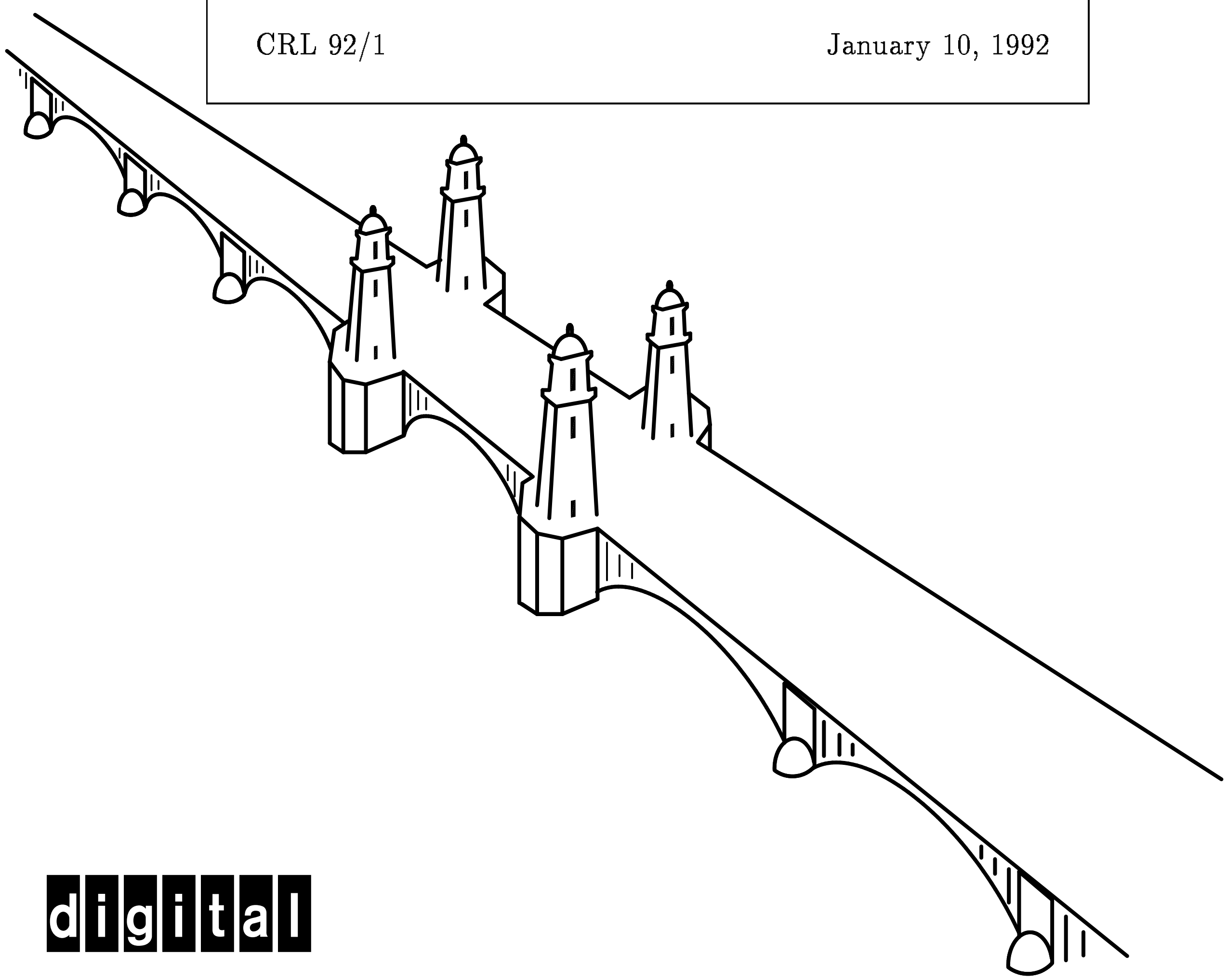
# ROLLBACK DATABASES

David Lomet Betty Salzberg

Digital Equipment Corporation
Cambridge Research Lab

CRL 92/1 January 10, 1992



**CAMBRIDGE RESEARCH LABORATORY**
Technical Report Series

# ROLLBACK DATABASES

David Lomet Betty Salzberg [1]

Digital Equipment Corporation
Cambridge Research Lab

CRL 92/1 January 10, 1992

**Abstract**

Rollback databases are multiversion databases that use transaction time to identify the versions of the data. Such databases support queries that request information "as-of" some specific past time. This report describes most of the technical aspects of supporting rollback databases. It shows how to to do the timestamping itself, including how to cope with distributed systems. Using the TSB-tree, we show how multi-version data can be efficiently indexed, efficient both in space and in query time. How one provides high concurrency while updating a TSB-tree is described next. Finally, we show how one can use a rollback database as a backup source to provide media failure recovery.

Keywords: rollback databases, multiversions, timestamping, concurrency, media recovery


---

[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

Over the last several years, multiversion databases have attracted increasing attention. Temporal databases, one form of multiversion database, have been studied with several notions of time [35]. One time of interest is the time at which a transaction executes, called "transaction time". Data is "stamped" with the time of interest, and this timestamp can be queried along with the ordinary data.

Transaction times and timestamping facilitate the realization of one particular form of multiversion database called a "rollback" database. Such databases support queries that ask to see the state of the database "as of" some particular time in the past. All updates made by a transaction to a rollback database are "stamped" with the same transaction time. This timestamp is an extra attribute of the data. The order of the timestamps must be a correct serialization of the transactions.

Current data are the most recent versions of data, i.e. the versions with the latest timestamp. Current data may continue to be updated. Hence this data is best stored on a stable read/write medium like the magnetic disk. We call this part of our data the current database.

Data that is no longer current can be stored separately from current data. This "historical" data is never updated, and hence could be stored on stable write-once, read many (WORM) optical disks [37, 20]. An inexpensive WORM medium changes dramatically the functionality/cost trade-off and makes multiversion support interesting for a large number of applications [21]. We call this part of our data the historical database.

In this report, we discuss how to realize a rollback database. The issues that we discuss are:

- TIMESTAMPING OF DATA: why timestamping at commit time is best, how timestamps are chosen in a distributed setting, and how to finish timestamping after commit (section 2).
- TSBTREE: a unified index for both current and historical data, that permits incremental migration of old data and index terms to historical the database, and the performance that is achievable (section 3).
- CONCURRENT INDEX UPDATING: integrating the TSB-tree with the concurrency and recovery system of a DBMS so as to achieve high

performance and concurrency (section 4).

- EXPLOITING HISTORY FOR BACKUP: information in an historical database can be used for backup as well as history, making this data play two roles, and hence increasing the value of historical data (section 5).

In the last section, we discuss the current state of rollback databases, how the technology we have described can be used to advance that state, and what is required to adapt this technology to the problem of supporting valid time.

# 2 Timestamping for Rollback Databases

## 2.1 The Timestamping of Data

### 2.1.1 Traditional Timestamping

Timestamps have a long history as a way of performing concurrency control [3]. Timestamping methods impose the serialization at the point when the timestamp is chosen. If this is when the transaction starts [3, 28], competing requests for the same data that are out of order result in one or the other of the competing transactions being aborted.

Timestamping concurrency control techniques have been suggested that require data to be stamped with the time of last read. This enables these protocols to determine when timestamp order has been violated for read-write conflicts, not merely write-read and write-write conflicts. This is a very serious negative, however, as it turns reads into writes, in order to post the timestamp.

Read-only transactions can use their start time to correctly serialize with other transactions against the same data by reading the most recent version with a timestamp before the start time of the transaction. This is called mixed multiversion concurrency [3].

### 2.1.2 Timestamping with Locking

Timestamping concurrency control is usually considered to be less robust and less effective than locking. It has not turned up in many system implementations, where two phase locking usually reigns supreme. Locking is well understood and has acceptable performance. A key advantage of locking is that the serialization order for a transaction is "chosen" when a transaction commits.

Our intent is to perform timestamping in the context of two phase locking. When locking is used as the primary method of concurrency control, timestamping of reads is unnecessary. Locking prevents access to such data while the transaction is active. The requirement is that the timestamps chosen for updates correctly reflect the serialization order imposed on the transactions by the locking protocol.

Note that locking is only necessary here for concurrency control involving the current versions of data. All historical versions cannot be updated, and hence no locking is needed. The timestamps are used to obtain a transaction consistent view of the data. Hence, mixed multiversion concurrency is effective for the reading of recent data, even when locking is used otherwise. Indeed, Rdb/VMS [29] provides just such a capability.

### 2.1.3 Nature of the Timestamp Attribute

Versions of data are timestamped with the time of the transaction that produced them, and are valid until a subsequent version, with a later timestamp, is entered into the database. It is not necessary to include both begin and end times with a version of data if the next version of the data is readily accessible. The TSB-tree (see the next section) is a data organization that ensures that versions are clustered so as to make determining version validity convenient using only start times.

### 2.1.4 Assigning a Correct Transaction Time

For the timestamping to correctly reflect the serialization order, we delay the choice of timestamp until the transaction is being committed. The timestamp is then chosen to be later than the timestamps of all conflicting transactions. A transaction conflicts with a preceding transaction if, for example, it reads data written by the preceding transaction or writes data read by the preceding transaction. In these cases, the transaction serializes after the preceding transaction. For a transaction $X$ at site $i$, we define $CONFLICT_i(X)$ to be a time guaranteed to be later than the time of all earlier conflicting transactions at site $i$. Hence, a timestamp associated with a transaction's updated data that is not earlier than $CONFLICT_i(X)$ will be correct.

There are several ways to choose an acceptable value for $CONFLICT_i(X)$. Probably the simplest is to use clock time at the time the transaction is to be committed as this value. This time, since it is after all previously committed transactions, is surely also after all conflicting ones as well. See [18] for a discussion of other ways of selecting a value for $CONFLICT_i(X)$.

Read-only transactions, i.e. those that are satisfiable by the reading of a recent, but not necessarily current version, may be given any time earlier than the earliest time already assigned to an active transaction. (If clock time of commit is used as a timestamp, the begin time of the read-only transaction will suffice.) This means that the read-only transaction can execute without locking because the version that it sees is immutable. All changes can only occur after its time.

## 2.2 Agreeing on a Timestamp

In a distributed system, we need to choose a timestamp for a transaction that is acceptable for all participants (cohorts) of the transaction. Choosing of a timestamp at transaction start permits all transaction cohorts to know and propagate it. However, as indicated above, this seriously impairs concurrency. Hence, we choose the transaction time at commit.

If a transaction $X$ is centralized (at one site), one might choose at commit a transaction time equal to $CONFLICT_i(X)$. By employing strict two phase locking (**2PL**), where locks are held until commit, any other transaction which conflicts with $X$ and is still active, will have a later commit time, for it must still acquire the conflicting locks.

However, if the transaction takes place in a distributed setting, where there are cohorts at several sites, some other technique is required. When one cohort has acquired all its necessary locks, another cohort may still be in the process of acquiring them at another site. To have one cohort unilaterally choose the commit time would risk timestamps that do not reflect the $CONFLICT_i(X)$ serialization requirements of all cohorts.

Hence, in distributed systems, for commit time timestamping, each cohort must have a role in deciding what that timestamp will be, so that each cohort can guarantee local agreement of serialization order and timestamp order. Further, all cohorts must be notified at commit time as to what a transaction's timestamp is. A method by which cohorts vote for a commit time has been suggested in [12, 38, 18], and is explained next. It uses the two phase commit protocol for this.

### 2.2.1 Two Phase Commit

Two phase commit (2PC) is the protocol used by cohorts of a distributed transaction to reach agreement on whether to commit or abort the transaction. We briefly review the protocol here:

1. A transaction coordinator notifies all cohorts to a distributed transaction with the PREPARE message that the transaction is now to be terminated.

2. Each cohort tries to become PREPAREd, i.e. guarantee that both the before state of the transaction and the after state are durably stored, and hence can be installed as required. Each cohort then **votes** on the disposition of the transaction by returing a VOTE message to the coordinator. If a cohort successfully becomes PREPAREd, it votes COMMIT. Any failure results in a vote to ABORT.

3. When the coordinator has received votes from all transaction cohorts, it commits the transaction if all cohorts have voted COMMIT. All other cases result in transaction abort. The coordinator sends the transaction disposition message (i.e. COMMIT or ABORT) to all cohorts.

4. When a cohort receives the transaction disposition message, it either commits by installing the after state of the transaction in the database, or aborts by installing the before state. Each cohort ACKs the disposition message upon completion, permitting the coordinator to purge its information concerning the transaction.

### 2.2.2 Voting for Transaction Time

We extend the two phase commit protocol to enable cohorts to agree on and propagate the transaction time. This can be done without extra message overhead. We augment the information conveyed on two of the 2PC messages. Each cohort informs the transaction coordinator of its requirements for transaction time on message two (the VOTE message). The coordinator then attempts to find a single time that satisfies all cohort requirements. If successful, it propagates, on message three, to all of the cohorts, both the disposition of the transaction and, if the disposition is COMMIT, the transaction time chosen. We outline one of several options suggested in [18] to accomplish this.

A cohort must determine, when it receives the PREPARE message, a time that is later than the time for any preceding transaction with which it may conflict. Enforcing that transaction time be later than the time of preceding conflicting transactions guarantees that timestamp order and serialization order agree. We assume that the local clocks at the sites are loosely synchronized with a global time source that reflects real world time, e.g. Greenwich Mean Time. Our intent is to assign times to transactions that reflect users' perceptions of when the transactions actually occurred.

A database system that acts as a transaction cohort expresses its transaction time requirement as the $EARLIEST$ time at which the transaction can be permitted to commit. This must be later than the time of any preceding conflicting transaction in that database. Further, a transaction's commit time should come no earlier than its start time, $START(X)$, to keep the transaction time synchronized with user expectations. Thus, cohort $i$ votes a time for transaction $X$ of

$EARLIEST_i(X) = \max\{CONFLICT_i(X), START(X)\}$

### 2.2.3 Picking a Common Transaction Time

The coordinator can pick a transaction time that is not earlier than the latest $EARLIEST$ time chosen for any cohort. In fact, it is desirable to choose exactly the lastest $EARLIEST$ time voted. This transaction time has the advantage of being the time that satisfies the constraints and that also is the closest such time to the times required by the cohorts. Thus the coordinator chooses a time for transaction $X$ of

$TIME(X) = \max\{EARLIEST_i(X) \mid COHORT_i(X)\}$

$CONFLICT_i(X)$ is later than the timestamps of any previously committed conflicting transaction at site $i$. A following transaction at site $i$ will thus vote an $EARLIEST_i(X)$ time that is later than this. The agreed upon time will thus be later than all conflicting transactions at all sites. This ensures that serialization order and timestamp order agree at each cohort. Since serialization order and timestamp order agree locally at each cohort, the common timestamps will agree with the global serialization order for all transactions, local and distributed.

In [18], a cohort may vote a time range $[EARLIEST_i(X), LATEST_i(X)]$. The coordinator can pick any time that is within all time intervals voted by cohorts, if such a time exists. If no such time exists, the transaction is aborted. These ranges permit the bounding of clock divergence, and make possible certain 2PC optimizations without requiring that all cohorts have completed their work prior to initiation of 2PC. For example, read locks may be released at site $i$ as of the $LATEST_i(X)$ time. The other transactions at site $i$ may then write on these records as long as their EARLIEST vote time is later than the LATEST vote time of any PREPAREd transactions at their site with which they might conflict. This is a new way that timestamps can be used in place of locks for concurrency.

## 2.3 Completing Timestamping After Commit

When transaction time is not known until after commit, as with distributed systems, it is, of course, impossible to post the transaction time with the data at the time of the update. What is needed is some persistent way of associating transaction time with a transaction identifier (TID) that IS stored with the data at the time of update. One can then replace this TID with its timestamp subsequently.

Hence, 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 finding transaction time given the TID.

So, a TID-TIME table is kept in addition in volatile memory, to make the look-up of transaction time more efficient. The TID-TIME table should contain 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 all versions (locations) that still need to be timestamped for this transaction. This last permits us to garbage collect the TID-TIME table by removing entries from it for which timestamping has been completed.

The TID-TIME table is periodically written to disk, for example in log checkpoint records, to make it stable. After a system crash, it can be reconstructed in memory from the stable version and the log records since the checkpoint. It is important then, that actions that involve timestamping of data be logged so that the list of versions awaiting timestamping can be maintained durably.

If we wait for subsequent updates to the same disk block in order to do timestamping, then the logging is already done for us. If writes to disk are logged, then timestamping can be done at this time as well, without extra log records. Since many pages from recent transactions will still be in the database buffer immediately after commit, this should take care of the majority of timestamping needed. Only timestamping that is done independently of these activities need generate separate log records. In a low priority background process, for example, data needing very old timestamps could be fetched, stamped and logged.

# 3 The Time-Split B-tree

## 3.1 Overview

To support a rollback database, one needs to be able to find versions of data as of any given transaction time. The Time-Split B-tree (TSB-tree)is a two dimensional search structure for doing this. Each node of the TSB-tree describes a rectangle in time-key space. The TSB-tree exploits the non-updatability of historical data to support its storage on a WORM medium, while keeping the current data on a write-many medium. (With other notions of time than transaction time, historical data may be subject to update. Supporting these forms of temporal database require a general purpose multi-attribute index, e.g. the hB-tree [22, 7]. We briefly expand on this idea in section 6.

What we describe initially is a TSB-tree used as a unified index structure. However, it is possible to adapt the TSB-tree to support access to historical data when current data is stored in essentially any format. (See section 3.4) The TSB-tree search algorithm and its split algorithms for index and data nodes are described below.

## 3.2 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.

## 3.3 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 section 3.5, and treated in depth in [21]. Here we describe only the mechanics of the splitting process. A sequence of splits is illustrated in Figure 2.

### 3.3.1 Time Splits

If a node is split by time $T$, all entries with time less than $T$ go in the history node, which we only write once so as to permit its storage on a WORM device. 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.

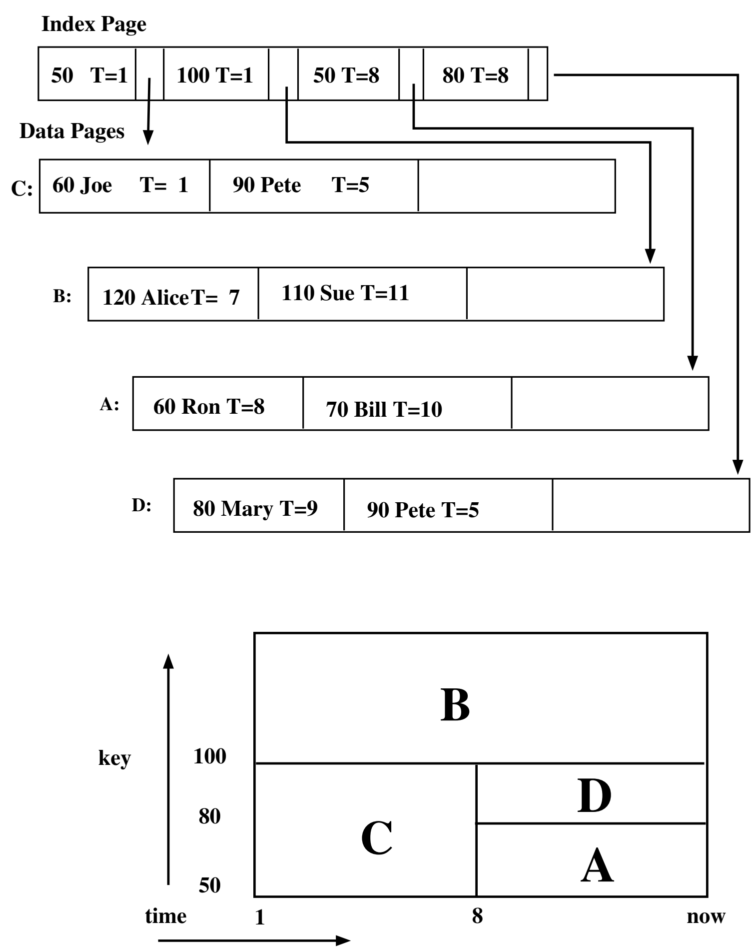



Figure 1: Each index entry is the lower key and earliest time for the time-space rectangle spanned by its child. To find a record with key 60, valid at time 7, ignore all entries in the index with time greater than 7. Find the largest key $\leq$ 60 with the largest time. This is the index entry (50 T=1). 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).

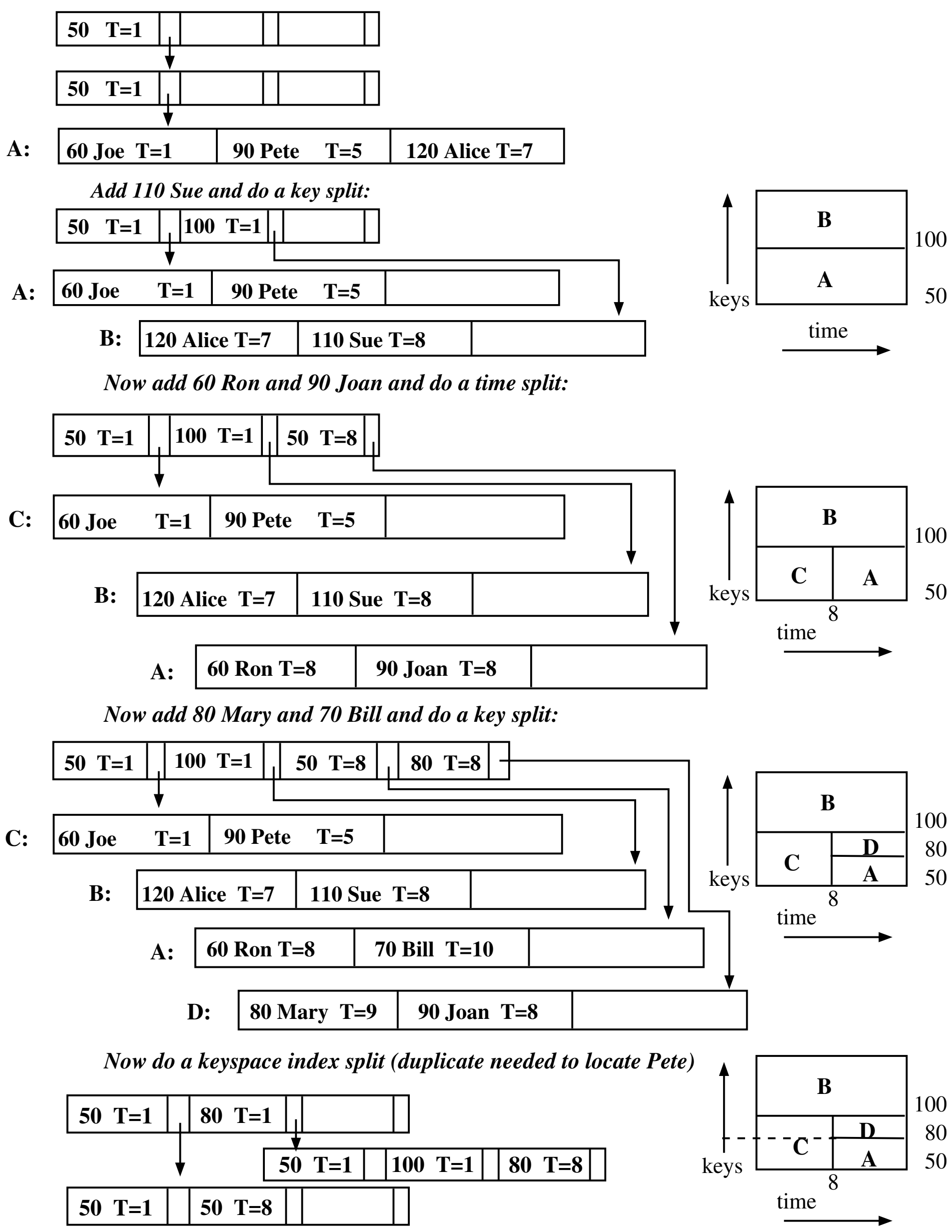



Figure 2: Illustrated is a sequence of splits ending in a key split for an index node. This key split requires that index entries for node C be present in both resulting nodes.

Any time after the begin time for a data node can be used for a data node time split. For an index node time split, the split time cannot be later than the begin time for any current child node so as to insure that history index nodes do not refer to current nodes. Hence, index entries are posted, at the time their referenced node is split, to only one parent index node. History nodes, which do not split, may have several parents. Current nodes have only one parent.

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, at least for historical data. This is important for two reasons:

1. Choosing an appropriate split time requires knowledge of the times.

2. We do not update history nodes (they may be on WORM devices) so this is our last chance to timestamp the history data.

### 3.3.2 Key Splits

A data record corresponds to a line segment (one key value and a time interval) in the key-time space. If a data node is split by key, the split is exactly the same as in a $B^{link}$-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.

## 3.4 Using TSB-trees for Any Rollback Databases

Not all database systems, or all tables within a database system, will necessarily be stored using a B-tree like index. Regardless of the way that current data is organized, the TSB-tree can be used to index the historical data so long as there exists a unique and durable record identifier (**RID**). The key used in this TSB-tree is this record identifier. We assume that all versions of the data, current and historical, are timestamped. The TSB-tree locates history versions of records via their record identifiers and the time of interest. The current database is accessed directly via the record identifier, using whatever organization is desired for this data.

Each update or delete (but not insert) in the current database generates a new historical version, which is the version being updated or deleted. The cost of posting old versions of records to the historical database will vary depending on the organization of the current data, and how well it meshes with the TSB-tree organization. Particularly important is whether the TSB-tree can exploit pages of the current organization as TSB-tree nodes for current data.

### 3.4.1 Current Pages as TSB-tree Nodes

Frequently, a database system stores the bulk of its data in an entry-ordered file. The RIDs here 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.

Pinned records are one example of an organization where it is possible to align the TSB-tree organization so as to permit batch updating of the history database via node splitting. We assume that current pages are not filled entirely as new records are inserted, but rather some space is left in each page. Until the page becomes full, the old versions remain in the original page.

When a current page overflows, a TSB-tree time split is made. Key splits are never made. This time split removes all versions with start times earlier than the split time from the current database, and posts them to the historical database. These historical versions cluster on PID. Each new history node consists of updates to previous versions with the same PID. Some of the recent but non-current versions will only have copies in the current database, but the TSB-tree index can be used to find records with a given PID valid at a given time, as usual, whether it is in the current or the historical database.

### 3.4.2 One Record at a Time Posting

Not all current organizations might be so conveniently handled. But even ones where the primary key is not clustered in a page structured organization, it is still possible to use the TSB-tree to store historical versions. Such a history-only TSB-tree is built gradually, one version at a time, as current data is updated. So long as current data is timestamped, enough information is present to perform "as of" queries, even those involving current data, though with a need to sometimes search both structures, because we have not yet bound the end time for the most recent historical versions.

We can bound the end time for historical versions by writing the timestamp for the current version to the historical database whenever we post a new historical entry. In fact, if we post this timestamp, this removes the need to carry timestamps with the current data. This permits us to support historical databases for existing data without any changes at all, even for non-timestamped data. The TSB-tree should be a better index organization for this than is the R-tree [11] that is used for this purpose in POSTGRES [37] because of its ability to keep contiguous versions of data together in a small number of data nodes, thus reducing the accesses needed to retrieve a time slice.

## 3.5 Search Performance and Space Utilization

There are two obvious dangers in constructing temporal indexes:

1. One can make repeated time slices, risking using $O(CS)$ space where $C$ is the number of current records and S is the number of slices. (S might be as large as $M$, the number of different record versions, if a new time slice is generated for every update.)

2. One can have some sort of linear search algorithm for finding a record with a given key and time, as one would have in merely logging all

updates to a temporal database. This is $O(M)$ search.

Clearly, what is desired is to use space that is proportional to the number of versions present, i.e. O(M), and search time for a record that is of the same order that is obtained from traditional index trees, i.e. O(log(M)). Further, we would like a search time for N records in a time slice to be of O(N). That is, one needs a constant time to retrieve each of the records, regardless of the time slice chosen. Finally, we would like to find all previous versions of any given record in time linear in the number of previous versions.

The time and space bounds are self-evident for the TSB-tree once we show that the number of redundant records is not too large. We did this in [21] where we both analyzed and simulated the performance of the Time-Split B-tree, providing asymptotic performance results under two assumptions:

1. **Uniform Growth Assumption**: A new record is equally likely to be between any two existing records. Hence, the probability that a record is inserted into a node is proportional to the number of records with unique keys in the node.

2. **Equal Probability Assumption**: Each record with a unique key is equally likely to be updated.

The two assumptions above do not imply that our results apply only to uniformly distributed keys. That is not the case, even though our simulation employed uniformly distributed keys. When analyzing index-based access methods, the purpose of a uniform distribution is to realize the uniform growth assumption. This is a standard procedure.

We used a form of fringe analysis [5, 1]. It involved computing a closure on node probabilities and produced asymptotic performance results directly. The analysis was confirmed by a detailed simulation entailing multiple trials, each trial involving the addition of 50,000 records. Both the analysis and the simulation were parameterized in terms of the percentage of updates versus insertions.

For the splitting policy we called "Independent Key Split," (most like what is described above), we made two decisions:

1. Always use the time of LAST UPDATE (i.e. not including inserts) as the splitting time.

2. Perform a key split whenever two thirds or more of the splitting node consists of current data. This is called the **key splitting threshold**.

Among the quantities measured were

$U_{svc}$: This is single version current utilization, the ratio of space consumed by the most recent versions of records to the space used to hold the current database. $U_{svc}$ captures the cost of retaining multiple versions of data [records] instead of performing update-in-place. In particular, it tells us how well our split policy is working in terms of minimizing the space for the current database. Since the current database is stored on a write-many medium, this is quite important. Space on the write-many medium may be a factor of three to ten times more expensive than space for the historical database on the write-once medium.

$F_{red}$: This is the fraction of redundancy, the number of redundant records divided by the total number of records. (Recall that redundancy is introduced by the need to copy versions of records that persist across the times used for time splitting.) This tells us how much redundancy is introduced by various choices for the details of node splitting.

$U_{mv}$: This is multiple version utilization, the ratio of the space consumed by all versions of data to the space that used to hold all versions of the data, i.e. the total space. $U_{mv}$, measures how effectively the TSB-tree, together with the particular split policy, is in supporting multiversion data. This is the measure that can be used to compare TSB-trees with other multiversion approaches. It reflects the cost of maintaining the integrated index to the entire collection of versions, and the cost of storing redundant copies of the versions so as to support "as-of" queries.

The results we obtained for the two extremes of insertion versus updates can be explained as follows (intermediate ratios of insertion versus update resulted in results that were intermediate between these end point results):

**All Insertions:** All update activity consists of adding new records to the database, not updating prior versions. The TSB-Tree behaves as a regular B-tree behaves with respect to $U_{svc}$. In the limit, as node size $b$ increases, $U_{svc} = \ln 2 = 0.693$. There are no redundant records so $F_{red}$ is zero. Finally, $U_{mv} = U_{svc}$ since there are no historical versions.

**All [99%] Updates** : For almost all updates, the splitting performed is almost entirely time splits. The maximum current utilization, $U_{svc-max}$, becomes the key splitting threshold. We set this to .666. Hence, $U_{svc}$, when averaged over all nodes, at these high update rates, is near $U_{svc-max} \ln 2 = 0.666 \ln 2 = 0.46$. This is the penalty for keeping old versions around. At each time split, no more than $U_{svc-max}$ of data is current and hence becomes redundant. The records added between time splits is not less than $1 - U_{svc-max}$. Thus, $F_{red}$ is bounded by $U_{svc-max}/(1 - U_{svc-max})$. $U_{mv}$ trails off, but does not get below 50%. This means the space occupied by data pages in the TSB-tree is at worst twice that needed for an organization with no redundancy and no empty space in data pages. It needs only at worst 50% more space than a standard $B^+$-tree which has on average 30% empty space in data pages. Hence, TSB-tree space is linear in M, i.e. O(M). This will not be true of any organization which makes an arbitrarily large number of time slices.

Our paper [21] describes several split policies, and presents results for these and other quantities with various update percents. In all the policies that we analysized, the fraction of redundant records was bounded. This means that space for all policies is O(M), that the index tree height is O(log(M)), and hence that is the cost of a random probe. And finally, the cost of retrieving N records in a time slice or N versions of a particular key value are both O(N) because of the clustering of records in key-time space. Only multi-attribute tree indexes with bounded redundancy can give results that are this good.

# 4 Concurrency Control for TSB-trees

## 4.1 Π-trees

Maintaining concurrent access to the TSB-tree while it undergoes structure changes induced by node splitting is important. We use a technique that is a generalization of the $B^{link}$-tree technique [14, 31, 32] which exploits what we call the Π-tree[23]. 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.

Informally, a Π-tree is a balanced tree, and we measure the level of a node by the number of child edges on any path between the node and a leaf node. More precisely, however, a Π-tree is a rooted DAG because, like the $B^{link}$-tree, nodes have edges to sibling nodes as well as child nodes. All these terms are defined below.

### 4.1.1 Within One Level

Each node is **responsible for** a specific part of the key space, and it retains that responsibility for as long as it is allocated. A node can meet its space responsibility in two ways. It can **directly contain** entries (data or index terms) for the space. Alternatively, it can **delegate** responsibility for part of the space to a **sibling node**.

A node delegates space to a new sibling node during a node split. A **sibling term** describes a key space for which a sibling node is responsible and includes a **side pointer** to the sibling. A node containing a sibling term is called the **containing node** and the sibling node to which it refers is called the **contained node**.

Any node except the root can contain sibling terms to contained nodes. Further, a Π-tree node is not constrained to have only a single sibling, but may have several. A **level** of the Π-tree is a maximal connected subgraph of nodes and side pointer edges. Each level of the Π-tree is responsible for the entire key space. The first node at each level is responsible for the whole space, i.e. it is the containing node for the whole key space.

### 4.1.2 Multiple Levels

The Π-tree is split from the bottom, like the B-tree. **Data nodes** (leaves) are at level 0. Data nodes contain only data records and/or sibling terms. As the Π-tree grows in height via splitting of a root, new levels are formed.

A split is normally described by an index term. Each **index term**, when posted, includes a **child pointer** to a **child node** and a description of a key space for which the child node is responsible. A node containing the index term for a child node is called a **parent** node. Hence, a parent node indicates the containment ordering of its children based on the spaces for which the children indexed are responsible.

A parent node directly contains the space for which it is responsible and which it has not delegated, exactly as with a data node. Parent nodes are **index nodes** which contain only index terms and/or sibling terms. Parents nodes are at a level one higher than their children. The same child can be referred to by more than one parent. This happens when the boundary of a parent split cuts across a child boundary. Then the union of the spaces that children nodes directly contain may be larger than the space the index node directly contains.

## 4.2 The TSB-tree as a Π-tree

A TSB-tree indexes records both by key and by time, and we can split by either of these attributes. We take advantage of the property that historical nodes (nodes created by a split in the time dimension) never split again. This implies that the historical nodes have constant boundaries and that key space is refined over time. History sibling pointers permit the current node directly containing a key space to access history nodes that contain the previous versions of records in that space.

In Figure 3, the region covered by a current node after a number of splits is in the lower right hand corner of the key space it started with. A time split produces a new (historical) node with the original node directly containing the more recent time. A history sibling pointer in the current node refers to the history node. The new history node contains a copy of prior history sibling pointers.

A key split produces a new (current) node with the original node directly containing the lower part of the key space. A key sibling pointer in the current node refers to the new current node containing the higher part of the key space. The new node will contain not only records with the appropriate keys, but also a copy of the history sibling pointer. It makes the new current node responsible for not merely its current key space, but for the entire history of this key space. This split duplicates the history sibling pointer.

These sibling pointers 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

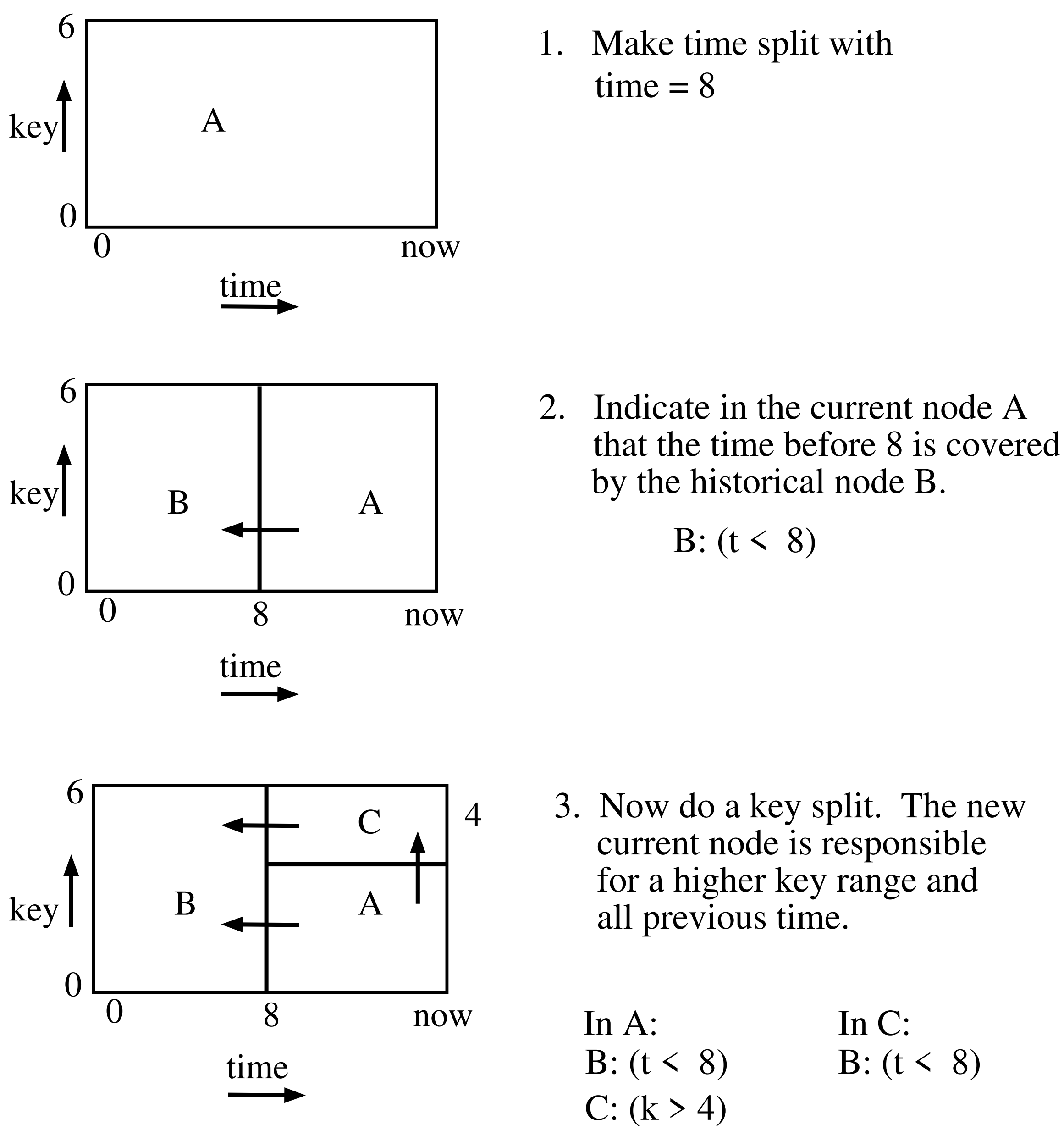



Figure 3: In the Time-Split B-tree, as a Π-tree, new current nodes contain copies of old history node pointers and old key pointers. New historic nodes contain copies of old history pointers. Current nodes are responsible for all previous time through their historical (side) pointers and all higher key ranges through their key (side) 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.

## 4.3 Multiple Atomic Actions for Tree Growth

Π-tree structure changes consist of a **sequence** of atomic actions [15]. These actions are serializable and are guaranteed to have the all or nothing property by the recovery method. Searchers can see the intermediate states of the Π-tree that exist between these atomic actions. Hence, complete structural changes are not serializable.

We define separate actions for performing updates at each level of the tree. Update actions on non-leaf nodes can be separate from any transaction whose update triggers a structure change. Only node-splitting at the leaves of a tree may need to be within an updating transaction. Even in this case, only for some recovery systems do locks on the split nodes need to be kept to the end of the transaction.

Non-leaf nodes in our concurrency algorithm do not have database locks (locks managed by the lock manager). Instead, index nodes have simple short-term semaphores we refer to (as in [27]) as **latches**. To use latches, the programmer assumes responsibility for ensuring that deadlock does not occur. This permits the lock manager to be bypassed.

Latches can have S mode (share), X mode (exclusive) or U mode (update). Exclusive mode latches are not compatible with any other latches. Share mode latches are compatible with other share mode latches and with update mode latches. Update mode latches are not compatible with other update latches. The reason for update latches is to allow more concurrency than exclusive latches while avoiding the deadlock problems of converting share latches to exclusive latches. Obtaining an update latch signals a likely later conversion to an exclusive latch.

When executing structure changes as multiple atomic actions, it is an important performance consideration to utilize information acquired in immediately previous atomic actions. We do this by remembering the path that was traversed by an earlier atomic action while searching for a particular key. This makes our sequence of atomic actions very efficient, not requiring much more overhead than that which would be needed if all structure changes were done in one action. We do not have to search for the parent of a node which has split in order to post the index term in a separate action. Should the posting go further up the tree, due to repeated splits, the rest of the path can be used.

## 4.4 Completing Structure Changes

There is a window between the time a node splits in one atomic action and the index term describing it is posted in another. Between these atomic actions, a $\Pi$-tree is said to be in an intermediate state. We try to complete structure changes because intermediate states may result in non-optimal search.

There are at least two reasons why we "lose track" of which structure changes need completion, and hence need an independent way of re-scheduling them.

1. A system crash may interrupt a structure change after some of its atomic actions have been executed, but not all. The key to this is to detect the intermediate states during normal processing, and then schedule atomic actions that remove them. Hence, database crash recovery does not need to know about interrupted structure changes.

2. We only schedule the posting of an index term to a single parent. We rely on subsequent detection of intermediate states to complete multi-parent structure changes. This avoids the considerable complexity of trying to post index terms to all parents, either atomically or via the scheduling of multiple atomic actions.

Structure changes are detected as being incomplete by a tree traversal that includes following a side pointer. At this time, we schedule an atomic action to post the index term. Several tree traversals may follow the same side pointer, and hence try to post the index term multiple times. These are acceptable because the state of the tree is **testable**. Before posting the index term, we test that the posting has not already been done and still needs to be done.

## 4.5 An Example Structure Change Action

To illustrate the detailed steps of an atomic action, we treat the case of posting index terms in the TSB-tree. A previous atomic action has split a current (index or data) node and scheduled index posting. If a system failure has intervened, index posting may have been scheduled by a searcher traversing a sibling pointer.

The arguments for the index term posting operation are: REMEMBERED PATH including the address of the PARENT node where the index term is expected to be posted and the node ADDRESS, TIME INTERVAL and KEY INTERVAL of the term to be posted.

Index term posting performs the following steps:

1. **Search:** The search starts with the PARENT. The PARENT is U-latched, and checked to see if the KEY INTERVAL intersects the PARENT'S directly contained space. If not, one traverses key space side pointers releasing U-latches and acquiring new ones until the correct NODE is U-latched. If this current index node does not intersect the TIME INTERVAL, the index-posting action is terminated as historical nodes are never updated. (This happens only when the posting of the index term occurs after a more recent time-split index term in this key range has already been posted and the index node has been time split. )

2. **Verify Split:**. If the index term has already been posted, the action is terminated.

3. **Space Test:** Test NODE for sufficient space to accommodate the update. If sufficient, proceed to **Update Node**

   Otherwise, split NODE: The space management information is X-latched and a new node is allocated. The time and key space directly contained by the current node is divided, such that the new node becomes responsible for a subspace of the time and key space. (This is either a time split or a key split.)

   A sibling term that references the new node is placed in NODE. The change to NODE and the creation of the new node are logged. If NODE is not the root, an index term is generated containing the new node's address as a child pointer. (At the end of the action, when all latches are released, an index posting operation using the rest of the REMEMBERED PATH is scheduled for the parent of NODE.)

If NODE is the root, a second node is allocated. NODE's contents are removed from the root and put into this new node. A pair of index terms is generated that describe the two new nodes and they are posted to the root. These changes are logged.

Then we check which resulting node has a directly contained space that intersects the KEY and TIME INTERVALS and make that NODE. (It is possible, but unlikely, that both nodes intersect the KEY and TIME INTERVALS. Both could be updated, or one could be, with the other triggered later by a traversal.) This can require descending one more level in the Π-tree should NODE have been the root where the split causes the tree to increase in height. Release the X-latch on the other node, but retain the X-latch on NODE. Repeat this **Space Test** step.

4. **Update NODE:** Post the index term in NODE. Post a log record describing the update to the log. Release all latches still held by the action.

# 5 Exploiting History for Database Backup

## 5.1 Overview

### 5.1.1 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. 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. The part of the log containing actions that may need to be applied is bounded by the **backup safe point**, identifying the earliest possibly unapplied action, and the end of the log.

The TSB-tree 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.

We show 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.

### 5.1.2 Incremental 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. 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.

When a backup occurs, all data nodes whose NCV bits are one are read and backed up. Other data nodes are not accessed. The NCV is updated to reflect this node backup. Conceptually, all updates before we visit a node during backup are before the backup and are indicated in an "old" NCV. Nodes updated after our visit are after the backup and these updates are indicated in a "new" NCV that determines what is backed up during the next cycle.

## 5.2 Forcing Current Data to History

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.

### 5.2.1 Entire Current Node as History Node

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(see 5.2.3). 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, backup could 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 what follows, we assume that current nodes are not changed, even if they need timestamping.

### 5.2.2 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. 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.

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.

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. When START is earlier than the start time in the index term for the node (which we shall call **ISTART**), SI versions may be present.

### 5.2.3 Choosing a Split Time

The choice of a split time determines ISTART. 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 choice, which is like the choice in WOB-trees [4], 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 (ISTART) the earliest prepare time of any such record.

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. Two ways of dealing with this might be to

1. make sure that the backup time is older than the oldest vote time for the system, or

2. mark in-doubt data in the backup history node as "possibly committed" with their vote time as a time stamp when the split time for the node is after the vote time. More recent nodes in the same key range will

contain the commit time, if there is one, and these are searched in AS-OF queries.

## 5.3 Splitting Index Nodes

### 5.3.1 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.

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. Like historical data nodes, we do not want historical index nodes to be updated. Since backup produces history nodes for the entire TSB-tree, the split time for index nodes will be after (or at) the time at which the backup started.

### 5.3.2 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. 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 sometimes only part of the child's space may lie inside 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. Note that with the TSB-tree, index terms for the current database are never covered.

### 5.3.3 Impact of Covering

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 (copying) of index nodes should frequently result in index-term covering at the next higher level in the TSB-tree. 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.

The backup process guarantees to advance ISTART for every current node. Even when no backup of a data node is required (NCV bit is zero), ISTART for the current node is advanced. See Figure 4. ISTART 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.

### 5.3.4 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. Further, a TSB-tree 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).

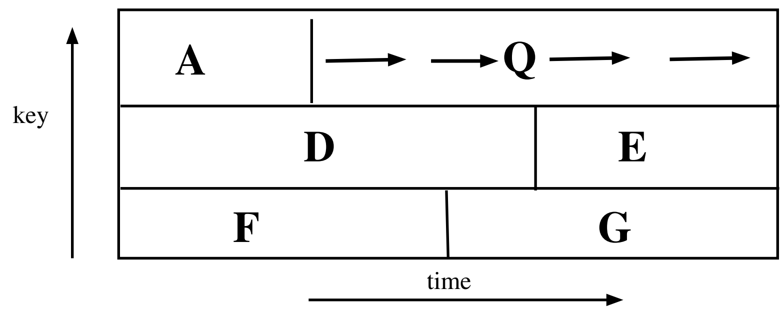



Figure 4: The current child node Q has not been changed since the last backup. All Q's records have timestamps $\leq$ the start time in its index entry. The records in A, valid at the previous backup time (previous start time for Q), are still valid at the new backup time. Thus the start time in Q's 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.

## 5.4 The Backup Process

The backup process is one of installing data current as of the time of the backup into a stable version of the database that is separate from the current database. [Note that we do not provide backup for historical data.] We do this by time-splitting nodes that have been updated since the prior backup. The **NCV** is maintained to ensure that only those data nodes which have changed since the last backup are written to the history database in the current backup.

We "sweep" through the current database in TSB-tree "post-order," backing up children before their parents. 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.

Normal database activity is concurrent with the backup process, so that what is described constitutes a "fuzzy" backup for media failure [3]. 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, non-full nodes may also be time-split to ensure 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 advance the backup safe point. It ensures that all changes to the database that precede the safe point are in the backup. The backup safe point is stored in the BSB.

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.

**Writing the History Node:** The current node is copied to form the history node, which is force written to the stable history database. Backup of a changed current data node can be done atomically. The node is share-latched by the buffer manager to ensure 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.

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 here work correctly for the backup of index nodes.

**Logging the Split:** 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 describing the posting of the index term is stable. The write-ahead log protocol is observed to ensure that the log record describing the update of the index node is stable before the index node itself is written.

A single log record describes the update to the parent index node. 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.

**Writing of the Index Node:** The parent index node of the split node is updated to make the new history node accessible. This updated index node is then subject to backup by writing it to the history database, as described above.

In [24], we explain how these steps are accomplished and why the ordering is important. In addition, we describe in detail how to ensure that backup is efficient and can continue across system crashes.

## 5.5 Media Recovery

### 5.5.1 Principles

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. These backups are accessible via the history root referenced in the BSB. After this restore step is completed, failed nodes that existed at the time of the last backup all have been restored with valid past states.

The log is then scanned, starting at the backup safe point that is stored in the BSB. The 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.

Each of the restored nodes can be rolled forward based solely on their log records and their restored state. The log also contains 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

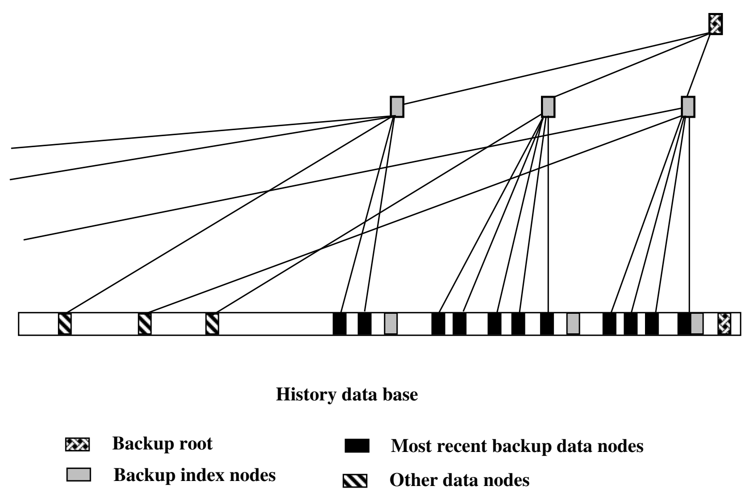



Figure 5: Many of the nodes needed for media recovery are clustered together in the area of the most recent backup. This includes the entire backup index and those data nodes that were in the last backup.

redo scan starts at the backup safe point, as opposed to the crash recovery safe point.

### 5.5.2 The Restoration Process

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.

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 illustrated in Figure 5.

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 low disk arm movement.

We describe in [24] how the history database is accessed to deal with different types of failures. The important issue that we deal with there 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.

# 6 Discussion

## 6.1 Current State

Although many proposals for temporal indexing have been made [20, 28, 35, 37], we know of none have been implemented in commercially available systems. The commercial versioning of which we are aware supports the ability to read a recent version without setting locks, even in the presence of ongoing update activity. This functionality is provided, e.g., in the snapshot feature of DEC's Rdb/VMS [29]. A snapshot is a transaction-consistent recent version. Snapshot versions are created only as needed for these transactions, and hence there is no complete history of versions. Neither are these versions retained any longer than needed by these specific transactions. Thus, there is no general ability to either index historical versions or to query them with "as of" queries.

The most ambitious research effort in implementing a system to support rollback databases is the POSTGRES system [37]. POSTGRES uses the R-tree [11], a general purpose multi-attribute index tree organization to store historical data by the time interval in which the data is valid, using transaction time. The R-tree is only used to reference the historical data, not the current data. A separate (background) "vacuuming" process migrates historical versions from the current database to the historical database.

While POSTGRES provides the functional capability of rollback databases, the design of its storage system limits the performance and flexibility of this capability. As indicated above, there is no unified index for both current and historical databases, meaning that some "as of" queries will need to access both databases, in separate searches. Because time splitting is not performed, hence avoiding redundancy in the vacuuming process, time slices are spread over more nodes. Without redundancy, the history database cannot be used as a backup in media recovery. Without time splitting (which exploits clipping), spliting by both start and end time is needed in order to partition the search space, requiring two stamps on each version (See also the discussion below on valid time.).

## 6.2 Providing More Capability

### 6.2.1 The Needs

In [34], the capabilities needed for next generation database systems were surveyed. Multi-versioning was one such capability. Another was the management of data on tertiary storage, and the ability to move data between levels of the storage hierarchy so as to be responsive to cost and performance trade-offs. These capabilities are needed to satisfy the requirements of large commercial organizations that wish, e.g., to find financial trends in their business records, doing "data mining" in which enormous collections of historical data are accessed. These capabilities are not yet available.

The same report also remarks that traditional algorithms for media recovery for large (say one terabyte) databases take days when restoration is done from a non-indexed archived dump, which is impossibly long for most customers.

### 6.2.2 The TSB-tree

We have presented a temporal indexing structure, the TSB-tree, that efficiently supports the "as of" queries of a database that uses transaction time. The TSB-tree groups together data present in the database state at the same or nearby times. Since some data can be present for long durations, this requires some redundancy. However, as we have indicated, the amount of redundancy is both bounded and reasonable. In addition, since all redundant information is historical, and therefore never updated, problems normally introduced by having to update multiple copies are avoided.

Along the way, we have discussed how the TSB-tree incrementally moves data from on-line disks to an archive. This attacks the problem of managing tertiary storage. That is, current data is on magnetic disk, while historical data, accessed much less frequently, and written only once, is stored on WORM disk.

Citing our prior work, we have also shown how, using a general index concurrency method, the TSB-tree can be maintained while supporting high concurrency. Such high concurrency is essential when multiversioning is introduced into general purpose database systems.

Finally, we have shown how to use the TSB-tree for media backup as well as for temporal queries. Using the TSB-tree as an indexed backup should make this recovery more efficient, particularly if only a portion of the database needs to be recovered. The TSB-tree historical data is maintained on-line, thus avoiding operator intervention. Its indexed nature makes it ideal when coping with partial media failures.

### 6.2.3 Dealing with Valid Time

The TSB-tree has been designed for efficient support of transaction time based rollback databases. But it can be used simultaneously for valid time by adding valid time begin and end fields to each record and by using any point-based multiattribute search structure such as the hB-tree [22, 7] as a secondary index.

It is argued in [19] that mapping intervals to their endpoints is efficient for spatial search. The short intervals are clustered together if they are close and the longer intervals are clustered with other longer intervals with approximately similar endpoints. So if an index is constructed for keys and valid time intervals using a point-based multiattribute structure, the references to versions valid in a time slice should exhibit reasonable clustering in the index, though not to the same extent as that provided by TSB-trees exploiting limited redundancy.

A secondary index maps a secondary key (in this case, key and valid time) to a primary key (in this case, key and transaction time). That is, given a key and a valid time $T$, one uses the index to find a transaction time for the record version valid at $T$ and with the given key. Since valid time is frequently close to transaction time, there should be some locality in the TSB-tree for valid time slices, even though the TSB-tree is organized for transaction time slices.

Should only valid time and not transaction time be needed, the multi-attribute index can be used as a primary index. Since using endpoints of intervals provides some clustering, there will be some locality for valid time slices. However, it is not clear how to gracefully separate such a structure into a current and a historical component so as to exploit WORM media since prior valid time versions may be updatable. A valid time database is not appropriate for media recovery as there is no redundancy and hence no way to provide for backups.

# References

[1] Baeza-Yates, R. and Larson, P.A. Performance of $B^+$-trees with Partial Expansions. *IEEE Trans. on Knowledge and Data Engineering* 1,2 (June 1989) pp. 258-257.

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

[3] Bernstein, P., Hadzilacos, V. and Goodman, N. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Co., Reading MA (1987).

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

[5] Eisenbarth, B., Ziviani, N., Gonnet, G., Mehlhorn, K. and Wood, D. The Theory of Fringe Analysis and Its Application to 2-3 Trees and B-Trees. *Inform. Contr.* 55, (1982) 125-174.

[6] Eswaren, K., Gray, J., Lorie, R. and Traiger, I. On the notions of consistency and predicate locks in a database system. *Communications of the ACM* 19,11 (Nov 1976) 624-633.

[7] Evangelidis, G., Lomet, D. and Salzberg, B. Modifications of the hB-tree for node consolidation and concurrency. (in preparation)

[8] Gray, J. Notes on Database Operating Systems. in "Operating Systems: an Advanced Course", *Lecture Notes in Computer Science* 60, Springer-Verlag, (1978) 393-481. also in IBM Research Report RJ2188 (Feb. 1978).

[9] Gray, J.N., Lorie, R. A., Putzulo, G. R., and Traiger, I. L., Granularity of locks and degrees of consistency in a shared data base. *IFIP Working Conf on Modeling of Data Base Management Systems* (1976) 1-29.

[10] Gray, J. and Reuter, A. *Transaction Processing: Techniques and Concepts*, Morgan Kaufmann (in preparation).

[11] Guttman, A. R-trees: a dynamic index structure for spatial searching. *Proc. ACM SIGMOD Conf.*, Boston, MA (May 1984) 47-57.

[12] Herlihy, M. Optimistic Concurrency Control for Abstract Data Types. *Proc. Symp. on Principles of Distributed Computing*, (1986) 206-217.

[13] Lampson, B. and Sturgis, H. Crash Recovery in a Distributed System. Tech Report (1976) Xerox PARC, Palo Alto CA.

[14] Lehman, P., Yao, S.B. Efficient locking for concurrent operations on B-trees. *ACM Trans. on Database Systems* l6,4 (Dec 1981) 650-670.

[15] Lomet, D. B. Process structuring, synchronization, and recovery using atomic actions. *Proc. ACM Conf. on Language Design for Reliable Software*, SIGPLAN Notices 12,3 (Mar 1977) 128-137.

[16] Lomet, D.B. Subsystems of processes with deadlock avoidance. *IEEE Trans. on Software Engineering* SE-6,3 (May 1980) 297-304.

[17] Lomet, D.B. Recovery for shared disk systems using multiple redo logs. Digital Equipment Corp. Tech Report CRL90/4 (Oct 1990) Cambridge Research Lab, Cambridge, MA.

[18] Lomet, D. Consistent Timestamping for Transactions in Distributed Systems. Digital Equipment Corp. Tech Report CRL90/3 (Sept. 1990) Cambridge Research Lab, Cambridge, MA.

[19] Lomet, D. Grow and Post Trees: Role, Techniques, and Future Potential. in "Advances in Spatial Databases", *Lecture Notes in Computer Science* 525 Springer-Verlag (1991) 183-206.

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

[21] Lomet, D. and Salzberg, B. The Performance of a Multiversion Access Method. *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ (May 1990) 353-363.

[22] Lomet, D. and Salzberg, B. The hB-tree: a multiattribute indexing method with good guaranteed performance. *ACM Trans. on Database Systems* 15,4 (Dec 1990) 625-658.

[23] Lomet, D. and Salzberg, B. Concurrency and Recovery for Index Trees. Digital Equipment Corp. Tech Report CRL91/8 (Aug 1991) Cambridge Research Lab, Cambridge, MA.

[24] Lomet, D. and Salzberg, B. Media Recovery with Time-Split B-trees. Digital Equipment Corp. Tech Report CRL 91/9 (Nov 1991) Cambridge Research Lab, Cambridge, MA.

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

[26] Mohan, C., Lindsay, B. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. *Proc. Symp. on Principles of Distributed Computing*, Montreal, CA (Aug. 1983)

[27] Mohan, C. and Levine, F. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. IBM Research Report RJ 6846, (Aug 1989) Almaden Research Center, San Jose, CA.

[28] Reed, D. Implementing Atomic Actions. *Proc. ACM Symp. on Operating System Principles*(1979) and in *ACM Trans. on Computer Systems* 1,1 (Feb. 1983) 3-23.

[29] Rengarajan, T., Spiro, P., and Wright, W. High availability mechanisms of VAX DBMS software. *Digital Technical Journal* 8, (Feb. 1989), 88-98.

[30] Rosenblum, M. and Ousterhout, J. The Design and Implementation of a Log-Structured File System. *Proc. ACM Symp. on Operating System Principles*, Asilomar, CA (1991) 1-15. also in *ACM Trans. on Computer Systems* (to appear).

[31] Sagiv, Y. Concurrent operations on B* trees with overtaking. *J Computer and System Sciences* 33,2 (1986) 275-296.

[32] Salzberg, B. Restructuring the Lehman-Yao tree. Tech Report TR BS-85-21 (1985) Northeastern University, Boston, MA.

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

[34] Silberschatz, A., Stonebraker, M., Ullman, J. (eds) Database Systems: Achievements and Opportunities. *Communications of the ACM* 34,10 (Oct. 1991) 110-120.

[35] Snodgrass, R. and Ahn, I. A Taxonomy of Time in Databases. *Proc. ACM SIGMOD Conf.*, Austin, TX (May 1985), 236-246.

[36] Srinivasan, V. and Carey, M. Performance of B-tree concurrency control algorithms. *Proc. ACM SIGMOD Conf.*, Denver, CO (May 1991) 416-425.

[37] Stonebraker, M. The Design of the POSTGRES Storage System. *Proc. Very Large Databases Conf.*, Brighton, UK (Sept. 1987), 289-300.

[38] Weihl, W. Distributed Version Management for Read-Only Actions. *IEEE Trans. on Software Engineering* SE-13,1 (Jan. 1987), 55-64.