35

Efficient Differential Timeslice Computation

Kristian Torp, Leo Mark, and Christian S. Jensen

Transaction-time databases support access to not only the current database state, but also previous database states. Supporting access to previous database states requires large quantities of data and necessitates efficient temporal query processing techniques.

In previous work, we have presented a log-based storage structure and algorithms for the differential computation of previous database states. Timeslices, i.e., previous database states, are computed by traversing a log of database changes, 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 using the log. The cost of this computation is determined by the size of the log between the outset and the new timeslice.

This paper proposes an efficient algorithm that identifies the cheaper outset for the differential computation. The basic idea is to compute the sizes of the two pieces of the log by maintaining and using a tree structure on the timestamps of the database changes in the log. The lack of a homogeneous node structure, a controllable and high fill-factor for nodes, and of appropriate node allocation in existing tree structures, e.g., B^+-trees, Monotonic B^+-trees, and Append-only trees, render existing tree structures unsuited for our use. Consequently, a specialized tree structure, the Pointer-less Insertion tree, is developed to support the algorithm. As a proof of concept, we have implemented a main memory version of the algorithm and its tree structure.

Keywords: transaction-time, data models, snapshots, timeslice, incremental computation

1 Introduction

A transaction-time database records the history of the database [9, 27] 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 [6] Applications also include the management of medical records [5].

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 [15, 16, 29], and significant effort has recently been devoted to implementation-related topics (e.g., see [18, 21, 28, 32]).

The timeslice operator [1, 25] is one of the central operators in temporal database systems. Indeed, most temporal relational algebras [19] 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 [10] 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* [11]. 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., insertions and deletions) to a relation are appended to its backlog. A relation is derived from a backlog by using the timeslice operator. In addition to the attributes of the associated relation, each tuple in a backlog contains attributes, e.g., a transaction timestamp, that make the implementation of the timeslice operator possible.

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., incremental or decremental computation of queries from the cached results of similar and previously computed queries [2, 3, 4, 13, 14, 22].

When given a time t_x for which a new timeslice $R(t_x)$ of relation R is requested, the times t_{x-1} and t_{x+1} of the nearest earlier and later cached timeslices, $R(t_{x-1})$ and $R(t_{x+1})$, respectively, are identified. Identifying the times t_{x-1} and t_{x+1} together with the page position in the backlog corresponding to these times is done through a very small memory-resident binary tree on the timestamps of the timeslices that have been previously cached. The Pointer-less Insertion tree (PLI-

tree) is then used to compute page positions for the time t_x in the backlog. The three resulting page positions can 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 PLI-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 are packed completely because node splitting never occurs. The PLI-tree contains no stored pointers; they are replaced by computation.

We maintain a PLI-tree 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 PLI-tree to find the page position of a change request corresponding to a given timestamp value, is exploited during timeslice computation. 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 also be predicted precisely and efficiently, which is useful in, e.g., real-time applications.

The paper is organized as follows. Section 2 first describes the data structures in the backlog model. It then defines the timeslice operator, introduces the concept of differential computation, and provides a top-level differential timeslice algorithm. The remainder of the paper fleshes out this algorithm. Section 3 defines the PLI-tree and covers insertion, search, and implementation aspects. Section 4 shows how a PLI-tree is used during timeslice computation to decide whether incremental or decremental computation is most efficient. Section 5 describes related work and includes a comparison of the PLI-tree with the AP-tree [8], a related index structure. Finally, Section 6 summarizes the paper and points to directions for future research.

2 Implementation Model for Transaction-Time Databases

A number of data models support transaction-time; for a survey, see [20, 26]. We use the backlog model [11, 13] as the basis for this work. This model is quite simple in that it stores temporal data in append-only logs. In addition, its query language is simple and has a formal semantics based on the relational algebra [26].

This section introduces the problem of differential timeslice computation and describes the general solution to the problem. To do so, we initially present the backlog transaction-time data model, with an emphasis on its storage structures and the timeslice operator.

2.1 Relations and Backlogs

In the backlog model, all base data is stored in backlogs. Here, we describe their format and how they are updated.

For each relation, R, defined by the user, the database system generates and maintains a backlog, B_R. The backlog B_R for relation R is simply a relation which contains the entire history of change requests to R. Specifically, assume that R has schema $R(A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$. The backlog B_R then has schema $B_R(\text{Id} : \text{Surrogate}, \text{Operation} : \{\text{Ins}, \text{Del}\}, \text{Time} : \text{TTime}, A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$. Thus, a backlog contains three attributes in addition to those defined in its corresponding relation. The attribute "Id" is surrogate-valued and is used as a tuple identifier for backlog tuples, termed change requests. Next, "Operation" is an indicator of whether the tuple is an insertion or a deletion request. Updates are modeled as a deletion/insertion pair with the same transaction timestamp. Finally, "Time," is an instant-valued transaction timestamp that records the time when the transaction that inserted the change request commited. It is assumed that each change request has a unique transaction timestamp (except updates) and that the backlog is stored in transaction-time order.

Table 1 shows how insertion, deletion, and update operations on user-defined relations are translated into insertions into the corresponding backlogs. Additional explanation follows.

operation on R	Effect on B_R
insert R("tuple")	insert B_R(new-id(), Ins, current-timestamp(), "tuple")
delete R(k)	insert B_R(new-id(), Del, current-timestamp(), tuple(k))
update R(k, "new values")	insert B_R(new-id(), Del, current-timestamp(), tuple(k)) insert B_R(new-id(), Ins, current-timestamp(), k, "new values")

Table 1: Operations on a Relation and Their Effect on the Backlog

The insertion of a tuple into R has the effect that an insertion change request is appended to B_R. The functions new-id() and current-timestamp() return a previously unused surrogate value and the time when the insertion transaction commits, respectively. The deletion of a tuple with key value k from R results in a deletion change request being appended to B_R. The function tuple() returns the tuple in R identified by k. We shall later introduce data structures that allow for the efficient computation of this function. An update of a tuple with key value k leads to two change requests being appended to B_R, namely a deletion request for the tuple with the key value k and an insertion request for the tuple with key value k and with the new attribute values.

The storage space requirements are $\mathcal{O}(n)$ where n is the total number of different versions of all tuples. The insertion or deletion of a tuple results in a single

change request being appended to the backlog, and the update of an existing tuple results in two change requests being appended.

2.2 The Timeslice Operator

The five basic relational operators are retained in the algebra for the backlog model. Before any of these operators can be applied to a relation, the relation must first be timesliced. The timeslice at time t_x of relation R is denoted $R(t_x)$ and intuitively computes the state of R current at time t_x. Only time arguments between when the relation was created, $t_{\text{initialization}}$, and the current time, *now*, are permitted.

A formal definition of the timeslice operator is given next [13]. Let relation R have attributes $A_1, A_2, \ldots, A_n$, with *Key*, a time-invariant key value, being one of these. The timeslice $R(t_x)$ is then defined as follows.

$$
\begin{aligned}
&R(t_x) = \\
&\quad \{y^{(n)} \mid \exists s\ (s \in B_R \wedge y[A_1] = s[A_1] \wedge \\
&\qquad\qquad y[A_2] = s[A_2] \wedge \cdots \wedge y[A_n] = s[A_n] \wedge \\
&\qquad\qquad s[\text{Time}] \leq t_x \wedge s[\text{Operation}] = \text{Ins} \wedge \\
&\qquad\qquad \neg\exists u(u \in B_R \wedge s[Key] = u[Key] \wedge \\
&\qquad\qquad\qquad s[\text{Time}] < u[\text{Time}] \leq t_x))\}
\end{aligned}
$$

As can be seen, the timeslice is computed from the backlog. In the first two lines, the attributes of the result are selected. In the third line, all insertions that are done before the timeslice time are identified. Line four and five serve to eliminate all those insertions that have been countered by deletions before the time of the timeslice.

2.3 Incremental and Decremental Computation of Timeslices

Having defined the timeslice operator and the underlying data structure, the next step is to consider the computation of timeslices. As a foundation for achieving efficiency, results of applying the timeslice operator, termed timeslices, are cached and subsequently reused for the computation of other timeslices. These results may be saved in a so-called *view-pointer caches* [22], which are disk-based data structures from which the results may be materialized. A view-pointer cache, c_R, for relation R, has the format described next.

```
record of (
    change-request-pointers: list of ( record of (
        PID : Pointer,
        list of ( TID : Surrogate ) ) ),
    slice-time: TTime,
    offset: Integer )
```

In the data structure, values of attribute *PID* point to pages in the backlog where change requests necessary for materialization of the view are stored. The *TID* values associated with a *PID* value identify the exact change requests within the particular page. The timeslice represented as a view-pointer cache is materialized using the backlog records thus identified. Finally, the attribute *slice-time* records the time when the timeslice was current, and *offset* indicates the number of disk pages occupied by change requests with a transaction time not exceeding *slice-time*. Their use will be explained in Section 2.4.

It is obvious that if a view-pointer cache is stored every single time a new timeslice is computed, then eventually the disk-space requirements will be prohibitive. To solve this problem, we assume that a fixed amount of disk-space is allocated for storing view-pointer caches. The finite set of all view-pointer caches for a relation R is denoted C_R. The choice of an appropriate cache replacement strategy is an orthogonal issue that is not addressed here.

Differential timeslice computations use view-pointer caches as outsets. Thus, the timeslice at a time t_x can be computed using any cached timeslice, earlier or later than t_x, as the outset. Initially, the outset is materialized. Starting from the outset, the change requests in the backlog are then reflected in turn in the outset until the desired timeslice is obtained, see Figure 1. With an earlier timeslice as the outset, we are in the incremental ("do" or "update") case, and with a later timeslice as the outset, we are in the decremental ("undo" or "downdate") case. Algorithms for incremental and decremental computation of timeslices have been described previously [12, 13].

2.4 The Problem of Starting From the Best Outset

We have now seen how a new timeslice, e.g., $R(t_x)$, may be computed with any cached timeslice as the outset. The problem addressed in this paper is how to efficiently select the best outset available in the cache. Making the reasonable assumption that a view-pointer cache for $R(now)$ is always present, there are always precisely two candidate outsets, namely the currently closest earlier and closest later cached timeslices. Note that the timeslice $R(t_{\text{initialization}})$, which is empty, is trivially in the cache and will always qualify as an earlier timeslice. The view-pointer cache for $R(now)$ makes it possible to compute the function tuple(), introduced in Table 1, without scanning the backlog.

Locating these two outsets in the cache is quite easy. We simply assume that a (small) binary tree, referencing the cached timeslices based on their *slice-time* values, is maintained in main memory. The candidate outsets are located by doing a nearest-neighbor search in the tree with the time of the timeslice to be computed as the search argument.

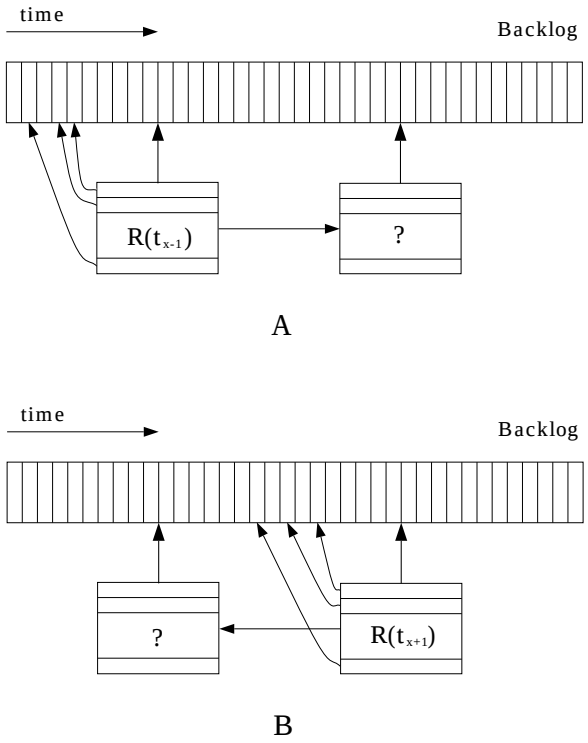

Figure 1: A) Incremental Computation B) Decremental Computation

The problem that remains is to decide which of the two outsets at hand is the better one to use. We will base our decision on the numbers of pages of change requests in the backlog that need to be processed when using one of the outsets. This yields the following conceptual, top-level algorithm for differential computation of the timeslice $R(t_x)$, which in addition to the timeslice time t_x takes as arguments a backlog B_R, and a corresponding cache C_R. Additional explanations follow the algorithm.

differentialTimeslice(t_x, B_R, C_R)

$t_{x-1} \leftarrow \max(\{c_R.\textit{slice-time} \mid c_R \in C_R \wedge c_R.\textit{slice-time} \leq t_x\})$ (1)

$t_{x+1} \leftarrow \min(\{c_R.\textit{slice-time} \mid c_R \in C_R \wedge c_R.\textit{slice-time} \geq t_x\})$ (2)

$P_{t_{x-1}} \leftarrow \text{pagePosition}(t_{x-1}, C_R)$ (3)

$P_{t_x} \leftarrow \text{pagePosition}(t_x, B_R)$ (4)

$P_{t_{x+1}} \leftarrow \text{pagePosition}(t_{x+1}, C_R)$ (5)

if $\mid P_{t_x} - P_{t_{x-1}} \mid \leq \mid P_{t_{x+1}} - P_{t_x} \mid$ **then** (6)

    incrementalTimeslice(t_{x-1}, t_x, B_R, C_R) (7)

else

    decrementalTimeslice(t_{x+1}, t_x, B_R, C_R) (8)

Steps 1–2 locate the times of the closest earlier and later cached timeslices, as already outlined above. Steps 3–5 compute page positions of the three timeslice times. The page positions corresponding to times t_{x-1} and t_{x+1} are recorded in the view-pointer caches as *offset* values and are easily obtained during Steps 1 and 2, respectively. Step 4 will be addressed later. With page positions for all three times, the number of backlog pages between the requested timeslice and the earlier and later cached timeslices, respectively, are compared in Step 6. The intermediate backlog pages are the ones that must be read to compute the timeslice. On the basis of the comparison, the timeslice is computed incrementally or decrementally, as outlined in Section 2.3.

In order to efficiently compute a timeslice, only one problem remains, namely that of computing the page position in the backlog of an arbitrary timeslice time, i.e., Step 4.

It is a fundamental requirement to a solution is that it be efficient. This rules out a solution where the backlog between $t_{\text{initialization}}$ and t_x (or the part between t_{x-1} and t_x and the part between t_x and t_{x+1}) is scanned. With that solution, always simply computing either incrementalTimeslice or decrementalTimeslice is more efficient than computing differentialTimeslice. Rather, the solution should require only a few disk accesses. Also observe that using the temporal proximity among the three times t_{x-1}, t_x, and t_{x+1} as the basis for computing the two page counts in Step 6 is not a good solution. This is so because it cannot be assumed that change requests are inserted into the backlog at a constant frequency. In many applications, e.g., financial applications such as stock trading, insertions occur at highly irregular rates.

In the remainder of the paper, we present a precise and efficient solution to the problem. With this solution, we have effectively added high-performance transaction-time support to incremental database systems such as ADMS [22].

3 PLI-trees

In this section, we describe the PLI-tree. For expository reasons, we introduce PLI-trees in two steps. First, we present a structure similar to the B^+-tree, with pointers between nodes, the I-tree. Second, we present a structure similar to the I-tree, with no explicit pointers between nodes, the PLI-tree.

3.1 I-trees, a Precursor to PLI-trees

The structure of an I-tree is described first. Then, updates are considered.

The I-tree Structure

The I-tree is essentially a degenerate B^+-tree designed to index append-only data on a sequential key, e.g., change requests in a backlog on their timestamps. An example of an I-tree is shown in Figure 2.

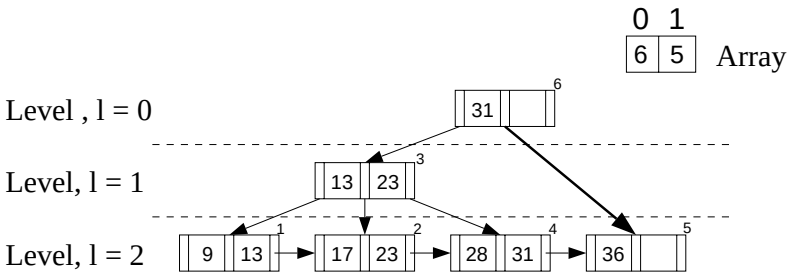

Figure 2: An I-tree of Height $h = 2$ and Order $d = 3$

The tree shown is of height $h = 2$ and order $d = 3$. As can be seen the structure of the nodes is identical to the structure of nodes in a B^+-tree. Both internal nodes and leaf nodes have the same structure, and leaf nodes are connected in search-key order.

The chain of pointers and nodes to the right is called the *right-most chain*. In Figure 2, the right-most chain consists of the root, the boldface pointer, and the right-most leaf, termed the *current node.*

Insertions are only made into the current node, and node splitting does not occur. The I-tree grows from the bottom and up in the right most subtree of the root, and internal nodes are not allocated before they are needed. These characteristics have three implications: First, all nodes, except nodes in the right-most chain, are filled. Second, all subtrees of the root, except the right-most subtree, are filled and balanced. Finally, the right-most subtree needs not be balanced.

The array, in Figure 2, is a dynamic array containing pointers to all nodes in the right-most chain [7]. These pointers are used when insertions are made to the tree. In Figure 2 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 indicate the allocation order of tree nodes and are used for illustrating the dynamics of the index later in the paper; they 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 2 also shows that the right-most subtree of the root needs not be balanced—there is no node at Level 1.

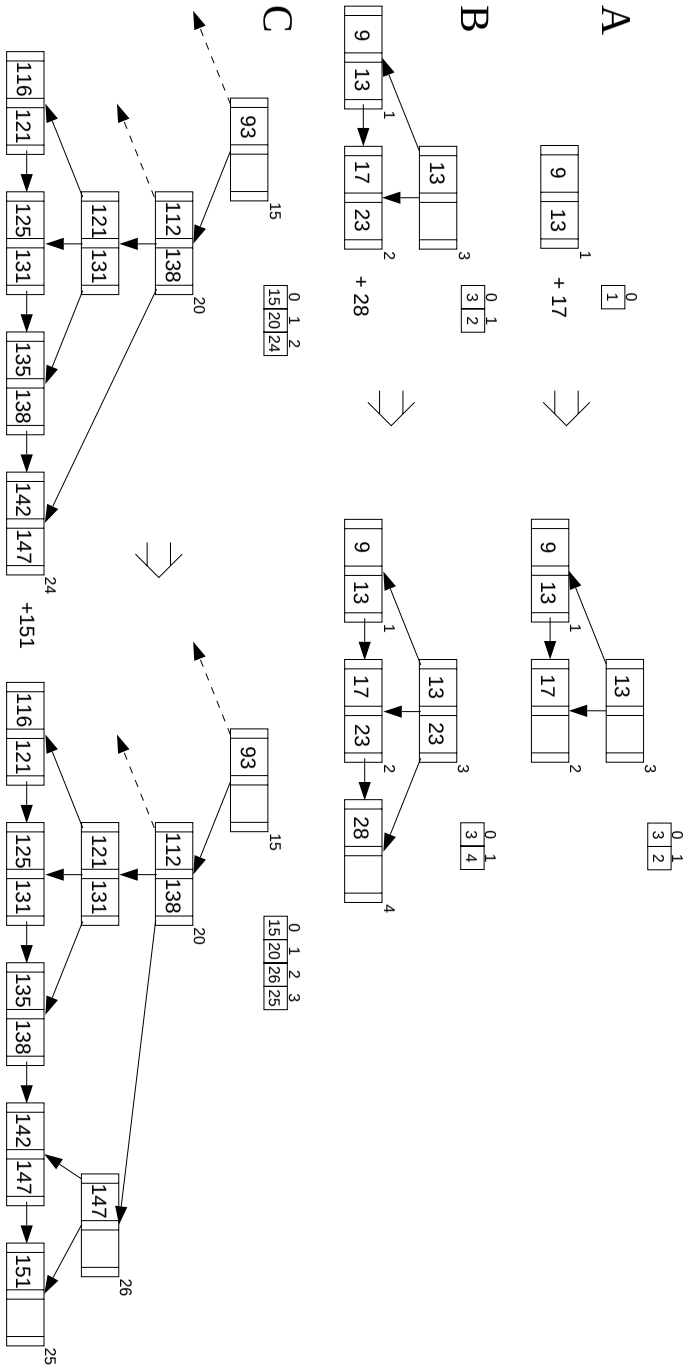

Figure 3: Examples of Insertions into the I-tree

Insertion into the I-tree

Next, 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 implementation of the I-tree are listed in [31].

Figure 3A 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.

Figure 3B 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.

Figure 3C 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 is 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.

3.2 PLI-trees

Next, we shall see that it is possible to eliminate all stored pointers from the I-tree. This increases the number of key values that fit in a single node and reduces the index size. Specifically, 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 the Pointer-less I-tree (PLI-tree).

First, we briefly cover the structure of the PLI-tree. Then, search using implicit, computed pointers is described, and finally, implementation aspects are considered. As the PLI-tree insertion algorithm is almost identical to that of the I-tree, it is not discussed here. Further, the insertion algorithm does not use the search algorithm to locate the node where a new key value is to be inserted, as does the B^+-tree. For reference, the complete insertion algorithms for the I-tree and PLI-tree are given elsewhere [31].

PLI-tree Structure

An example of a PLI-tree of height $h = 2$ and order $d = 3$ is shown in Figure 4. Compare this figure to Figure 2.

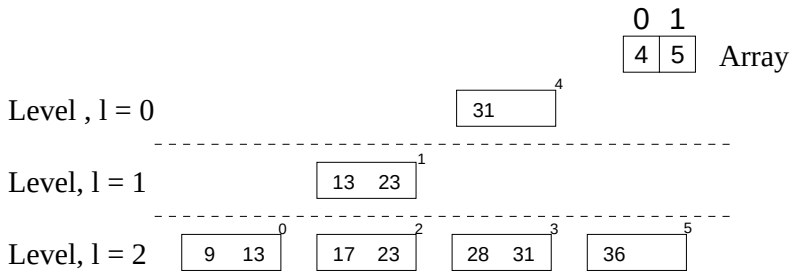

Figure 4: A PLI-tree of Height $h = 2$ and Order $d = 3$

Compared to the I-tree, all explicit tree pointers are eliminated. The dynamic array used in I-trees is retained and contains pointers to all nodes in the right-most chain. As before, the numbers shown above each node are not part of the tree; they indicate the allocation order of tree nodes and are used in the subsequent discussions.

Search in the PLI-trees

The search algorithm for the PLI-tree is *logically* the same as for the B^+ and I-trees. The only difference is that all pointers are implicit and must be computed. When we show how to compute the pointers in the following, we use "pointer" to mean an "implicit pointer" between the nodes.

Figure 5A shows a PLI-tree. The node numbers above each node indicate the allocation order, and it can be seen that the nodes are allocated in in-order. Figure 5B thus shows how the nodes are stored sequentially 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 parameters listed in Table 2 are needed; see also Figure 5 for further explanation. The search algorithm is called with a key value when the PLI-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.

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

The PLI-tree is full except for the right-most subtree. When the nodes are allocated

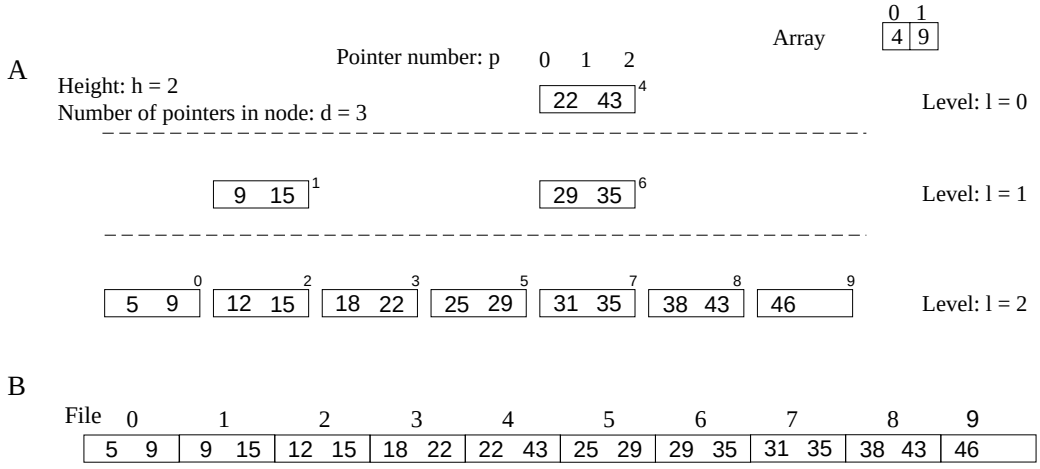

Figure 5: An Example of a PLI-tree and How Nodes are Stored in a File

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

Table 2: Parameters Used to Calculate Node Numbers

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.

If the left-most pointer of a node is to be 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.

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

In this formula, $d-1$ 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 $+1$ accounts for the old node.

If the pointer followed is not the left-most pointer, we are going 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.

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

Here, 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 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.

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

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 levels 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.

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.

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 disk. This makes it possible to access a node in a file on disk by a start address and an offset. This assumption may be too restrictive in a multi-user environment where nodes are allocated dynamically. Here, disk space may be allocated in chunks, termed *extents* [23]. An extent is a number of consecutive disk pages. All extents contain 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 6B shows how nodes of a PLI-tree are stored in extents. In the figure, an extent consists of three disk pages. Compare this to Figure 5. The extent array in Figure 6A contains the start address of extents. Slot number zero points to extent number zero, etc.

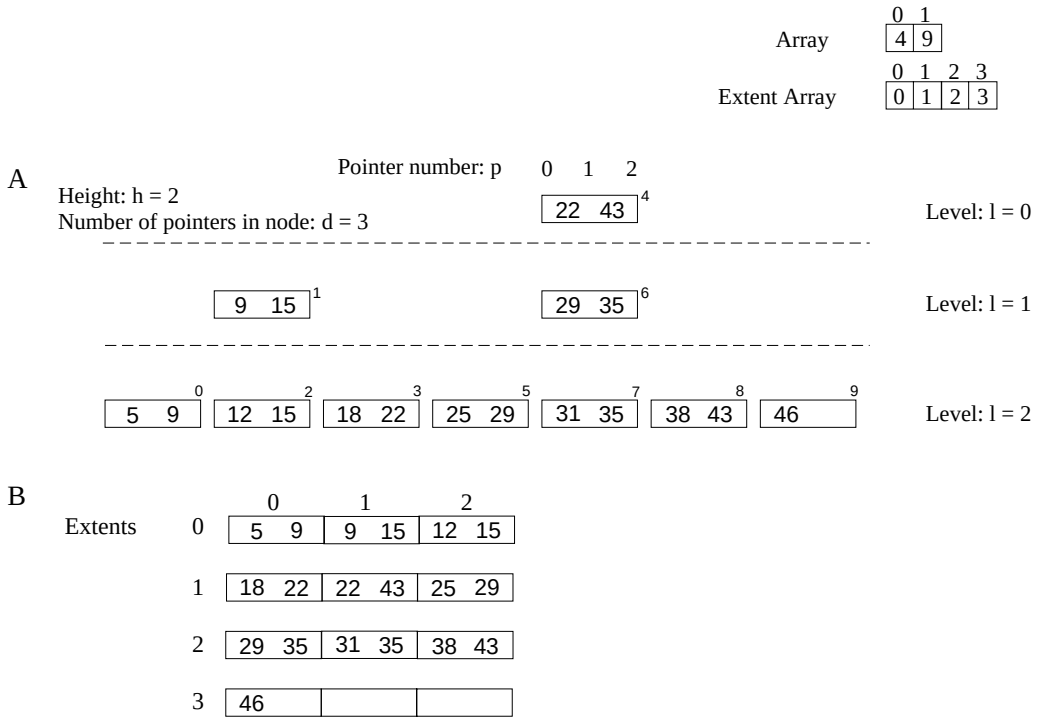

Figure 6: An Example of a PLI-tree and How Nodes are Stored in Extents

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 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.

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

The offset of a node within an extent is given as follows.

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

4 Using PLI-trees for Differential Timeslice Computation

In this section, we first describe how to find a page position in the backlog using the PLI-tree. Second, we compare the performance of our algorithm to the only

alternative, that of a linear scan. Finally, we estimate the size of the PLI-tree and the I/O-cost of maintaining it.

4.1 Finding Page Positions Using the PLI-tree

The idea is to maintain a PLI-tree on the transaction-time attribute of the change requests of a backlog. Figure 7 shows a PLI-tree on a backlog. When computing a timeslice at time t_x, the tree is used to find the *page position* of the corresponding change request(s) in the backlog.

The following formula is used.

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

This formula uses four parameters, namely h, d, p, and l, which are explained in Table 2 and Figure 7. The p_l values and the p_h value are obtained by searching the PLI-tree with t_x as the search value. The value p_i denotes the pointer number to be followed to the next level during the search in the PLI-tree node at level i. Value p_h thus denotes the pointer number at the leaf level that points to the appropriate backlog page.

The formula may then be explained as follows. The PLI-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, $(d-1)\,d^{h-(l+1)}$ pointers to disk pages of the backlog are passed at the leaf level. The formula sums up the number of disk pages passed at each level from the root to the level just above the 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 I/O-cost of computing a page position is h disk accesses, the height of the tree. In summary, we are now able to efficiently choose the best outset for either incremental or decremental computation of a timeslice. We have thus accounted for Step 4 in the algorithm listed in Section 2.4. As this was the only remaining step to account for, the full algorithm has now been covered.

4.2 Comparison with Linear Scan

With no PLI-tree available, the *only* reasonable, existing way to find the cost of computing a timeslice is to actually compute it. Therefore, to investigate if timeslice computation using a PLI-tree is cost effective, we compare it to the timeslice computed using a linear scan of the backlog from $P_{t_{x-1}}$ 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 page positions $P_{t_{x-1}}$ and $P_{t_{x+1}}$ is zero because they can be found

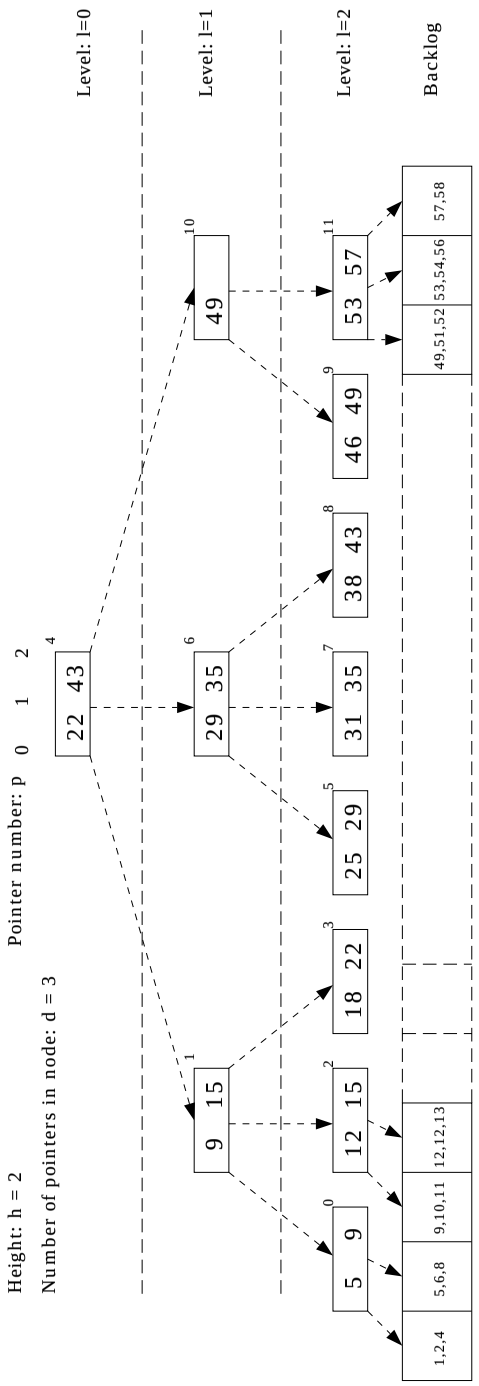

Figure 7: A PLI-tree Indexing Transaction Timestamps on a Backlog (the Pointers Shown are Implicit)

by lookups in the main-memory binary tree that indexes the computed and cached timeslices. The cost of finding the position P_{t_x} in the PLI-tree is h disk accesses.

Figure 8 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 page positions of t_x and the closest cached timeslices; the y-axis indicates the number of disk accesses needed to compute the timeslice. The dashed line assumes that the timeslice $R(t_x)$ is computed incrementally from $P_{t_{x-1}}$, while the solid curve assumes that the PLI-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.

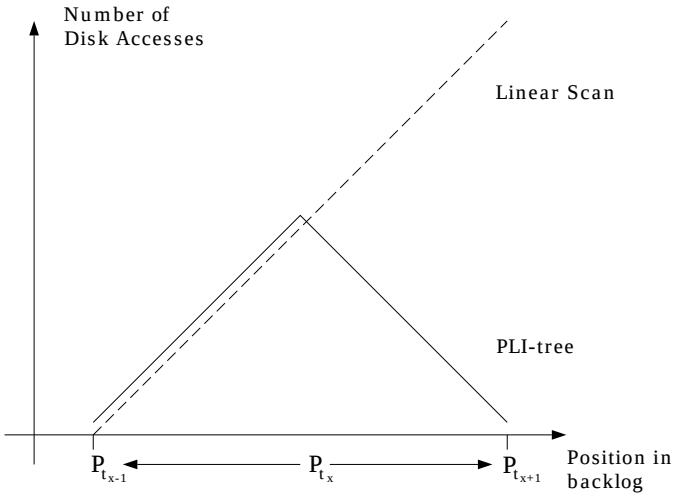

Figure 8: Cost Comparison of Timeslice Computation Using Linear Scans Versus PLI-trees

The figure indicates that the PLI-tree is almost as fast as linear scan in approximately 50% of the cases. The difference is h disk accesses (in practice, h is 2 or 3). At the same time, it shows that in the other approximately 50% of the cases, there can be very substantial efficiency gains when using the PLI-tree. For realistic situations, this means that using the PLI-tree to choose the outset for the differential computation, instead of using linear scan, never performs worse and in approximately 50% of the cases performs significantly better.

4.3 Maintenance of the PLI-tree

In this section, we estimate the size of the PLI-tree and the I/O-cost for maintaining the tree.

PLI-tree Size versus Backlog Size

The PLI-tree does not need to record the transaction timestamps of all the change requests in the backlog. Only approximately one transaction timestamp for each disk page is needed—the PLI-tree is a sparse index. Two PLI-trees of the same height are shown in Figure 9. For this height, Figure 9A shows a worst-case situation where the PLI-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 9B shows a best-case situation where the size of the PLI-tree is the smallest possible compared to the size of the backlog. This PLI-tree is full and balanced.

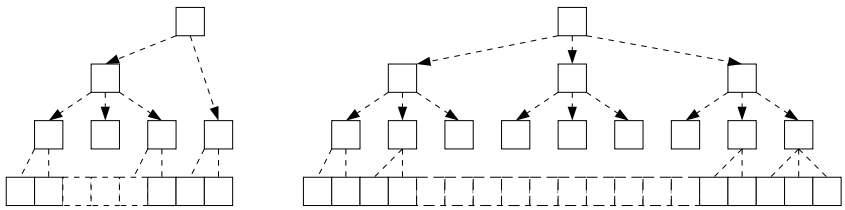

Figure 9: A) Worst-case Situation B) Best-case Situation

The following expression is valid for height $h \geq 1$ and order $d \geq 2$. In the worst-case situation, the size of the PLI-tree compared with the size of the backlog is given as follows.

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

Examples of the worst-case and the best-case are shown in Table 3 for height, $h = 3$ and order, $d = 100$ [23]. As can be seen, the backlog is approximately d times larger than the PLI-tree. The size of the index is very small. The difference for the worst-case and best-case is insignificant for realistic trees.

	PLI-Tree	Backlog	PLI-Tree/Backlog (%)
Worst-case	10,103	990,002	1.0205
Best-case	1,010,101	99,000,001	1.0203

Table 3: Worst-Case and Best-Case for the Size of the Backlog vs. the Size of the PLI-Tree

I/O-Cost of Maintaining the PLI-tree

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

There is an insertion into the PLI-tree each time a new disk page is allocated for the backlog. Two cases should be distinguished: when the right-most leaf of the PLI-tree is not full, and when that leaf is full. The latter situation is rare. It occurs only once for each $(d - 1)\ N_{ch}$ appends to the backlog, where d is the order of the PLI-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 PLI-tree is zero. In the second case, the three possibilities shown in Figure 3 exist. Their I/O-costs are shown in Table 4.

Description	Fig. 3A	Fig. 3B	Fig. 3C
Allocate new leaf node	√	√	√
Allocate new root/node	√		√
Read internal node		√	√
Write internal node		√	√
Write old root	√		
Write new node			√
Write current node	√	√	√
Total I/O-cost	4	4	6

Table 4: Number of Disk Accesses for Different Update Cases

In the worst case, it will require 6 disk access to update the PLI-tree when the backlog is updated. (Note that the I/O-cost is independent of the height of the PLI-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 PLI-tree index is given by $((d - 1)\ N_{ch}/6)$.

Table 5 shows examples of how many change requests can be appended to the backlog for each PLI-tree I/O operation, for different realistic page sizes. It is assumed that transaction timestamps occupy 64 bits [26], pointers (Unix) 32 bits, and change requests 128 bytes.

The number of appends per extra disk access grows with the square of the page size because both d and N_{ch} depend on the page size. In conclusion the PLI-tree is cheap to maintain for realistic page sizes.

	Page size (bytes)				
	512	1,024	2,048	4,096	8,192
Order d	64	128	256	512	1,024
N_{ch}	4	8	16	32	64
Appends per I/O	42	169	680	2,725	10,912

Table 5: Fanout of PLI-tree for Different Page Sizes

		Page size (bytes)				
		512	1,024	2,048	4,096	8,192
Fanout	PLI-tree	64	128	256	512	1,024
	AP-tree	41	84	169	340	681
Height	PLI-tree	3	2	2	2	1
	AP-tree	3	3	2	2	2
No. of pages	PLI-tree	15,876	7,876	3,924	1,959	978
	AP-tree	25,002	12,050	5,954	2,952	1,471

Table 6: A Comparison of AP-trees and PLI-trees

5 Related Work and Comparison with the AP-tree

To the best of our knowledge, no other efficient algorithms exist that address the problem of selecting the best outset for the differential computation of timeslices. The algorithm presented in this paper makes essential use of a new tree structure, the PLI-tree. In this section, we review the existing tree structures that are most similar to the PLI-tree, and we explain why we have designed a new tree structure. We then compare the PLI-tree in more detail to the structure that resembles it the most, namely the AP-tree.

5.1 Related Tree Structures

The tree structures most closely related to the PLI-tree are the B^+-tree, the Monotonic B^+-tree (MB-tree) [7] and the Append-only tree (AP-tree) [8].

For an overview of other, less related, structures, see [24, 30]. These structures are all intended for a more general use than the PLI-tree which is designed for the single specific purpose of indexing the timestamps of the entries in a log. For this purpose, the PLI-tree is superior.

If a regular B^+-tree was used in place of the PLI-tree, the nodes would only be approximately 50% full [8].

There are primarily three reasons why we have chosen to not use the MB-

tree. 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* [7]. This extra generality, not needed for our use, unnecessarily complicates the insertion algorithm.

For the AP-tree, there are also three reasons why it is not well-suited for our use. 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.

Being designed for a single purpose, the PLI-tree eliminates these problems. In addition, the PLI-tree uses easily computed "pointers" while the three related indices use stored pointers.

5.2 Comparison with the AP-tree

Being a more general index than the PLI-tree, the AP-tree allows both insertions and deletions. The AP-tree favors insertions at the expense of more complicated deletions. The PLI-tree takes the full step and completely sacrifices deletion. This has yielded a more compact and regularly structured tree where it is feasible to compute "pointers".

Table 6 shows the fanout, the height, and the number of pages used to store an AP-tree and a PLI-tree for different page sizes.

The fanout of the PLI-tree is larger compared to the fanout of the equivalent AP-tree because the pointers in the nodes are eliminated. If it is assumed that timestamps occupy 64 bits [26] and pointers (Unix) 32 bits then the fanout of a PLI-tree is approximately 50% larger than that of an equivalent AP-tree. The height of PLI-trees and AP-trees is calculated for an index on a backlog consisting of 1 million pages. As can be seen, the height of the PLI-tree is smaller than or the same as the height of the equivalent AP-tree, depending on the page size. The number of disk pages used for storing the PLI-tree and the AP-tree when indexing 1 million pages, are also shown in Table 6 for different page sizes. As can be seen, the PLI-tree is approximately 33% smaller than the AP-tree.

Finally, the maximum number of disk accesses for making an insertion into an AP-tree is ten [8]. We have shown in Section 4.3, that the maximum number of disk access to make an insertion into a PLI-tree is six.

6 Summary and Future Research

In this paper, we have taken a step in the direction of realizing efficient timeslice queries in transaction-time databases. We have presented an efficient algorithm which can precisely predict whether it is going to be more efficient to incrementally or decrementally compute a *timeslice* from a previously computed and cached timeslice.

The algorithm uses a *Pointer-Less Insertion tree* (PLI-tree) as an index on the transaction timestamps of the entries of a *backlog*, a log-like storage structure for transaction-time data. The algorithm improves, possibly quite substantially, the performance of the timeslice operation in approximately 50% of all cases. In the remaining cases, there is a very small, constant overhead.

The PLI-tree is similar to the B^+-tree, but has been designed for the specific purpose of being an ideal part of the algorithm. The tree has a regular node structure; all nodes, root, internal, and leaf, have the same format. Next, all nodes in non-rightmost subtrees are completely filled, and no node is allocated before it is used. With these properties, it has been relatively straightforward to also eliminate all pointers from the tree and replace them by computation. This gives a maximum fanout. Put together, the result is a very compact and flat index that does not waste any space.

A main memory version of the PLI-tree has been implemented as a proof of concept.

Several interesting directions for future research exist. First, it would be interesting to implement the tree in an extensible database system. Second, it may be observed that the proposed tree is not helpful for all queries. Extensions or new indices are needed for, e.g., point and range queries. Specifically, the combined use of other indices, e.g., the Time-Split B-tree [17], together with the PLI-tree is an open problem. Third, we believe that further insight may be gained from more elaborate performance studies with real data.

Acknowledgements

The authors wish to thank Richard Snodgrass for comments that lead us to design the Pointer-Less I-Tree. K. 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. C. S. Jensen was supported in part by the Danish Natural Science Research Council, grants no. 11–1089–1 SE, 11–0061–1 SE, and 9400911.

References

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

[2] 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.

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

[4] M. Carey, E. Shekita, G. Lapis, B. Lindsay, and J. McPherson. *An Incremental Join Attachment for Starburst.* Proceedings of VLDB, pp. 662-673, 1990.

[5] S. M. Downs, M. G. Walker, and R. L. Blum. *Automated Summarization of On-line Medical Records.* Medinfo'86, N-H, pp. 800–804, 1986.

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

[7] R. Elmasri, G. T. J. Wuu, and V. Kouramaijian. *The Time Index and the Monotonic B^+-tree.* Ch. 18, pp. 433–456, in [30].

[8] 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.

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

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

[11] 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.

[12] 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, pp. 461–473 December 1991.

[13] C. S. Jensen and L. Mark. *Differential Query Processing in Transaction-Time Databases.* Ch. 19, pp. 457–492, in [30].

[14] 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.

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

[16] 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.

[17] D. Lomet and B. Salzberg. *Transaction-Time Databases.* Ch. 16, pp. 388–417, in [30].

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

[19] 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.

[20] G. Özsoyoğlu and R. T. Snodgrass. *Temporal and Real-Time Databases: A Survey.* IEEE Transaction on Knowledge and Data Engineering, Vol. 7, No. 4, pp. 513-532, August 1995.

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

[22] 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.

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

[24] B. Salzberg and V. J. Tsotras. *A Comparison of Access Methods for Time Evolving Data.* Technical Report NU-CSS-94-21, Northeastern University, 1994.

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

[26] 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.

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

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

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

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

[31] K. Torp, L. Mark, and C. S. Jensen *Efficient Differential Timeslice Computation* Technical report no R-94-2055 Aalborg University, Department of Computer Science, Frederik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark, 1994.

[32] V. J. Tsotras, A. Kumar. *Temporal Database Bibliography Update.* ACM SIGMOD Record, Vol. 25, No. 1, pp. 41–51, 1996.