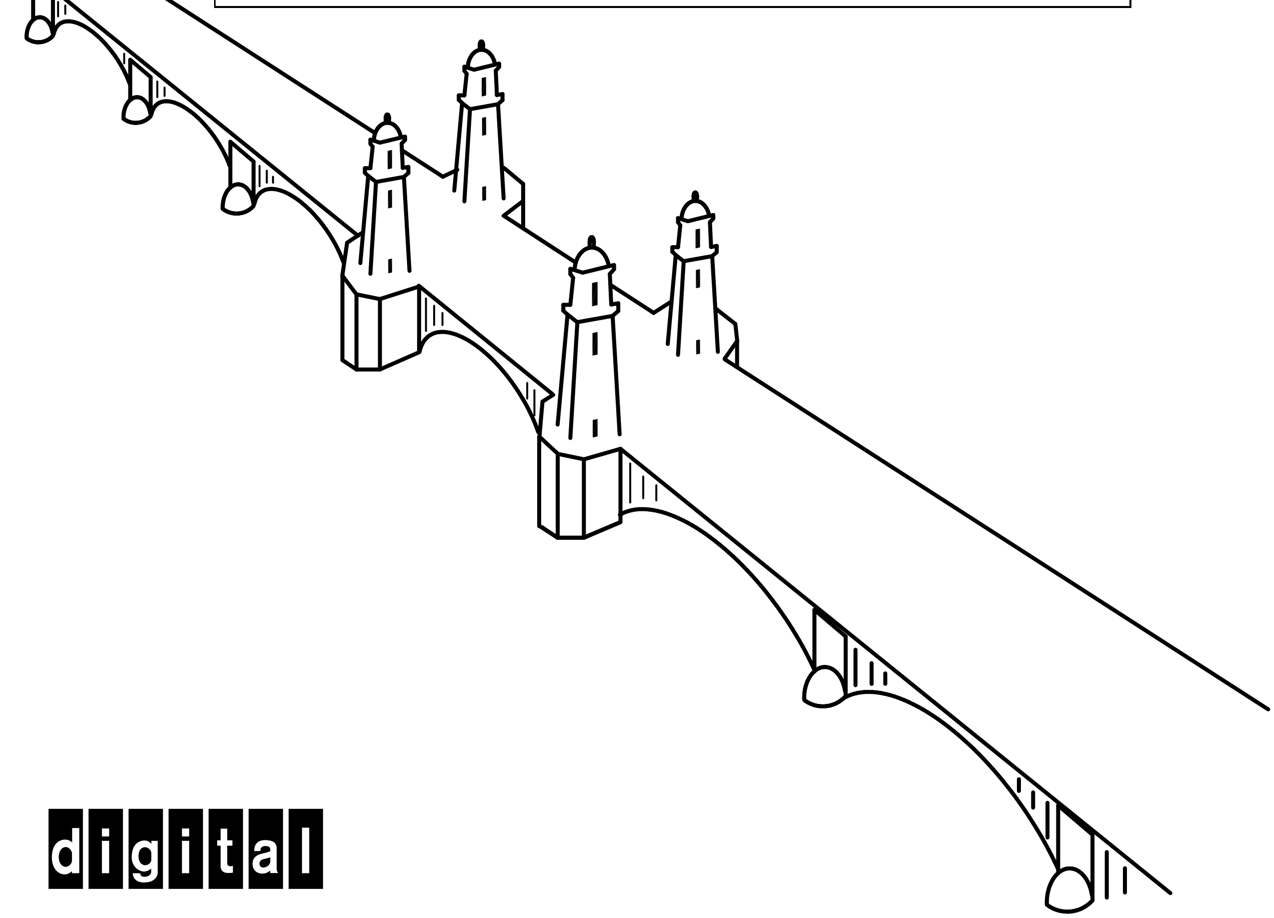
Media Recovery with Time-Split B-trees

David Lomet Betty Salzberg

Digital Equipment Corporation
Cambridge Research Lab

CRL 91/9 October 16, 1991



CAMBRIDGE RESEARCH LABORATORY
Technical Report Series

Digital Equipment Corporation has four research facilities: the Systems Research Center and the Western Research Laboratory, both in Palo Alto, California; the Paris Research Laboratory, in Paris; and the Cambridge Research Laboratory, in Cambridge, Massachusetts.

The Cambridge laboratory became operational in 1988 and is located at One Kendall Square, near MIT. CRL engages in computing research to extend the state of the computing art in areas likely to be important to Digital and its customers in future years. CRL's main focus is applications technology; that is, the creation of knowledge and tools useful for the preparation of important classes of applications.

CRL Technical Reports can be ordered by electronic mail. To receive instructions, send a message to one of the following addresses, with the word **help** in the Subject line:

On Digital's EASYnet:	CRL::TECHREPORTS
On the Internet:	techreports@crl.dec.com

digital™

Cambridge Research Laboratory
One Kendall Square
Cambridge, Massachusetts 02139

Media Recovery with Time-Split B-trees

David Lomet Betty Salzberg [1]

Digital Equipment Corporation
Cambridge Research Lab

CRL 91/9 October 16, 1991

Abstract

Modern database systems provide media recovery by taking periodic backups and applying a transaction log to the backup to bring the data up-to-date. A multi-versioned database is one that retains and provides access to historical versions of data. This paper shows how a history database, supported by the Time-Split B-tree, can be used to 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 backup is comparable to the cost of a good differential backup method, where only changed data is backed up. The media recovery cost, especially when the media failure is only partial, e.g. a single disk page, will usually be lower.

Keywords: media recovery, multiversioned data, access methods

[1] College of Computer Science, Northeastern University, Boston MA. This work was partially supported by NSF grant IRI-88-15707 and IRI-91-02821.

1 Introduction

1.1 Background

Traditionally, database systems take periodic backups to insure against media (disk) failures. The backup reflects the state of the data at a previous time. If a media failure occurs, the backup and the transaction log are used to recover the current (lost) database.

In [7,8], we describe a partitioned temporal database accessible by a common index. We call this the *time-split B-tree* or *TSB-tree*. The database has two components: a history database and a current database. The history database is kept in a separate random access medium; this could be a WORM device or simply another magnetic disk drive. A failure in the current database will not affect the history database, since it is stored separately.

The history database contains the same kind of information found in a backup, information that describes a previous state of the database. If media failure occurs in the current database, we want to use the history database as the backup. What we propose here can be viewed in either of two ways.

1. Database backups are organized so as to permit their use as a history database.

2. A history database, with appropriate protocols, is used for database backup.

In any event, the history nodes of the TSB-tree serve two purposes.

1.2 Overview

Media failure recovery is essentially redo recovery in which actions on the log that might not be reflected in an available stable version of data (in the backup) are applied to that version to bring it up-to-date. This paper shows how to modify the TSB-tree so that the history database has at least one copy of each version created by a given time. This permits the history database of the TSB-tree to be used as a backup for the current database. This is accomplished with very little overhead beyond that needed for traditional backup.

The transaction log contains a record of the changes since the backup was made. These changes are applied to the backup state to recreate the state of the database at the time of the failure. The log is scanned, starting at some **safe point** where it is known that all updates logged before the safe point are in the backup. It is important that the redo safe point for media recovery be controllable by the backup process. This requires that all updates logged prior to a proposed new safe point be stably recorded by the backup process.

The backup process is one of installing data current as of the time of the backup into a separate stable version of the database, in our case, the history database accessible through the TSB-tree. We "sweep" through the current database and ensure that all updates not yet in the history database are written to it. A **sweep cursor** is used to keep track of the progress of the backup.

A **Backup Status Block** is stably maintained so that the information needed for media recovery—the location of the last history root, for example—can always be found.

Normal database activity is concurrent with the backup process, so that what is described constitutes a "fuzzy" backup for media failure [1]. That is, the history database that results does not represent a transaction consistent view of the database in that some updates from some transactions may only be partially installed in the history database.

Originally, TSB-tree nodes were only split when they became full. When using a TSB-tree for backups, (potentially non-full) nodes may also be time-split to assure that the required versions of data are in history nodes of the tree. An entire recently changed current node is copied, no matter what split time is chosen. This is what enables us to set the safe point. It assures that all changes to the database that precede the redo safe point are in the backup.

We are careful to ensure that all data structures used by the backup process can be reconstructed from the log and the history database so as to permit backup to resume after a crash. Recovery from system crashes is almost entirely conventional. Should backup be in progress when the system crashes, a small amount of additional work is needed to permit backup to resume.

1.3 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.

1.3.1 Writing to the History Database

Over time, multiple backups are performed so as to shorten media recovery time by advancing the redo safe point. We maintain access to prior backups via the TSB-tree, indeed treating the backup as a history database. When backup begins, we sweep through the current database and time-split only nodes updated since the last backup. These newly time-split nodes, together with the prior time-split nodes in the history database, placed there during either normal time-splitting activity or previous backups, include a complete copy of all current data as of the time of the redo safe point.

A bit vector, called the **Node Change Vector**(**ncv**), is maintained to assure that only those data nodes which have changed since the last backup are written to the history database in the current 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.

1.3.2 Writing to the Log and the Current Database

Normally, when a node is time-split, versions of data are removed from the current node. These are the versions whose period of existence precedes the time chosen for the split. But because we wish to make our backup process as efficient as possible, we do not remove these versions from the current data nodes. Hence, current data nodes are not written during backup. Since current data nodes are not changed, no log records are needed to record their changes.

Only index nodes in the current database are changed during backup, which is required to make the backup usable as a history database. During our backup sweep of the current database, the order in which the nodes are backed up—key order with all children before the parent (in binary trees, this would be called “post order”)—enables the index posting to be batched. That is, 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 this updating of the index. Versions of data are never logged. In particular, we do not log changes to the history database. The ordering of the splitting process—(1) write history node (2) log the split (3) post index term—is what enables us to minimize the amount of backup-induced logging activity. Further, the writing of log records describing the index changes can be batched, permitting efficient sequential I/Os. While the WAL protocol must be followed, the backup process is not transactional, and hence few log forces are required.

1.3.3 Index Maintenance

Every backup is incorporated into the history database. All this data needs to be indexed in some coherent way in order for the backups to be usable as an integrated history. This all contributes to a potentially large expansion in the size of the index needed to access not only history data but also the current data, both of which are indexed in the same tree structure. Reducing index growth constitutes but a minor space saving, but is extremely important in order to minimize the access path to data.

TSB-tree index growth is minimized by exploiting the redundancy inherent in the backup process to purge unneeded index terms from the index nodes. By not removing data or index terms from current nodes during backup, we encourages the possibility that an index term created by the current backup "covers" a previous index term. By "covers", we mean that all the data accessible via the covered index term, pointing to one node, is available through the covering index term in the node that it points to. In this case, the covered index term can be replaced with the covering term. This limits the amount of backup-induced index growth.

Finally, the index nodes themselves must be backed up. To do this effectively requires that the split time chosen be as recent as possible. We can, without re-writing nodes, advance the split time associated with unchanged nodes so as to facilitate this effective split of the index nodes that refers to them. Without this, differential backup would be very difficult to achieve.

1.3.4 Media Recovery

To make the media recovery efficient, we exploit sequential reading and writing whenever possible. We take advantage of the proximity on the backup disk of the most recent backup records to do some of the reconstruction with sequential scanning. We can then write the restored nodes sequentially on the new current database if we use a **Relocation Table** to locate the recovered nodes.

1.4 Organization of Paper

In section 2, the TSB-tree is reviewed. The update process for records in a TSB-tree (as it relates to backup) is described in section 3. Included here are the steps that must be taken in order to prepare for backup. Section 4 shows how to modify TSB-tree node data node splitting to accommodate backup so that backup cost is minimized. In section 5, how to handle the index during backup is described, and, in particular, how to minimize the increase in index size that results from backup. Section 6 describes the the details of the backup-induced splitting, showing the particular steps and their order. Section 7 describes the overall backup process, and how it is possible to optimize the node splitting costs because backup is a "batch" operation. Section 8 outlines media recovery, how one can find the backup copies of nodes and how the log is applied. We end with a brief discussion of our results and their range of applicability.

2 The TSB-tree

2.1 Overview

The TSB-tree is a two dimensional search structure. Each node of the TSB-tree describes a rectangle in time-key space. This space must be searched to find appropriate records, and it must be partitioned to adjust to changing numbers of entries. The TSB-tree search algorithm and its split algorithms for index and data nodes, as originally presented in [7], are described below. Using the TSB-tree for backup requires different splitting algorithms. Backup-induced splitting algorithms are described in subsequent sections.

2.2 TSB-tree Searching

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. A search for a record with a given key valid during a given time proceeds as follows.

One begins at the root of the tree. 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. Repeat this algorithm at each level of the tree until a leaf is reached. At the leaf, look for an exact match on key when doing a point search. For a range search, look for the smallest key larger than or equal to the search key, now playing the role of lower bound on the key range. 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 [8]. Here we describe only the mechanics of the splitting process. A sequence of splits is illustrated in Figure 2, which is Figure 7 from [7].

2.3.1 Time Splits

If a node is split by time 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 (of a record in a data node or an entry in an index node), the entry with the largest time smaller than or equal to T must be in the current node. Thus records which are valid both strictly before and strictly after T have copies in both nodes. A record whose start time is exactly the split time will be found only in the current node. For index nodes, entries referring to lower level nodes whose regions span T are copied to both the history index node and the current index node.

Any time after the begin time for a data node can be used for a data node

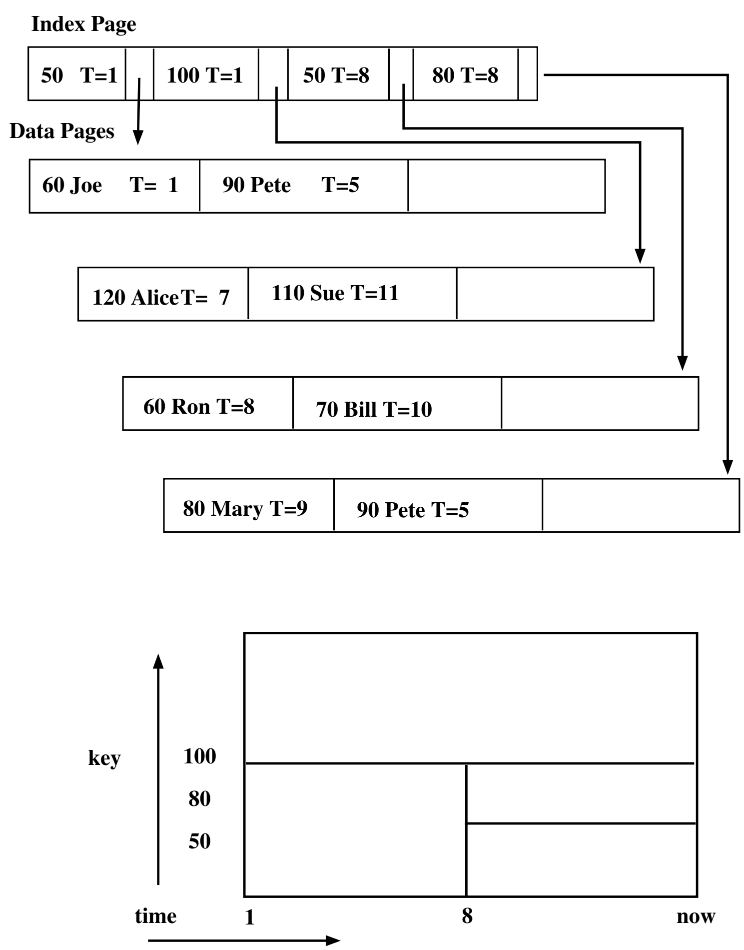


Figure 1: Each index entry is the lower key and earliest time for the time-space rectangle spanned by its child. To find a record with key 60, valid at time 7, ignore all entries in the index with time greater than 7. Find the largest key $\leq$ 60 with the largest time. This is the index entry (50 T=1). The record (60 Joe T=1) satisfies the search. It is valid until the next version (60 Ron T=8). (90 Pete T=5) is in two data pages because it is valid across the split time (T=8).

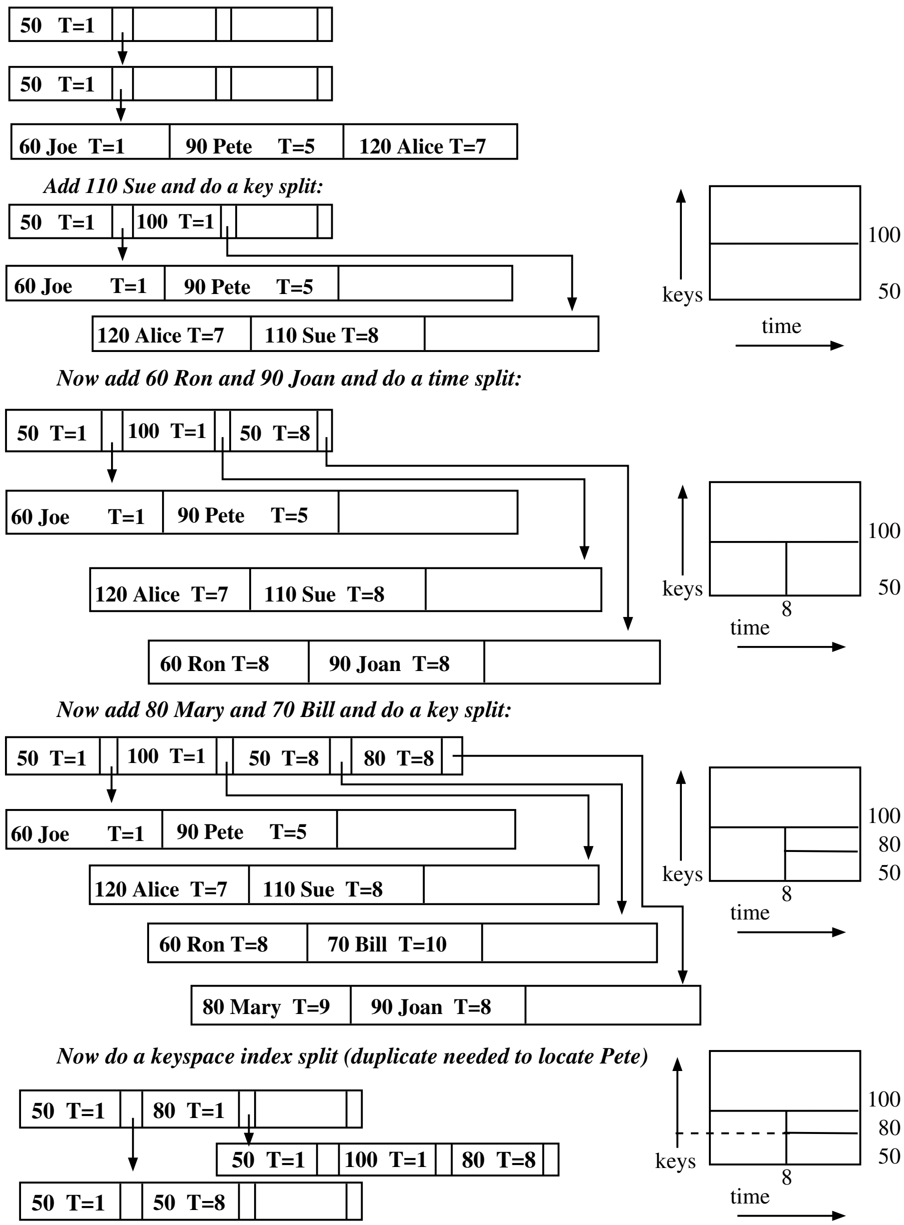


Figure 2: Illustrated is a sequence of splits ending in a key split for an index node.

time split. For an index node time split, the split time cannot be later than the begin time for any current child node to insure that history index nodes do not refer to current nodes. 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

Data records correspond to a line segment (one key value) in the time-key space. If a data node is split by key, the split is exactly the same as in 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 a little different since the elements referred to by the index entries are rectangles in time-key space. To split an index node, a key value from one of the index entries is chosen as the split value. References to lower level nodes whose key range upper bound is less than or equal to the split value stay in the old node. Those whose lower bounds are greater than or equal to the split value go in the new node (higher keys). Any lower level node whose key range strictly contains the split value must be copied to both nodes (see Figure 2). Note that because the key space is refined over time, any entry which is copied to both the new and the old node must be a reference to a history node.

2.3.3 Concurrent Node Splitting

We are faced with the usual concurrent B-tree update problem. That is, how do we keep the TSB-tree consistent when there are concurrent tree modifications (node splits) such that the index term for a node may need to be in another index node by the time that the node split is complete. This can be accomplished with a form of B-link-tree technique [4,13,14]. This involves lazily posting index terms to index pages sometime after a lower level node split. Search capability is preserved by leaving a forwarding address for the new, split-generated node in the original node.

We describe a concurrent node splitting method, which copes with system failures as well, for very general index tree structures in [9]. TSB-trees are among the structures handled. For the TSB-tree, forwarding addresses inserted with time splits form a linked list from the most recent node in the key range backwards in time. This implies that (1) searches for all versions of a given record will be fast and (2) a history node whose index term is not yet posted can be reached via sibling pointers from a more recent node in the same key range. For key-splits, the lower key range node will contain a pointer to the higher key range sibling, forming a linked list of current nodes from lowest key to highest key.

An additional complication arises when using B-links with the TSB-tree. If there is a time split, the "new" split-generated node is a history node, which cannot be updated. If a time split of the parent node occurs between the time a child node is split and its index term is posted, it is possible, but very rare, that the index term belongs in the new history index node. In this case, since history nodes are not updated, we do not post the index term. This requires that a forwarding address be used permanently to access the node.

Since we constrain the split time for an index node to be before the begin time of any current child node (so that the new history index node does not refer to current children), this anomaly only occurs when a second time split is made on a current child node before the index term for its first time split is posted. This unusual sequence of events is illustrated in Figure 3. Note that the search (following child and sibling pointers) will still be correct.

3 The Updating Process

We describe here the elements of the updating of data in a TSB-tree that are particularly relevant to the backup of a database. It is during updates to the database that we must prepare for database backup.

3.1 Timestamping

Timestamps distinguish the versions of data and enable the TSB-tree to readily support AS-OF queries. We explore briefly how data is timestamped below.

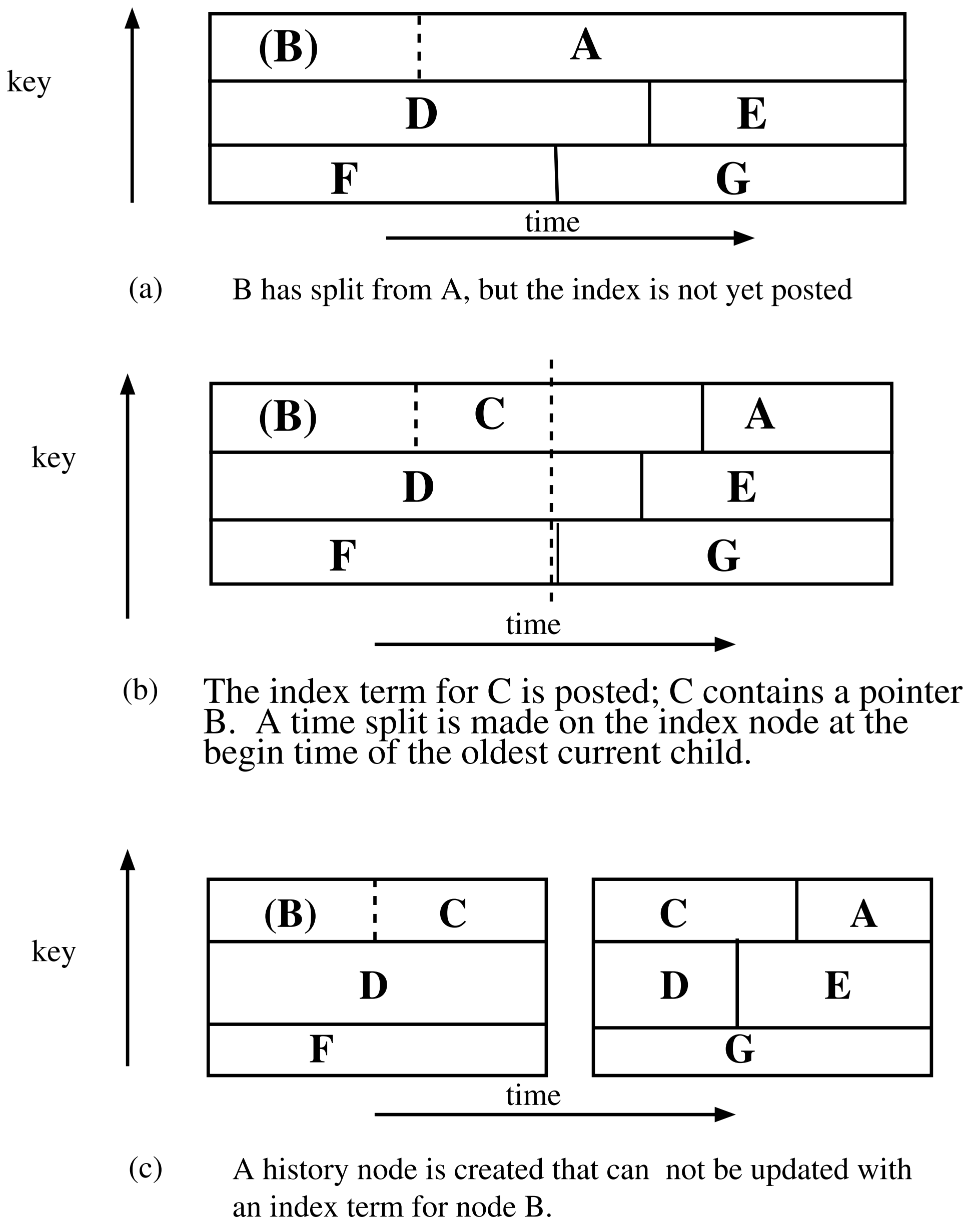


Figure 3: A time when the index term is not posted before the index node splits.

3.1.1 Concurrency Control Considerations

Choosing a time for a transaction at its start has the advantage that it can be easily propagated to all transaction cohorts and is available at the time that updates to the database are being made. Nonetheless, early choice of time excessively constrains serialization order, leading to more transaction aborts than is desirable.

We delay the choice of time until a transaction is committing. Choosing the time then permits it to reflect the serialization order actually experienced by the transaction. However, changed data must be visited twice. On first visit, the data changed by a transaction is stamped with the transaction's identifier (TID). In a second visit, after commit, the TID is replaced with the time chosen for the transaction. Thus, all stamped data is committed data, but not the converse.

3.1.2 Two-phase commit

With distributed systems, cohorts of a distributed transaction must negotiate the transaction time and this time must be distributed to cohorts so that they can stamp their changed data. Two phase commit(2PC), or some variant, must be executed to guarantee atomicity of distributed transactions. The 2PC protocol messages can be augmented so that cohorts can agree not only on whether to commit a transaction, but also on transaction time [5]. In this augmented protocol, the transaction coordinator chooses the transaction time based on the times voted by cohorts and distributes this chosen time along with the commit message.

Two-phase commit has an "in-doubt" phase, where a participant in a transaction has declared itself ready to commit, but has not received the commit message from the coordinator. Therefore, there may be records in TSB-tree nodes whose transaction time is earlier than the current time, i.e., the time has already been chosen, but the information has not been received by the cohort. This impacts the time-splitting of nodes. We exploit the fact that the time chosen by the coordinator is not earlier than the times voted by cohorts.

3.1.3 Timestamping of versions

Since timestamping goes on AFTER a transaction has committed, the association between a transaction and its time must be stably stored. This permits timestamping to be completed across system crashes. Storing the transaction time in the commit record for the transaction on the recovery log is one effective way of accomplishing this. However, this is not convenient for looking up the transaction commit time given the TID. Thus a TID-TIME table is kept in addition in volatile memory, to make the look-up more efficient. The TID-TIME table can be periodically written to disk, for example in log checkpoint records, to make it stable. Then after a system crash it can be reconstructed in memory from the stable version and the log records since the checkpoint.

All versions in a history node that can be encountered via TSB-tree search must be stamped with their transaction times, not their TIDs. If the timestamping has not been completed for versions in the current node when the node is being time split, it must be completed during the split. This is important for two reasons:

1. Choosing an appropriate split time requires this knowledge.
2. We do not update history nodes (they may be on WORM devices) so this is our last chance to timestamp the history data.

To facilitate dealing with prepared transactions, whose commit time is not known, it is useful to also include in the TID-TIME table prepared transactions and their time of prepare. Thus, this table contains the attributes (i) TID, (ii) TIME: either transaction time or time voted during prepare, and (iii) STATUS: either committed or prepared.

To reduce the space overhead involved in keeping TID-TIME table entries, we expect to garbage-collect entries for transactions when all the records they have updated have been timestamped. This is discussed in [10].

3.2 Marking Nodes for Backup

It is not necessary to copy all current data nodes to history nodes. Only data nodes that have changed since the last backup, plus the index nodes need be backed up and split. Copies of the other data nodes are already in the history database via a previous backup.

To avoid having to read data nodes to determine whether they have been updated since the last backup, we use a **Node Change Vector** or **NCV**, which is a bit vector with one bit for each data node in the TSB-tree. We associate a NCV with each backup sweep. During a backup, conceptually, there will be both a current NCV and the start of a next NCV for the next backup. The bits corresponding to nodes which have already been visited (in key order) are in the new NCV and the bits corresponding to nodes which have not been visited are in the current NCV. The NCV is ordered by physical position of the nodes on the disk. Thus an index is not necessary to find the bit corresponding to a given node. To change a bit, the backup process (or an ongoing transaction) must hold a latch on the node. The backup process always holds share latches. Updating transactions hold exclusive latches.

When a data node is changed, its need for backup is indicated by setting its designated bit to one. 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 which contain records whose transactions may commit before the next backup time must be read and copied in the next backup even if there is no change made in the node. This will enable the next backup pass to write the timestamped data in the correct time interval in the backup.

Current nodes with records which are not timestamped, but whose transactions have committed, may have their NCV bit cleared during the backup process, since the current backup already contains the correct timestamped version.

Other nodes with no changes in them are not read in the next backup, but their index terms will be moved forward, indicating that the historical version is still accurate at a later time. This is illustrated in Figure 4.

To summarize the cases:

1. A new version is entered in a current data node. At this time, its NCV bit for the next backup is set to 1.

2. A data node is backed up and all its timestamping is complete in the backup. Its NCV bit is cleared by the backup.

3. A data node is backed up, but some of the record versions are from uncommitted transactions whose timestamp cannot be entered in the

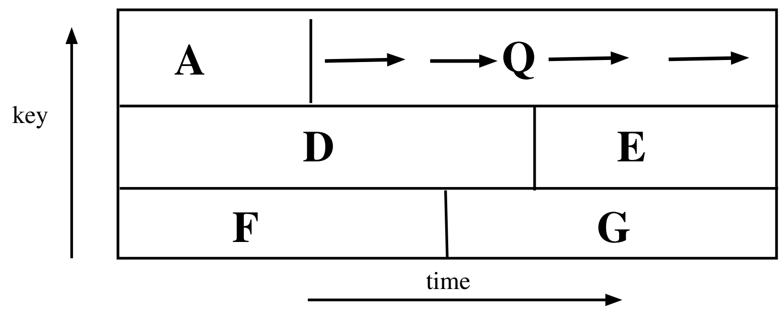


Figure 4: The current child node Q has not been changed since the last backup. All Q's records have timestamps $\leq$ the start time in its index entry. The records in A, valid at the previous backup time (previous start time for Q), are still valid at the new backup time. Thus the start time in Q's index entry (which is the end time for A) can be moved forward to the new backup time. This enables the index node to split at a later time than would otherwise be possible.

backup. Its NCV bit is not cleared; this data node must be processed again in the next backup.

4. A data node with a 0 NCV bit is not copied to the backup, but its index term time is moved forward.

When a backup occurs, all data nodes whose bits are one are read and backed up. Other data nodes are not accessed. Instead of a read-all sweep of the TSB-tree, we consult the NCV to determine which nodes need backup. Conceptually, all updates before we visit a node during backup are before the backup and go in the "old" NCV. Nodes updated after our visit are after the backup and go in the "new" NCV that determines what is backed up during the next cycle.

4 Backup Splitting of Data Nodes

New considerations govern the details of data node splitting for backup. In particular, we wish to ensure that all changes since the last backup are successfully placed in the history database, and we wish to avoid writing into the current database. How data nodes are time split during backup so as to accomplish these aims are discussed below.

4.1 Entire Current Node as History Node

Previously[7,8], we minimized redundancy in the history database, or made sure that increased redundancy led to reduced overall storage costs. For backup, we want to exploit redundancy. Backup **requires** that versions of data that are only in the current database be redundantly stored in the history database.

Thus, during backup, the entire current node is written to the history database. This ensures that all updates logged prior to the backup safe point in the log will be present in the backup copy of the database represented by the history nodes. This may involve writing data to history nodes from active (unprepared) transactions or from transactions which have prepared but not committed (in-doubt transactions). It does not cause a problem because the split times chosen for the index terms direct us to the current node when this data is desired. Such data in history nodes is harmless with respect to searches and useful with respect to backup and recovery.

We call any data which is not within the time-space region described by the index term referring to it **Search Invisible** or **SI**. Data copied from current nodes during backup which is not valid at the split time posted in the index (because their creating transactions have not committed by that time) is SI data.

If there are records in the current node which are not timestamped, but whose creating transactions have committed, we replace the TIDs with the transaction timestamp 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. If efficiency of the backup process is not a priority, the backup can be modified to update the current database in this case. Then, the entries in the TID-TIME table for transactions committing before the backup process begins can be erased at the end of the backup. In the rest of this paper, we assume that current nodes are not changed, even if they need timestamping.

4.2 No Change to the Current Node

Backup-induced time-splits do not remove data from current data nodes. Hence, current data nodes do not require re-writing. Backup makes no changes to the current database except for the posting of appropriate index terms that refer to the new history nodes.

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.

When normally inserting new versions of data into a current node, the SI versions left by backup-induced time-splitting can be removed if their space is needed. However, it is desirable to NOT remove SI versions UNLESS their space is needed. The presence of SI versions means that a current node continues to include all versions that were in its last backup history node. Once SI versions are removed, this "covering" ceases. This redundancy can be used to reduce the number of index terms. See section 5.2.

To detect SI versions, a **START** time is kept in each current node. START is the earliest time covered by data in the node. The current node includes all data versions in its key interval from START until the current time. START indicates the last time used in removing versions from the node. When this time is earlier than the start time in the index term for the node, SI versions may be present.

The start time for a current node posted with an index term resulting from a backup induced time-split will not match START in the node since the current node is not written during backup. This indicates that some cleanup is possible in which the too-old (SI) entries can be eliminated to make room for new versions.

4.3 Choosing a Split Time

The choice of a split time determines the start time that is posted with the index term that describes the split. How the index is handled is discussed in the next section. Here we discuss how the split time is chosen for data nodes.

In the absence of records of prepared (in-doubt) transactions, whose commit status and transaction time are unknown, current time can serve as the split time. This is the choice made for the WOB-tree[2], and which we have referred to as the WOB-policy[7,8]. It is the ideal choice as it maximizes the number of SI versions in the current node. This choice is possible even when there are versions from active transactions in the node. These transactions will commit (if they commit) after the current time, and hence their versions are never encountered in a search where the specified time is before or at the split time.

When there is a record in the node from an in-doubt transaction, we do not know whether its transaction time is before or after the current time. We can know, however, at what time the transaction was prepared locally, and what the local cohort voted as an earliest acceptable time. This voted time is found in the TID-TIME table. We choose as split time the earliest prepare time of any such record. This choice ensures that all searches for unstamped versions go to the current node, where they will eventually be stamped with their transaction time. The copies of these versions in the history database will be uncommitted as of the split time. Hence, in the history database, they are SI versions.

There is some chance that the earliest prepare time for a data node will be before the start time of the backup. This will limit the choice of split time for the index node which is its parent, and hence the backup split times of other ancestors. We offer two suggestions to deal with this problem:

1. Choose a backup time which is older than the oldest vote time for the system. Do not make a backup if there are in-doubt transactions with very old vote times.

   or

2. Mark in-doubt data in the backup history node as "possibly committed" if the split time is after the earliest vote time for the current node. More recent nodes in the same key range will contain the commit time if there is one. This will require some further search to answer AS-OF queries. Including the vote time with the data will reduce the frequency of these searches.

5 Handling the Index During Backup

5.1 Unique Properties of the Index

Index nodes are treated differently from data nodes because

1. Index terms for the backup-induced new history nodes must be posted to the correct index node. Thus, the index is changed by the backup process. The posting of these terms is done in the same basic way that index terms are usually posted in a TSB-tree. However, there are unique considerations, which we discuss below.

2. The split time chosen for index nodes is never later than the start time of the oldest current index entry. This time reflects the oldest (earliest) time any current descendent node could split. All subsequent splits of the current children will be posted to the current index node. (We note one exception below.) This is important because, like historical data nodes, we do not want historical index nodes to be updated. The way in which backup produces history nodes for the entire TSB-tree permits the split time for index nodes to be after (or at) the time at which the backup started (see section 6.4).

5.2 Index Term Covering

The number of new index terms generated by the backup process is troublesome. If these are simply posted to TSB-tree index nodes, they will cause the index to grow substantially larger than would be the case if backup were not being done. And the cost of each additional backup would be yet another substantial increase in index size. While this is somewhat unfortunate in the increased access path length to historical data, it is a very serious performance degradation for current data. Fortunately, many of the backup induced index terms are, or can be made, redundant. And redundant index terms can be dropped.

Redundant index terms are those that are said to be covered by other index terms. In a TSB-tree, index nodes as well as data nodes describe rectangles in time-key space. Index terms within an index node refer to that portion of the child's rectangle which intersects its parent's rectangle. Often, this is the whole child space, but as we see in Figure 5, part of the child's space may lie outside the boundaries of the parent. We shall 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.

Readers do not care whether they read data from the node referred to by the covered index term or from the node referred to by the covering index term. Both contain the same data for the given rectangle. Thus, we can systematically eliminate covered index terms from index nodes. The node whose reference is erased in this index node may become inaccessible, but no information is lost.

5.3 Calculating Child Boundaries in the TSB-tree

5.3.1 Index Terms and Index Nodes Terms

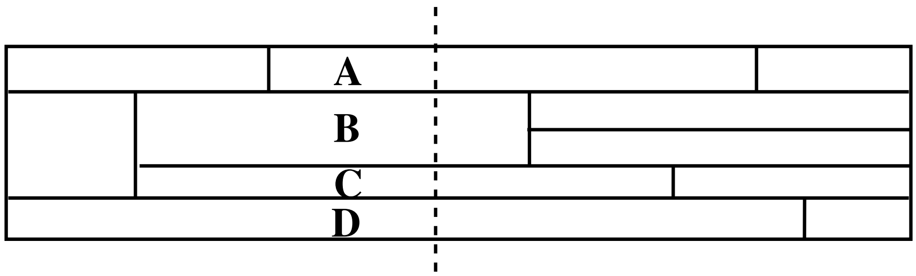
To take advantage of covering, each TSB-tree index node must contain sufficient information so that the key and time boundaries (within the space of the index node) of each referenced child node can be determined. START and END times and LOWKEY and HIGHKEY for each must be available (or derivable) so that we can detect when index terms for a newly posted split cover index terms already present. A TSB-tree index entry, however, only lists LOWKEY and START. In this section we show how to derive HIGHKEY and END for each index entry.

The index-splitting algorithms for the TSB-tree imply that the union of the time-space rectangles described by the children of a node may be larger than the time-space rectangle of the parent index node. Time splits which create a new history index node allow the current node to refer to children whose START is before the split time (START for the current node). This is illustrated in Figure 5(a).

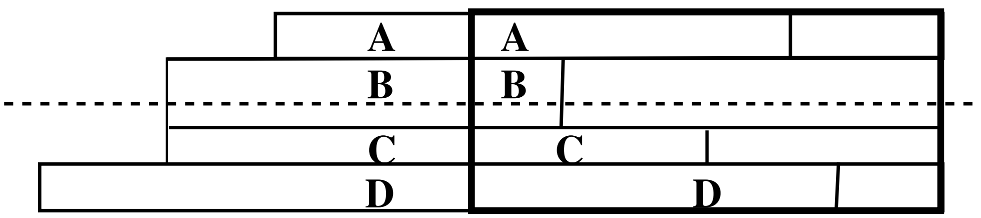
Key splits can cause a history child to have two parents. In this case, one parent refers to a history child whose LOWKEY is lower than the parent's LOWKEY. The other parent refers to a history child whose HIGHKEY is higher than the parent's HIGHKEY. This is illustrated in Figure 5(b).

5.3.2 Boundaries for Index Terms

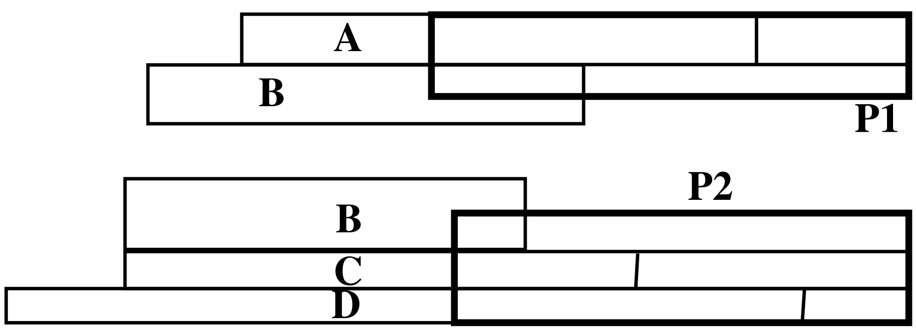
For current children of an index node, HIGHKEY is the next highest LOWKEY in the index node. The HIGHKEY of the entry with the highest LOWKEY



(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 5:

is the HIGHKEY of the parent. This is true because the key space is refined over time. Old key divisions are not lost and the current nodes reflect the finest partition of the key space. END for all current children is "now." A current child can be recognized as such because there is no more recent entry in the index node with the same LOWKEY.

Calculation of END and HIGHKEY for history children of current index nodes is more subtle. Suppose LOWKEY of a child is K, and START is T. END will be the earliest START after T of the smallest LOWKEY greater than or equal to K. Except when K is the smallest LOWKEY in the parent node and is less than LOWKEY of the parent (as is Figure 5(b) with child B in parent P1), there will always be a more recent entry with LOWKEY K. END for history child nodes can always be calculated from the information in the index node.

HIGHKEY of a history child will be LOWKEY of the entry with lowest LOWKEY greater than K and START before END for the child unless HIGHKEY of the child is greater than or equal to HIGHKEY for the parent. (See Figure 5(b) with child B in parent P2 or with child A in parent P1). One cannot calculate HIGHKEY for a history child which covers the highest key range for its time range. However, even in this case, the intersection of the child's space with the parents space can be calculated. (HIGHKEY of the intersected space is the HIGHKEY of the parent.) This is all that is needed to detect whether a newly posted index term covers or is covered by previously posted terms.

5.3.3 A Rare Case

When new index terms are posted, as long as they are posted in the order of creation, one can always tell from the information in the index node whether any covering occurs. However, our concurrent index posting algorithm [9] does not guarantee that the terms will be posted in order as we have seen in Figure 3. This can result in some of the boundary information being missing. The consequence of this is that a child appears to cover a larger space than it actually contains. To eliminate redundant index terms, where the directly contained space of one covers the directly contained space of the other, we need more information.

There are two possible responses to this difficulty. One is to ignore it and forsake the ability to eliminate a covered index term in such cases. The other is to supply the missing information when posting an out-of-order index term. We assume that the full boundaries of the index term being posted are known. When these boundaries are smaller than those that result from our previous calculations, we can record those boundaries in the index term posted.

5.4 Cases Encountered During Backup

The index-term covering optimization is important for both data and index nodes because of the frequency of one index term covering another as a result of backups. There are several possibilities for the node referenced by an index term.

No change: The node has not changed. The index term "covers" itself; We can advance the time demarcating the most recent history node and the current node to the current time without writing a new history node. These are the nodes which have a zero bit in the NCV (see Figure 4).

Only added versions: The node was changed between backups, but did not split. Nor were "SI versions" removed. The new backup-induced history node index term covers the prior history node index term generated at the last backup.

Versions removed: The node has had versions removed as a result of a time split or because SI versions were removed. The history node index term generated during backup will not cover a previous history node index term. Hence, its index posting results in an additional index term.

Key split without versions removed: The union of two (or more) index terms resulting from a key split and then a back-up could cover one previous history index term (if no SI versions have been removed). The key range of the previous history term could be reduced as each new back-up index term is posted. The previous history term can be eliminated when the last back-up index term in its key range is posted. (This is not an unusual case. Current nodes may contain only current data. When new data is to be inserted, there are no SI versions to be removed. If more space is required, a key split must be made.)

Covering normal node: A "normal" time split has occurred after the backup node is written, but before its index entry is posted. If this "normal" history node index term covers the backup index term, the backup index entry will not be posted. The "normal" history node index term covers this period of history, instead. (This case is rare.)

5.5 Index Node Time Splitting

The reduction in number of index terms, brought about by exploiting index-term covering, greatly reduces the frequency with which index nodes split. Because only splits of index nodes reduce their time-key space, backup of index nodes should frequently result in index-term covering at the next higher level in the TSB-tree. That is, index nodes can frequently be instances of **Only added versions**. SI versions of index terms should not be removed from the current index node during a backup so as to enhance the frequency with which this occurs. However, SI versions can be removed should the space be needed.

It is always possible to advance START for a current index node. The backup process guarantees that START for each index term for a current node is advanced by the backup process. Even when no backup of a node is required (NCV bit is zero), START for the index term for the current node is advanced. START is never older than the oldest prepared transaction in the data node at the time that it is backed up. Hence, a current index node can be time-split, generating its new backup history node, using the time of the oldest prepared transaction whose versions are in the subtree spanned by the index node.

6 Steps in Backup Splitting

6.1 The Steps in Backup-Induced Splitting

In order to make our backup process as efficient as conventional backup, we need to exercise care to minimize data contention and the amount of data placed on the system recovery logs. Also, backup must work correctly, leaving a well structured TSB-tree, should a failure occur at any arbitrary time during the execution of the backup process.

Essentially, a backup-induced time-split requires that three steps be accomplished. As usual for the permanence of an activity, the activity must be logged stably. We require the following three steps to be made stable in the order given below. The subsequent subsections explain how these steps are accomplished and why the ordering is important.

Writing the History Node: The current node is copied to form the history node, which is force written to the stable history database. Should the history database be on a WORM device, the space used is permanently consumed. A flag identifies the history node as a backup node.

Logging the Split: That a backup-induced time-split is done for a node is logged so that the split is durable. This log record describes the update to the parent index node.

Writing of the Index Node: The parent index node of the split node is updated to make the new history node accessible.

6.2 Special Treatment for History Nodes

Recovery for nodes of the history database is handled differently from recovery for nodes of the current database. This is true for both media failure recovery and crash recovery.

Media failure: We do not discuss history node media failure. The method described here does not support such recovery, though enough information may persist to recover from some failures.

System crash: Once history nodes are created, they undergo no further changes and hence, they need no further crash recovery support. The redo recovery process need not be aware of history nodes. Redo is applied solely to the nodes of the current database.

To assure crash recovery for history node creation, the history node is force written to the stable history database prior to recording stably any other information about the time-split that generated it. This differs from current nodes, whose write to stable storage can be extensively delayed. The early forced write of the history node means that creation of the history node will never need redoing when log records documenting the time-split are encountered during redo recovery. Hence we do not have to log the initial contents of history nodes. This is a significant performance enhancement. The size of the log is often a problem in transaction processing systems.

6.3 Logging the Backup Split

We wish to provide independent redo recovery [5,11] for current nodes. Independent recovery permits each node's recovery to be done by applying its log records to its previous states, independent of the log records and states of any other nodes. Hence, we need separate log records for each current node changed by a split.

Three log records are needed for a key split. One lists the records that are placed in the newly created node. The second describes the removal of records from the original node. The third describes the update to the parent index node. All changes to the structure of the index tree must leave the tree well-formed, even in the presence of system crashes. Ensuring this requires enforcing an appropriate unit of atomicity, which can be non-trivial. (In [9], we describe how this can be done efficiently and with high concurrency.)

For a backup-induced time split, much less needs to be logged. First, the current (original) node is unchanged. Second, by force writing the history node first, its creation never needs redoing. Neither of these nodes needs a log record. Hence, a backup split needs only a single log record describing the posting of the index term for the split. Hence, backup-induced time-splits are trivially atomic. If the log record is not present, the split has not been done. If the log record is present, the split has been done and is durable. The log never contains a partial backup-induced time-split which might have required undo recovery. This simplifies crash recovery.

The log information describing the posting of the index term for a backup-induced time-split includes the following.

NODE: index node being updated;

OPERATION: the fact that this is the posting of index terms for a backup-induced time-split;

LOWKEY: the low key for the range of keys in the split nodes;

HIGHKEY the high key for the range of keys in the split nodes;

HSTART: start time for versions in the history node;

CSTART: start time for versions in the current node; This is the end time for the history node. (The current node contains SI versions between HSTART and CSTART.)

CURRENT: location of the current node being split;

HISTORY: location of the history node; The history node is either a pre-existing node (when a new index term is posted for a node whose NCV bit was zero) or a new node.

6.4 Node Backup

6.4.1 Data Nodes

Backup of a changed current data node can be done atomically. The node is share-latched by the buffer manager 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, making for minimum interference with ongoing transactions. The necessary timestamping for the history node can be done after the copy.

Once the history node is stably written, the parent index node is updated. This requires an exclusive latch on the index node. The split is durable when the log record is stable. The write-ahead log protocol is observed to ensure that the log record describing the updated index node is stable by the time the index node itself is stable. Index node updating during backup may require that the index node be split. This complication is dealt with in the next subsection.

6.4.2 Index Nodes

Index node backup is more complex than data node backup because backup itself updates index nodes. It is essential to capture these updates in the backup induced history index node so that this node will correctly reference all backup nodes for its descendents. Thus, an index node is backed up only after the completion of backup for all its descendents, the writing of the history nodes, the posting of the index terms, and the logging of each split. After this, the ordering steps described in section 6.1 work correctly for the backup of index nodes.

An index node is updated by the backup of its descendents over a relatively extended period of time. Each update is only protected by a short-term exclusive latch on the node so that ongoing transactions can make updates to an index node interleaved with the updates caused by the backup process. Index node backup needs to deal with both forms of update. It is the posting of index terms for backup history nodes that makes these nodes available as history nodes in the multiversioned database.

Prior to backing up an index node, we back up the subtree of which it is a root. Backup sweeps through an index node, backing up descendent nodes. For index nodes whose children are themselves index nodes, all children require new history nodes to be created. For the lowest level index node, only the data nodes indicated in the NCV as having changed result in new history nodes. The descendent node backup is complete when the parent index node is updated to reflect its backup. In effect, above the leaf (data) level, we create an entirely new subtree in the history database. This subtree shares (history) data nodes that haven't changed with the previous subtree.

Should an index node time-split during its backup, no special measures are required. Time splitting does not remove **current** versions. Hence current index terms needed for the backup of the index node continue to be present in the changed index node.

The biggest complication involves the key splitting of an index node during its backup. There are two cases:

1. Our backup sweep has already finished with the index terms in one of the nodes resulting from the split. In this case, immediately perform backup on this completed node, and continue the sweep of the incompletely processed node.

2. We have not completed a sweep of either of the resulting nodes. Proceed as if the split hadn't occurred, and backup the index node that we are currently working in (after the split) when we complete its sweep.

6.5 Handling the Root

Backing up the root of a TSB-tree needs to be handled somewhat differently than the backup of an ordinary index node. 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. We call this information the Backup Status Block (BSB). It is described in section 7.3. We also log this update to the BSB, making the information recoverable from the log.

7 The Backup Process

We wish to make the backup process as inexpensive as possible, comparable to the cost of ordinary backup as done in non-versioned databases. For that reason, 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. This is described below.

7.1 TSB-tree Traversal for Backup Sweep

The time-splits needed to provide database backup are performed in a tree traversal of the TSB-tree. 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. Upon backup completion, all backup versions are accessible from the root of the history tree. It is this property that ensures a new log safe point for media recovery.

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.

Since we use sibling pointers so as to permit lazy posting of index terms to index nodes in the TSB-tree, it is possible that some changed nodes that need backup do not have index terms posted. The sibling from which they were split has been updated in the process of performing the split and appears as an updated node in the NCV. In the process of backing up this node, we need also to be sure to back up any new siblings that have been produced as results of node splitting. Further, since we wish that all backed up nodes be recorded in the backup for the index node, we insist on posting these index terms during the backup process. This ensures that only updated nodes, as marked in the NCV, ever have unposted siblings.

7.2 Batch Writing of Backup Data

Aside from posting index terms to current index nodes (discussed above), the writing done by backup consists of logging the index posting and writing the history nodes. Both of these can be done in sequential batches.

Because of this sequential allocation of history file space, we can 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. This gives it a location and causes it be written in location order when we perform sequential writes. Nodes are entered into the output history buffer before their ancestor nodes. The ancestor nodes are backed up as soon as we have finished with all their descendents. Hence they will be written after their descendents.

A history node must be written prior to the log record that describes the split. To enforce this protocol AND batch the writing of history nodes and log records suggests that several history nodes be written prior to the posting of their index terms. Then, the index node can be updated with a group of backup-induced index terms, producing log records for these updates in a batch as well. Thus, batch writing of history nodes and log records is feasible while observing the ordering requirements for correct concurrency and recovery.

7.3 Backup Data Structures

Backup maintains two data structures, the Sweep Cursor and the Backup Status Block. These are described below.

7.3.1 The Sweep Cursor

The **Sweep Cursor** encapsulates the instantaneous state of the backup. Its information 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.

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

Unposted Terms: Associated with each history node that has been written but whose corresponding index term has not yet been posted, a record is entered here of the form of the index term to be posted.

To enforce the writing of the history node prior to the logging of the split, we require History Key $\geq$ Log Key. The log cannot be written to stable storage, updating Log Key, until sufficient history nodes have been written. The History Key can get arbitrarily far ahead of the Log Key, should that be convenient, e.g., to facilitate batch writing of the history database.

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 6.

After batch writing a group of history nodes, the backup process posts the Unposted Terms recorded in the Sweep Cursor in a batch.

7.3.2 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. While much of the information is present

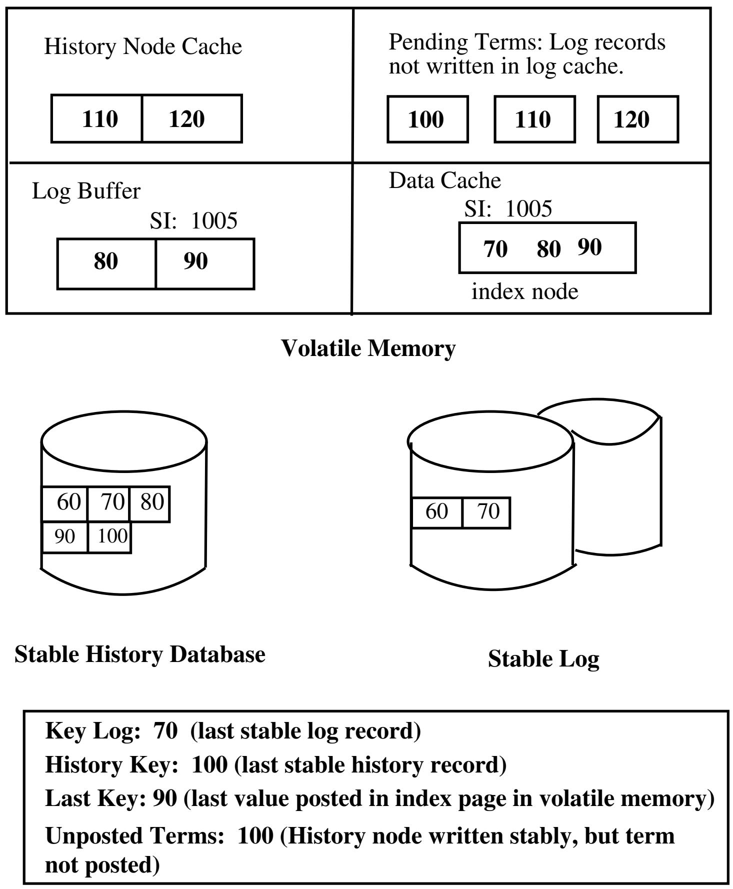


Figure 6: The Sweep Cursor: The unposted term 100 can be entered into the log buffer and posted to the index node in volatile memory. Once the log buffer is flushed to the stable log, the index page can be copied to the stable current database.

redundantly on the log as well, having it in the BSB avoids the scanning of most of the log to recover it. 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.

NCV LSN: the LSN of the last checkpointed copy of the NCV. During normal processing, we checkpoint the NCV to the log whenever a crash recovery log checkpoint is performed.

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.

The BSB may contain other information relevant to backup, such as when the next backup is scheduled.

The BSB is forced whenever a transaction recovery checkpoint is taken. This assures that its information can be fully brought up-to-date by scans that involve only the crash recovery log. Backup resumption is treated below. Media recovery is the subject of section 8.

7.4 Continuing Backup Across System Crashes

It is unacceptable to undo backup to its start following a crash. Indeed, once history nodes have been written to a WORM device, it is not possible to completely undo the backup. Rather, we want to resume backup from the point that was reached thus far.

Backup produces stable testable state that allows it to be resumed across system crashes. The history nodes written are stable, as are the log records describing the updating of index nodes. The nodes changed since the last checkpoint are all recorded on the crash recovery log. Hence, the NCV can be recovered. The BSB redundantly contains recent backup state information that makes resumption fast.

First, normal database recovery is performed, bringing all nodes up to the state as of the time of failure. Recall that TSB-tree backup splits are never undone, and hence all such splits will be redone where necessary. Normal database activity can resume at this point. BSB updates are described via log records. Hence, the BSB is restored as a result of database recovery. The NCV is also restored during database recovery.

If backup was in progress when the crash occurred, we need to restore the Sweep Cursor. The crash recovery log is searched back from its end. The first backup log record encountered indicates the last completed backup and its HIGHKEY provides the value for Log Key and for Last Key.

To determine the value for History Key, the history database is scanned from the location of the history node stored as the HISTORY attribute in the log record above to history database end. The high key of the last backup node encountered in this scan becomes the History Key attribute of the Sweep Cursor. The Unposted Terms are re-created during this scan by examining the key and time attributes of the history nodes. Once the Sweep Cursor is re-generated in its entirety, normal backup resumes.

8 Media Recovery Process

8.1 Fundamentals

When there is a media failure, the first step is to restore all damaged current nodes that have backups from the most recent accessible history nodes. After this restore step is completed, and prior to applying the log, failed nodes that existed at the time of the last backup all have been restored with valid past states.

With independent redo for current nodes, each of the restored nodes can be rolled forward based solely on their log records and their restored state. The log will also contain sufficient information to regenerate all nodes that do not have backup copies in the history database. These were created only via key splitting. Their initial contents have been stored in the log and they can be re-created without access to backup versions. Applying the media recovery log proceeds exactly like ordinary redo recovery after a system crash. The only difference is that the media failure redo scan starts at the media safe point, stored in the BSB, as opposed to the crash recovery safe point.

What we consider next is how the history database is accessed to deal with different types of failures. The important issue that we need to deal with is how to minimize the number of I/O accesses. We want to minimize (i) the read accesses to the history database, which might be stored on a WORM device with a slow access rate; (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.

8.2 Full Database Media Recovery

8.2.1 Minimizing Accesses to the Archival Medium

Recall that a backup is generated via a TSB-tree traversal. 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. For each index node, the backup history nodes for its descendents are clustered into a set of regions of almost contiguous backup nodes on the medium in which the history database resides. There will be one such region for each backup sweep whose history nodes are among the still relevant backup set for the current database.

History nodes from the most recent backup will exist at very high density in their region because only occasional ongoing activity during the backup will break the sequence of backup history nodes. As the regions associated with increasingly older backups are accessed, the density of occurrence of still relevant history nodes declines. This is because more and more of the history nodes written by these backups will have been superceded by more recent backups. This is illustrated in Figure 7.

Despite the only approximate nature of the contiguity of backup history nodes, we can use this locality to minimize the number of accesses required to read the backup nodes. We can identify regions of backup storage space where the density of nodes that need to be read for restoration exceeds some threshold, e.g. 50 %. We can read these regions in large sequential reads, spending some data transfer in order to save the access times. Further, performing access such that nodes from each backup sweep are processed together results in small disk arm movements.

Should the backup medium be a WORM device, it will frequently be the case that all data for nodes written in a given backup is on a small number

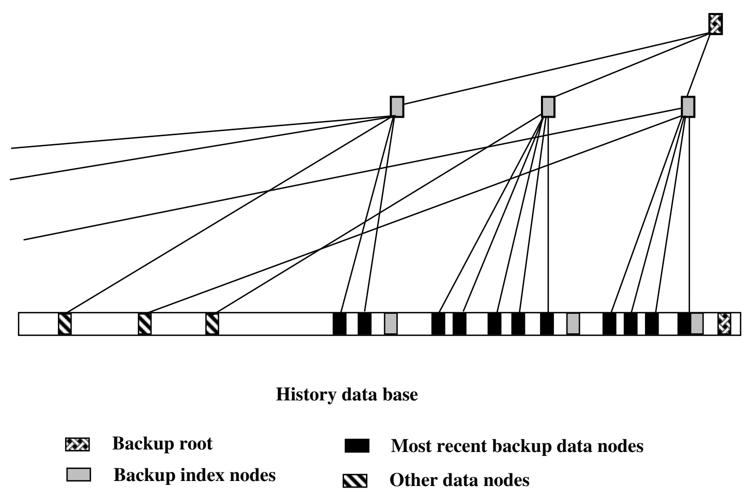


Figure 7: Many of the nodes needed for media recovery are clustered together in the area of the most recent backup. This includes the entire backup index and those data nodes that were in the last backup.

of tracks. Since adjacent tracks can be reached without arm movement by pivoting the mirror at the head, all such data can be accessed without moving the access arm. Thus, even without a large sequential read, we can take advantage of the proximity introduced by the backup process to minimize costly arm movement during database restore.

The same strategy used here for a full database restore can be used to restore an entire subtree of the TSB-tree. The only difference is that the root of the subtree is found by searching the current database for the index term that refers to the damaged subtree. Then, the most recent history node associated with that index node becomes the "root" of the history time slice traversed.

8.2.2 Sequentially Writing the Restored Database

A restored database, or substantial part thereof, needs to be relocated to new stable (disk) storage during the restoration step. When the recovery log is applied to the backup database, 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.

Of course, one could do the above translation in a trivial manner by organizing the restored database so that "relative addresses" within some storage area are preserved. This makes the size of the translation information very small. However, to minimize access arm movement when writing the current database, it would be better to build a RELOCATION TABLE while traversing the history database.

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 is done, 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 is essentially the suggestion in [12] for log-structured files, which are written sequentially in new locations after update, rather than being randomly written to do update-in-place. Here also an index must be kept to locate the moved pages.

Thus, the current database is restored by copying the backup data nodes to a new sequential area and rebuilding the current index nodes based on the historical index nodes, but updated with the new locations of their restored child nodes. The RELOCATION TABLE is used to provide this translation.

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.

The RELOCATION TABLE is built in main memory as database restoration proceeds. The write-optimized RELOCATION TABLE requires an entry per node to be restored. When recovery is complete we write the relocation table to our archive and post an entry to the BSB that references it. The RELOCATION TABLE will be used to permit restoration of the relocated (restored) data should there be a second failure before the next backup is taken.

8.2.3 Applying the Log

As with conventional media recovery, the media recovery log can be periodically processed so as to optimize the roll forward of the database. This involves what is called "change accumulation" [3]. 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. This minimizes the number of times a restored node needs to be visited during the roll forward process. 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 this step, at the cost of the log pre-processing.

The bottom line is that there need be only a modest number of access arm movements to read the backup nodes from the backup medium, a few per backup that is still reflected within an index node. 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.

8.3 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 to us 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 fully 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. And usually, some information is available about the possible range of keys.

9 Discussion

9.1 Impact on TSB-tree Attributes

Our backup process has been designed to have performance competitive with conventional 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 usual role of accessing multiple versions of data.

1. Single version current utilization (the proportion of the current database space occupied by current versions of data) is not adversely impacted since current data node contents are not changed.

2. The height of the TSB-tree index should not be much affected because of the systematic replacement of redundant history node index terms with history node index terms that "cover" them. This should be very effective in avoiding tree growth. Index-term covering should occur with high frequency.

3. Index node time-splitting has enhanced effectiveness. Backup permits the time chosen for the splitting of an index node to advance because the oldest current index term is only as old (approximately) as the last backup. The splitting of index nodes permits history index terms to be removed from the index nodes, which works to keep the height of the current tree small.

4. Multiversion total utilization (the proportion of the entire—historical and current—database occupied by current versions of data) should be comparable to that achieved by using WOB splitting, i.e. the splitting regime of the WOB-tree[2] in which the entire current node becomes the history node, when the nodes whose index terms are covered are ignored. Thus, utilization, in its impact on search performance and on the nodes encountered in a search, is not adversely affected. Of course, the existence of these nodes means that additional history space is consumed. (Even if the backup is not on a WORM device, the space consumed cannot be reclaimed if the sibling pointers in the nodes are needed. For example, in Figure 3, the only pointer to B in the TSB-tree

is in C. Even if the index term for C is covered, the space for C cannot be reclaimed, for then B would be inaccessible.)

9.2 Physically Organized Backup

The process that we have described organizes the backup process by subtree. That is, nodes that share close ancestors are written close to each other physically in the history database. Media failures are more likely to corrupt physically contiguous parts of the current database than logically close nodes in the TSB-tree.

It is possible to organize the backup process to deal with this. The cost of the backup is likely to be higher so as to make recovery faster. One can backup entire contiguous storage areas. Then one can update the TSB-tree index to reflect that this backing up has been done. This requires finding the logical nodes that correspond to the physical space. Usually, this is written into the nodes themselves. More data will probably be written to the history database because nodes that haven't changed will be written as part of the storage area.

One caveat here is that this technique should probably be limited to the data nodes. The index nodes must be written after their descendents if they are to reference the backup versions of their descendents. Storing the index in a separate storage area helps to accomplish this.

At recovery time, the entire storage can be restored via a single or a small number of sequential reads. Hence, recovery can be done very rapidly. Relocating the storage area is also simplified since the new origin of the restored storage area can serve to relocate all nodes in the storage area.

9.3 TSB-trees for Pinned Record History

Frequently, a database system stores the bulk of its data in an entry-ordered file and accesses that data via record identifiers (RIDs). The RIDs typically have two parts, a page identifier (PID) and slot identifier within the page (SID). Thus an RID is a pair <PID,SID>. This organization is called a "pinned record" organization as the records are always located by going to the PID named page and looking at the SID slot. The record itself can move, but it must be accessed via its slot.

A TSB-tree can be used as a history database for a current database of the pinned record form. The key used in this TSB-tree is the PID. The TSB-tree locates history versions of records via their PID and the time of interest. The current database is accessed directly via the PID.

Such a tree is built gradually as current database pages time-split. The time-splitting of a page causes its entry into the TSB-tree with an appropriate time. This page can be written directly to the backup medium (e.g. a WORM device) since it is not further updated. This process is entirely conventional for the TSB-tree for time splits. Key splits never occur in data pages.

Having used the TSB-tree to gain access to history versions of pinned records, it now becomes possible to use the TSB-tree for backup purposes as well. In this case, there is the added advantage that physical and logical locality are synonymous. Thus, the tree traversal and the storage area approach come together. Backup remains fast and recovery speed is enhanced.

Bibliography

1. Bernstein, P., Hadzilacos, V., and Goodman, N. Concurrency Control and Recovery in Database Systems. Addison Wesley, Reading MA. 1987.

2. Easton, M. Key-sequence data sets on indelible storage. IBM J. of R.D. 30,3 (May 1986) 230-241.

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

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

5. Lomet, D. Consistent timestamping for transactions in distributed systems. Digital Equipment Corp. Technical Report CRL90/3, Cambridge Research Laboratory, Cambridge, MA (Sept. 1990)

6. Lomet, D. Recovery for shared disk systems using multiple redo logs. Digital Equipment Corp. Technical Report CRL90/4, Cambridge Research Laboratory, Cambridge, MA (Oct. 1990)

7. Lomet, D. and Salzberg, B. Access methods for multiversion data. Proc. ACM SIGMOD Conference, Portland, OR (June 1989) 315-324.

8. Lomet, D. and Salzberg, B. The performance of a multiversion access method. Proc. ACM SIGMOD Conference, Atlantic City, NJ (June 1990) 354-363.

9. Lomet, D. and Salzberg, B. Concurrency and recovery for index trees. DEC TR CRL 91/8 (August 1991) Cambridge Research Lab, Cambridge Ma.

10. Lomet, D. and Salzberg, B. Managing Timestamping in a Multiversion Database. (In preparation).

11. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H. and Schwarz, P. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. Research Report RJ6649(Jan 1989) IBM Almaden Research Center, San Jose, CA and ACM Trans. on Database Systs. (to appear).

12. Rosenblum, M. and Ousterhout, J. The Design and Implementation of a Log-Structured File System. 13th ACM Symposium on Operating Systems Principles, 1991 and Transactions on Computer Systems (to appear).

13. Sagiv, Y. Concurrent operations on B*trees with overtaking. Proc. ACM SIGACT-SIGMOD PODS Conference, Portland, OR (June 1985) 28-37.

14. Salzberg, B. Restructuring the Lehman-Yao tree. Northeastern U. Technical Report 85-21 (1985).

15. Stonebraker, M. The design of the Postgres storage system. Proc. 13th VLDB Conf., Brighton, UK (Sept 1987) 289-300.