Efficient Differential Timeslice Computation

Kristian Torp Leo Mark
College of Computing
Georgia Institute of Technology
Georgia, GA 30332, USA
{kristian,leomark}@cc.gatech.edu

Christian S. Jensen
Department of Mathematics
and Computer Science
Aalborg University
Fredrik Bajers Vej 7E
DK-9220 Aalborg Ø, DENMARK
csj@iesd.auc.dk

February 19, 1994

Abstract

Transaction-time databases record all previous database states and are ever-groving, leading to potentially huge quantities of data. For that reason, efficient query processing is of particular importance.

Due to the large size of transaction-time relations, it is advantageous to utilize cheap write-once storage media for storage. 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. We provide efficient means of always picking the optimal one. Specifically, we define and investigate the use of a new data structure, the B$^+$-tree-like Insertion Tree (I-tree), for this purpose. The cost of using an I-tree for picking the optimal outset is similar to that of using a B$^+$-tree. Being sparse, I-trees require little space overhead, and they are cheap to maintain as the underlying relations are updated.

I-trees also provide a basis for precisely and efficiently estimating the costs of performing timeslices in advance. This is useful for query optimization and can be essential in real-time applications. Finally, it is demonstrated how I-trees can be used in the computation of other types of queries.

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

1 Introduction

A transaction-time database records the time when each stored tuple is current in the database, and it thus records the history of the database [SA 85]. 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, etc. [DWB 86].

Recent and continuing advances in hardware have made the storage of ever-growing and potentially huge transaction-time databases a practical possibility. As a result, significant effort has recently been devoted to this topic see, e.g. [RS 87, McK 86, SS 88, Soo 91].

Research focus is moving from conceptual data modeling aspects to implementation-related aspects . In order to make transaction-time systems practical, the hardware advances must be combined with advances in query processing techniques.

An essential operator in transaction-time databases is the *timeslice operator* [Sch 77]. The timeslice *R(t)*, of a relation R at time $t \leq now$, is the snapshot state of the relation R as of time t [AB 80].

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 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.

In this paper we use an I-tree as a permanent index on the transaction-time attribute in a backlog. The I-tree is updated every time a new page of change requests is allocated for the backlog. Given a transaction timestamp, the I-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-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 transaction 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-tree is then used to compute page positions for these times 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 estimated precisely, which is useful in e.g., real-time applications. Also, the I-tree can be used to find the exact number of change requests appended to a backlog in a given time interval. Such statistics 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 shows how an I-tree index on a backlog can be used to decide whether incremental or decremental computation is most efficient. Finally, Section 5 shows other situations where the I-tree can be used. Section 6 briefly summarizes the paper and points to directions for future research.

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 semantics based on the relational algebra [Sno 92].

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:	$A_1 : D_1$	...	$A_n : D_n$

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

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 backlogs 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 timestamp); and *Time*, a 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.

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

When an insertion to R is requested, an insertion request is appended to B_R. When a deletion from R of a tuple with key value k is requested, an insertion 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.

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 refering 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.

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.

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

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, ..., A_n where *Key*, a time-invariant key, is one of these. The timeslice $R(t_x)$ is given by

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

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, Jen 91] . 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.

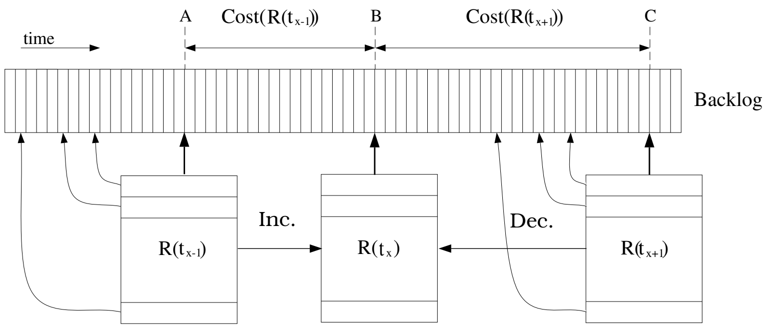

Figure 3: Differential Computation of the Timeslice Operator

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 "down-date" $R(t_{x+1})$ to $R(t_x)$ the change requests between C and B must be traversed. Algorithms for incremental and decremental computation of timeslices may be found in [TMJ 94].

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 indexes may be used to index monotonically increasing key values; for an overview, please see [T 93]. Here, we simply 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 inhomogeneity is not necessary for our purposes. Second, internal nodes in the right-most subtree are allocated before they are needed, yielding a space overhead that we can avoid. Third, in the insertion algorithm extra parameters are given to be able to implement a *Time Index* [EWK 93]. This extra generality makes the insertion algorithm more complicated.

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.

In the I-tree, designed for our specific problem, it has been possible to eliminate the above characteristics, 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 key values.

2. A root node has $i + 1$ (non-null) pointers and i (non-null) key values if it is not at leaf, $1 \leq i \leq (d$ - $2)$. 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 of the is full and balanced. The last pointer on each leaf page, 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 1 and at most h - 1.
   - When the n'th subtree is full, a new subtree of height 1 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 1.

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.

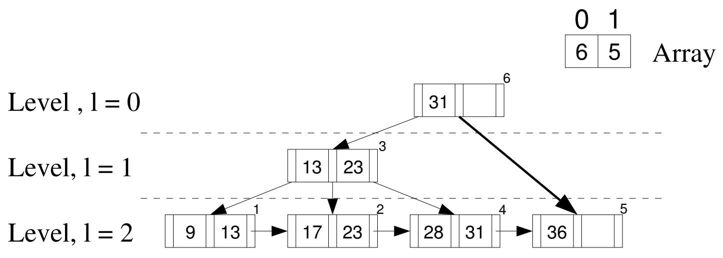

Figure 4: An Example of a I-tree of Height h = 2 and Order d = 3

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 algorithm that formed the basis for implementing the I-tree is given elsewhere [TMJ 94a].

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.

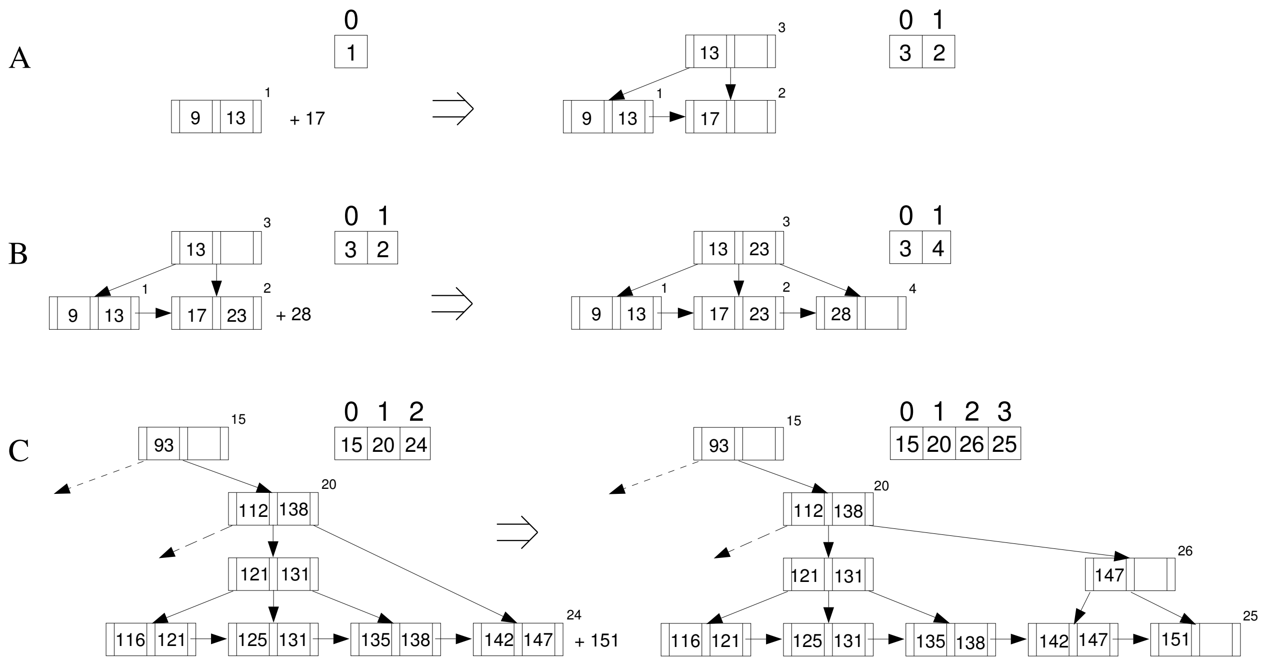

Figure 5: Examples of Appends to the I-tree

4 Use of I-trees for Efficient Differential Timeslice Computation

In this section, we first describe how to find a page position in the backlog using the I-tree. Next, we estimate the size and the I/O-cost of maintaining the I-tree. Finally, we present an algorithm for choosing between incremental and decremental computation of timeslices.

The basic idea is to build 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 6 shows an I-tree on a backlog.

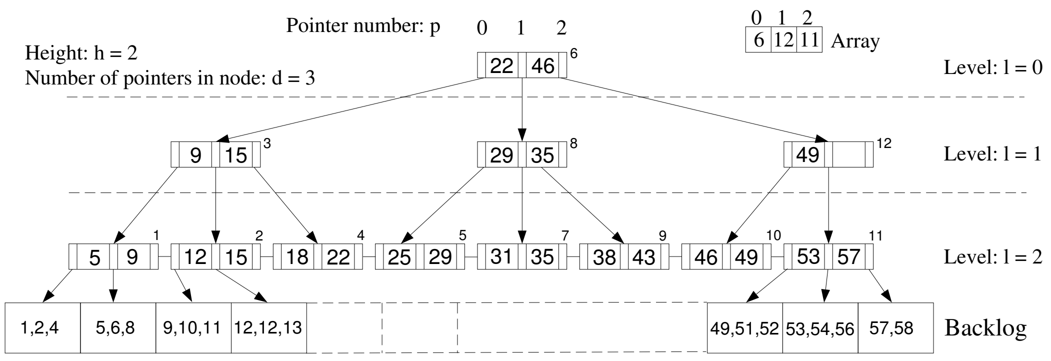

Figure 6: An I-tree and a Backlog

4.1 Finding Page Positions Using the I-tree

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.

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 compute this page position, the following parameters are needed, see also Figure 6.

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

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 Formula 1.

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

Formula 1 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. Formula 1 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 6. 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.

4.2 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.

4.2.1 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 7. For this height, Figure 7A shows a worst-case situation where the I-tree's size is the largest possible compared to the size of the backlog. The rightmost leaf has just been allocated. For the same height, Figure 7B 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.

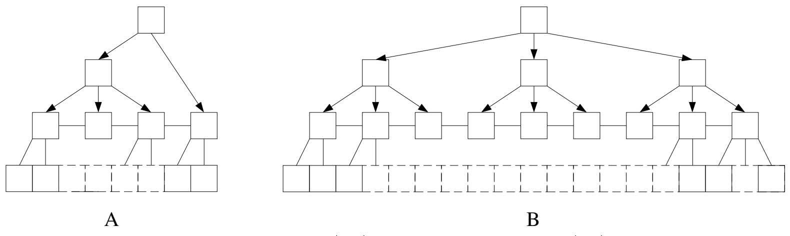

Figure 7: Worst-case (A) and Best-case (B) Situations

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 by Formulas 2 and 3.

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

$$\textit{Best case:} \quad \sum_{m=0}^{h} d^m \Big/ \left((d-1) \cdot d^h + 1\right) \tag{3}$$

Examples of the worst-case and the best-case are shown in the table below for height, $h = 3$ and order, $d = 100$. 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

4.2.2 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 request 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 request can be appended to the backlog for each I-tree I/O operation, for different realistic page sizes. It is 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	42	64
appends per I/O	27	112	451	2,380	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 $6 \cdot \mid B_R \mid/(d - 1) \cdot N_{ch}$

4.3 Choosing between Incremental and Decremental Timeslice Computation

We now present the algorithm for choosing between incremental and decremental computation of timeslices in transaction time databases.

4.3.1 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($t \leq t_x \wedge \exists$ timeslice at time $t$)
    Find R($t_{x+1}$) where $t_{x+1}$ = min($t \geq t_x \wedge \exists$ 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.

4.3.2 Comparison with Linear Scan

If no I-tree is available on the backlog, the only way to find the cost of computing the timeslice is to actually compute the timeslice. Therefore, to investigate if the I-tree computation method is cost efficient, we compare it to the cost of 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 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 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 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 8 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

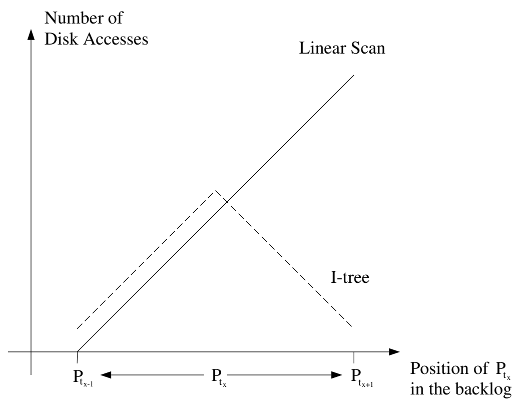

Figure 8: Cost Comparison When Computing Timeslices When Using Linear Scans versus I-trees

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 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. 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 Formula 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 point 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 *time-varying queries*, *the moving window operator* [NA 93] and the Σ-operator [JMR 91, Jen 91]. Details can be found in [TMJ 94].

6 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 I-tree as a permanent index on transaction timestamps on each backlog. An I-tree is a sparse B^+-tree-like index with high space utilization where the format is the same for all nodes (root, internal, and leaf). Nodes are connected by single pointers which eventually all will be used, and the nodes are not allocated before they are needed.

The size of the I-tree in number of disk pages has been estimated, and we have shown that its size is very small compared to the size of the corresponding backlog. The I/O-cost for maintaining the I-tree has been estimated and found to be relatively small for trees of high order. Finally, the I-tree can be used in the actual computation of a wide range of other queries such as, point queries, time-varying queries, the moving window operator, and the Σ operator.

Acknowledgement

Kristian 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.

References

[AB 80] Michel E. Adiba and Bruce G. Lindsay. *Database Snapshots.* Proceedings of the Sixth International Conference on Very Large Databases pp. 86-91, 1980

[AS 88] Ilsoo Ahn and Richard Snodgrass. *Partitioned Storage for Temporal Databases.* IEEE Transaction on Information Systems, Vol. 13, No. 4, pp. 369-391, 1988

[BCL 89] Jose A. Blakeley, Neil Coburn, and Per-Åke 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] Jose A. Blakeley, Per-Åke Larson, and Frank W. Tompa. *Efficiently Updating Materialized Views.* Proceedings of the ACM SIGMOD '86, pp. 61-71, 1986

[BM 93] Lars Bækgaard and Leo Mark. *Incremental Evaluation of Time-Varying Queries Using Predicate Caches.* Technical report CS-TR-2912, UMIACS-TR-92-64, Department of Computer Science, University of Maryland, College Park, MD 20742, June 1992

[CSLLM 90] Michael Carey, Eugene Shekita, George Lapis, Bruce Lindsay, and John McPherson. *An Incremental Join Attachment for Starburst*, Computer Sciences Department, University of Wisconsin — 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.* The Benjamin/Cummings Publishing Company Inc. ISBN 0-8053-0145-3, 1989

[EWK 93] Ramez Elmasri, Gene T.J. Wuu, and Vram Kouramaijian. *The Time Index and the Monotonic B^+-tree.* In *Temporal Databases Theory, Design and Implementation.* The Benjamin/Cummings Publishing Company Inc. ISBN 0-8053-2413-5, pp. 433-456, 1993

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

[Jen 91] Christian S. Jensen. *Towards the Realization of Transaction Time Database Systems.* PhD dissertation, Technical Report R 91-11 Aalborg University, March 1991

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

[JMR 91] Christian S. Jensen, Leo Mark, and Nick 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] Christian S. Jensen and Leo Mark. *Differential Query Processing in Transaction-Time Databases.* In *Temporal Databases Theory, Design and Implementation.* The Benjamin/Cummings Publishing Company Inc. ISBN 0-8053-2413-5, pp. 457-491 , 1993

[LHMPW 86] Bruce Lindsay, Laura Hass, C. Mohan, Hamid Pirahesh, and Paul Wilms. *A Snapshot Differential Refresh Algorithm.* Proceedings of the ACM SIGMOD '86, pp 53-60, 1986

[LJ 88] Nikos A. Lorentzos and Roger G. Johnson. *Extending Relational Algebra to Manipulate Temporal Data.* Information Systems Vol. 13, No.3 pp. 289-296 1988

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

[NA 93] Shamkant B. Navathe and Rafi Ahmed *Temporal Extensions to the Relational Model and SQL.* In *Temporal Databases Theory, Design and Implementation.* The Benjamin/Cummings Publishing Company Inc. ISBN 0-8053-2413-5, pp. 92-109, 1993

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

[Rou 91] Nicholas 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

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

[SG 89] Arie Segev and Himawan Gunadhi *Event-Join Optimization in Temporal Relational Databases.* In Proceedings of the Fifteenth International Conference on Very Large Data Bases, pp. 205-215, Amsterdam 1989

[SA 85] R. Snodgrass and Ahn, I. *A Taxonomy of Time in Database.* ACM SIGMOD, pp. 235-246, 1985

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

[Soo 91] M.D. Soo. *Bibliography on Temporal Databases.* SIMMOD, Vol 20, No 1, pp. 14-23, 1991

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

[Sto 87] Michael Stonebaker. *The Design of The Postgres Storage System.* In Proceedings of the 13th international conference on Very Large Data Bases, pp. 289-300, 1987

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

[TMJ 94] Kristian Torp, Leo Mark, and Christian S. Jensen. Efficient Differential Timeslice Computation in Transaction Time Databases. Technical Report, Georgia Tech 1994.

[TMJ 94a] Kristian Torp, Leo Mark, and Christian S. Jensen. Temporal Indexes for Transaction Time Databases. In preparation.

[Ull 82] Jeffery D. Ullman. *Principles of Database Systems.* Volume of Computer Software Engineering Series, Computer Science Press, second edition, 1982