge as the relation size increases.

Fig.14 shows two graphs pertaining to the performance of the four algorithms in terms of the average fill factor of buckets, over the same range of parameters as in Fig. 13. Alg. TB has the lowest fill factor, exceeding 100% only in the extreme. The low average utilization of the disk pages is another explanation for the high number of buckets required for the method. On the other hand, Alg. S almost always has an overflow, although it is usually below 100% -- this means that most of the time, an additional page is needed for each bucket. The two dynamic algorithms are the most efficient in space utilization -- there is little distinction between Alg. DO and DU otherwise.



(a) $\rho_s = 0.7$; $N_T = 1{,}000$

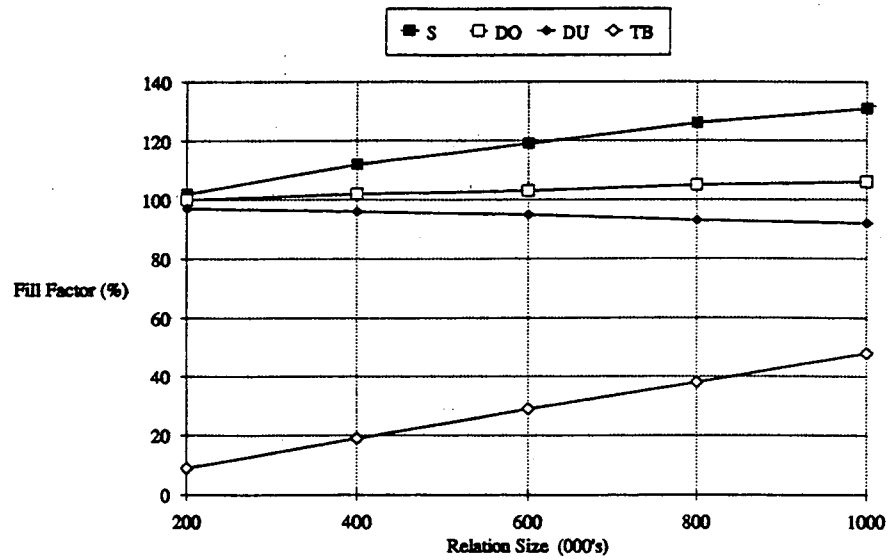


(b) $\rho_s = 0.9$; $N_T = 1{,}000$

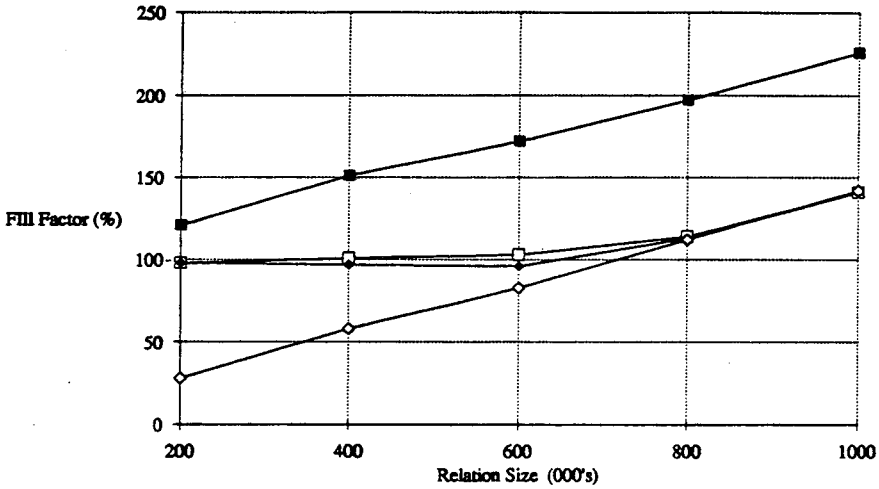
Figure 13: Comparison of Number of Buckets Used

5.7. Performance of Balancing Routines

The first three algorithms included balancing routines. It was found that the first method required balancing most. The result of the balancing was not so much a reduction in the total pointer count, but in the distribution of pointers across buckets. In most tests, the total count remained about the same, but the high count and standard deviation of bucket count dropped within the range of 10% to 40%. On the other hand, Alg. DO and DU did not require balancing, with minor exceptions; even when balancing took place, no noticeable difference occurred in the high count and standard deviation of allocations. One intervening factor is uniformity of the



(a) $\rho_s = 0.7$; $N_T = 1{,}000$



(b) $\rho_s = 0.9$; $N_T = 1{,}000$

Figure 14: Comparison of Bucket Fill Factors

generated sample data, which means that the intervals associated with tuples were uniformly distributed across the lifespan. Nonetheless, these two methods can be easily adapted to dynamic databases, since new data will only extend the ending time of the lifespan, requiring new segments to be created independent of the existing assignment.

5.8. Implications for Query Performance

We reject the second dynamic algorithm – Alg. Dynamic-Underflow, since it never performs better than its variant, the Dynamic-Overflow. Among the three remaining algorithms, we have to distinguish Alg. Simple from the other two: the former is more suitable for static data, since the partitioning is a function of the total count of tuples and a balancing routine is needed to optimize its performance. It clearly outperforms the others, especially when there is a larger density of tuples for a given time point being considered, and also the interval of a tuple is longer.

On the other hand, Alg. Dynamic-Overflow and Tight-Bound are more suitable for dynamic databases. The difference between the two is the tradeoff between the number of tuples retrieved in order to respond to a given query on the one hand, and the number of index pages that have to be retrieved on the other hand. Although Alg. Tight-Bound should provide only data that is valid for a specified query interval, its partitions are degenerate, i.e., it quickly converges towards the worst case of one partition for each time point. Thus, it has very low selectivity for an arbitrary query. Both algorithms generally have no overflow: thus, Alg. Dynamic-Overflow is clearly superior. Even if we take into consideration the desirability of adding a pair of time-stamps for each pointer in a given bucket, the reduced bucket capacity utilization in Alg. DO does not negate its advantages over Alg. TB.

The difficulties of allocating intervals is clearly the result of the fact that time intervals, unlike geometric objects, do not have natural clustering tendencies. Therefore, partitioning methods similar to those in spatial indexes, such as R-tree [9] and R^+ tree [25], need not be suitable, as discovered in [10].

6. CONCLUSIONS

In this paper we have investigated various issues associated with the indexing of temporal databases. We looked at the organization of a database in terms of current and historical data, the functional requirements of queries, and the links between physical order of data and indexing. We then looked at several structures, each aimed at optimizing data retrieval for a specific context. The AP-tree is aimed at indexing data for append-only databases, in order to help event-join optimization and queries that can exploit the inherent time ordering of such databases. Two variable indexing for the surrogate and time was studied -- we showed that a nested index could be a very efficient structure in this context, and overall is preferable to a composite B-tree or an index that involves linear lists of historical tuples. We discussed in detail the problems of indexing time intervals, as related to non-surrogate joint-indexing. Several algorithms to partition the time line were introduced, and we concluded that the Dynamic-Overflow method seems to yield good performance for the case of a dynamic

database, while the Simple algorithm can be effective for static data organizations. We also outlined a two-variable *AT* index based on nested indexing. Indexing of temporal attributes and time has not been explored in the literature before, where focus has been on the surrogate and time.

For future research, we would like to explore further the issue of time partitioning, primarily in the context of multidimensional search structures. It has already been mentioned, how difficult it can be in partitioning time jointly with other variables, since the latter are inherently point data, and further, there is no natural ordering within them. Thus, we should not follow too closely the research already done in the multidimensional partitioning area, since they are more often than not applied to spatial/CAD/geometric data. Another topic of interest is the construction and use of indexing for certain classes of temporal queries, which by their complex nature, may benefit from indexing. Finally, the organization and maintenance of a multi-level storage structure for temporal data is an important topic worth exploring.

REFERENCES

[1] I. Ahn, "Towards an implementation of database management systems with temporal support," in *Proc. Int. Conf. Data Eng.*, Feb. 1986, pp. 374-381.

[2] I. Ahn, R. Snodgrass, "Performance evaluation of a temporal database management system," in *Proc. ACM-SIGMOD Conf. Management of Data*, May 1987, pp. 96-107.

[3] I. Ahn, R. Snodgrass, "Partitioned storage for temporal databases," *Inform. Sys.*, vol. 13, 4, 1988, pp. 269-391.

[4] G. Ariav, "A temporally oriented data model," *ACM Trans. Database Syst.*, vol. 11, 4, Dec. 1986, pp. 499-527.

[5] J. Clifford, A. Croker, "The historical relational data model (HRDM) and algebra based on lifespans," in *Proc. Int. Conf. Data Eng.*, Feb. 1987, pp. 528-537.

[6] J. Clifford, A. Tansel, "On an algebra for historical relational databases: two views," in *Proc. ACM-SIGMOD Conf. Management of Data*, May 1985, pp. 247-265.

[7] H. Gunadhi, A. Segev, "Physical design of temporal databases," Dept. Computing Sci. Res. and Devt., Lawrence Berkeley Lab., CA, Tech. Rep., LBL-24578, Jan. 1988.

[8] H. Gunadhi, A. Segev, "A framework for query optimization in temporal databases," in *Lecture Notes in Computer Science*, 1990, forthcoming.

[9] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proc. ACM-SIGMOD Conf. Management of Data*, June 1984, pp. 47-57.

[10] C. P. Kolovson, M. Stonebraker, "Indexing techniques for historical databases," in *Proc. Int. Conf. Data Eng.*, Feb. 1989, pp. 127-139.

[11] H. P. Kriegel, "Peformance comparison of index structures for multi-key retrieval," in *Proc. ACM-SIGMOD Conf. Management of Data*, May 1984, pp. 186-196.

[12] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner and J. Woodfill, "Designing DBMS support for the temporal dimension," in *Proc. ACM-SIGMOD Conf. Management of Data*, May 1984, pp. 115-130.

[13] S. B. Navathe, R. Ahmed, "TSQL -- a language interface for history data bases," in *Proc. Int. Conf. Temporal Aspects of Inform. Syst.*, North-Holland, 1987, pp. 113-128.

[14] D. Rotem, A. Segev, "Physical organization of temporal data," in *Proc. Int. Conf. Data Eng.*, Feb. 1987, pp. 547-553.

[15] A. Segev, H. Gunadhi, "Event-join optimization in temporal relational databases," in *Proc. Int. Conf. Very Large Data Bases*, Sept. 1989, pp. 205-215.

[16] A. Segev, A. Shoshani, "Logical modeling of temporal databases," in *Proc. ACM-SIGMOD Conf. Management of Data*, May 1987, pp. 454-466.

[17] A. Segev, A. Shoshani, "The representation of a temporal data model in the relational environment," in *Lecture Notes in Computer Science*, vol. 339, M. Rafanelli, J.C. Klensin and P. Svensso, Eds. New York: Springer-Verlag, 1988, pp. 39-61.

[18] A. Shoshani, K. Kawagoe, "Temporal data management," in *Proc. Int. Conf. Very Large Data Bases*, Aug. 1986, pp. 79-88.

[19] R. Snodgrass, "The temporal query language TQuel," *ACM Trans. Database Syst.*, vol. 12, 2, June 1987, pp. 247-298.

[20] R. Snodgrass, I. Ahn, "A taxonomy of time in databases," in *Proc. ACM-SIGMOD Conf. Management of Data*, May 1985, pp. 236-246.

[21] R. Snodgrass, I. Ahn, "Performance analysis of temporal queries," Dept. of Comp. Sci., Univ. of North Carolina, Chappel-Hill, TempIS Doc. No. 17, Aug. 1987.

[22] Special issue of *IEEE Database Eng.*, vol. 11, 4, 1988.

[23] M. Stonebraker, "The design of the POSTGRES storage system," in *Proc. Int. Conf. Very Large Data Bases*, Sept. 1987, pp. 289-300.

[24] M. Stonebraker, L. Rowe, "The design of POSTGRES," in *Proc. ACM-SIGMOD Conf. Management of Data*, May 1986, pp. 340-355.

[25] M. Stonebraker, T. Sellis, and E. Hanson, "An analysis of rule indexing implementations in data base systems," in *Proc. Int. Conf. Expert Database Syst.*, April 1986, pp. 353-364.

[26] T. J. Teorey, J. P. Fry, *Design of database structures*. Englewood Cliffs: Prentice-Hall, 1982.

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
INFORMATION RESOURCES DEPARTMENT
1 CYCLOTRON ROAD
BERKELEY, CALIFORNIA 94720