Exploiting A History Database for Backup

David Lomet
DEC Cambridge Research Lab
One Kendall Square, Bldg 700
Cambridge, MA 02139

Betty Salzberg
College of Computer Science
Northeastern University
Boston MA 02115 *

Abstract

Database systems provide media recovery by taking periodic backups and applying a recovery log to a backup to restore the failed media. A transaction-time database is one that retains multiple versions of data, recording with each version the time of the transaction that created it. Such a database provides access to historical versions based on transaction time, and permits reconstruction of timeslices representing the data that were valid at past times. This paper shows how a TSB-tree supported transaction-time database can also provide the backup function of media recovery. Thus, the same versions used for database history are used for database backup. The cost of taking a TSB-tree backup is comparable to that of a conventional differential backup. The media recovery cost, especially when the media failure is partial, e.g. a single disk page, will usually be lower.

1 Introduction

1.1 Background

There is increasing interest in providing "data mining" capabilities, where one may wish to discover patterns over time, or to explore the database state at some time in the past. To support such temporal queries, transaction-time databases [15] store not only the current database but the history of the database as well. It is natural to ask whether this history data can also be used to provide backup for the current database in case of a media failure.

Traditionally, database systems take periodic backups to permit recovery from media (disk) failures. The backup reflects the state of the data at a previous time. If media failure occurs, the backup and the recovery log are used to restore the database. The preferred backup method takes a "fuzzy dump". Backups are "fuzzy" when normal transaction activity is concurrent with the backup and the copied data does not represent a transaction consistent picture of the database.

For a full backup, the entire database is copied in a background process. When the database is large (multiple gigabytes), a complete fuzzy dump becomes very expensive. Then differential backups, also fuzzy, which only copy pages changed since the last backup become very desirable.

In this paper, we show how to use a temporal indexing method, the *time-split B-tree* or *TSB-tree* [6, 7], for differential backup of the database. The TSB-tree is particularly suited to the task of supporting a transaction-time database. It is a multiattribute access method that accesses data by key and by transaction time. It partitions the transaction-time database into two components, a history database (archive) and a current database, while providing a single unified index to all versions of data. The two components may be kept on separate random access media. The history database can be on WORM media (write-once, read-many) as history data in a transaction-time database is never updated.

The TSB-tree incrementally migrates history records and copies of long-lived current records to the history database. We modify TSB-tree maintenance so that we can guarantee that every node containing a change made since the last backup is copied to the archive during the backup process. Normal database activity is concurrent with the backup process. The

*This work was partially supported by NSF grant IRI-88-15707 and IRI-91-02821.

**Proceedings of the 19th VLDB Conference,
Dublin, Ireland, 1993**

history database can then be used as a backup. We know of no other proposals along these lines. It provides an added role for the history records, hence increasing the value of retaining them, and reducing the incremental cost of supporting a transaction-time database.

It is important to emphasize the unique suitability of the TSB-tree for our purpose. It is the only temporal access method that simultaneously provides a $O(log(n))$ search and insertion cost, an $O(n)$ size, excellent clustering of time-slices in its nodes, and an integrated index for both current and history databases. Because data can move incrementally from current to history database, and because clipping introduces redundancy that is useful in backup, the TSB-tree is readily adapted to provide the backup functionality.

1.2 Optimizing Backup and Recovery

We have made a considerable and detailed effort to make backup and recovery efficient. Below, we describe the optimizations that we exploit.

1.2.1 Backup for Data Nodes

Records are clustered by time and by key in the leaf nodes (data nodes) of the TSB-tree. All nodes (whether data nodes or index nodes) of the TSB-tree are disk pages (contiguous areas of a disk). TSB-tree nodes are split either by transaction time or by key, as in the B^+-tree. Originally, nodes were split only when full. When using a TSB-tree for backup, (potentially non-full) nodes may also be time-split to ensure that the required versions of data are in history nodes of the tree.

During backup, we sweep through the current database and time-split only nodes updated since the last backup. The history database is written sequentially, with backed up nodes being written in large groups. These large sequential writes are very important for both the execution path length and the elapsed time of the backup. It is possible for the sweep to be done in parallel, with the writing of the new history nodes also done in parallel, as with conventional backups.

Normally, when a node is time-split, versions of data are removed from the current node. Indeed, the "normal" reason for doing a node split is to make space available in a node. Because we wish to make our backup process as efficient as possible, we do not remove data from the current data nodes. Current data nodes are not updated by the backup process.

1.2.2 Index Maintenance

In the current database, only index nodes are changed by the backup process. This is required to make the backup usable as a history database. The index also makes recovery from partial media failures fast by providing fast access to the specific backup node(s) required.

Our backup sweep of the current database is in tree traversal order. Then, all index terms for one index node are posted before proceeding to the next index node. This clustering of index updates reduces the I/O cost of performing them. The only log records produced by backup describe the updating of the index. This is necessary to permit us to guarantee the recovery of the index. We do not log changes to the history database made by the backup.

Every backup is incorporated into the history database and must be indexed. This contributes to a potentially large expansion in the size of the index. Reducing index growth is important to reduce the access path to data. TSB-tree index growth is restricted by exploiting the redundancy inherent in the backup process to purge unneeded index terms from the index nodes.

1.2.3 Media Recovery

We take advantage of proximity of the most recently backed up nodes on the history medium to use sequential access during media recovery. We can use write optimization via relocation to permit sequential writing of the restored nodes.

1.3 Organization of Paper

In section 2, the TSB-tree is reviewed. Section 3 shows how to modify TSB-tree node splitting so that backup cost is minimized. In section 4, how to handle the index during backup is described. Section 5 describes the overall backup process, and how it is possible to optimize the node splitting costs because backup is a "batch" operation. Section 6 outlines media recovery, how one can find the backup copies of nodes and how the log is applied. We end with a brief discussion.

2 The TSB-tree

2.1 Overview

The TSB-tree search algorithm and its split algorithms for index and data nodes, as originally described in [6], are recapped below. Using the TSB-tree for backup requires a specific (and different)

time-splitting strategy. Backup-induced splitting is described in subsequent sections.

Data records in a transaction-time database correspond to a line segment (one key value) in the time-key space. The line segment in the temporal dimension represents the lifespan [15] of the version of the data record, the start point being the transaction time at which it was entered, and the end point being the transaction time at which its successor version was entered, where its successor can be a "delete stub" indicating that the record has been deleted, not updated. In the TSB-tree, a version of a record consists of a pair, a timestamp denoting the transaction time of entry, and the data, including the key, associated with the version. The end time of a version is given by the start time of its successor version with the same key value.

The TSB-tree index entries are triples consisting of time, key, and pointer to lower-level tree node. Time and key respectively indicate the low time value and the low key value for the rectangular region of time-key space covered by the associated lower-level node. Upper bounds on these indexed rectangular regions are given by the lower bounds of other entries (with the same key but a later time, or with the same time but a higher key) or by the boundaries of the parent index node.

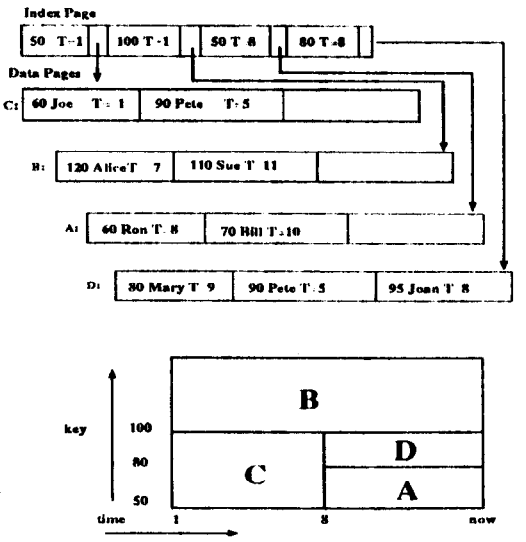


Figure 1: An index entry is the low key and time for the child's time-space rectangle. To find a record with key 60, valid at time 7, find the index entry with largest key $\leq$ 60 among the entries with time *leq* 7, i.e. (50 T=1). The record (60 Joe T=1) satisfies the search. (90 Pete T=5) is in two data pages because it is valid across the split time (T=8).

2.2 TSB-tree Searching

A search in a temporal index for a version with a given key, and valid at a given time, starts at the root and involves following index terms whose space contains the <key,time> point requested until the leaf of the index tree is reached. Then the versions of records in that leaf are searched for a record with the given key whose lifespan includes the time requested in the search.

In a TSB-tree, the search within an index node proceeds as follows. All index entries with times later than the search time are ignored. Within a node, look for the largest key smaller than or equal to the search key. Find the most recent entry with that key (among the non-ignored entries with time not later than the search time). Follow the associated address pointer. This is repeated until a leaf is reached. At the leaf, look for a version of data with an exact match on key and with the largest transaction timestamp less than or equal to the requested time. Searching is illustrated in Figure 1.

2.3 TSB-tree Node Splitting

A node of the TSB-tree can be split by time or by key. Deciding whether to split by time, or by key, or by both time and key, impacts the characteristics of the resulting TSB-tree. The implications of splitting policy are explored in depth in [7]. Here we describe only the mechanics of the splitting process. A sequence of splits is illustrated in Figure 2.

The TSB-tree exploits clipping [3] to cope with its entries that have extents. For versions of records, the extent only exists in the temporal dimension, while index terms are key-time rectangles, and hence have extent in both attributes. The regions indexed in a TSB-tree are maintained as rectangles only because entries are copied (i.e. clipped) and appear in both resultant nodes, when their extent crosses the node split boundary (line).

Clipping in TSB-trees introduces no redundancy for the current data of the current database or the index entries for the nodes of the current database. Since only the current database is subject to update, no update complications of the sort present with the R+tree are introduced by TSB-tree clipping.

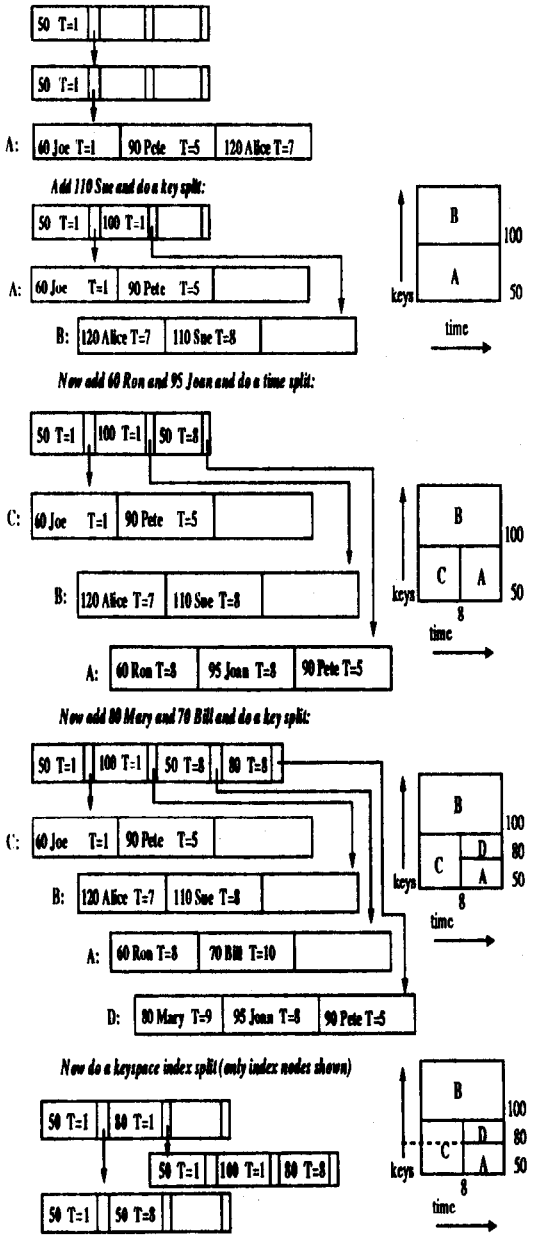


Figure 2: Illustrated is a sequence of splits ending in a key split for an index node.

2.3.1 Time Splits

It is time splitting that posts nodes (and hence migrates data) to the history database. To time split a TSB-tree node at T, all entries with time less than T go in the history node. All entries with time greater than or equal to T go in the current node. For each key-identified entry, the version with the largest time smaller than or equal to T must be in the current node. Thus records with lifespans that cross T are clipped and have copies in both nodes.

Any time after the begin time for a data node can be used to split a data node. For an index node, the split time cannot be later than the begin time for any index term referencing a current node to insure that history index nodes do not reference current nodes. This prevents us from having to deal with history node updates when a current node splits. Hence, index entries are posted to only one node. History nodes, which do not split, may have several parents. Current nodes have only one parent.

2.3.2 Key Splits

A TSB-tree data node is split by key exactly like a B^+-tree. All the records with key greater than or equal to the split value go in the new node and the records with key value less than the split value remain in the old node.

Key splitting for index nodes is like time splitting in that index entries have extents in the key space dimension. A key K of an index term is chosen as the split key value. All entries with keys less than K are placed in the old (low keys) node. Entries with keys greater than or equal to K go in the new (high keys) node. Each entry with the largest key, K_1, less than or equal to K as of any time included in the node must be in the new (high keys) node as the extent of such an entry includes K. If K_1 is less than (not equal to) K, this entry is thus in both nodes (see Figure 2). Note that because the key space is refined over time, any such clipped index entry will reference a history node.

2.3.3 Concurrent Node Splitting

Essentially any concurrency algorithm used for B^+-trees, such as those listed in [13], could be used to provide concurrent updating and node splitting for TSB-trees. The current nodes of the TSB-tree are the only ones that are updated. The current partition of the TSB-tree looks like a B^+-tree when it comes to node splits, except that the new history nodes are placed in the history database. However, concurrency is particularly high when using a method based on the

B-link tree [4]. Such a method which also provides for recovery as well, is described in [8]. Changes made on the leaf level are in separate atomic actions from those made above the leaf level. Details of the use of the concurrency method of [8] in TSB-trees can be found in [9].

2.4 Timestamping

The time chosen as the transaction time for timestamping record versions in the TSB-tree is the commit time of the the transaction which created that version. Choosing transaction time at transaction start has the advantage that it is available at the time that updates to the database are being made. However, this excessively constrains serialization order, leading to more transaction aborts than is desirable. However, choosing transaction time at commit implies that changed data must be visited twice. The first visit adds the new version and stamps it with the transaction's identifier (TID). The second visit, after commit, replaces the TID with the transaction time chosen.

Since timestamping goes on *after* a transaction has committed, the association between a transaction and its time must be stably stored. Storing the transaction time in the commit record on the recovery log accomplishes this. For efficiently finding a transaction's time, a TID-TIME table is kept in addition in volatile memory. The TID-TIME table can be periodically written to disk to speed its recovery after a crash. It is brought up-to-date using the logged commit records.

To be able to garbage collect the TID-TIME table, we suggest that it also contain a list of pages whose records require timestamping. We expect most of the pages to be timestamped will be in the database cache right after the transaction commits. Timestamping them will be part of the disk write, which will be logged. As suggested in [14], timestamping can be piggy-backed on subsequent updates to the same page. Any timestamping that remains can be done by a background process, and must also be logged. Garbage-collecting the TID-TIME table is outlined in [10].

2.5 Distributed Transactions

Agreeing on and distributing transaction time to the cohorts of a distributed transaction can be handled by augmenting the two-phase commit protocol messages. Cohorts vote not only on whether to commit a transaction, but also on transaction time [5]. The time chosen by the coordinator is not earlier than the times voted by cohorts. Hence, while a transaction is PREPARED, a lower bound for its commit time is known, i.e. the time voted by the cohort. This lower bound on commit time is of use when dealing with backup induced time-splits for nodes containing data from prepared transactions. Thus, we also include prepared transactions and the time voted by the cohort in the TID-TIME table. Thus, this table contains the attributes (i) TID, (ii) TIME: either transaction time or time voted during prepare, (iii) STATUS: either committed or prepared and (iv) a list of pages in need of timestamping by this transaction.

3 Node Backup

3.1 Data Nodes

New considerations govern the details of data node time-splitting for backup. In particular, all changes since the last backup are placed in the history database, and the current database is not written.

3.1.1 Identifying Nodes for Backup

Data nodes updated since the last backup are indicated by a **Node Change Vector** or **NCV**, a bit vector with one bit for each data node in the TSB-tree. Similar bookkeeping will be necessary for any method of differential backup that does not read the entire database. The NCV is ordered by physical position of the nodes on the disk.

When a data node is changed, its need for backup is indicated by setting its NCV bit. This bit is cleared after a node is split for backup, if there are no records in the node from uncommitted (prepared or active) transactions. Nodes with records of uncommitted transactions, that may commit before the next backup time, must be copied in the next backup even if there is no subsequent change. The next backup pass will write the timestamped data in the correct time interval in the backup. Hence the NCV bit for these nodes is not cleared.

The NCV is treated like a database system table with respect to system crash recovery. That is, its changes are logged (via log records indicating updates to TSB-tree nodes) and it is written to disk like other system tables as required by the recovery checkpointing process.

3.1.2 Current Node as History Node

When using the TSB-tree for media recovery, all time-splits copy the entire current node to the history

database. This ensures that all updates prior to the backup will be present in the history nodes. This may require writing uncommitted data to history nodes, where the uncommitted transaction may be given a transaction time later than the backup. This data is marked as uncommitted in the history node. TSB-tree index terms will direct us to the current data node when this data is desired. Hence, this data in history nodes is harmless with respect to searches and is necessary with respect to recovery. We call such data which is not within the time-space region described by the index term referring to it **Search Invisible** or **SI**.

If there are records of committed transactions in the current node which are not timestamped, we replace their TIDs with the transaction times in the backup history node. The copy of the record in the current database still needs to be stamped as the current database is not written during the backup process.

3.1.3 No Change to the Current Node

Backup makes no changes to data nodes in the current database. However, a new index term describing the backup time-split is posted. Hence like a history node, a current node can contain SI versions of data. The SI versions in the current node are versions which are no longer valid at the new start time for the current node indicated in the index. These SI versions have been superceded by more recent versions at (or before) the start time in the index term.

To make detection of SI versions easy, a **START** time is kept in each current node. START is the earliest time covered by data in the node. When START is earlier than the time in the index term for the node, SI versions may be present.

3.1.4 Data Node Split Times

If a data node contains no updates from prepared transactions, then current time can serve as the split time for data nodes. When there is a record in the node from a prepared transaction, we do not know whether its transaction time is before or after the current time. We do know what the local cohort voted as transaction time. We choose as split time the earliest such voted time of any such record.

3.2 Index Nodes

3.2.1 Unique Properties of the Index

Index nodes are treated differently from data nodes because

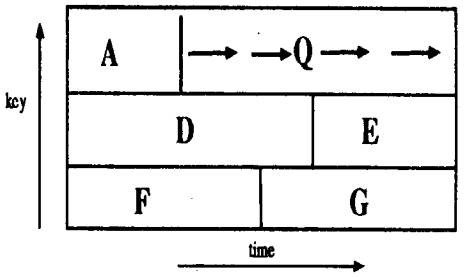


Figure 3: The current child node Q has not been changed since the last backup. All Q's records have timestamps $\leq$ the start time in its index entry. The records in A, valid the last time the node was backed up, (previous start time for Q), are still valid. Thus the start time in Q's index entry (which is the end time for A) can be moved forward to the current time.

1. Index terms for the backup-induced new history nodes must be posted. Thus, the index is changed by the backup process.

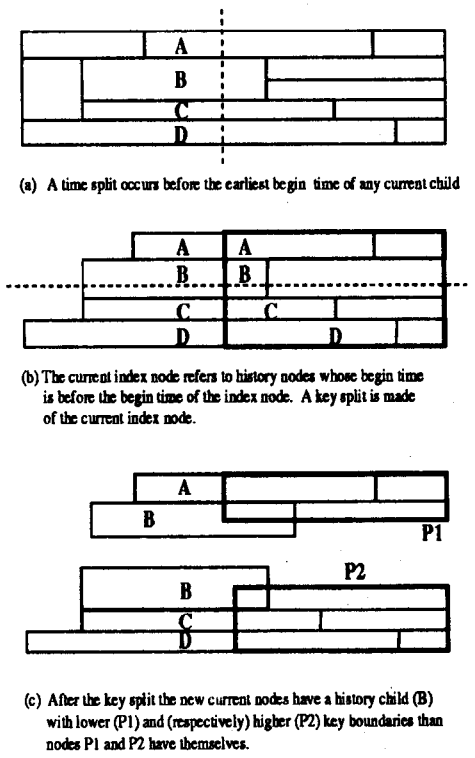
2. The split time for index nodes is never later than the start time of the oldest current index entry to ensure that history index nodes do not reference current nodes.

3.2.2 Split Time for Index Nodes

Unchanged data nodes are not read during backup. However, the start times in the index terms for these nodes are set to the current time. This indicates that the last historical nodes for these current data nodes include all changes up to the current time. This is illustrated in Figure 3.

Updated data nodes are read and split by the backup process, including the writing of a new historical node for them. The split time associated with the new index term for a current node is the oldest voted transaction time of any prepared transactions with updates in the node, or the current time if there are no prepared transactions.

Recall that an index node can be time-split using the time of its oldest current child. Hence, an index node which is the root of a subtree with updates of prepared transactions can be time-split, generating a new backup history index node, using the oldest transaction time voted by any prepared transaction. If the subtree has no updates from prepared transactions, the current time becomes the split time. The split time becomes the start time in the index term for this current index node.



(a) A time split occurs before the earliest begin time of any current child

(b) The current index node refers to history nodes whose begin time is before the begin time of the index node. A key split is made of the current index node.

(c) After the key split the new current nodes have a history child (B) with lower (P1) and (respectively) higher (P2) key boundaries than nodes P1 and P2 have themselves.

Figure 4: Index terms may refer to a proper subset of child space.

3.2.3 Index Term Covering

If all the new index terms generated during backup are posted to TSB-tree index nodes, they will cause the index to grow larger than would be the case if backup were not being done. Fortunately, many of the backup induced index terms are, or can be made, redundant.

In a TSB-tree, nodes denote rectangles in time-key space. An index term in an index node refers to that portion of the child's rectangle which intersects its parent's rectangle. Usually this is the whole child space. Sometimes, as we see in Figure 4, part of the child's space lies outside the boundaries of the parent. We say that an index term T_1 covers another index term T_2 if the space (a subset of the index node space) to which T_1 refers includes the space referred to by T_2.

We systematically eliminate covered index terms from index nodes. The node whose reference is erased in this index node may become inaccessible via a TSB-tree search, but all search relevant information remains accessible. Details of calculating child boundaries for index covering is in [9]. When using a WORM device for history nodes, space cannot be re-used. No attempt need then be made to recover space in the historical database used by inaccessible nodes.

4 The Backup Process

We wish to make the backup process comparable in cost to a conventional differential backup. Thus, we take pains to minimize the number of nodes read and written, and also perform batch writes of the information that backup needs to store stably. (Although we do not discuss parallelism *per se*, the TSB-tree can be partitioned by key range and these algorithms can be applied to each partition in parallel.)

4.1 Backup Splitting Steps

We require that a backup-induced time-split be done in three steps in the order given below.

Writing the History Node: The current node is copied to form the history node, which is written to the stable history database before the next step is executed.

Logging the Split: A single log record for the backup-induced split is written. This log record describes the posting of new index terms that update the parent index node.

Writing of the Index Node: The index terms describing the split are posted to the parent index node, making the new history node accessible. The updated index node is not made durable until the log record is durable, following the usual write-ahead-log rule.

4.2 Writing History Nodes

4.2.1 Non-root Nodes

A current node which is to be backed up is share-latched to assure read consistency while it is copied to the history database buffer to become the history node. This latch can be dropped as soon as the copy is complete. The necessary timestamping for a history data node can be done after the copy. Once the history node is stably written, the parent index node is updated and this update is logged. This usually requires only a short-term exclusive latch on the parent index node. If the parent splits, latches on higher level nodes will also be needed [8].

4.2.2 The Root

Backing up the root is handled somewhat differently. There is no index node above the root into which to store the index terms describing the backup-induced time-split of the root. Normal B-tree splits of a root cause the creation of a new root, but this new root, of necessity, is in the current database, hence requiring backup itself. We must break this recursion. When the backup copy of the root is made, we place a reference to it and to the split time into stable storage as part of our backup status information.

4.3 The Backup Sweep

The backup is done in key order, nodes with lower keys being backed up before nodes with higher keys (or consistently the reverse). This has two desirable results.

1. An index node is backed up immediately after its descendents. This assures that the backup version of the index node references the new backup versions of all descendent nodes.

2. An index node will most likely remain in cache and available while its descendents are being split. Hence, the effect is to batch the updates to the index node.

We write the backup history nodes as part of large sequential writes. During backup, as soon as a node is split, we place its history node in the output history buffer, which serves as an output queue. A history node must be written prior to the log record that describes the split. We write several history nodes prior to posting of their index terms. The index node is then updated with a group of backup-induced index terms, producing log records for these updates in a batch as well.

4.4 The Sweep Cursor

The **Sweep Cursor** permits the ordering requirements of backup to be enforced. Its restoration after a system crash permits an interrupted backup to be resumed. The Sweep Cursor contains the following information.

Log Key: the key for the last node whose backup is recorded stably in the log.

History Key: the key for the last node whose history node has been written stably to the history database.

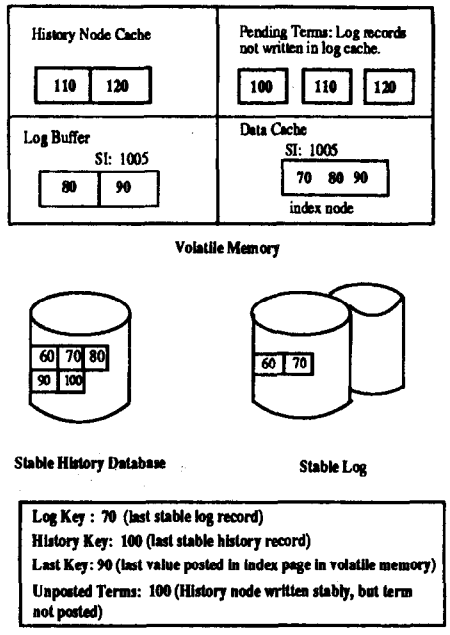


Figure 5: The Sweep Cursor: The unposted term 100 can be entered into the log buffer and posted to the index node in volatile memory. Once the log buffer is flushed to the stable log, the index page can be copied to the stable current database.

Last Key: the key for the last node whose backup has been completed. This includes the posting of the index entry in volatile memory.

Unposted Terms: the set of index terms as yet unposted for history nodes that have been written.

To enforce the writing of the history node prior to the logging of the split, we require History Key $\geq$ Log Key. Since parent index nodes are not updated until history nodes are stable, Last Key $\leq$ History Key. The write-ahead log rule implies Log Key $\leq$ Last Key. The Sweep Cursor is illustrated in Figure 5.

4.5 The Backup Status Block

The **Backup Status Block (BSB)** provides a durable repository in a known location for information related to backup. This information is of two types: (i) information needed to quickly initiate recovery from media failures; and (ii) information that assists in making the resumption of backup fast should the system crash during backup. The BSB relates a backup with the log information needed to

roll forward from the backup to regenerate the current database. This information includes log sequence numbers (LSNs), essentially addresses within the log, that indicate the beginning of media recovery related information on the log. The BSB contains the following:

BACKUP ROOT: the location of the history root of the last complete backup. This determines where media recovery finds the backup that should be restored.

SAFE POINT LSN: the redo safe point LSN associated with the last complete backup. This determines where media recovery should start its redo scan of the log.

NEW SAFE LSN: the redo safe point LSN associated with the in-progress backup. This will become the SAFE POINT LSN when the current backup is complete. It is NIL when backup is not in progress.

NCV: the location of the most recent stable copy of the NCV.

4.6 Backup Across System Crashes

It is unacceptable to undo backup to its start following a crash. Rather, if a backup is in progress when the system crashes, we want to resume backup from the point that was reached thus far. How we accomplish this is described here.

First, normal database recovery is performed, bringing all nodes up to the state as of the time of failure. Normal database activity can resume at this point. The BSB and the NCV are restored as a result of database recovery.

What remains is to restore the Sweep Cursor. The recovery log is searched back from its end. The first backup log record encountered indicates the last completed node backup and its HIGHKEY provides the values for Log Key and for Last Key. This same log record provides the history node where a search in the history database starts to find the value for History Key. The search continues until the end of the history database, the high key of the last backup node becoming the History Key of the Sweep Cursor. The Unposted Terms are re-created during this scan by examining the key and time attributes of the history nodes encountered. Once the Sweep Cursor is regenerated, normal backup resumes.

5 Media Recovery Process

5.1 Fundamentals

When there is a media failure in the current database, the first step is to restore all damaged nodes that have backups from the most recent accessible history nodes. Each of the restored nodes can be rolled forward from this restored states by applying their log records. Nodes without backups in the history database can be restored solely from their log records. (These are nodes created by key splits.)

What we consider next is how the history database is accessed to deal with different types of failures. We want to minimize (i) the read accesses to the history database; (ii) the write accesses needed to restore the backups to the current database; and (iii) the read accesses to the media log when rolling the restored database forward.

5.2 Full Database Media Recovery

5.2.1 Minimizing Backup Accesses

For a full restoration, traversing the history tree, starting at the history root has the advantage of encountering substantial clustering of history nodes needed for database restoration on the backup medium. Relevant history nodes from the most recent backup will exist at very high density in a small region corresponding to the backup time. As the regions associated with increasingly older backups are accessed, the density of occurrence of still relevant history nodes declines. This is illustrated in Figure 6.

We can identify regions of the backup medium where the density of nodes needed for a restore exceeds some threshold, e.g. 50 %. We can read these regions in large sequential reads, spending some data transfer time in order to save access times.

5.2.2 Sequential Writes to Restore Data

Full database media recovery may require that the restored data be relocated to new stable (disk) storage. When the recovery log is applied, we translate the old locations of current nodes, as recorded in the log, to the relocated locations of the restored backup versions, and apply the log records to the relocated nodes.

One could do the above translation by organizing the restored database so that "relative addresses" within restored data are preserved. This makes the size of the translation information very small. However, access arm movement when restoring the data

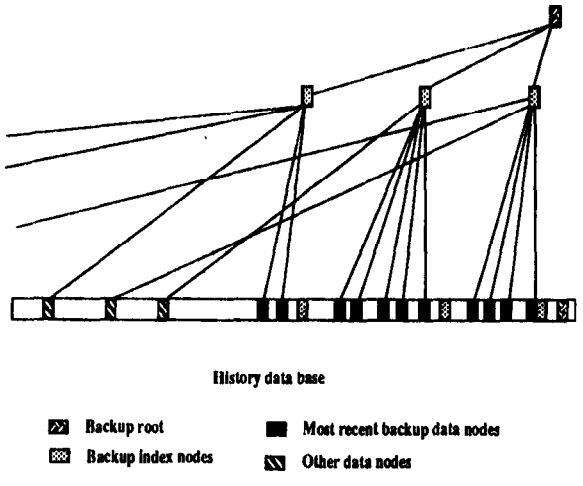


Figure 6: Many of the nodes needed for media recovery are clustered together in the area of the most recent backup. This includes the entire backup index and those data nodes that were in the last backup.

can be minimized by building a RELOCATION TABLE. As the backup tree is read, subtrees of the current database are reconstructed in memory and written to the disk in the same order as the backup, i.e. the children of an index node are written prior to the index node itself. The writing of the restored database can then require a very small number of large sequential writes. This exploits relocation to optimize writing in the same way as is done with log-structured files [12].

5.2.3 Applying the Log

The RELOCATION TABLE is also needed to permit the log to be successfully applied to the restored database. Log records refer to the pre-failure locations of the data, and need to be translated so as to correctly update the restored nodes. Further, node addresses for the pre-failure nodes that appear in index term log records need to be translated so that these addresses refer to the restored nodes.

As with conventional media recovery, the log can be processed so as to optimize the roll forward of the database. This involves what is called "change accumulation" [2]. The log is sorted by node and within node by time. The result is that the part of the log relevant to the rolling forward of a node is stored contiguously. If the RELOCATION TABLE approach is taken, it is useful to sort the log by the relocated addresses of the restored database. This permits a single sequential scan of the restored database for roll forward.

5.2.4 Summary

The bottom line is that there need be only a modest number of access arm movements to read the backup nodes from the history database. Writing the backup nodes to a restored current database can be done nearly sequentially. Hence, restoration after media failure can be done with high performance as well.

5.3 Single Node Restoration

Media failure may involve only a single node. For these localized media failures, using the TSB-tree's history nodes as the source of the backup is a substantial advantage. The TSB-tree's index, which is available in the current database, can be used to locate the backup node.

If we know the key range of data in the corrupted node, we can readily find a backup version in the history database. We use the key range information to search the TSB-tree for the most recent history node with that range. We use this history node to restore the corrupted current node. This node is then rolled forward by applying the recovery log. If the corrupted current node does not have a history node, then it was produced as a result of a key split. Such a node can be restored solely from the recovery log.

If a corrupted node is encountered in such a way that its key range is not known, then more extensive searching is required. The locations of current nodes that have backups are all in history *index* nodes of the TSB-tree. With an index node fan-out of around 200, the TSB-tree index represents about 0.5% (.005) of the database.. Even scanning all the current entries in the most recent history index for the failed node requires only a small fraction of the I/Os needed were we to search the entire backup database.

6 Discussion

6.1 Impact on TSB-tree Attributes

Our backup process has been designed to have performance competitive with conventional differential database backup while permitting the backup to also be used as a history database. We want to emphasize here that doing this has not compromised the performance of the TSB-tree in its support of a transaction-time database.

1. Single version current utilization (the proportion of the current database space occupied by cur-

rent versions of data) is little changed since current nodes are not updated during backup.

2. The height of the TSB-tree index is only modestly affected because history node index terms are eliminated whenever they are covered by other index terms. Index-term covering should occur with high frequency.

3. Index node time-splitting is enhanced. Backup permits the time chosen for the splitting of an index node to advance, which permits history index terms to be removed from current index nodes. This works to keep the height of the TSB-tree small.

6.2 Differential Backup Comparison

A conventional differential backup algorithm must also keep track of which pages were changed since the last backup. It would also have to take measures to protect against redoing the entire backup if the system failed while backup was in progress. It would also be necessary to located the most recent backed-up versions of each page to begin media recovery. This would lead one to believe that TSB-tree backup is comparable in cost to that of conventional differential backup. The advantage of the TSB-tree is that it can also be used to support a transaction-time database.

References

[1] Bayer, R. and Schkolnick, M. Concurrency of operations on B-trees. *Acta Informatica* 9 (1977) 1-21.

[2] Gray, J. Notes on database operating systems. IBM Research Report RJ2188 (Feb. 1978), IBM Research Division, San Jose, CA.

[3] Guenther, O. and Buchmann, A. Research issues in spatial databases. SIGMOD Record 19,4 (Dec. 1990) 61-68.

[4] Lehman, P. and Yao, B. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Systems* 6,4 (Dec. 1981) 650-670.

[5] Lomet, D. Consistent timestamping for transactions in distributed systems. Proc. of Conf. on Parallel and Distributed Information Systems, San Diego, CA (Jan. 1993) 48-55.

[6] Lomet, D. and Salzberg, B. Access methods for multiversion data. *Proc. ACM SIGMOD Conf.*, Portland, OR (June 1989) 315-324.

[7] Lomet, D. and Salzberg, B. The performance of a multiversion access method. *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ (June 1990) 354-363.

[8] Lomet, D. Salzberg, B. Access method concurrency with recovery. *Proc. ACM SIGMOD Conf.*, San Diego, (June 1992) 351-360.

[9] Lomet D. and Salzberg, B. Media recovery with time-split B-trees. Digital Equipment Corp. Tech Report CRL 91/9 (September 1991), Cambridge Research Lab, Cambridge, MA

[10] Lomet, D. and Salzberg, B. Transaction-time databases. in *Temporal Databases: Theory, Design and Implementation*, A. Benjamin Cummings, Redwood City, January 1993.

[11] Reed, D. Implementing atomic actions on decentralized data. *ACM Trans. Computing Systems* (Feb. 1983) 3-23.

[12] Rosenblum, M. and Ousterhout, J. The design and implementation of a log-structured file system. 13th ACM Symposium on Operating Systems Principles (1991)

[13] Shasha, D. and Goodman, N. Concurrent search structure algorithms. *ACM Trans. Database Systems* 13,1 (March 1988) 53-90.

[14] Stonebraker, M. The design of the Postgres storage system. *Proc. Very Large Databases Conf.*, Brighton, UK (Sept. 1987) 289-300.

[15] A. Tansel, J. Clifford, S. Jajodia, A. Segev and R. Snodgrass, editors, *Temporal Databases: Theory, Design and Implementation*, A. Benjamin Cummings, Redwood City, January 1993.