Efficient Differential Timeslice Computation[*]

Kristian Torp   Leo Mark   Christian S. Jensen

Abstract

Transaction-time databases record all previous database states and are ever-growing, leading to potentially huge quantities of data. For that reason, efficient query processing and the utilization of cheap write-once storage media is of particular importance. This is facilitated by adopting a log-based storage structure. Timeslices, i.e., relation states or snapshots, are computed by traversing the logs, using previously computed and cached timeslices as outsets. When computing a new timeslice, the cache will contain two candidate outsets: an earlier outset and a later outset. The new timeslice can be computed by either incrementally updating the earlier outset or decrementally downdating the later outset. This paper proposes an efficient algorithm that efficiently identifies the cheaper outset.

The perhaps most obvious algorithm uses the proximity in time between the earlier and later outsets and the new timeslice as the basis for its measure of cost. Unfortunately, this is not a reliable measure of the in the number of changes recorded in the logs between each of the two outsets and the new timeslice. The amount of change to the database may vary substantially over time. We subsequently investigated a number of index structures on the timestamps in the logs, including B$^+$-trees, Monotonic B$^+$-trees, and Append-only trees. The fundamental idea was that determining the relative positioning, of the timestamps of the earlier, the new, and the later timeslices, in an index would allow a computation of the corresponding number of changes recorded in the logs between these times. Unfortunately, the lack in these index structures of either a homogeneous node structure, a controllable fill-factor for nodes, or of an appropriate node allocation algorithm greatly complicated the computation.

Consequently, a specialized index structure was developed to support the algorithm. We present and analyze two variations of this index structure, the Insertion tree (I-tree) and the Pointer-less Insertion tree (PLI-tree). The cost of using one of these trees for picking the optimal outset for timeslice computation is only slightly lower than that of using a B$^+$-tree. However, being sparse and packed, I-trees and PLI-trees require little space overhead, and they are cheap to maintain as the underlying relations are updated. The trees also provide a basis for an algorithm that precisely and efficiently predicts the actual costs of computing timeslices in advance. This is useful for query optimization and can be essential in real-time applications. Finally, it is demonstrated how the trees can be used in the computation of other types of queries. As a proof of the functionality of the I- and PLI-tree we have implemented main memory versions of both.

Keywords: Transaction-time, data models, snapshots, timeslice, incremental computation.

[*]Mr. Torp and Dr. Mark are with College of Computing, Georgia Institute of Technology, Georgia, GA 30332, USA, {kristian,leomark}@cc.gatech.edu. Dr. Jensen is with Department of Mathematics and Computer Science, Aalborg University, Fredrik Bajers Vej 7E, DK-9220 Aalborg Ø, DENMARK, csj@iesd.auc.dk.

1 Introduction

A transaction-time database records for each tuple the time when it was current in the database and thus records the history of the database [SA 85, JCEGHJ 94]. Database systems supporting transaction time are useful in a wide range of applications, including accounting and banking, where transactions on accounts are stored, as well as in many other systems where audit trails are important [EN 89]. Applications also include the management of medical records [DWB 86].

Recent and continuing advances in hardware have made the storage of ever-growing and potentially huge transaction-time databases a practical possibility. In order to make transaction-time systems practical, the hardware advances must be combined with advances in query processing techniques. Research focus has spread from conceptual data modeling aspects to also include implementation-related aspects [Sto 87, LS 90a, LS 90b], and significant effort has recently been devoted to implementation-related topics (e.g., see [RS 87, McK 86, SS 88, Soo 91]).

The timeslice operator [Sch 77, AB 80] is one of the central operators in temporal database systems. Indeed, most temporal relational algebras [MS 91] proposed to date contain a variation of this operator, and user-level, temporal query language proposals frequently provide special syntax for timeslice queries. Further, a substantial portion of the natural-language queries in a recent consensus test suite for temporal query languages [J 93] may be implemented using the timeslice operator. The timeslice *R(t)*, of a relation R at a time, t, not exceeding the current time, is the snapshot state of the relation R as of time t.

The transaction-time data model used in this paper is *the backlog model* [JM 92]. In this model, a backlog is generated and maintained by the system for each relation defined in the database schema. The change requests (i.e., inserts and deletes) to a relation are appended to its backlog. A relation is derived from a backlog by using the timeslice operator. Besides the attributes of the associated relation, each tuple in a backlog contains attributes which make the implementation of the timeslice operator possible, such as a transaction timestamp.

Data is, at least in principle, never deleted from a transaction-time database, meaning that it may eventually contain very large amounts of data. For transaction-time databases to be useful, queries must be processed efficiently. One way to improve efficiency is to use *differential computation*, i.e., incrementally or decrementally compute queries from the cached results of similar and previously computed queries [Rou 91, JM 93, BCL 89, BLT 86, LHMPW 86, CSLLM 90].

The I-tree is a degenerate, sparse B$^+$-tree designed for append-only relations, such as backlogs where data is inserted in transaction timestamp order. Thus, insertions are done in the right-most leaf only, and nodes may be packed completely because node splitting never occurs. The PLI-tree is a specialization of the I-tree which contains no pointers; they are replaced by computation.

In this paper we use an I- or PLI-tree as a permanent index on the transaction timestamps in a backlog. The tree is updated every time a new page of change requests is allocated for the backlog. Given a transaction timestamp, the tree efficiently locates the disk pages containing the change request(s) with that timestamp or the most recent earlier timestamp. This ability of the I- and PLI-tree, to find the *page position* of a change request corresponding to a given timestamp value, is exploited during timeslice computation as described next.

When given a time t_x at which to timeslice relation R, the times t_{x-1} and t_{x+1} of the nearest earlier and later timeslices, i.e., $R(t_{x-1})$ and $R(t_{x+1})$, respectively, in the cache are identified. The I- or PLI-tree is then used to compute page positions for the times t_{x-1}, t_x, and t_{x+1} in the backlog. These positions can then be used to predict whether it is going to be more efficient to incrementally compute $R(t_x)$ from $R(t_{x-1})$ or decrementally compute $R(t_x)$ from $R(t_{x+1})$.

The most efficient outset for differential computation of the timeslice operator can be chosen with little overhead, and the cost of computing the timeslice can be predicted precisely and efficiently, which is useful in e.g., real-time applications. Also, the I- or PLI-tree can be used to find the exact position of a change request in the backlog. This can be used to retrieve statistics which are useful for query optimization.

The paper is organized as follows. Section 2 describes the data structures and operators in the backlog model. Section 3 defines the I-tree and explains why it is well-suited for backlogs. This section also compares the I-tree with related indices. Section 4 defines the PLI-tree and compare it with the I-tree. Section 5 shows how an I- or PLI-tree index on a backlog can be used to decide whether incremental or decremental computation is most efficient. Section 6 shows other situations where the I- or PLI-tree can be used. Finally, Section 7 summarizes the paper and points to directions for future research. An appendix contains the algorithms that formed the basis for implementing the I- and the PLI-tree.

2 Implementation Model for Transaction-Time Databases

Several data models support transaction-time; for a recent survey, please see [Sno 92]. In this paper we have chosen the backlog data model because it is a very simple *first normal form* data model and because the simple query language has a formal semantic based on the relational algebra [Sno 92]. In this section we review the structures used in the backlog model [JM 92, JM 93]. Next, the timeslice operators is formally defined, and finally differential computation is informally introduced.

2.1 Backlog and Cache

For each relation R defined in the database schema, the database system generates and maintains a backlog B_R. A backlog B_R is a relation which contains the entire history of change requests to the relation R. The schema of a relation R and its corresponding backlog B_R is shown in Figure 1.

R:	$A_1 : D_1$	$\ldots$	$A_n : D_n$

B_R:	Id : *surr*	Operation : $\{Ins, Del\}$	Time : $TTIME$	$A_1 : D_1$	$\ldots$	$A_n : D_n$

Figure 1: The attributes of the relation R and its backlog B_R.

The backlog schema contains all attributes A_1 to A_n defined in the corresponding relation. In addition the backlog has three new attributes: "Id," a surrogate used as a tuple identifier for change requests in the backlog; "Operation," an indicator of whether the change request is an insertion or a deletion (updates are modeled as a deletion/insertion pairs with the same transaction time-stamp); and "Time," an instant-valued transaction timestamp. It is assumed that each change request has a unique transaction timestamp (except updates) and that the backlog is ordered in transaction-time order.

Figure 2 shows the effect on the backlog B_R resulting from a change request to the relation R.

When an insertion to R is requested, an insertion change request is appended to B_R. When a deletion from R of a tuple with key value k is requested, a deletion change request is appended to B_R. The function *tuple(k)* returns the tuple identified by k, a primary key value in relation R. To efficiently compute this function, we assume that a cache for $R(now)$ is maintained eagerly. When an update of a tuple with key value k is requested, two change requests are appended to the backlog, namely a deletion of the tuple with the key value k and an insertion of the tuple with key value k and the new attribute values.

Change request to R	Effect on B_R
insert R(tuple)	insert B_R(id, *Ins*, time, tuple)
delete $R(k)$	insert B_R(id, *Del*, time, *tuple(k)*)
update $R(k$, *new values*)	insert B_R(id, *Del*, time, *tuple(k)*) insert B_R(id, *Ins*, time, k, *new values*)

Figure 2: Operations on a relation and their effect on the backlog.

Note that the storage space requirements are $\mathcal{O}(n)$ where n is the total number of different versions of all tuples. The insertion or a deletion of a tuple results in a single change request being appended to the backlog. The modification of an existing tuple results in two change requests being appended.

The results of previously computed timeslices are saved in *view pointer caches* [Rou 91]. A view pointer cache is a stored data structure containing a collection of pointers to change requests needed to materialize the result of the corresponding timeslice.

The structure of a *view pointer cache* is

$$(TID' : \text{surr}, TID : \text{surr}, PID : \text{ptr})$$

where values of the attribute TID' are surrogates used as tuple identifiers in the view pointer cache. Values of the attribute TID are surrogates referring to change requests in the backlog. Finally, values of the attribute PID are pointers to pages in the backlog where the change requests are stored. View pointer caches may also be used for caching results of more general queries than timeslices, but this is beyond the scope of this paper.

It is obvious, that if we store a cache every single time a new timeslice is computed and never reclaim space, then eventually the space requirements will be prohibitive. To solve this problem, we assume that a fixed amount of space will be allocated for storing caches. The choice of an appropriate cache replacement strategy is beyond the scope of this paper.

2.2 Algebra Operators

The five basic relational operators are retained in the algebra for the backlog model. A relation must be timesliced before it is used as an operand in an algebra expression. The idea of timeslicing can be expressed as follows.

$$\begin{array}{rll} R(t_x) & \text{denotes} & R \text{ at time } t_x,\ t_{init} \leq t_x \leq \textit{now} \\ R & \text{denotes} & R(\textit{now}) \end{array}$$

The variable *now* has the value of the current time, and t_{init} is the time when the database was initialized. A formal definition of the timeslice operator is given next [JM 93].

Let R be a relation having attributes $A_1, A_2, \ldots, A_n$ where *Key*, a time-invariant key value, is one of these. The timeslice $R(t_x)$ is given by

$$\begin{aligned} R(t_x) = \{y^{(n)} \mid \exists s\, (& s \in B_R \wedge y[1] = s[1] \wedge y[2] = s[2] \wedge \ldots \wedge y[n] = s[n] \wedge \\ & s[\text{Time}] \leq t_x \wedge s[\text{Operation}] = \textit{Ins} \wedge \\ & \neg\exists u (u \in B_R \wedge s[\text{Key}] = u[\text{Key}] \wedge s[\text{Time}] < u[\text{Time}] \leq t_x))\} \end{aligned}$$

As can be seen, the timeslice is computed from the backlog. First, the attributes of the result are selected. In the second line, all insertions that are before the timeslice time are identified. The third line serves to eliminate all those insertions that have been countered by deletions before the time of the timeslice.

2.3 Incremental and Decremental Computation

Informally, the idea of differential computation is to reuse the cached results of previously computed timeslices together with intermediate change requests to compute the results of new timeslices. When a timeslice query is issued against the database, the closest earlier and later timeslices that have been previously computed and cached are located. Note that it is straightforward to locate these cached timeslices; for example, a B$^+$-tree on the times of the timeslices can be used. The problem of query matching when more general queries may be cached for reuse is addressed elsewhere [Rou 91, JM 93]. This paper addresses the problem of how to efficiently determine whether incremental computation, using the earlier timeslice, or decremental computation, using the later timeslice, is most cost-efficient.

Figure 3: Differential computation of the timeslice operator.

The idea of differential computation of timeslices is shown in Figure 3. The structures $R(t_{x-1})$ and $R(t_{x+1})$ are view pointer caches containing the results of previously computed timeslices, and $R(t_x)$ is the timeslice to be computed. To incrementally update $R(t_{x-1})$ to $R(t_x)$ the change requests between A and B in Figure 3 must be traversed. Similarly, to decrementally "downdate" $R(t_{x+1})$ to $R(t_x)$ the change requests between C and B must be traversed. Algorithms for incremental and decremental computation of timeslices have been described previously [JMR 91, JM 93].

3 I-trees

In this section, we first review related work on indices and explain why a new index is needed. We then describe the structure of the I-tree and show how the tree is maintained during insertions to backlogs.

3.1 Related Work

The I-tree is a degenerate B$^+$-tree similar to and inspired by the *Monotonic B$^+$-tree* (MB-tree) [EWK 93] and the *Append-only tree* (AP-tree) [GS 93]. The I-tree is specifically designed to index monotonically increasing values. If a regular B$^+$-tree was used for this, the nodes would only be approximately 50% full [GS 93].

Several other indices may be used to index monotonically increasing key values; for an overview, please see [T 93]. Here, we describe why we have defined a new index instead of adopting any of the two most closely related indices, the MB-tree and the AP-tree.

The MB-tree has not been used for three reasons. First, the internal nodes and the leaf nodes have different formats, and the leaf nodes can be different in size. This lack of homogeneity is not desirable for our purposes. Second, internal nodes in the right-most subtree are allocated before they are needed, yielding an avoidable space overhead. Third, in the insertion algorithm extra parameters are given to be able to implement a *Time Index* [EWK 93]. This extra generality, not needed for our use, complicates the insertion algorithm.

The AP-tree has not been used for three reasons. First, all pointers between nodes in the AP-tree are double pointers. For our problem, single pointers will do. Second, not all pointers are used in the internal nodes of the AP-tree, giving a slight waste of space. Third, when nodes in the right-most subtree of the root are appended, the chain from the root to the right-most leaf must be traversed. This requires that these nodes are stored in main memory or read from secondary storage.

The I-tree and PLI-tree are designed for our specific problem, and it has been possible to eliminate the problems mentioned above, as we shall see next.

3.2 Characteristics of the I-tree

An I-tree of order d has the following properties. An example follows the properties.

1. All nodes (including the root, internal nodes, and leaf nodes) are of the format

$$< \; P_0, \; t_0, \; P_1, \; t_1, \ldots, P_{d-2}, \; t_{d-2}, \; P_{d-1} \; >$$

   where the P_i are pointers and the t_i are (search) key values.

2. A root node has $i + 1$ (non-null) pointers and i (non-null) key values if it is not a leaf, $1 \leq i \leq d - 1$. The root has one pointer if it is also a leaf.

3. All internal nodes and leafs, except the right-most node/leaf at each level, have exactly d (non-null) pointers and $d - 1$ (non-null) key values. Such nodes are full.

4. Within each full node, $t_0 < t_1 < \ldots < t_{d-2}$. In general, all non-null key values are ordered this way.

5. All leaf nodes that are descendants of non-right-most subtrees of the root are at the same level. This portion is full and balanced. The last pointer on each leaf node, P_{d-1}, is used to chain leaf nodes together in search key order.

6. Insertions are always done in the right-most leaf, deletions do not occur, and search is as in B^+-trees.

7. For a tree of height h and with a root that has n subtrees, $2 \leq n \leq d - 1$, these properties hold.

   - The first $n - 1$ subtrees are full and balanced and have height $h - 1$.
   - The n'th subtree has at least height zero and at most $h - 1$.
   - When the n'th subtree is full, a new subtree of height zero is created (explained further in the next section). If the height of the tree is increased by one, the new root only has two subtrees: one that is filled and one of height zero.

Figure 4: An example of an I-tree of height $h = 2$ and order $d = 3$.

An example of an I-tree is shown in Figure 4. The tree shown is of height $h = 2$ and order $d = 3$. The chain of pointers and nodes to the right is called the *right-most chain.* In Figure 4, the right-most chain consists of the root, the boldface pointer, and the right-most leaf, termed the *current node.* The array is a dynamic array containing pointers to all nodes in the right-most chain [EWK 93]. These pointers are used when insertions are made to the tree. In Figure 4 Position 0 of the array points to 6 and Position 1 points to 5. The numbers 5 and 6 refer to the numbers shown above the right corner of each node. These numbers are used for illustrating the dynamics of the index and are not part of the data structure. Furthermore, for each position in the array, the level of the node is stored along with an indication of whether the node is full or not. Figure 4 also shows that the right-most subtree of the root needs not be balanced—there is no node at Level 1.

3.3 Insertion into the I-tree

We give a comprehensive description of insertion into the I-tree by means of three examples that cover all possible combinations. The algorithms that formed the basis for the implemention of the I-tree are listed in appendix A.

1. Figure 5A shows an example of the general case where the whole tree is completely full and balanced. The key value 17 must be inserted. A new leaf is created and the new value inserted. A new root is created and the last key value in the old current node is inserted. The left-most pointer of the new root is set to point to the old root, and the right-most pointer is set to point to the new leaf. The array is properly updated.

2. Figure 5B shows an example of the general case where a non-full node is found in the right-most chain. The number of levels between the closest non-full node and the next node in the right-most chain is one. The key value 28 must be inserted. A new leaf is created and the value 28 is inserted. The last key value in the old current node is inserted in the non-full node, and a new right-most pointer in the node is set to point to the new leaf. The array is updated to reflect the new leaf node.

3. Figure 5C shows an example of the general case where all nodes in the right-most subtree of the root are full, but the subtree is not balanced. This case also covers the situation when the root is full but the right-most subtree of the root not balanced. The key value 151 is to be inserted. A new leaf is created and the new key value is inserted. The last key value of the old current node is inserted in a new node created between the right-most node, found to point directly to a leaf, and the new leaf. The array is properly updated.

Figure 5: Examples of appends to the I-tree

4 PLI-trees

To increase the number of key values in each node of the I-tree, thereby reducing the index size, all pointers in nodes may be eliminated. Assuming that the nodes (disk pages) of an I-tree are stored on disk in allocation order and consecutively, pointers may be replaced by computation. The resulting data structure is called the Pointer Less I-tree (PLI-tree).

In this section, we first describe the characteristics of the PLI-tree. Then we give an example of the tree and explain how search is accomplished. Next, the requirement of consecutive storage is relaxed. Finally, we compare I-trees and PLI-trees.

4.1 Characteristics of the PLI-tree

An PLI-tree of order d has the following properties. An example follows the properties.

1. All nodes (including the root, internal nodes, and leaf nodes) are of the format

$$< \; t_0, \; t_1, \ldots, \; t_{d-2} >$$

   where the t_i are key values.

2. A root node has i (non-null) key values if it is not a leaf, $1 \leq i \leq d-1$.

3. All internal nodes and leafs, except the right-most node/leaf at each level, have exactly $d-1$ (non-null) key values. Such nodes are full.

4. Within each full node, $t_0 < t_1 < \ldots < t_{d-2}$. In general, all non-null key values are ordered this way.

5. All leaf nodes that are descendants of non-right-most subtrees of the root are at the same level. This portion is full and balanced.

6. Insertions are always done in the right-most leaf, deletions do not occur, and search is *logically* done as in B$^+$-trees.

7. For a tree of height h and with a root that has n subtrees, $2 \leq n \leq d-1$, these properties hold.

    - The first $n-1$ subtrees are full and balanced and have height $h-1$.
    - The n'th subtree has at least height zero and at most $h-1$.
    - When the n'th subtree is full, a new subtree of height zero is created. If the height of the tree is increased by one, the new root only has two subtrees: one that is filled and one of height zero.

Compared to the I-tree, all properties regarding pointers are eliminated. Notice that there are no pointers between leaf nodes in a PLI-tree, and that the search algorithm is *logically* the same as for the B$^+$-tree.

Figure 6: An example of a PLI-tree of height $h = 2$ and order $d = 3$.

An example of PLI-tree of height $h = 2$ and order $d = 3$ is shown in Figure 6. Compare this figure to Figure 4. The dynamic array used in I-trees is retained and contains pointers to all nodes in the right-most chain. The numbers shown above each node are *node numbers.* They are not a part of the data structure, but are essential in the following explanation because they indicate the allocation order of nodes.

4.2 Insertion and Search in the PLI-trees

The PLI-tree insertion algorithm is almost identical to that of the I-tree. The difference is that nodes are allocated in in-order and the pointer manipulation between nodes is eliminated. Appendix A lists the insertion algorithms for both the I-tree and the PLI-tree, and compares them.

The search algorithm for the PLI-tree is different from the search algorithm for the B$^+$-tree and I-tree. This is due to the fact that no pointers exists between nodes. We discuss how searching is accomplished below; further details are provided in the appendix. In the following, we use the word "pointer" to mean the "implicit pointers" between the nodes.

Figure 7A shows a PLI-tree. The numbers above each node show the order in which nodes are allocated. It can be seen that the nodes are allocated in in-order. Figure 7B shows how we assume that the nodes are stored in a file. The start address of a file is always known. The node numbers make searching without pointers possible because they are the offset within the file.

To describe search in the PLI-tree, the following parameters are needed, see also Figure 7 for further explanation.

Figure 7: An example of a PLI-tree and how nodes are stored in a file.

Name	Description
h	height of the PLI-tree
d	order of the PLI-tree
p	pointer number in a node
l	level number in the tree

The search algorithm is called with a key value when the tree contains more than one node. First the root and the level of the root is found in the dynamic array. The node number of the root can be computed as follows.

$$\textit{root number} \ = \ \Sigma_{i=0}^{h-1} d^i$$

Explanation: The PLI-tree is full except for the right-most subtree. When the nodes are allocated in in-order, a subtree of height one smaller than the height of the PLI-tree is allocated before the root. Notice the first node number is 0. In Figure 7 the root number is:

$$\textit{root number} \ = \ \Sigma_{i=0}^{2-1} 3^i \ = \ 1 \ + \ 3 \ = \ 4$$

If the left-most pointer of a node is followed, we are going to a node that was allocated earlier; we thus subtract all nodes that were allocated between the old node and the new node. This number is computed as follows.

$$\textit{new number} = \begin{cases} \textit{old number} - ((d-1)(\Sigma_{i=0}^{h-(l+2)} d^i) + 1) & \text{if } l+2 \leq h \\ \textit{old number} - 1 & \text{if } l+2 > h \end{cases}$$

Explanation: $d-1$ is the number of subtrees of the new node that were allocated between the old and the new node. The sum finds the number of nodes in a subtree of the new node, thus the $h-(l+2)$. The second case is needed because the new node may have no subtrees. The $+1$ is for the old node. If we follow the left-most pointer of the root in Figure 7 the new node number is given by:

$$\textit{new number} \ = \ 4 - ((3-1)(\Sigma_{i=0}^{2-(0+2)} 3^i) + 1) \ = \ 4 - (2 \cdot 1 + 1) \ = \ 1$$

If the pointer followed is not the left-most pointer, we go to a node that was allocated later. Two possibilities exist. We are in the right-most subtree or we are not in the right-most subtree. In the latter case, the new node number is found by the formula below.

$$new\ number = \begin{cases} old\ number + (p_l - 1)\Sigma_{i=0}^{h-(l+1)} d^i + \Sigma_{i=0}^{h-(l+2)} d^i + 1 & \text{if } l+2 \leq h \\ old\ number + (p_l - 1)\Sigma_{i=0}^{h-(l+1)} d^i + 1 & \text{if } l+2 > h \end{cases}$$

Explanation: We name the pointer number followed at level l, p_l. Between the old and the new node, we have allocated $p_l - 1$ subtrees of nodes of height one smaller than the height of the old node; thus the first sum. We have also allocated the left-most subtree of the new node and the old node itself, thus the second sum and the +1. Again, two cases are needed to account for empty subtrees. If we follow pointer number two in node number 1 in Figure 7, the new node number is given by:

$$new\ number \; = \; 1 + (2-1)\Sigma_{i=0}^{2-(1+1)} 3^i + 1 \; = \; 1 + 1 \cdot 1 + 1 \; = \; 3$$

If we are in the right-most chain the dynamic array must be used to find the level of the new node. The number of the new node if found as follows.

$$new\ number = \begin{cases} old\ number + (d - 2)\Sigma_{i=0}^{h-(l+1)} d^i + \Sigma_{i=0}^{h-(l+jump+1)} d^i + 1 & \text{if } l+jump+1 \leq h \\ old\ number + (d - 2)\Sigma_{i=0}^{h-(l+1)} d^i + 1 & \text{if } l+jump+1 > h \end{cases}$$

Explanation: The change from the previous formula is that the left-most subtree of the new node may be a smaller tree depending on the number of levels between the old node and the new node. Thus in the second sum, we use the number of level between the nodes, the *jump*. Notice that if the levels are only one apart then the sum yields the same as in the previous formula If we go from the root to the right-most leaf in Figure 7 then the new number is given by:

$$new\ number \; = \; 4 + (3-2)\Sigma_{i=0}^{2-(0+1)} 3^i + 1 \; = \; 4 + 1 \cdot (1+3) + 1 \; = \; 9$$

From the node number calculated, the new node is retrieved from the file containing the nodes of the PLI-tree, using the start address of the file and the node number (the offset). This continues until a leaf is reached. Note that the new node has no leftmost subtree.

4.3 Implementing the PLI-tree Using Extents

In the design of the PLI-tree, we have assumed that nodes (disk pages) in the tree are stored consecutively on a disk. This makes it possible to access a node in a file on disk by a start address and an offset. This assumption can normally not be made in a multiuser database system where nodes are allocated dynamically. Here disk space is allocated in chunks called *extents* [Sal 88]. An extent is a number of consecutive disk pages. All extents contains the same fixed number of disk pages. Within an extent, disk pages can be accessed via a start address and an offset.

To make it possible to search a PLI-tree, without extra I/O-cost, an array containing start addresses of all extents, in which the PLI-tree is stored, must be in main memory. The first slot of this *extent array* stores the start address of the first extent allocated for the PLI-tree. Figure 8B shows how nodes of the PLI-tree are stored in extents. In the figure, an extent consists of three disk pages. Compare this to Figure 7. The extent array in Figure 8A contains the start address of extents. Slot number zero points to extent number zero, etc.

From the node number, the start address and the offset must be computed to retrieve the node from disk. The start address can be found by computing in which logical extent number the node

Figure 8: An example of a PLI-tree and how nodes are stored in extents.

is stored and then making a lookup in the extent array. From the extent number, the number of pages in each extent, e_{size}, and the node number the offset can be found.

The extent number (start address) is given by the following formula.

$$\textit{extent number} \ = \ \left\lfloor \frac{\textit{node number}}{e_{size}} \right\rfloor$$

The offset is given af follows.

$$\textit{offset} \ = \ \textit{node number} \ - \ \textit{extent number} \ \cdot \ e_{size}$$

As an example consider finding the extent number and the offset for node number 8 in Figure 8. The extent number is $\textit{extent number} \ = \ \left\lfloor \frac{8}{3} \right\rfloor \ = \ 2$ and the offset is $\textit{offset} \ = \ 8 \ - \ 2 \ \cdot \ 3 \ = \ 2$.

When we in the following specifically refer to the PLI-tree stored in extents, we term the data structure an ePLI-tree.

4.4 I-tree versus PLI-tree

The fanout of the PLI-tree is larger compared to the fanout of the equivalent I-tree because the pointers in the nodes are eliminated. If it is assumed that timestamps are 64 bit [Sno 92] and pointers (Unix) 32 bit, then the fanout of a PLI-tree is approximately 50% higher than the fanout of an equivalent I-tree. The fanout of the PLI-tree and the ePLI-tree are the same. The table below shows the fanout for an I-tree and a PLI-tree for different page sizes.

Page size	512	1024	2048	4096	8192
PLI-tree	64	128	256	512	1024
I-tree	42	85	170	341	682

The I/O-costs of the insertion algorithm is the same for I-trees, PLI-trees, and ePLI-trees because we assume that the dynamic array and extent array are in main memory. The I/O-cost of the search algorithm for PLI-trees and ePLI-trees is smaller than that of I-trees due to the higher fanout of PLI-trees and ePLI-trees.

The I-tree and PLI-tree are further compared in Section 5.4.

5 Use of I-trees/PLI-trees for Differential Timeslice Computation

The I-tree and the PLI-tree are very similar, so we use the I-tree as an example in this section and explain when there are differences. We first describe how to find a page position in the backlog using the I-tree. Second, we estimate the size of the I-tree and the I/O-cost of maintaining it. Third, we present an algorithm for choosing between incremental and decremental computation of timeslices. Finally, the I-tree and the PLI-tree are compared.

The basic idea is to maintain an I-tree on a backlog and use it to determine if it is more cost efficient to incrementally update a cached timeslice $R(t_{x-1})$ to $R(t_x)$, $t_{x-1} \leq t_x$, or decrementally "downdate" a cached timeslice $R(t_{x+1})$ to $R(t_x)$, $t_{x+1} \geq t_x$, see Figure 3. The nodes in the I-tree contain transaction timestamps. Figure 9 shows an I-tree on a backlog.

Figure 9: An I-tree and a backlog.

5.1 Finding Page Positions Using the I-tree

Given a change request with transaction timestamp t, its *page position* in the backlog is found by using the I-tree. By page position, we mean the logical number of the disk page of the backlog on which the change request is stored. To find the number of change requests between two times, the page positions in the backlog of change requests with approximately those times are computed using the I-tree. Then the desired number can be found by subtraction. To compute a page position, the four parameters, h, d, p, and l listed on page 10 are needed, see also Figure 9. The page position is found by searching the I-tree, with t as a parameter. For each level, it is recorded which pointer number is followed to the next level. The page position is calculated from this information, by this formula:

$$Page\ Position = (d-1) \cdot \sum_{l=0}^{h-1} (d^{h-(l+1)} \cdot p_l) + p_h$$

The formula may be explained as follows. The I-tree is balanced and all nodes are full to the left. This means that each time a pointer is skipped in a node at Level l, $(d - 1) \cdot d^{h-l-1}$ pointers to disk pages of the backlog are passed at the leaf level. The factor p_l is the pointer number p followed to the next level, at level l. The formula sums up the number of disk pages passed at each level from the root to the level just above leaf level. At the leaf level, one disk page is passed each time the pointer number p is increased by one—this is p_h. The parameters (p_l values) needed to compute the page position are provided using the I-tree. Notice the first disk-page number is 0. As an example, consider finding the page position on which the change request with transaction timestamp = 13 is stored, see Figure 9. In this example $h = 2$ and $d = 3$. The page position is therefore $(3 - 1) \cdot (3^{2-0-1} \cdot 0 + 3^{2-1-1} \cdot 1) + 1 = 2 \cdot (0 + 1) + 1 = 3$.

The I/O-cost of computing a page position is h disk accesses, the height of the tree.

That the page position can be calculated is the reason no pointers are needed at the leaf level in a PLI-tree indexing a backlog.

5.2 Maintenance of the I-tree

In this subsection we estimate the size of the I-tree and the I/O-cost for maintaining the I-tree as a permanent index on the backlog.

5.2.1 I-tree Size versus Backlog Size

The I-tree does not need to contain the transaction timestamps of all the change requests in the backlog. Only approximately one transaction timestamp for each disk page is needed—the I-tree is a sparse index. Two I-trees of the same height are shown in Figure 10. For this height, Figure 10A shows a worst-case situation where the I-tree's size is the largest possible compared to the size of the backlog. The right-most leaf has just been allocated. For the same height, Figure 10B shows a best-case situation where the size of the I-tree is the smallest possible compared to the size of the backlog. This I-tree is full and balanced.

Figure 10: A) Worst-case situation. B) Best-case situation.

The following two expressions are valid for height $h \geq 1$ and order $d \geq 2$. In the worst-case and best-case situations, the size of the I-tree compared with the size of the backlog is described next.

$$\textit{Worst case:} \quad \left(\left(\sum_{m=0}^{h-1} d^m\right) + 2\right) \Big/ \left((d-1) \cdot d^{h-1} + 2\right)$$

$$\textit{Best case:} \quad \sum_{m=0}^{h} d^m \Big/ \left((d-1) \cdot d^h + 1\right)$$

Examples of the worst-case and the best-case are shown in the table below for height, $h = 3$ and order, $d = 100$ [Sal 88]. As can be seen, the backlog is approximately d times larger than the I-tree. The size of the index is very small. The difference for the worst-case and best-case is insignificant for realistic trees.

	I-tree	Backlog	% I-tree/Backlog
worst-case	10,103	990,002	1.0205
best-case	1,010,101	99,000,001	1.0203

5.2.2 I/O-Cost of Maintaining the I-tree

The I/O needed to maintain the backlog itself is independent of whether or not there is an I-tree index defined on it. There is an extra I/O-cost related to maintaining the I-tree as an index on the backlog. Here we describe that cost. We assume that the root, the right-most leaf, and the dynamic array of the I-tree are in main memory; and that the costs for allocating, reading, and writing a disk page are all the same.

There is an insertion into the I-tree each time a new disk page is allocated for the backlog. Two cases should be distinguished: when the right-most leaf of the I-tree is not full, and when that leaf is full. The latter situation is rare. It occurs only once for each $(d - 1) \cdot N_{ch}$ appends to the backlog, where d is the order of the I-tree and N_{ch} is the number of change requests which fit in one disk page. In the frequent, first case, the I/O-cost for updating the I-tree is zero. In the second case, the three possibilities shown in Figure 5 exist. The I/O-cost for these three possibilities are shown in the table below.

Description	Figure 5A	Figure 5B	Figure 5C
Allocate new leaf node	$\surd$	$\surd$	$\surd$
Allocate new root/node	$\surd$		$\surd$
Read internal node		$\surd$	$\surd$
Write internal node		$\surd$	$\surd$
Write old root	$\surd$		
Write new node			$\surd$
Write current node	$\surd$	$\surd$	$\surd$
Total I/O-cost	4	4	6

In the worst case, it will require 6 disk access to update the I-tree when the backlog is updated. It should be noted that the I/O-cost is independent of the height of the I-tree.

The smallest (i.e., worst case) average number of change requests that can be appended to a backlog per I/O operation needed to maintain the I-tree index is given by $((d - 1) \cdot N_{ch}/6)$.

The table below shows examples of how many change requests can be appended to the backlog for each I-tree I/O operation, for different realistic page sizes. It is again assumed that transaction timestamps occupy 64 bits [Sno 92], pointers (Unix) 32 bits, and change requests 128 bytes.

Page size (bytes)	512	1,024	2,048	4,096	8,192
Order d	42	85	170	341	682
N_{ch}	4	8	16	32	64
appends per I/O	27	112	450	1,813	7,264

The number of appends per extra disk access grow with the square of the page size because both d and N_{ch} depend on the page size. It is clear that the I-tree is cheap to maintain for realistic page sizes.

Finally, the total cost of maintaining an I-tree for a backlog with $\mid B_R \mid$ change requests is less than $6 \cdot \mid B_R \mid /((d - 1) \cdot N_{ch})$.

5.3 Choosing between Incremental and Decremental Timeslice Computation

We first present the algorithm for choosing between incremental and decremental computation of timeslices in transaction time databases, then we compare the cost of using the algorithm to the cost of doing linear scan.

5.3.1 The Algorithm

To decide whether to use incremental or decremental computation, we must know which one is more efficient in a given situation. In the algorithm presented below, the time t_x is the time of the timeslice to be computed, and t_{x-1} and t_{x+1} are the times of the cached timeslices which are closest to t_x, before and after t_x, respectively.

Observe that there will always exist both a newer timeslice and an older timeslice in the cache, as shown in Figure 3. The timeslice $R(t_{init})$ which is empty is trivially in the cache and will always qualify as an earlier timeslice. Next, we made the assumption that $R(now)$ is eagerly maintained. This timeslice will always qualify as a later timeslice.

The page positions of the times t_{x-1}, t_x, and t_{x+1} are denoted $P_{t_{x-1}}$, P_{t_x}, and $P_{t_{x+1}}$ in the following. The number of pages that must be read in order to do an incremental computation is $\mid P_{t_x} - P_{t_{x-1}} \mid + 1$, and the number of pages that must be read in order to do a decremental computation is $\mid P_{t_{x+1}} - P_{t_x} \mid + 1$. The two values are compared and the result reveals the cheapest computation method. The pseudo algorithm for choosing between incremental and decremental timeslice computation follows.

```
incremental_or_decremental($B_R$, $t_x$)
    Find R($t_{x-1}$) where $t_{x-1} = \max(t \leq t_x \wedge \exists$ a timeslice at time $t)$
    Find R($t_{x+1}$) where $t_{x+1} = \min(t \geq t_x \wedge \exists$ a timeslice at time $t)$
    $P_{t_{x-1}} \leftarrow$ page position of time $t_{x-1}$
    $P_{t_{x'}} \leftarrow$ page position of time $t_{x'}$
    $P_{t_{x+1}} \leftarrow$ page position of time $t_{x+1}$
    **if** $\mid P_{t_{x'}} - P_{t_{x-1}} \mid \leq \mid P_{t_{x+1}} - P_{t_{x'}} \mid$
        incremental_timeslice($B_R$, R($t_{x-1}$), $t_{x-1}$, $t_x$) - - function call
    **else**
        decremental_timeslice($B_R$, R($t_{x+1}$), $t_{x+1}$, $t_x$) - - function call
```

Two comments are in order. To efficiently locate the cached timeslices in the first two lines, one may maintain a B$^+$-tree index on the times of the cached timeslices. Next, it may be the case that no change request exists in the backlog with timestamp t_x. Thus, the page position of the change request with the largest timestamp $t_{x'}$ that does not exceed t_x is computed.

5.3.2 Comparison with Linear Scan

If no I-tree is available on the backlog, the *only* existing way to find the cost of computing the timeslice is to actually compute the timeslice. Due to a lack of regularity, B$^+$-trees, MB-trees, and AP-trees do not allow a precise and efficient prediction of the cost of computing time-slices in advance. Therefore, to investigate if the timeslice computation using an I-tree is cost efficient, we compare it to the timeslice computed by a linear scan of the backlog from $P_{t_{x-1}}$ up to $P_{t_{x'}}$.

To find the two timeslices closest to the desired time t_x, a lookup is done in the cache. We assume that the cost for this is the same in both situations. The cost of finding the positions $P_{t_{x-1}}$, $P_{t_{x'}}$, and $P_{t_{x+1}}$ in the I-tree is h disk accesses each. In the average-case, the linear scan will be $3 \cdot h$ disk accesses better in approximately 50% of the cases. There is a cost for the estimate, but this cost can reduce the total cost of computing the timeslice in the other approximately 50% of the cases. Figure 11 shows the general case, where the cost of computing a timeslice by a linear scan is compared to the cost of computing a timeslice using differential computation.

The x-axis indicates the position of P_{t_x} and the y-axis indicates the number of disk accesses needed to compute the timeslice. The solid line assumes that the timeslice $R(t_x)$ is computed

Figure 11: Cost comparison when computing timeslices when using linear scans versus I-trees.

incrementally from $P_{t_{x-1}}$, while the dashed curve assumes that the I-tree is being used to determine whether to do either incremental or decremental update, from position $P_{t_{x-1}}$ or $P_{t_{x+1}}$, respectively. Figure 11 indicates that the I-tree approach looses slightly if the distances between the timeslices are small. At the same time, it shows that there can be significant efficiency gains using the I-tree.

Note that the temporal closeness of the times of timeslices is not a reliable measure for their physical closeness, i.e., for how many change requests that were entered into the backlog between the two timeslice times. To see the problem, consider an example where we assume that no I-tree is available and that the outset closest in time to t_x is chosen. If the database is updated from 8 a.m. to 4 p.m. only and we want to compute a 5 p.m. timeslice for day 5 and we have a day 5, 2 p.m. cached result and a day 6, 9 a.m. cached result then the day 5 outset will be chosen, but the day 6 outset is the best.

5.4 I-tree versus PLI-tree

The I/O cost of maintaining a PLI-tree and an ePLI-tree obey the same formulas as those derived in Section 5.2.2 for the I-tree. The number of appends in the backlog per I/O-cost used for the I-tree and PLI-tree are listed in the table below. It is assumed that timestamps are 64 bit, pointers are 32 bit, and change requests 128 bytes.

Page size	512	1024	2048	4096	8192
PLI-tree	42	169	680	2,725	10,912
I-tree	27	112	450	1,813	7,264

The number of change requests which can be appended per I/O-cost is approximately 50% higher for the PLI-tree compared to the I-tree because the fanout of a PLI-tree is approximately 50% higher than the equivalent I-tree.

The height of PLI-trees and I-trees indexing a backlog consisting of 1 million pages are shown below for for different page sizes.

Page size	512	1024	2048	4096	8192
PLI-tree	3	2	2	2	1
I-tree	3	3	2	2	2

As can be seen, the height of the PLI-tree is at most one smaller than the height of the equivalent I-tree.

The number of disk pages used for storing the PLI-tree, the ePLI-tree, and the I-tree when indexing 1 million pages, are shown below for different page sizes. It is assumed that the extent size is 8 disk pages:

Page size	512	1024	2048	4096	8192
PLI-tree	15,876	7,876	3,924	1,959	978
ePLI-tree	16,869	8,369	4,170	2,082	1,040
I-tree	24,392	11,907	5,919	2,943	1,471

Compared with an I-tree, an equivalent PLI-tree is approximately 33% smaller because pointers are eliminated from the PLI-tree. Compared with a PLI-tree, an equivalent ePLI-tree are larger because pointers to extents must be stored in the extent array. We explore the size of the extent array next.

The table below shows the number of disk pages in the extent array for different page sizes and different extent sizes when an ePLI-tree is used to index a backlog consisting of 1 million disk pages. Further the fanouts of I-trees for the different page sizes are listed in the last row.

	Page size				
Extent size	512	1024	2048	4096	8192
8	993	493	246	123	62
16	497	247	123	62	31
32	249	124	62	31	16
Fanout I-tree	42	85	170	341	692

If the number of disk pages in the extent array is larger than the fanout for the I-tree, for the same page size, it means that instead of storing the extent array in main memory at least the two top levels of an I-tree could be stored in main memory instead of. If this is done then the I/O-cost of using the I-tree will be smaller than the I/O-cost of using the ePLI-tree. This shows that the page size and extent size must be chosen carefully for ePLI-trees.

6 Additional Uses of the I-tree

The I-tree may be used in other situations to make the database system more efficient. Here we show that it can be used to compute an *exact position* of a change request. We discuss how this information can be used in different situations and show how it can be used in *random sampling.*

6.1 Exact Position

Whereas the page position of a change request is the logical number of the disk page on which the change request is stored, the exact position of a change request consists of this page position and an offset. The exact position of a change request is useful when retrieving statistics about a backlog, e.g., which day were the most change requests appended to the backlog.

The exact position of a change request with transaction timestamp t_x can be computed by finding the (logical) page position where the change request resides, using the formula presented in Section 5.1, then read the corresponding disk page and compute the change-request position, called p_{ch}, within that disk page. When the number of change request in a full disk page, N_{ch}, and the number of disk pages skipped at the leaf level of the I-tree, N_{pages}, are known, the change request's exact position in the backlog is given by $N_{pages} \cdot N_{ch} + p_{ch}$. The cost of finding the exact position is $h + 1$ disk accesses.

If a database administrator wants to find out how many change request were appended to a backlog for the past day, he could issue the following backlog-based query.

$$\text{count } (\sigma_{[now\ -\ 1\ day\ \leq\ Time\ \leq\ now]} B_R)$$

where count is an aggregate function which returns the cardinality of its parameter. This query can be converted to two queries which find the exact positions of a change request with transaction timestamp $t = now$ *- 1 day*, and $t = now$. The number of change requests between these positions can be computed from the results of the two exact-position queries. Generally all queries which count the total number of change requests in a transaction time interval can be converted to exact-position queries. This also applies to so-called *time-varying queries* [BM 93], *the moving window operator* [NA 93], and the Σ*-operator* [JM 92, JMR 91].

6.2 Random Sampling

In random sampling a small part of the tuples of a relation are retrieved. Further actions are based on the contents of the tuples in this *sample set*. Random sampling can be used if the processing of the whole relation is too expensive or not necessary. It is typically used to support statistical analysis of a data set, either to estimate parameters of interest for testing hypotheses [ORX 90]. Furthermore, financial databases are subject to annual audits, where random sampling of tuples from databases is used [OR 89].

Simple random sampling (SRS) is sampling where the inclusion probabilities for each tuple in the relation is uniform. SRS is *fixed sized* if the sample size has been specified by the user. Further, the samples can be drawn with or without replacement. If it is drawn with replacement duplicates may occur in the sample set. If it is drawn without replacement duplicates are not allowed.

SRS can be done via indices on the relation [OR 89, ORX 90]. The problem with e.g. a regular B^+-tree is to produce uniform inclusion probabilities for all tuples in the relations. If a pointer number is randomly picked for each node from the root to a leaf in a B^+-tree, tuples reached from nodes with low fanout are more likely to be include in the sample set.

Fixed size SRS on transaction time in a backlog can be supported without modification to the data structures and without increasing the cost of insertions and search in the I-tree. If we have a sample size of j and the number of change request in the backlog is n we generate j numbers between 1 and n. This numbers correspond to a change request's logical number in the backlog. The exact position method is used to find the j numbers. This is similar to the method for SRS in *Ranked* B^+*-trees*. SRS without replacement can be implemented by not allowing the same random number to be generated twice in the same sample.

Assuming the root of the I-tree is in main memory, the I/O-cost for a SRS of fixed size, j, is given by this formula:

$$Cost\ Sampling = j \cdot h$$

The size of the sample times the height of the I-tree, which is the cost of using the exact position method.

If only part of the backlog must be used in the sampling, e.g. a bank is sampling on the last years transaction, the I-tree can be used to narrow the parts of the backlog used by first retrieving the lower and upper bounds using the exact position method.

7 Summary and Future Research

In this paper we have taken one step in the direction of realizing a global query optimizer for temporal queries. We have given a method which can predict whether it is going to be more efficient to incrementally or decrementally compute a *timeslice* from a previously computed and cached result in a transaction-time data model. This has been done by using an *Insertion tree* (I-tree) or a *Pointer-Less Insertion tree* (PLI-tree) as a permanent index on transaction timestamps on *backlogs*, a log-based storage structure. Timeslices are derived from backlogs by using the important *timeslice operator*.

Using the I- or PLI-trees as indices on the transaction timestamps of the backlogs, we have demonstrated that it is possible to choose the optimal outset for incrementally or decrementally computing timeslices. This approach improves, possibly dramatically, the performance of the timeslice operation in approximately 50% of all cases. In the remaining cases, there is a small, constant overhead.

We have defined the two B^+-tree-like data structures the I-tree and the PLI-tree. The I-tree is a specialization of the B^+-tree designed for append-only relations. The PLI-tree is a specialization of the I-tree.

The I-tree index has a very regular structure and high space utilization. All nodes, root, internal, and leaf, have the same format. No node is allocated before it is used, and each allocated node is eventually fully packed. As in the B^+-tree, nodes are connected via single pointers. The PLI-tree is similar to the I-tree, but the nodes store no pointers. Instead, the pointers are computed, which is possible due to the regular structure of the tree. An optimal fanout results, yielding a very flat index. Main memory versions have been implemented as a proof of functionality.

The I- and PLI-trees eliminate problems found in existing append-only indices, e.g., Append-only trees [GS 93] and Monotonic B^+-trees [EWK 93], where nodes are not identical, double pointers are used, or not all pointer positions in allocated nodes are used.

The sizes of the I- and PLI-trees have been shown to be very small in comparison to the backlogs they index. The I/O-cost of maintaining the trees is small for realistic high order trees. A PLI-tree is smaller than and cheaper to maintain than an equivalent I-tree. However it requires a special search algorithm, and leaf nodes cannot be linked together. Further extra information must be stored in main memory for a PLI-tree to make the I/O-cost of the search algorithm the same as for an equivalent I-tree.

Finally, we have shown that the trees are useful when computing a wide range of other queries, such as, time-point queries, time-varying queries, moving window operator queries, Σ queries, and random sampling.

Several interesting directions for future research exist. First, it would be interesting to implement the trees in an extensible database system. Second, it may be observed that the proposed trees are not helpful for all queries. Extensions or new indices are needed for, e.g., point and range queries. Specifically, the use of an index like the Time-Split B-tree [LS 93] together with our proposed trees is an open problem. Third, we believe that further insight may be gained from more elaborate performance studies with real data.

Acknowledgement

The authors wish to thank Richard Snodgrass for his comments that lead to the design of the Pointer-Less I-Tree. Mr. Torp was supported by The Fulbright Commission, Henry og Mary Skovs Fond, Grosserer L. F. Foghts Fond, Reinholdt W. Jorck og Hustrus Fond, and Aalborg University. Dr. Jensen was supported in part by the Danish Natural Science Research Council, grants no. 11–1089–1 SE and no. 11–0061–1 SE.

References

[AB 80] M. E. Adiba and B. G. Lindsay. *Database Snapshots.* Proceedings of VLDB, pp. 86-91, 1980.

[BCL 89] J. A. Blakeley, N. Coburn, and P.-Å. Larson *Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates.* ACM Transactions on Database Systems, Vol. 14, No. 3, pp. 369-400, 1989.

[BLT 86] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. *Efficiently Updating Materialized Views.* Proceedings of the ACM SIGMOD, pp. 61-71, 1986.

[BM 93] L. Bækgaard and L. Mark. *Incremental Evaluation of Time-Varying Queries.* Technical Report. Department of Mathematics and Computer Science, Aalborg University, 1993.

[CSLLM 90] M. Carey, E. Shekita, G. Lapis, B. Lindsay, and J. McPherson. *An Incremental Join Attachment for Starburst*, Computer Sciences Department, University of Wisconsin at Madison, Technical report no. 937, 1990.

[DWB 86] S. M. Downs, M. G. Walker, and R. L. Blum. *Automated Summarization of On-line Medical Records.* Stanford Memo KSL-86-6, Stanford University, 1986.

[EN 89] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Second Edition. The Benjamin/Cummings Publishing Company, Inc. ISBN 0–8053–1748–1, 1994.

[EWK 93] R. Elmasri, G. T. J. Wuu, and V. Kouramaijian. *The Time Index and the Monotonic B^+-tree.* In [T 93].

[GS 93] H. Gunadhi and A. Segev. *Efficient Indexing Methods for Temporal Relations.* IEEE Transaction On Knowledge and Data Engineering Vol. 5, No. 3, pp. 496-509, June 1993.

[JCEGHJ 94] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, and S. Jajodia (editors). *A Consensus Glossary of Temporal Database Concepts.* SIGMOD Record, Vol. 23, No. 1, pp. 52–64, March 1994.

[J 93] C. S. Jensen (editor) et al. *A Consensus Test Suite of Temporal Database Queries.* Technical Report R 93–2034, Aalborg University, Department of Mathematics and Computer Science, Frederik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark, November 1993.

[JM 92] C. S. Jensen and L. Mark. *Queries on Change in an Extended Relational Model.* IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 2, pp. 192–200, April 1992.

[JMR 91] C. S. Jensen, L. Mark, and N. Roussopoulos. *Incremental Implementation Model of Relational Database with Transaction time.* IEEE Transactions on Knowledge and Data Engineering, Vol. 3, No. 4, December 1991, 461–473.

[JM 93] C S. Jensen and L. Mark. *Differential Query Processing in Transaction-Time Databases.* In [T 93].

[LHMPW 86] B. Lindsay, L. Hass, C. Mohan, H. Pirahesh, and P. Wilms. *A Snapshot Differential Refresh Algorithm.* Proceedings of the ACM SIGMOD, pp. 53-60, 1986.

[LS 90a] D. Lomet and B. Salzberg. *The Performance of a Multiversion Access Method.* Proceedings of the ACM SIGMOD, pp. 353–363, May 1990.

[LS 90b] D. Lomet and B. Salzberg. *The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance.* ACM Transaction on Database Systems. Vol. 15, No. 4, pp. 625–658, December 1990.

[LS 93] D. Lomet and B. Salzberg. *Transaction-Time Databases.* In [T 93].

[McK 86] E. McKenzie *Bibliography: Temporal Database.* SIGMOD, Vol. 15, No. 4, pp. 40–52, 1986.

[MS 91] E. McKenzie and R. T. Snodgrass. *An Evaluation of Relational Algebras Incorporating the Time Dimension in Databases.* ACM Computing Surveys, Vol. 23 No. 4, pp. 501–543, December 1991.

[NA 93] S. B. Navathe and R. Ahmed. *Temporal Extensions to the Relational Model and SQL.* In [T 93].

[ORX 90] F. Olken, D. Rotem, and P. Xu. *Random Sampling from Hash Files.* SIGMOD Record, Vol. 19, No. 2, pp. 375–386, June 1990.

[OR 89] F. Olken and D. Rotem. *Random Sampling from B^+-trees.* Proceedings of VLDB, pp. 269–277, 1989.

[RS 87] L. Rowe and M. Stonebraker. *The Postgres Papers.* UCB/ERL M86/85, University of California, Berkeley, 1987.

[Rou 91] N. Roussopoulos. *An Incremental Access Method of ViewCache: Concept, Algorithms, and Cost Analysis.* ACM Transaction on Database Systems, Vol. 16 No. 3, pp. 535–563, September 1991.

[Sal 88] B. Salzberg. *File Structures: an analytic approach.* Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0–1331–4550–6, 1988.

[Sch 77] B. Schueler. *Update Reconsidered.* In G. M. Nijssen (ed.) *Architecture and Models in Data Base Management Systems.* North Holland Publishing Co, 1977.

[Sno 92] R. T. Snodgrass. *Temporal Databases.* In A. U. Frank, I. Campari, and U. Formentini (editors) *Theories and Methods of Spatio-Temporal Reasoning In Geographic Space.* Springer-Verlag, Lecture Notes in Computer Science 639, pp. 22–64, 1992.

[SA 85] R. T. Snodgrass and I. Ahn. *A Taxonomy of time in Databases.* Proceedings of ACM SIGMOD, pp. 236–246, 1985.

[Soo 91] M. D. Soo. *Bibliography on Temporal Databases.* ACM SIGMOD Record, Vol. 20, No. 1, pp. 14–23, 1991.

[SS 88] R. Stam and R. T. Snodgrass. *A Bibliography on Temporal Databases.* Database Engineering, Vol. 7, No. 4, pp. 231–239, 1988.

[Sto 87] M. Stonebraker. *The Design of The Postgres Storage System.* Proceedings of VLDB, pp. 289–300, 1987.

[T 93] A. U. Tansel et al. (editors) *Temporal Database, Theory Design and Implementation.* The Benjamin/Cummings Publishing Company, Inc. ISBN 0–8053–2413–5, 1993.

A Algorithms For the I-tree and the PLI-tree

This appendix list the insertion algorithms for the I-tree and the PLI-tree. Insertion into these trees is accomplished by two functions, `append` and `adjust_tree`. For the PLI-tree, the search algorithm is also given.

A.1 Insertion Algorithm for I-trees

The algorithm for appending to an I-tree is listed and explained below.

```
append(NewValue)
    if (CurrentNode full)
        create NewLeaf
        insert NewValue into NewLeaf
        CurrentNode.pointer[d-1] = NewLeaf - - connect CurrentNode to NewLeaf
        adjust_tree(NewLeaf, Temp)         - - function call
    else - - CurrentNode not full
        insert NewValue into CurrentNode[y]
        y = y + 1
        Temp = NewValue
```

The algorithm is called with a `NewValue`. If the current node is full, a new leaf is allocated, and the new value inserted here. The right-most pointer of the current node is set to point to the new leaf and the algorithm `adjust_tree` is called to correctly insert the new leaf. Otherwise, if the current node is not full, the new value is inserted, and the value is saved in the variable `Temp`, to be used in the algorithm `adjust_tree` listed next and briefly explained in the following.

```
adjust_tree(NewLeaf, NewValue)
    x = Max           - - how many entries in Array
    node_full = true
    jump = true
    - -  scan Array for non-full node or jump
    while (x > 0 and node_full == true and jump == true)
        if (Array[x] not full)
            node_full = false
        else
            if (x+1 <= Max and level(Array[x]) - level(Array[x+1]) > 1)
                jump = false
            else
                x = x - 1
    - - the tree is full
```

```
Y   if (x == 0 and Array[0] full and level(Array[0]) - level(Array[1]) <= 1)
        create NewRoot
        insert NewValue into NewRoot
        NewRoot.pointer[0] = Root    - - left most pointer to old root
        NewRoot.pointer[1] = NewLeaf - - right most pointer to new leaf
        Array[0] = NewRoot
        Max = 1
        Root = NewRoot
Y   else - - the tree is not full
Z       if (level(Array[x]) - level(Array[x+1]) > 1) - - a jump between levels
            create NewNode
            insert NewValue into NewNode
            NewNode.pointer[0] = Array[x+1]  - - sets left most pointer
            NewNode.pointer[1] = NewLeaf     - - sets right most pointer
            Array[x].pointer[y] = NewNode    - - change right most pointer
            Array[x+1] = NewNode
            Max = x + 2
Z       else - - there is a node which is not full
            insert NewValue into Array[x]
            Array[x].pointer[y+1] = NewLeaf  - - new right most pointer is used
            Max = x + 1
    Array[Max] = NewLeaf
    CurrentNode = NewLeaf
```

The varable `Max` records how many slots are used in the dynamic array, `Array[]`. In the `while` loop, the right-most chain is scanned for a non-full node or a `jump` between levels in the (right-most) chain from the right-most leaf to the root. The three situations shown in Figure 5 on page 8 are handled in the subsequent `if` statements. The `if` marked `Y` handles the case where the I-tree is full, and a new root is allocated. See Figure 5A. The `if` marked `Z` handles the case where the is a `jump` between levels in the right-most chain. See Figure 5C. The `else` at `Z` handles the case where a non-full node is found before a `jump`. See Figure 5B.

A.2 Insertion Algorithm for PLI-trees

The algorithm for appending to a PLI-tree is listed below. We indicate the differences from the previous algorithm.

```
append(NewValue)
    if (CurrentNode full)
        adjust_tree(Temp)           -- function call
        create NewLeaf
        insert NewValue into NewLeaf
        ApArray[NodeNo] = NewLeaf    -- save to NewLeaf in ApArray
        NodeNo = NodeNo + 1
        Array[Max] = NewLeaf
        CurrentNode = NewLeaf
    else -- CurrentNode not full
        insert NewValue into CurrentNode[y]
        y = y + 1
        Temp = NewValue
```

In comparison with the append algorithm for the I-tree, the function `adjust_tree` is called before a new leaf is created. Further, the maintenance of `Array[max]` and `CurrentNode` is move from `adjust_tree` to `append` because `adjust_tree` is no longer called with a `NewLeaf` parameter. The algorithm for adjusting PLI-trees is listed below.

```
adjust_tree(NewValue)
    x = Max
    node_full = true
    jump = true
    while (x > 0 and node_full == true and jump == true)
        if (Array[x] not full)
            node_full = false
        else
            if (x+1 <= Max and level(Array[x]) - level(Array[x+1]) > 1)
                jump = false
            else
                x = x - 1
Y   if (x == 0 and Array[0] full and level(Array[0]) - level(Array[1]) <= 1)
        create NewRoot
        insert NewValue into NewRoot
        ApArray[NodeNo] = NewRoot
        NodeNo = NodeNo + 1
        Array[0] = NewRoot
        Max = 1
        Root = NewRoot
Y   else
Z       if (level(Array[x]) - level(Array[x+1]) > 1)
            create NewNode
            insert NewValue into NewNode
            ApArray[NodeNo] = NewNode
            NodeNo = NodeNo + 1
            Array[x+1] = NewNode
            Max = x + 2
Z       else
            insert NewValue into Array[x]
            Max = x + 1
```

In comparison with the algorithm for the I-tree, the maintainance of the pointers between nodes in the tree is eliminated.

A.3 Search Algorithm for PLI-trees

A high-level algorithm for search in PLI-trees is listed below. Explanations follow the algorithm.

```
search(search_key)
node = Array[0]->node - - points to the root node of the tree
level_no = node->level - - the level of the root
```

```
in_rmt = true - - to detect if search continues in the right-most subtree of the root
array_no = 0
number = $\Sigma_{i=0}^{level\_no-1} d^i$ - - the root's logical number
while (level_no > 0) {
    pointer_no = find_ptr_no(node, search_key) - - find the implicit pointer to follow
    if (pointer_no == 0) {
        $number \;=\; number - ((d-1)(\Sigma_{i=0}^{level\_no-2} d^i) + 1)$
        in_rmt = false
    }
    else {
        if (pointer_no == d and rmt == true){
            jump = level_no - Array[array_no + 1]->level
            $number \;=\; number + (pointer\_no - 2)\Sigma_{i=0}^{level\_no-1} d^i + \Sigma_{i=0}^{level\_no-jump-1} d^i + 1$
            array_no = array_no + 1
            level_no = Array[array_no]->level
        }
        else {
            rmt = false
            $number \;=\; number + (pointer\_no - 1)\Sigma_{i=0}^{level\_no-1} d^i + \Sigma_{i=0}^{level\_no-2} d^i + 1$
        }
    }
    node = get_ph_address(number, ap_array)
    if (in_rmt == false)
        level_no = level_no - 1
}
return(node)
```

The algorithm accepts a transaction-time argument. First, the root node and the level of the root are found in the dynamic array, "Array." The variable "in_rmt" is used to indicate if the search is in the right-most subtree which is not necessarily full. The variable "array_no" records which entry in "Array" that is being used. The logical number of the root is computed as explained in Section 4.2.

In the "while" loop, it is first calculated which implicit pointer to follow to the next level. If the left-most pointer is followed, then we are going to a node that was allocated earlier; we thus subtract the all nodes that were allocated between the current node and the new node. The formula for computing this was given in Section 4.2.

If the logical pointer number is not the left-most we go to a node that was allocated later. Two possibilities exists. We are in the right-most subtree or not. If we are not in the right-most subtree the new node number is again found using a formula developed in Section 4.2.

If we are in the the right-most chain, "Array" must be used to find the height of the new node. The formula for computing the new number may be found in Section 4.2. Further, the entry in "Array" must be taken into account if we are in the right-most chain. This is were "array_no" is used.

From the number calculated, the new node is retrieved from the array on nodes, "ap_array," and the level number is decreased by one if we are not in the right-most chain. The while loop continues until a leaf is reached.