ACCESS METHODS FOR MULTIVERSION DATA

by

David Lomet†
Digital Equipment Corporation
110 Spitbrook Road
Nashua, New Hampshire, 03062

Betty Salzberg *
College of Computer Science
Northeastern University
Boston, Massachusetts 02115

ABSTRACT

We present an access method designed to provide a single integrated index structure for a versioned timestamped database with a non-deletion policy. Historical data (superceded versions) is stored separately from current data. Our access method is called the *Time-Split B-tree.* It is an index structure based on Malcolm Easton's Write Once B-tree.

The Write Once B-tree was developed for data stored entirely on a Write-Once Read-Many or *WORM* optical disk. The Time-Split B-tree differs from the Write Once B-tree in the following ways:

- Current data *must* be stored on an *erasable* random-access device.
- Historical data *may* be stored on *any* random-access device, including WORMs, erasable optical disks, and magnetic disks. The point is to use a faster and more expensive device for the current data and a slower cheaper device for the historical data.
- The splitting policies have been changed to reduce redundancy in the structure—the option of pure key splits as in B^+-trees and a choice of split times for time-based splits enable this performance enhancement.
- When data is migrated from the current to the historical database, it is consolidated and appended to the end of the historical database, allowing for high space utilization in WORM disk sectors.

† This research was done at the Wang Institute of Graduate Studies, Tyngsboro, Massachusetts

* This research was partially supported by NSF Research Grant IRI-88-15707

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0-89791-317-5/89/0005/0315 $1.50

1. INTRODUCTION

There are many database application areas where a policy of non-deletion is required. These include financial transactions, transcript archives in universities, multiple version histories in engineering design, legal records, medical records, and so forth. One usually wants faster access to the most recent records while tolerating slower access to the older, historical records. It would therefore be convenient to store the most recent versions of records in one area, and keep this *current* database small, while storing the historical part of the database in a separate area, possibly on a slower more archival medium.

We have developed an access method for just such a situation. We assume that the current database is stored on a random access erasable medium such as a magnetic disk drive. The historical database may be stored on any random access device, such as write-once optical disks, erasable optical disks or magnetic disks. A single unified index enables retrieval from both the historical and the current database. Further, data is written to the historical component sequentially, appended to the end of the historical database.

The current database, and all parts of the index which refer to it, must be on an erasable medium for two reasons. First, we must be able to change references to data which migrate from the current to the historical database. Second, we wish to be able to erase temporary data, such as that which is created by transactions which abort.

We view the current database as one which changes over time, with new data replacing older data. The historical database, in contrast, merely grows as records are added to it. No data is ever removed from the historical database. It is thus possible to store the historical database on a device such as a write-once optical disk.

Currently available write-once optical disks have high storage capacity, long life and reasonable access

time. They can be removed from the disk drive, enabling very inexpensive libraries to be created. These are usually served by a robot, which can mount archived disks on the disk drives. Further, as we shall show, the two characteristics of currently available optical disks which could limit their usefulness—slower seek times and a smallest writable unit—are not so limiting with our access method.

The first possibly limiting characteristic is the slow seek time. Optical drives have longer seek times on average (by about a factor of three) than magnetic disk drives. And if robot disk libraries are used, around 20 seconds are needed to mount a disk which is not already on line. But if these optical archiving systems are used only for historical data, which is accessed less often, these longer seek times may be tolerable, especially when viewed as a trade-off for cheaper storage.

Second, as with magnetic drives, when a sector or block is written, an error-correcting code is appended to the sector. On a write-once device, this error-correcting code is burned into the disk. Thus, even when a small amount of data is written, the rest of the sector is unusable. This implies that small incremental changes such as updating an index entry will waste a large amount of space. Each such increment must occupy an entire sector, typically about one kilobyte (1024 bytes). However, since we only append data to the historical database after it has been organized and consolidated in the current database, we shall be able to write data to the optical disk in units which nearly approximate the sector size.

In this paper, we describe a system for multiversion data which utilizes storage space and access time most efficiently. It takes advantage of the best aspects of two storage mediums: cheap, sturdy, possibly non-erasable, but slow-access optical disk storage, and more expensive but faster-access magnetic disk storage. Recent data is stored on magnetic disk where it can be updated quickly, while older data is incrementally moved to the slower optical disk as it matures.

This system can be also be used for any multiversion database, even if the historical part of the database is also stored on a magnetic disk. That is, while providing a method for efficient use of currently available storage mediums, the system is not restricted to these mediums. We only require that both the current and the historical database be stored on random-access devices. The current database *must* be stored on an erasable medium to permit it to be flexibly updated and reorganized. The historical portion of the database *may* be stored on a write-once medium.

Data in the historical database is never deleted. Data in the current database may be deleted, allowing for non-permanent current entries, such as those made by non-committed transactions.

POSTGRES [Ston] has done pioneering work in this area. We attack this problem using a variation of the "Write Once B-tree" [East] . We call this variation the *Time-Split B-tree.*

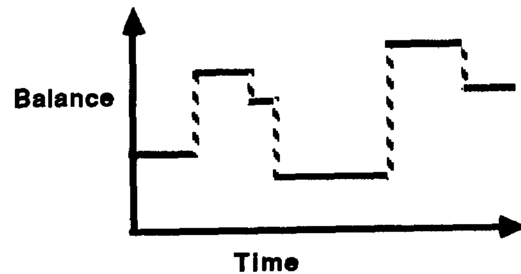

Figure 1. Stepwise constant data. The account balance remains constant between transactions.

Using the classification of time sequence semantics of [SeSh], we assume that we are dealing with *step-wise constant* data. Account balances or employee salaries exhibit this behavior as illustrated in Figure 1. To find the balance of an account at a given time T, we look at the last entry made before T. We assume that the balance is constant until another update is made.

In addition, we assume that we have what [McKe, SnAh] call a *rollback database.* This means that records are stamped with the transaction commit time rather than with the effective time for the information.

The rest of this paper is organized as follows. In section 2,.we give an explanation of the Write-Once B-tree. The Write-Once B-tree is for data stored entirely on a write-once optical disk. In section 3, we outline the design of the Time-Split B-tree, our new structure. The Time-Split B-tree is for data which is partitioned into an historical component, possibly stored on a write-once medium, and a current component, stored on an erasable medium. In section 4, we describe the features of Time-Split B-trees which support database transaction processing. In section 5, we present our conclusions.

2. THE WOBT

Several index structures have been suggested for write-once optical disks [Vitt, Chri, Rath, SaTa, East]. For our purposes, to obtain a single version of a record by time and key, to be able to access snapshots of the database, to retrieve all versions of a given record, we believe that Write Once B-trees [East, Salz (section 8.5)] are an excellent foundation for the access method that we have in mind.

The Write-Once B-tree, or WOBT, is a modification of the B^+-tree which can be implemented completely on a write-once medium. Basically, WOBTs do node splitting on a time basis as well as on a key space basis. This way, the most recent versions of records are kept in a small number of nodes, enabling search in the

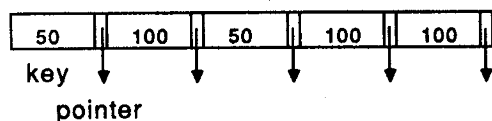

Figure 2. A WOBT index node. Entries are in insertion order. The same key may occur several times.

current database (the database of all versions of records which are valid at the current time) to be efficient.

2.1 Description of WOBT Nodes

We first describe how WOBTs can be used to provide single-version B^+-tree functionality on a write-once medium. That is, we show how to store and retrieve the most recent versions of each record.

The leaf nodes of the WOBT, like the B^+-tree, contain the records. Nodes are logical constructions which may be implemented with a sequence of consecutive sectors on an optical disk. The records are in insertion order, with possibly many versions of the same record in the same leaf node. That is, an update on a record is treated like an insertion of a new version. The key will be the same as on the old version, which remains in the database. If the record size is smaller than the sector size, there is exactly one newly inserted record in a sector of a leaf node, even if there is room for more than one record in a sector. This is due to the fact that the sector is the smallest writable unit. That is, when a new record is added to the database, it cannot use less than one sector of space. However, when nodes are split, several records will be copied into the new nodes at the same time, so the copied-over records can be consolidated. Thus a typical data node has several records per sector for the first few sectors, then one record per sector for rest of the node.

Similarly, the contents of an index node in a WOBT are in insertion order. The same key may occur twice, with the first occurrence in the node being the earliest occurrence as illustrated in Figure 2. Since new index entries are made one at a time, and a sector is the smallest writable unit, there may be only one new index entry in a sector. When index nodes are split, the copied index entries are condensed in the new index nodes, just as when data nodes are split, copied records are condensed in the new data nodes. Thus, the older index entries in an index node are packed together filling the sector space while the new index entries are placed one to a sector, wasting most of the sector space.

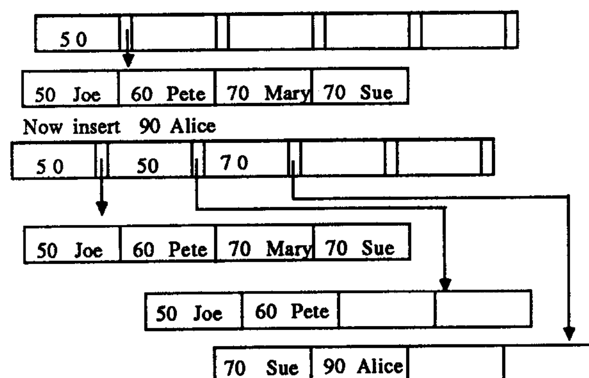

Figure 3. Splitting data nodes by key value and current time in a WOBT. The old node remains in the data base. Two new data nodes and two new index entries are written.

2.2 Search for Current Data

To find the most recent version of a record, begin at the current root of the WOBT. Find the key-and-pointer pair such that the key is the largest one which does not exceed the search key, and the pair is the last one listed in that node with that key. Follow the pointer down the tree. Repeat this process in each WOBT node visited until a leaf is reached. For a given key, exactly one path from the root to the leaf will be followed. If the record is in the database, the latest version is the one listed last in the leaf node.

2.3 Insertion in the WOBT

Insertion in the WOBT is very similar to insertion in the B^+-tree. The search process is followed to find the correct leaf for insertion. If there is room, the new record is inserted in the next available sector in that leaf. If there is no room, a "split" takes place and new leaf nodes are allocated and one or more new index terms are posted to the parent. Similarly, if index nodes are full, they too are split.

To "split" a WOBT leaf node or index node we have two choices. We may split by key value *and* current time or by current time only. In both cases, only the most recent versions of records or index entries are copied to the new nodes.

When the split is by key value (and current time), we create two new nodes. The two new addresses and the old key value and new split value are placed in the next available space in the parent node. Splitting data nodes by key value and current time is illustrated in Figure 3. The split value for the key determines which records go in which of the two new nodes. Since only

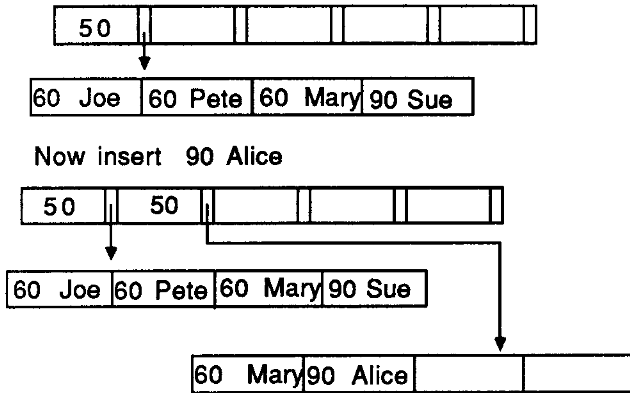

Figure 4. A pure time split in the WOBT. There are not enough current records to make two new nodes. The split is by current time.

current versions of records are copied, this is also a split with respect to current time.

If there have been many updates, the number of current versions may be so small that we may choose to split only by current time. In this case only one new node is constructed, consisting only of the current versions. This is a "split" entirely by time, not by key value. This type of split is illustrated in Figure 4.

We make two observations. First, records are repeated or copied several times. A version which lasts a long time has many copies in the database. Second, since we follow the same algorithm for splitting index nodes, the WOBT is a DAG, not a tree. Both the old and the new index nodes may contain copies of the same pointers.

2.4 New Root Nodes

As with the B$^+$-tree, sometimes the root node itself must be split. When a root splits in the WOBT, the new root refers to the old root. If the split has been by time only, both entries in the new root node will have the lowest possible key value, with the first entry pointing to the old root and the second pointing to the most recent versions of the entries from the old root.

If it has been a split by both time and keyspace, the new root has three entries: one with the lowest key value pointing to the old root, one with the lowest key value pointing to the most recent entries from the old root less than the split key, and one with the split key pointing to the rest of the most recent entries from the old root. Since this is all on optical disk, a list of successive addresses for the root nodes must also be kept.

2.5 Using the WOBT as a Rollback Database

The previous description of WOBTs when they provided single-version B$^+$-tree functionality, did not require that time be stored in the database. If we are to use WOBTs to support rollback databases, we need to provide each newly inserted record with a timestamp indicating the commit time of the transaction that inserted it. When a node split occurs, the current time must be used to timestamp the new index terms.

With a WOBT providing a rollback database, we can find the state of the database as it was at any given time in the past. We can find the records with a given key valid at a given point in time. We can find all past versions of a given record. Let us look at how these temporal queries are supported by the WOBT.

To find the record with a given key K valid at time T, begin with the current root node. Ignore all entries with timestamp greater than T, then follow the algorithm for latest version of a record. That is, ignoring all entries with timestamps greater than T, look for the largest key smaller than or equal to K in the node. Then find the last key-and-pointer entry with that key value in the node, and so on. Follow the pointer down to the next level of the WOBT. Repeat the same search pattern in every node visited. You are guaranteed to find the record in question, if it exists, in one path down the WOBT, just as in the search for a current record by key.

The current root node will have one pointer stored with the lowest key value (minus infinity) and the lowest time value. This is inserted into the initial root. The splitting and updating process assures its propagation to subsequent roots, and the pointer in the current root will point to the previous root, if there is one. The search path may take us through successively older roots, but this is handled by the search algorithm without making special cases.

To obtain a snapshot of the database at any given past time T, begin at the root as usual. Ignore all entries with timestamps after T. Then working down the WOBT, obtain the last entries in each index node for each key before or at T, and finally, the last copies of each record before or at T.

To find all previous versions of a given record, backward pointers in leaf nodes to the nodes they were split from are suggested. Begin at the leaf node containing the record. Follow the backwards pointers until a leaf node is encountered which contains no earlier version of the record. There will be several optical-disk seeks. But also several versions of the record are likely to be in each node accessed.

2.6 Conclusions on the WOBT

Search algorithms for many typical temporal queries are simple on the WOBT. This is an elegant, clean structure. Time-domain splitting concentrates the current data in a small number of nodes. However, this

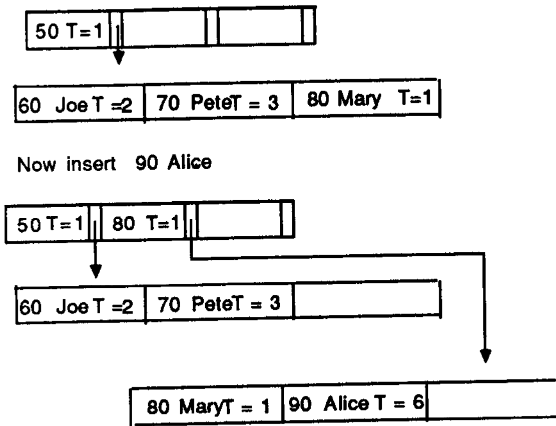

Figure 5. Data node split entirely by key. Timestamp in index entry is the same as the previous index entry timestamp.

means that many records have redundant copies in the database.

Further, storing a WOBT solely on a write-once medium means that new entries must use entire sectors, possibly wasting a great deal of space. Also, temporary data, such as that created by uncommitted transactions, cannot be discarded if a write-once medium is used.

On the other hand, using the WOBT solely on *magnetic* disk loses the advantages of less expensive per-byte storage cost, permanence, reliability and portability available with optical disks. One solution is to store current data on magnetic disk and migrate older permanent data to optical disk. The *Time-Split B-tree* has been developed to provide this solution.

3. THE TIME-SPLIT B-TREE

The Time-Split B-tree is a variant of the WOBT which will migrate data incrementally from a magnetic disk to an optical disk. In this section we shall explain how we change the basic WOBT structure, and then, in section 4, we show how this new structure can be used to support transaction processing.

We modify the split algorithm of the WOBT in several ways. First, we only split nodes which are on the magnetic disk. There are again time splits and key splits. But the key splits on magnetic disk are more like those in B^+-trees since we need not keep the old node intact. The records with keys smaller than the split value stay in the old node. Those with keys larger or equal to the split value go in the one new node.

When we split by time, we no longer need to split by the current time. We may split by any convenient time. In this case, the "older" records are migrated to the optical disk and the newer records are kept on the magnetic disk. Migration occurs incrementally, one node at a time, only when nodes are time-split. In spite of the changes in the splitting policy, the search process is exactly the same as in the WOBT.

3.1 Data Node Splitting

Suppose that in a given data node, each record has only one version. This means that all changes have been insertions of new records, with new keys. There have been no updates of existing records. It does not make sense to make a time split in this case. In this case, we make a keyspace split. The timestamp in the new index entry is the same as the timestamp of the previous index entry referring to the old data node. This is illustrated in Figure 5.

In case there have been a number of updates to existing records so that several versions of some of the records are in the node, we may make a time split, but with a twist. Instead of always splitting with respect to the current time, we split by a time T, depending on the actual timestamp values in the records in the node. Then the node with the older timestamp values is migrated to the historical database, while the node with the new values remains in the current database. *Note that migration is one node at a time.*

The time-split rule is as follows: If a split is made with timestamp T,

---

TIME-SPLIT RULE

1. All entries with time less than T go in the old node.

2. All entries with time greater or equal to T go in the new node.

3. For each key used in some entry, the entry with the largest time smaller than or equal to T must be in the new node. That is, the version valid at the split time must be in the new node.

This forces some redundancy, as all records which persist through the split time have copies in both nodes. However, this feature makes it possible for records valid at the same time to be clustered in a small number of nodes. If one does not have redundancy, long-lived records can only be stored in one place. No matter what strategy is chosen for storing such a long-lived record without redundancy, some snapshot queries will be inefficient. Also, as we shall see, the ability to choose the split time permits optimization choices to be made. We give some examples of data-node time splits in Figure 6.

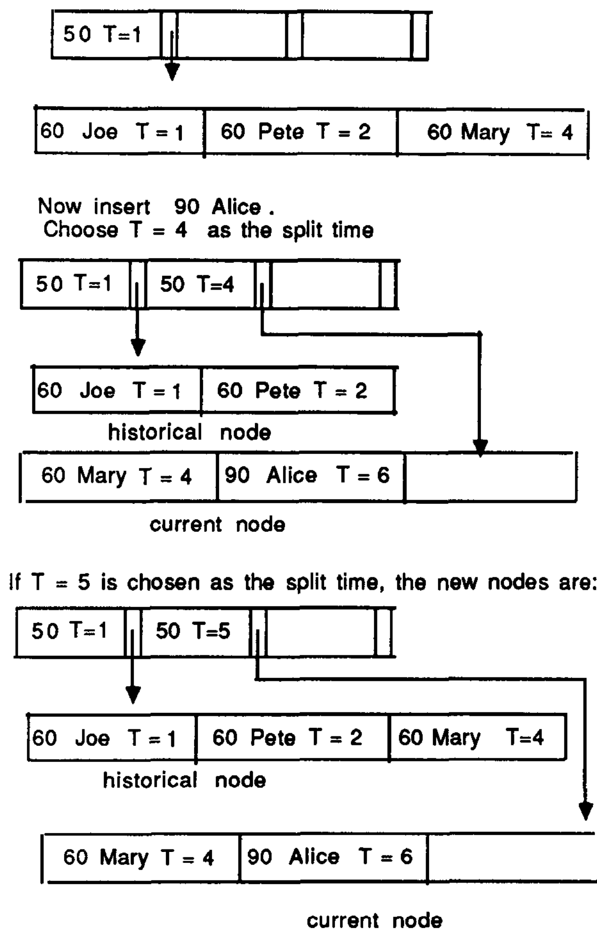

Figure 6. Time-Split B-tree time splits. If T=4 is chosen there is no redundancy in this example. If T=5 is chosen, the record with "Mary" is in both the historical and current nodes.

3.2 Deciding Whether to Split by Time or by Key

The question to be answered is the criteria by which we decide whether to do a key space split or a time split. One object of a storage system is to try to minimize the total space consumed. A second one is to minimize storage for the current version, which is subject to updating, has the highest expectancy of reads, and will be stored on the more costly write-many magnetic disk.

If total space minimization is the only goal, data node splitting by key space would always be favored. If current version space minimization is the only goal, time splitting would always be used. What we want is a storage system that does a good job for both of these requirements, and one that can be parameterized so as to be responsive to an adjustable cost function. One possible cost function is

$$C_S = \text{Space}_M \times C_M + \text{Space}_O \times C_O$$

where C_S is the total storage cost, C_M is the cost for storage on the magnetic disk and C_O is the cost for storage on the optical disk. The goal, in splitting, is to minimize the cost function.

At the same time, the kind of split used depends on the what is in the node to be split. As noted above, if only insertion has occurred in a full node requiring splitting, there is no reason to do time splitting. All data is relevant to the current version and hence must remain in the current node. Thus, time splitting by itself is useless. Key space splitting must be done.

On the other hand, if only repeated updating of a single record has occurred in a full node requiring splitting, there is no reason to do keyspace splitting. All data is associated with the same key value and so cannot be split. Thus keyspace splitting is useless and time splitting must be done.

These boundary conditions determine the kind of splitting that should be used. The more out-of-date (historical) data is on a node, the more likely it is that time splitting should be used while the less historical data there is (or the more current data there is), the more likely it is that key space splitting should be used. Let us examine some more consequences of the different splitting forms.

3.3 Time Splitting

Time splitting results in redundancy. If time splitting is chosen, a further decision has to be made about what time value to use for the split. The WOBT always used the current time as the value of the split because the old node had already been written on the optical disk. With magnetic disks, this restrictive approach can be overcome. Any convenient time more recent than the last time split for the node can be chosen as the split value.

As an example of using this flexibility, consider a situation where there are a number of insertions which were done *after* the last update of existing data. In this case, choosing the split time to be the same time as the last update avoids having to carry the final inserted data in the historical node. Note that the contents of the current version node are not affected by this choice of time splitting value, and remain at the minimum, i.e., only the current version data, and no historical data, is stored in the current node.

This does not mean that there is no redundancy. Some of the current data persisted across the split time. Only if the current data had all been created at or after the split time is redundancy avoided. Data created before the split time and persisting through the split time is in both the historical node and the current node.

When the split time is pushed back past updates in addition to insertions, some historical data must be stored in the current version node. This can still result in a smaller amount of redundant data overall as more data may be removed from the historical node than must be added to the current node. But now, we are making a trade-off between minimizing the amount of

redundant data, and minimizing the space used by the current database.

3.4 Size of Historical Node

In the choice of time value on which to split, we have been treating the amount of data stored in the historical node as an important consideration. Note that in WOBTs, the historical node size is fixed, and the data in a node which is to be split is indelibly written there on the write-once optical disk. This is not the situation with the Time-Split B-tree. Here, currently updatable data is stored on magnetic disk. And the node size for the optical disk can easily be set to be different from the magnetic disk node size. Even more to the point, the historical data can be appended to a sequential file.

The index pointer to a historical node needs only to record its address on the optical disk and its length. While there might be some minimum granularity that prevents us from matching precisely optical disk space consumed with historical data size, it is possible to come close. The risk of disastrously low optical disk storage utilization is thus entirely removed.

3.5 Index Node Splitting

Index nodes can always be keyspace split. To do this, since index nodes, unlike data nodes, reference entities involving a range of key values as well as a range of times, we must make a rule similar to the time-split rule. That is, record versions in data nodes have one key and span a time interval. Entries in index nodes refer to lower-level time-split B-tree nodes which span a keyspace interval as well as a time interval. We call a keyspace interval spanned by a time-split B-tree node a *key range*. We therefore make the following rule:

Index Node Keyspace Split Rule

1. The split value may be any key value actually used in an index entry in the node. This key value and a copy of the time used for the previous reference to the node to be split are posted to the parent index node.
2. References to key ranges where the upper bound for keys is less than or equal to the split value go in the new left node.
3. References to key ranges where the lower bound for keys is greater than or equal to the split value go in the new right node.
4. All others (which are guaranteed to be references to the historical database) are copied to both nodes. These reference key ranges which strictly include the split value.

The references where the split value is strictly contained in the key range are guaranteed to be historical because key splits are successive refinements of the key range over time. That is, the only case where a key

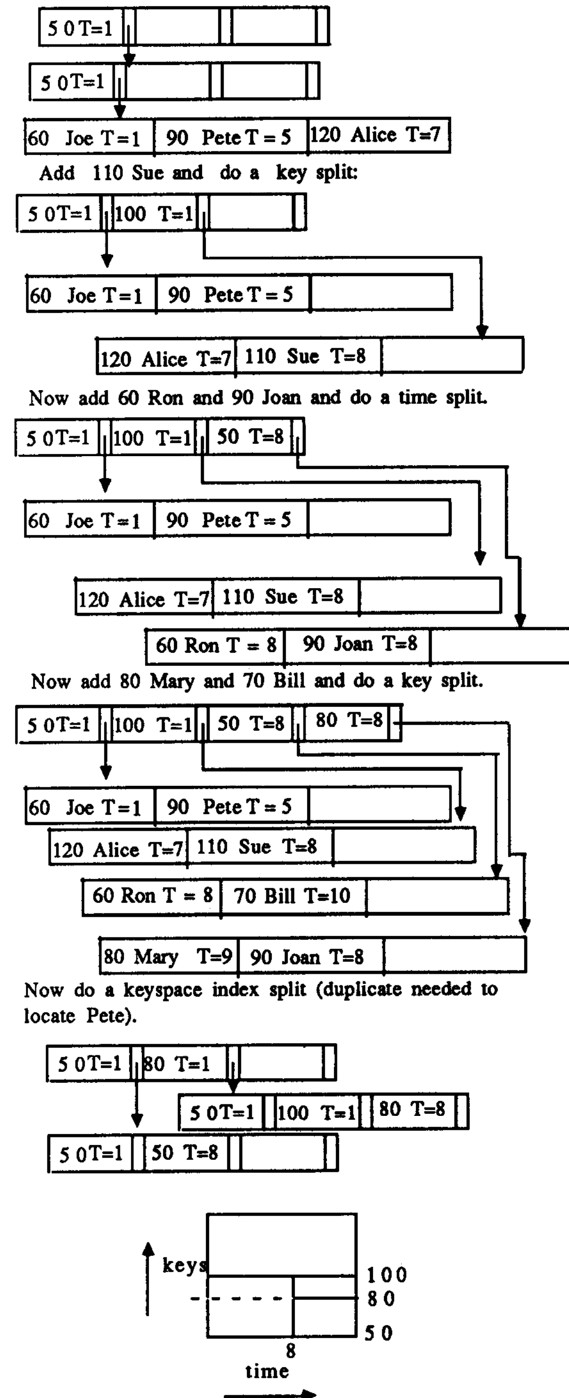

Figure 7. Successive changes in a Time-Split B-tree index node. At the end, a keyspace split is made, showing the key ranges and the time ranges referenced.

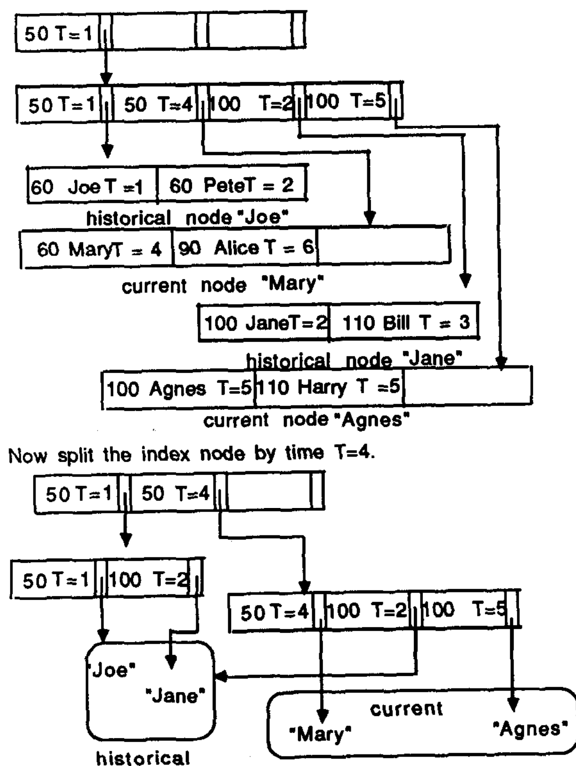

Figure 8. A Time-Split B-tree index node where the time split is entirely local. Only one index node migrates to the optical disk. This can be done whenever there is a time before which all references are to the historical database.

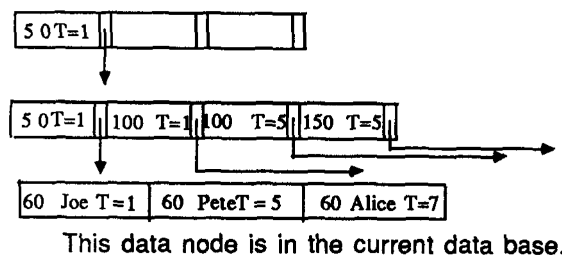

Figure 9. Here there is no time before which all entries of the index node point to the historical database. Either the index node must be keyspace split, or else lower nodes must also be split.

value is in an index entry X in the node to be split (hence a lower bound for the range referred to by X), and also is strictly contained in the key range referred to by Y (another index entry in the same node), is when the key range of Y was later affected by at least one time split, followed by at least one keyspace split. We show this phenomenon in figure 7. Note that, like the original WOBT, the Time-Split B-tree is a DAG rather than a tree. However, only historical nodes have more than one parent.

We can also time split index nodes by finding a time before which only historical versions exist, i.e., no version with an earlier time resides on magnetic disk. There may be no such time, of course. But most likely, over 'time' there will be some such point. This puts an additional constraint on index node splitting. Not only must current entries be retained in the current version index node, but no entries that reference current nodes can go into the historical index node. This is so because the current nodes can split, requiring index node updates, which cannot be accommodated in historical index nodes.

When these conditions are met, index splitting is local. In this case the redundant index entries are all pointing to historical nodes. Again, this makes the Time-Split B-tree a DAG rather than a tree, which, as we have noted, is also true of the original WOBT. Again, historical nodes may have more than one parent. This is illustrated in Figure 8.

When the conditions are not met, attempting to time split index nodes would force splitting in nodes lower in the tree. This would make splitting non-local. That is, the split cascades down the tree. A node which cannot be locally time split is illustrated in Figure 9.

When an old data node which has not undergone a time split prevents us from doing a *local* time split at an index node higher in the tree (as illustrated in Figure 9), this old data node could be marked to be time split at the next opportunity. This is just an optimization choice for the Time-Split B-tree. On the average, we should be able to do time splits with index nodes gradually, as there will usually be a time before which all entries point to historical data. There should not be many recalcitrant index nodes without this property.

3.6 Secondary Indexes

Secondary indexes can be implemented as Time-Split B-trees as well. Secondary indexes are modified when a new record is created, or when the secondary field is updated in any data record. Each entry inherits the timestamp from the record which caused the change. Secondary indexes, like the primary index Time-Split B-tree, span the historical (optical disk) and current (magnetic disk) databases.

The secondary indexes contain records of the form

$$< \text{timestamp, secondary key, primary key} >$$

The primary key and the timestamp are used to find the primary data record being referenced. When splits occur to the primary data, secondary indexes do not change. The timestamps also serve to answer

queries about the secondary values which do not require searching for primary data records. For example, one can answer the question of how many records had a given secondary key at a given time using only the secondary time-split B-tree.

3.7 Summary of Time-Split B-tree Characteristics

The Time-Split B-tree uses the best features of the optical disk WOBT—simple structure, easy access for many temporal queries and locality of access for records valid at a given time. It migrates data incrementally from the magnetic disk to the optical disk. One indexing structure handles both the current and the historical part of each relation.

Optimization choices can be made to limit the total space cost, the space used for current data or the amount of redundancy in historical data. This is all made possible by the flexibility for choosing whether to make a time split or a keyspace split, and the ability to choose the value for the time split.

Historical data space use is excellent as historical node size can vary and many entries can be consolidated on one sector. This is a consequence of integrating storage on magnetic disks with storage on optical disks. No small incremental data need be written to large optical disk sectors; it can be consolidated first and then migrated.

4. SUPPORT FOR TRANSACTION PROCESSING

Most concurrency methods based on versioning can be used with the time-split B-tree. For example, suppose that timestamps of committed transactions are used on records as in POSTGRES [Ston]. Records created by uncommitted transactions have no timestamps, so that they are never written to the historical database during a time split. This means that uncommitted data can always be erased.

4.1 Read-Only Concurrency Control

A read-only transaction, e.g., one that does file backup, can run without concurrency control, in terms of logical record [database] locks, if it is given a timestamp when it is initiated, as opposed to when it commits. It will then 'see' only versions that are not locked by an updater. Thus, it will never have to wait for an updater to commit.

If the latest version has a timestamp, the read-only transaction knows, based on its timestamp, which version to use. No updater can post a timestamp earlier than the read-only timestamp since that point in time has come and gone.

Similarly, if a version exists with a timestamp later than that of the read-only transaction, the read-only transaction will read the earlier version appropriate to its timestamp. This will be true even if there is a non-timestamped version.

This capability enables database unloading and backups to be efficient, since they do not require locks. This can be used in any versioning system; it is not unique to the Time-Split B-tree.

5. CONCLUSIONS AND ONGOING WORK

Space use in the WOBT on write-once disks can be poor when small amounts of information, such as index entries or delta records, occupy an entire sector. Further, if only write-once devices are used, as in the WOBT, "reorganization" of information (as occurs in node splits even when all the entries are insertions rather than updates) involves duplication of all the current data. Also, temporary data cannot be discarded.

By using both a magnetic disk and an optical disk, and the new node splitting policies, the Time-Split B-tree solves these problems. We can consolidate information before placing it on a write-once device. The erasability of the magnetic disk permits "normal" B-tree node splitting. Data can be data written by uncommitted transactions and erased if the transaction aborts. The adjustable splitting policy allows for different space costs in the magnetic and the optical disks—more time splits to lower magnetic-disk space use, and more key splits to lower total space use and data redundancy.

The Time-Split B-tree incrementally moves data from the current database stored on magnetic disk to the historical database on optical disk, one node at a time. Efficient concurrency methods based on versioning can be applied to allow read-only transactions to run without locks. Splitting policies can be parameterized to optimize for different cost formulas. The Time-Split B-tree should be an attractive storage option for multiversioned historical databases where there is a non-deletion policy.

We are currently in the process of implementing Time-Split B-trees at Northeastern University. This implementation effort is supported by the NSF (IRI-88-15707). We expect to measure total space use, space use in the current database, and amount of redundancy, under different splitting policies and with different rates of update versus insertion.

References

[Chri] Christodoulakis, S., "Analysis of Retrieval Performance for Records and Objects Using Optical Disk Technology," ACM-TODS, **12:2**, June 1987, pp. 137-169.

[East] Easton, M., "Key-Sequence Data Sets on Indelible Storage," *IBM J. Res. Develop.*, **30:3**, May 1986, pp 230-241.

[Lome] Lomet, D., "Partial Expansions for File Organizations with an Index," ACM-TODS, **12:1**, March 1987, pp. 65-84.

[McKe] McKenzie, E., "Bibliography: Temporal Databases," *SIGMOD Record,* **15:2**, Dec. 1986, pp. 40-52.

[Rath] Rathmann, P., "Dynamic Data Structures on Optical Disks," Computer Data Engineering Conference, April 1984, Los Angeles.

[Salz] Salzberg, B., *File Structures: An Analytic Approach*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[SaTa] Sarnak, N., and Tarjan, R., "Planar Point Location Using Persistent Search Trees," *Communications of the ACM*, **29::7**, July 1986, pp. 669-679.

[SeSh] Segev, A. and Shoshani, A., "Logical Modeling of Temporal Data," *Proc ACM SIGMOD*, May 1987, pp. 454-466.

[SnAh] Snodgrass, R., and Ahn, I., "A Taxonomy of Time in Databases," *Proc ACM SIGMOD*, March 1985, pp. 236-246.

[Ston] Stonebraker, M., "The Design of the POSTGRES Storage System," *Proc. 13th VLDB Conference*, Brighton, 1987, pp.289-300.

[Vitt] Vitter, J., "An Efficient I/O Interface for Optical Disks," ACM-TODS, June 1985, pp 129-162.