Key-sequence data sets on indelible storage

by Malcolm C. Easton

Methods for creating and maintaining key-sequence data sets without overwriting the storage medium are described. These methods may be applied to erasable or to write-once storage devices, and they are compatible with conventional device error-management techniques. All past values of data records are preserved in a data structure called a Write-Once B-Tree. Rapid random access is available to records by key value; rapid sequential access is available to records in key-sequence order. Moreover, queries requesting data as of a previous time are processed as rapidly as requests for current data. Access time is proportional to the logarithm of the number of *current* records in the database. Efficient methods for inserting, updating, and deleting records are described. Upper bounds for tree depth and for storage consumption are given and compared with results from simulation. It is concluded that, with rapidly improving storage technologies, indelible databases will become practical for many applications.

Introduction

Optical disks, both write-once and reversible, are expected to provide high-density storage at a low cost per bit [1]. At the same time, magnetic storage costs are rapidly dropping. Copeland [2] suggests that these developments in technology may change data management policies. Present practice is to keep on line only current data, while archiving historical data. Recovery from data processing errors typically requires a time-consuming process of restoring the database to an earlier state and then rerunning all subsequent transactions. Current methodology is also prone to human error and is vulnerable to unauthorized alteration of data.

Copeland also argues that the practice of erasing is contrary to basic accounting principles. If storage costs permit, he writes, then no erasing should occur in databases holding business records. He cites accounting practices that require a reversing entry to alter an existing entry. Svobodova [3] also notes the utility of saving all values that a data field ever takes. The possibility of using write-once optical disks for database applications is discussed by Maier [4].

These considerations give rise to the notion of *indelible* data. An indelible data set is one in which all old values of data records are retained. Such data sets are useful if rapid access is needed to historical data. They also provide a built-in audit trail.

A data structure that serves as the basis for many database management schemes is the B-tree of Bayer and McCreight [5]. IBM's VSAM as well as many other database management systems use variations of this approach to maintain *key-sequence data sets* (*KSDS*), that is, sets of records that can be rapidly accessed either by primary key value or in key-sequence order. We show here how the B-tree approach can be modified to provide efficient, *indelible* management of a KSDS on a rotating storage device. The performance benefits of B-trees are retained, and moreover the evolution of the database in time is preserved.

Related work

Management of data on write-once disks has been approached by several authors. Vitter [6] expands on an earlier report [7] of the B-tree methods described here but

makes different assumptions about the device. (Our assumptions are described in the next section.) If a variable-size record can be appended to a track with reasonable efficiency, then Vitter's scheme improves storage utilization by encoding changes rather than rewriting entire new records. Vitter also generalizes the methodology to apply to a class of data management problems.

Rathmann [8] has proposed an alternative tree management scheme that requires no overwriting. Rathmann implicitly restricts his device to writing all records sequentially from the first track, and thus must rewrite the full path from root to data record whenever an update is made. He also proposes a trie scheme that provides better performance than the tree scheme if keys are short. On a less restrictive device, however, Rathmann's schemes are less efficient than the methods described here. On the other hand, the "append-only" device model offers the advantage of an extremely simple method for "undoing" the most recent changes to the data set.

Terminology and device characteristics

Let the storage space consist of a number of *tracks*. To include both optical and magnetic disks under a single terminology, we use the term *sector* to refer to the unit of data that is written within a track. On write-once disks, the sector size is typically fixed.

We assume that the device allows random access to the start of any track. Following such a "seek" operation, we require only the capability to write into the first free space on the track. We do not assume the possibility of what Maier calls *backwriting*, that is, writing into a gap that precedes a previously written area on a track. One reason for avoiding the use of backwriting is the difficulty of handling, on a write-once disk, a write error when there is no space left in the gap.

For simplicity, we assume that a *data record* can be stored in a single sector. A *data set* is a collection of data records. A specific field within each data record is designated as the *primary key*, and no two data records within a particular data set have the same primary key. In addition, other fields of a data record may be designated as secondary keys.

Device error strategies—write-once disks

If a write-once optical disk is the storage medium, then special considerations apply to error management. Each sector is written with a redundancy check (CRC). Typically, each written sector is checked during or just after the write operation. The reason for this check is the relatively high bit error rate of optical recording. If a write error occurs, then the sector is copied to a new area. In addition, the device controller may overwrite, or otherwise mark, the bad sector to provide an unambiguous indication that the sector is to be ignored in future readings. We call this overwrite procedure *demarking*.

The details of handling the bad sector on readback depend on the controller and software. In some systems, the bad sector is automatically hidden from the application program, but the number of sectors in the track is reduced. The data management techniques described here are compatible with such systems.

We also consider the case where the software that manages the database must handle the recovery operations on encountering a read error. If demarking is used to indicate invalid sectors, then the problem is simplified. Any demarked sectors are ignored on readback. Any other type of read error must be retried. We assume in the main discussion that, whenever a write error occurs, the sector is immediately demarked. Issues arising when no such capability exists are discussed in Appendix A.

Basic operations

We assume that hardware and I/O software provide the following operations:

1. Write into the first free space of a track or determine that no additional sector can be written. (This may be implemented by first reading the track.)
2. Read all valid sectors of a track.

Database operations

The following database operations are required:

1. Insert a data record into the data set.
2. Given a primary key, find the record having that primary key.
3. Retrieve data records in primary key-sequence order starting with a specified key.
4. Given a primary key, erase the data record which has that primary key.

Many methods that require overwriting the storage medium have been described for accomplishing these tasks efficiently. We describe techniques for doing them without overwriting data or index records. (One reason that index records are not overwritten is to preserve historical paths to data.)

An essential feature of the methods described is that they are compatible with the use of error-detection and error-correction codes. Other approaches to managing data on write-once disks have not dealt with the possibility of data errors [4, 9].

Write-once balanced tree data set

The B-tree of Bayer and McCreight [5] is modified for application to indelible data sets. One important alteration comes from the need for different splitting rules than are used with conventional B-trees. Also, the structure of data within a node (bucket) is necessarily different in the Write-Once Balanced Tree (WOBT).

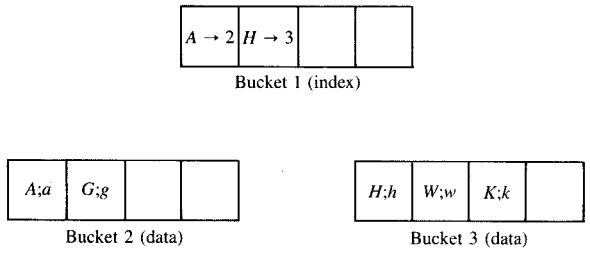

Figure 1

Sample WOBT containing primary keys *A, G, H, K, W.*

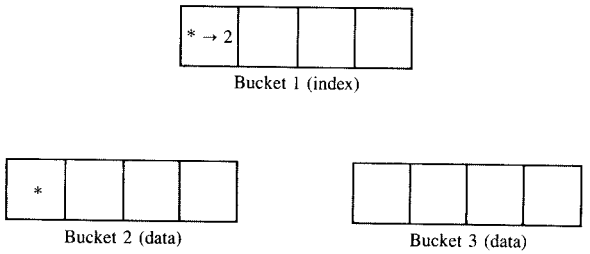

Figure 2

Initial WOBT.

Each data record contains a primary key as well as possible secondary keys and data. We use $X;x$ to indicate an example of a record with primary key X whose remaining part is x. The records are organized in buckets. Each bucket is implemented as one or more physical tracks and contains all records *of the data set* whose primary keys fall (in collating order) between the smallest and largest primary keys stored in the bucket. The bucket *header* is the smallest key in the bucket. Each bucket's header appears in the index along with a pointer to the bucket.

The index itself is organized in a similar manner. Each index bucket contains all keys of the index whose values fall between the smallest and largest keys in the index bucket. Each such bucket in turn has its header in a higher index bucket until there is a single (root) bucket that holds the highest level of index keys. Within each index bucket, the entries appear as key-pointer pairs, each pointer giving the location (bucket address) of a header key in the next lowest level. **Figure 1** shows an example with four sectors per bucket and one index level. In the example, each sector can hold either one key-pointer pair or one data record. The contents of bucket 1 indicate that all primary keys equal to or greater than A and less than H are found in bucket 2. All primary keys equal to or greater than H appear in bucket 3.

The first entry of a bucket holds the bucket's smallest key; otherwise entries are in no particular order. Keys may appear more than once in a bucket, because updating is done by writing the key again with its new pointer or record values. The last pointer or the last data entry with a particular primary key is deemed to be the current or *active* value.

Whenever a bucket is filled, its contents are processed and then rewritten into a new bucket or buckets. This step constitutes a local reorganization of the data. As in the case of the conventional B-tree, it will be shown that the depth of the tree grows as the logarithm to some large base of the number of primary keys in the database. (That is, the depth depends on the number of *current* entries rather than on the number of updates made to the database.) The fanout at each level of the tree is controlled by the recopying algorithm and depends on the bucket size.

We now describe the operations for key-sequenced data sets.

Initialization of data set

The initial tree has a single bucket for its one index level and a single data bucket. A null (dummy) record whose primary key is all zeros (denoted $*$) is the initial entry in the one data bucket, and its key is the initial entry in the index, paired with a pointer to the data bucket. No real key may have this (all-zeros) value. (See **Figure 2**.)

Handling write errors on write-once disk

In the procedures below, the only write operation is to write into the next free sector in a bucket. If a write error occurs, then the bad sector is demarked and the contents rewritten into the next position in the bucket. If there is no room, then the bucket is "full." See "Reorganize a bucket" below for the handling of this case.

Insert a record whose key is *X*

Begin with the root as the current index bucket.

1. Search the keys in the current index bucket for the largest key that does not exceed X. Find the last copy of that key. Follow the pointer with that key to the next level of index.

Repeat Step 1 until a data bucket is reached.

- If space exists in the bucket, write the record into the first available space.
- If not, then reorganize the bucket (see below) and write the new record into the appropriate new bucket.

Example showing insertions

In the figures that follow, four sectors can be written per bucket, and one sector can hold either a key-pointer pair or

a data record. The symbol * designates the key that is stored as all binary zeros. The entry $A \rightarrow 2$ means that key A points to bucket address 2. The dummy key * is the initial key in the data set and has no data with it; the first index entry points to the bucket holding this key, as in Figure 2.

We then insert records with primary keys F, E, B, as in **Figure 3**. Next the record with primary key W is inserted. Since bucket 2 is full, its contents are reorganized and then written out into new buckets 3 and 4, as in **Figure 4**. Bucket 2 is discarded. The index gets new pointers. Note that the second pointer for * supersedes the earlier entry.

Now the record with primary key W is updated three times. When the third update is attempted, bucket 4 is found to be full. The contents, including the new record with key W, are reorganized. Since only two distinct keys are present, the contents are recopied into a single new bucket, bucket 5 in **Figure 5**. Bucket 4 is discarded. A new index entry is written for key F. The most recent data record with key W is denoted $w\#$.

Suppose that at a later time another entry, $Z \rightarrow 8$, comes to bucket 1, which is then reorganized and copied into *two* buckets (to allow adequate free space). The index must now grow by one level. The two buckets derived from the old root form the second level of the new index, and a new first level (root) is written containing the pointers to the second-level buckets.

Retrieve record having primary key X

Let the current index bucket be the root.

1. Search the keys in the current index bucket for the largest key that does not exceed X. Follow the most recent pointer stored with that key to the next level of index.
2. Repeat Step 1 until a data bucket is reached.
3. Search the data bucket for the last record with key X. Or, if none is found, report that the record is absent from the data set.

Retrieve records in (ascending) primary key-sequence order

The start of the sequence to be retrieved is specified by an initial search key. The procedure does not require horizontal pointers in the tree structure to link together buckets on the same level. By holding in memory the index bucket that is one level higher than the data bucket being processed, the number of accesses required to read the entire set of records is kept to a small percentage increase over the number of accesses required to read each data bucket once. In essence, the algorithm provides what Knuth calls a *preorder* traversal of the tree [10].

The procedure uses a list, called a Search Sequence. This list has one entry for each level of the index. Each entry in the list is a pair consisting of the address of the last bucket visited on that level and a key value. The last-bucket-visited

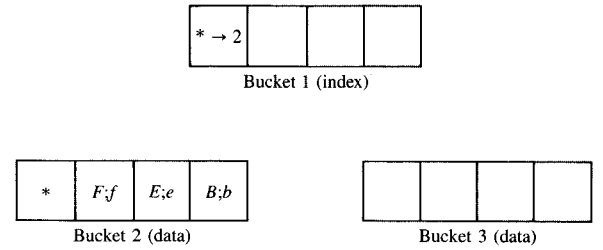

Figure 3

WOBT after inserting primary keys F, E, B.

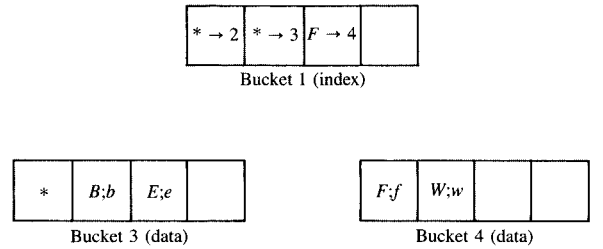

Figure 4

WOBT after inserting key W.

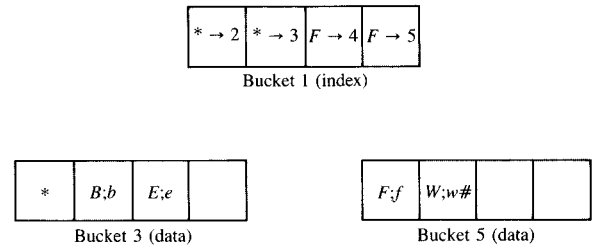

Figure 5

WOBT after three updates of key W.

entry for a level is updated whenever a bucket is visited. The updates to the key entry are described below:

1. Using the initial search key, locate its data bucket D. (This may be the dummy record associated with the all-zeros key.) While descending to D, at each index level of the tree, set the key entry in the Search Sequence to be the smallest nondeleted key of the bucket that exceeds the

search key. Save the contents of D and the contents of the last (index) bucket searched before reaching D. Sort this index bucket by keys, retaining only the latest copy of each record. The current key is the first key in the sorted list.

2. The current key points to the current data bucket. Sort the records of this data bucket by key in ascending order, retaining only the latest copy of each. Output the ordered records.
3. In the (sorted) saved index bucket, let the current key be the next key in sequence. If no keys remain, go to Step 4. Otherwise, go to Step 2.
4. Obtain from the Search Sequence the entry pair (B,K) for the tree level just above that of the saved index bucket. [If K is NULL, then go to the next highest entry in the Search Sequence until entry (B,K) with non-NULL key K is found. If the entire Search Sequence list is exhausted, then terminate since all records have been retrieved.] K is the new search key for the level.
5. Proceed from bucket B to search the tree for data bucket D that contains the record whose primary key is K. Update the Search Sequence at each level passed as in Step 1.
6. Sort by key the contents of the last bucket searched before reaching the data bucket, retaining only the latest copy of each. The current key is the first key in the sorted list. Go to Step 2.

Reorganize a bucket

Reorganizing of buckets is essential for eliminating obsolete records from the search path and for keeping the number of levels of the tree at a minimum. The procedure is basically the same for data buckets and index buckets. Reorganization occurs when an attempt is made to add a record to a full bucket. *The new record is included in the contents that are reorganized.* First, the records are sorted by key; records marked for deletion and outdated records are eliminated. However, the most recent record with a bucket header key is never eliminated.

The next step is to rewrite the contents into a new bucket or buckets. The original bucket becomes inactive. A significant design feature is the method used to determine the number of new buckets that will be written from a filled bucket. (The need for such a method arises from the fact that the number of retained entries after reorganization is data-dependent.)

The specific method studied here for determining how many new buckets to copy into is based on two parameters, $TD > 1$ and $TI > 1$. If the reorganized bucket is a data bucket and holds fewer than TD entries, then one new bucket is written. If it is a data bucket and holds at least TD entries, then two new buckets are written. The index bucket decision is similar, but based on parameter TI.

Each new bucket receives, as closely as possible, an equal share of the contents, which are stored sorted in key-sequence order. A new header-pointer pair for each of these new buckets is sent to the next highest level of index.

If such a pair is sent to the root, and that is full, then the root is reorganized in the same manner. If the root bucket is written to a single bucket, then that becomes the new root. Otherwise, the tree must grow one level. The buckets written from the old root constitute the new second level, and another bucket, the new root, is created to hold the pointers to the second level's buckets.

The implications of this approach for storage consumption and for performance are discussed below.

Erase a data record

1. Find the data bucket holding the record, as in "Retrieve record."
2. Write a new record in that bucket with the same primary key but with an indication that the record is deleted. (Call this a *delete record.*)

Retrieval operations report "no record found" if a delete record is the latest record for a particular primary key.

Handling of delete records

On reorganizing a data bucket, a delete record normally is not copied to a new bucket. There are two exceptions:

- The record's primary key is the old bucket's header key. (Reason: The header is retained to simplify management of the index.)
- The delete record was the last record added to the bucket contents that were reorganized. (Reason: The design philosophy requires that every data entry be stored at least once in the database.)

Note that there are no delete records for index entries.

Initialize a tree with records that are already in key-sequence order

If a data set already exists on some other storage medium, then the records should be sorted in key-sequence order and loaded onto the write-once device in one operation. This saves disk space by eliminating rewrites of full buckets and also saves time. The procedure for doing this is straightforward. Conceptually, "Insert a record whose key is X" is invoked once for each record. Instead of filling a bucket, when a certain number of records have been written into a bucket, the writing halts there and continues in the next bucket in the same level of the tree. An index entry for the new bucket is sent to the index. Free space is left in each bucket for updates or additions. The same approach is used at each index level, with a new root created whenever the topmost index level expands beyond one bucket.

Performance—depth of tree

The depth of the tree is a measure of performance since this determines the number of random accesses for insertion and retrieval. In this discussion, we ignore the possibility of shortening of buckets by write errors. We also ignore deletion. Retrieving a record or inserting a record requires at most one disk access for each index level of the tree other than the root (which can reside in memory), plus one for the data record. Each disk access consists of seeking to and reading a (fixed-size) bucket. We now estimate the capacity of a tree of a given number of levels.

Recall that $TI > 1$ and $TD > 1$. Let F be the greatest integer that does not exceed $TI/2$. Let J be the greatest integer that does not exceed $TD/2$. For simplicity, we only consider the case where the tree has grown larger than its initial configuration, so that it consists of at least two active data buckets in addition to the root. When a bucket is reorganized to two new buckets, the number of distinct entries in each new bucket is at least F for index buckets and at least J for data buckets. When reorganization is to a single new bucket, the number of distinct entries in the new bucket does not decrease. When the tree first grows, each new data bucket holds at least J distinct entries. It follows that every active data bucket holds at least J distinct entries. Similarly, if there exist active nonroot index buckets, then each holds at least F distinct entries.

We call a tree *full* if inserting some key in it will force the tree to grow to a greater depth. We compute the minimum number of data records stored in a full tree having h index levels (including the root) and thus find a maximum number of levels required in a tree that holds that many or fewer data records.

If a reorganized bucket is copied into one new bucket, then one index record is sent to the level above. If a reorganized bucket is copied into two buckets, then only one of the index records sent to the level above holds a key not already stored in that level. In a full tree, the root bucket's contents, plus the root's newly arriving index record key(s) resulting from an insertion, amount to at least TI distinct keys. However, the new key(s) add at most one distinct key. Therefore, a full tree has at least $TI - 1$ distinct keys in the root.

From this we conclude that a full tree with h index levels holds at least $(TI - 1)F^{h-1}J$ active data records.

Indexing options

The B-tree can be used for indexing only, keeping in the bottom level of the tree the pointers to the data records. In this case, the data records themselves are written sequentially as they arrive and need never move. The disadvantage of this approach is that retrieval of records in key-sequence order would typically require a seek for each record, rather than a seek per bucket of data. The advantage is a possible storage savings. The index must in this case contain an entry for each data record, rather than for each data bucket; this extra space in the index is compared with the savings in data record storage. As shown below, the recopying of data records to maintain the data organization consumes on average, depending on key distribution and use of data compression, 0.5 to 2 extra record slots per original record stored.

By not copying data records we have a data access technique with characteristics similar to those of extendible hashing [11] used on erasable storage. While the application of hashing to indelible data sets is beyond the scope of this paper, we note that a performance problem arises if we have many updates to one record. Each entry uses the same key and thus gets hashed to the same bucket. The approach taken here provides for elimination of the obsolete values from the search path and thus offers fast access regardless of the number of updates.

Storage consumption

The values of TD and TI can be used to trade access time against storage consumption. If the incoming sequence of E data records (inserts or updates) were simply written sequentially onto the disk, then E data sectors would be consumed. Extra storage is required by the WOBT in order to provide fast access in key-sequence order. We argue here that the storage consumption by index entries is negligible compared with that by data sectors (if the data buckets are of reasonable size) and that, for a range of values of TD, the worst-case storage consumption for data is $4E$ sectors. Typical behavior is better than the worst case, and so consumption of storage will usually fall between $2E$ and $3E$; that is, between 1 and 2 extra copies of a data record are written on average. In the next section we show how data compression can reduce this amount.

For performance reasons, the bucket size should be chosen so that the fanout parameter F discussed above is large enough to keep the tree to three levels. Typically, this value of F will be so large that storage in the index part of the tree is a negligible part of the whole. Filling of data buckets will send pointer updates to the index, and thus many duplicates will occur in the sequence of keys sent to index buckets. Choosing TI close to the capacity of a bucket will result in extra copying in the index buckets but will increase fanout; high fanout reduces the required bucket size and hence the access time. Extra copying in the higher-level index buckets will cost little in comparison with the total space consumption. Therefore, a good value for TI is 80 to 90 percent of the capacity of a bucket.

Choosing TD close to the capacity of a bucket, however, may result in excessive storage consumption. An experimental study of the impact of TD values on consumption is described in a later section. Theoretical and experimental arguments suggest that a reasonable value of TD is 1/2 to 3/4 the capacity of a bucket.

Consider first the special case in which data records are never updated. Then a filled bucket will always be rewritten into two buckets (because a single bucket cannot hold the contents) and the value of $TD > 1$ has no effect. A simple argument (see Appendix B) shows that at any time after the first bucket has been rewritten, the number of active buckets is slightly greater than the number of inactive (discarded) buckets. Therefore, about half the disk space allocated to the WOBT holds active buckets when the disk becomes full (that is, when no tracks are available for new buckets). Assume that the keys arrive in random order. An active bucket will start out half full and so, at the time the disk is filled, about 3/4 of the space in the active buckets will be occupied. This leads to the conclusion that 3/8 of the data sectors contain active data records when the disk is filled. Therefore, about $2.7E$ sectors are consumed in storing E data records. In a conventional B-tree, on average, about 1/4 of each node is empty also at that time. Therefore, if a record is exactly the size of a sector, then the WOBT requires about double the storage used by a conventional B-tree.

If the WOBT is initially loaded from sorted records, then the additional storage consumption depends on the amount of updating since the initial load. Use of free space in the WOBT initial load should be more generous than with a conventional B-tree since updates must be accommodated.

The following result, which makes no assumption about the keys or update rates of the incoming records, provides a theoretical guide to selecting the value of TD. Let m be the number of sectors in a data bucket. In Appendix B, we show that at most (about) $4E$ sectors are consumed in storing E data records provided that $TD \leq 3m/4 + 2$. This bound holds regardless of the number of updates. The worst-case situation is approached by introducing data records in *descending* sorted order by primary key. Experiments with realistic as well as simulated workloads (see below) have shown that $2E$ to $3E$ is a reasonable estimate of consumption for randomly ordered keys and that storage consumption shows little sensitivity to the exact value of TD.

Data compression

Data compression provides a way to "buy back" some of the storage lost in reorganization without sacrificing the added value of on-line historical data. When a bucket is reorganized, the amount of data carried over to the new bucket or buckets is typically half a track or more. Compression techniques applied to 20000-byte blocks have been found typically to reduce size by factors from 2 to 3 [12]. One reason that such techniques are not commonly used in conventional database systems is that when a block is altered, its compression ratio changes and it may not fit back into the same storage location. With indelible data, however, the problem does not occur. The approximate calculation of the previous section is now extended. Again, assume that there are no updates and that keys arrive in random order. Also consider only the case where every reorganization produces two new buckets. If compression reduces the storage required by a factor $1/R$, then the fraction of space initially occupied in either first new bucket is $1/2R$. Copying carries forward the compressed entries, but a simple calculation shows that the fraction of initial space filled in a new bucket does not grow beyond $1/(1 + R)$. Once this value is reached, successive buckets have nearly identical initial occupancies.

Each bucket of m slots initially has at least $[1 - 1/(1 + R)]m = Rm/(1 + R)$ free entries. When the disk is filled, all the initial free slots in each discarded bucket have been consumed by new entries. About half the buckets in the tree are active, and about half the initial free slots in the active buckets have been consumed by new entries. Therefore, the average number of new entries per allocated bucket is at least $0.75Rm/(1 + R)$. For example, if $R = 2$, then on average at most 2 storage slots are consumed for each new data entry. This compares with an average of 8/3 when no compression is used.

Comparison with storage costs of conventional B-tree data sets

The amount of storage consumed by a WOBT is necessarily greater than that consumed by a conventional B-tree on an erasable medium. For the extra cost, we gain the advantage of additional on-line information. We now estimate this increase in storage requirement.

Let u be the fraction of data set entries that have new keys (that is, the fractions of entries that are inserts). It was argued above that storing D (distinct) records in a conventional B-tree requires approximately $4D/3$ storage spaces. With $D = uE$, the worst-case storage requirement for a WOBT is $4D/u$ sectors. Thus, if the record size exactly equals the sector size, then the WOBT storage required is at most $3/u$ times as great, and typically $2/u$ times as great.

If compression by a factor of 2 is achieved on the recopied blocks of data, then typical WOBT storage is reduced to $1.5/u$ times that required by conventional B-tree storage.

Space can also be saved, if the data records are larger than the index records, by not recopying data records. This results in extra access costs when retrieving records in key-sequence order.

Timestamps and historical data

Each index and data entry can be timestamped. The only requirement on the timestamps is that each data entry carry a timestamp that is later than that of the previous entry to the database. The timestamp for an index entry is that of the most recent data record. The retrieval procedures previously described can be used to perform the task *as of* a particular time and thus reflect the state of the database at any chosen past time T. The search starts at the bucket that was the root of the tree at time T. (To find the root that pertains to a

particular time, use the list of successive roots described under "Finding the root" below.) The only other change in the retrieval algorithms is that entries with timestamps exceeding T are ignored in the searched buckets. This is the same as saying that the search of each bucket halts when a record is found with a timestamp larger than T.

A general facility for retrieving successively older versions of a data record is obtained as follows. Include in each data bucket a pointer back to the bucket from whose reorganization it came. Then, after retrieving the current version of a record, it is easy to follow the pointers back to find each earlier version. The same technique can be used to find earlier versions of a particular index entry.

Copying to a new disk

If the storage medium is filled, then the current records (only) can be copied to a new disk using the procedure "Retrieve records in primary key-sequence order." The old disk can be kept on or off line as facilities permit. A new disk can also be created which will provide all historical data back to a specified time.

Secondary indexes

A write-once B-tree can be used to provide an index to secondary keys. Such keys need not be unique. The key SEC from the record whose primary key is PRIME is stored under the index entry SEC.PRIME. This makes possible deletion of specific instances of a secondary key by the method described above. All secondary keys with the same value are clustered in a minimum number of tracks by the insertion procedure and are quickly found. To obtain all instances of SEC.???, for example, begin by searching for key SEC and then continue to retrieve in key-sequence order.

Choosing bucket size

One basic assumption made at the start was that a bucket is a multiple number of tracks. This assumption is tied to the requirement that writing must be sequential within a track. For a device that has no such requirement, however, it may be possible to use an arbitrary number of sectors as the bucket size.

Finding the current root (or root as of time T)

Starting from track 1 (or another track that is agreed on by convention) a method must be provided by which a program rapidly locates the current root (or root as of time T) of the write-once B-tree. For most applications it is sufficient to keep a time-stamped log of successive roots, using linear or binary search to find the required entry. For the current root, this search need only be done when a new disk is mounted. During subsequent processing, the address of the current root is held in memory.

Concurrency control

The availability of earlier versions of records permits transactions to read while other transactions are updating. The development of appropriate concurrency controls is beyond the scope of this paper, but offers a promise of performance gains [3].

Performance summary

Through use of a value of TI close to that of the bucket capacity, we can achieve a fanout close to that of a conventional B-tree with the same bucket size. Thus the number of accesses for random insert or retrieval is about the same as if overwriting were allowed.

Search time within a bucket will be slower than with standard B-trees because of the need for some linear search. After a bucket has been copied, however, its initial entries are in sorted order. Binary search can thus be used on the old entries and linear search on the new entries within a bucket. Because of rapid advances in processor speeds relative to disk access times, the search cost should be a small part of the cost of operating the algorithm.

Summary of design features

The design was planned to minimize the need for updating. The policy of not deleting a data bucket's header key avoids the need for changing the index as a result of deletions. The avoidance of horizontal pointers eliminates the need to keep these pointers current as buckets are reorganized.

Control of storage space and fanout is accomplished through the reorganization algorithm. The use of a high threshold for copying an index bucket into *two* buckets keeps the fanout in the tree relatively high. Use of a somewhat lower threshold in the data buckets provides space for updates in newly copied buckets.

Finally, certain features of the design take into account application to write-once disks. In particular, there is no forward chaining of tracks or buckets using a pointer that is written into the track or bucket that is being chained. This avoids the case where an error occurs while the pointer is being written, and the space in the track or bucket is exhausted.

Experimental study

A write-once B-tree algorithm was implemented using magnetic disk as the storage device but treating the medium as indelible.

The first set of experiments was based on a workload from a possible application of write-once disks to archiving. The workload comes from a system that periodically copies all changed user files to tape. If the files were instead stored on write-once disk, then a WOBT could be used to maintain the index to these files on the disk as well. For the test of this assumed application, the workload was a chronological list of 3000 file names. This is the "File Names" workload.

A second workload was generated by the following method. Keys were selected at random from a space of values. The frequency of repetition of each key was chosen in accordance with Zipf's law [13, 14] which has been found to describe many empirical frequency distributions. In this case, the frequency of the ith most frequent key was $1/i$ times that of the most frequent key for $i = 1$ to 175. Each remaining key had a single appearance. The keys with these frequencies were sent in random order to the file.

A third workload ("Uniform") was generated by making three copies of each of 1000 keys and arranging the 3000 values in random order.

Table 1 shows for each workload the number of distinct keys, the number of occurrences of the most frequently appearing key, and the number of keys that appear exactly once.

In this experiment, each bucket (index or data) held exactly 30 entries and there were no write errors. **Table 2** summarizes the storage consumption found as a function of the value of parameter *TD*. The parameter *TI* was set at 25 for each case. While 100 data buckets would be needed to store the unorganized data as received, the WOBT used from 206 to 307 buckets. It is interesting that the storage consumption was not very sensitive to the value of *TD*. Although the number of buckets "split" (copied to two buckets) decreased with increasing *TD*, the total consumption remained fairly constant for a large range of values of *TD*. A setting of *TD* between 1/2 and 3/4 the capacity of a bucket gave about the best results in all cases. A setting closer to the capacity generally gave worse results. The relative insensitivity of storage consumption to the distribution of updates indicates the robustness of the technique.

In comparison with the theoretical arguments above, we found that the storage consumption was 2 to 3 times E, rather than the worst-case $4E$.

The tree in all cases consisted of a root, a second level of index, and the data level. **Table 3** gives the fanout results for the second level of the index, where fanout is the number of distinct entries in an index bucket. (The fanout at the root was typically only 4.) Again, we saw little sensitivity to distribution. In all cases, $TI = 25$, so the theoretical minimum fanout was 12. The observed values were 14.6 to 21.3.

Table 1 Numbers of distinct keys, occurrences of most frequently appearing key, and keys that appeared exactly once for three workloads. Each workload consisted of 3000 keys.

Workload	*Number of distinct keys*	*Number of appearances of most frequently appearing key*	*Number of keys that appear once*
File names	2300	10	1879
Zipf distribution	1253	349	1078
Uniform distribution	1000	3	0

Conclusions

We have described methods for indelible management of key-sequence data sets. The performance, measured in terms of number of seeks and reads, is about the same as for conventional B-tree management.

Storage consumption depends on the update rate and involves an additional cost because information is rewritten to maintain data records in key-sequence order and to keep index records clustered for efficient retrieval. Compared with conventional structures, the ratio of storage consumed is about $1.5/u$ to $2/u$, where u is the fraction of entries that have new keys. The storage requirement can be reduced by using better data compression, by not recopying data records and thus accepting longer retrieval times for records in key-sequence order, and by sorting the initial load of data records before entering them.

Depending on the update rate and the cost of storage, it appears that indelible key-sequence data sets will be practical

Table 2 Storage consumption as a function of *TD*.

Workload	*TD*	*Active data buckets*	*Data buckets split*	*Data buckets copied*	*Total data buckets*
File names	6	134	133	0	264
	15	131	130	3	264
	24	126	125	16	267
	28	120	119	65	304
Zipf distribution	6	97	96	14	207
	15	85	84	37	206
	24	71	70	99	240
	28	62	61	184	307
Uniform distribution	6	117	116	0	233
	15	99	98	18	215
	24	61	60	110	231
	28	51	50	204	305

Table 3 Fanout results for second level of index, where $TI = 25$.

Workload	*TD*	*Average fanout in second level of index*
File names	6	16.8
	15	16.4
	24	15.8
	28	15.0
Zipf distribution	6	16.2
	15	21.3
	24	17.8
	28	15.5
Uniform distribution	6	14.6
	15	16.5
	24	15.3
	28	17.0

for many applications. The high density of optical storage and advanced magnetic recording may lead to keeping on line historical data that formerly were relegated to archives.

Appendix A: Write-once disk with no demarking

We discuss methods for implementing the WOBT scheme on a write-once disk in the absence of a "demarking" capability for bad writes. The crucial problem is, on encountering a read error, to distinguish between a previously discarded sector and one that was valid when written. In the former case, the error can be ignored. In the latter case, the operation must be retried.

One strategy for handling write errors is to copy the entire contents of the bucket where the error occurred to a new bucket and then write the new bucket's address in the index. An advantage of this approach is that discarded sectors are never subsequently read. The use of powerful error-correcting codes may keep the write error rate sufficiently low so that little space is wasted by this approach.

Another approach does not discard buckets. Instead, we rewrite a bad sector in the next position of the bucket if there is room. (Otherwise, the bucket is "full.") Each sector written into a bucket contains a bucket sequence number (BSN). The BSNs are 1, 2, 3, $\cdots$, increasing by one for each sector written successfully to the bucket. On reading a bucket, read errors on all but the last written sector in the bucket can be ignored so long as there are no gaps in the BSNs read. A read error on the last sector of a bucket that is not recovered by retries indicates a nonrecoverable error.

Appendix B: Upper bound on storage consumption

Assume that a reorganized data bucket in a WOBT is copied to a single node if it holds fewer than *TD current* entries and is copied to two nodes otherwise. (An *entry* is a stored record with its key.) There are no deletions of entries in the WOBT.

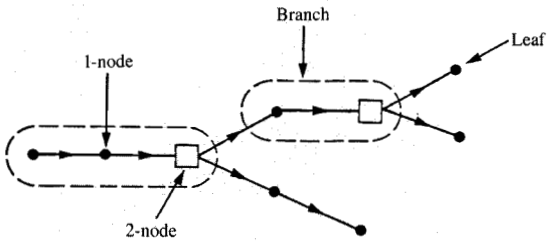

Figure 6

Example of graph showing movement of bucket contents after reorganization.

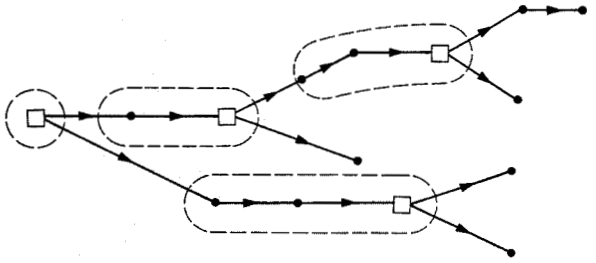

Figure 7

Another example of graph showing movement of bucket contents after reorganization.

Let each data bucket hold m entries, and let there initially be a single data bucket.

Claim

If E data entries (including the initial dummy entry) are stored in the WOBT, then at most $\lceil 4E/m \rceil$ data buckets are allocated if $TD \leq (3m/4) + 2$.

Proof Represent the initial data bucket as the root node in a directed graph. When a bucket is reorganized, draw an edge from its node to the node(s) representing the new bucket(s). Examples of resulting graphs are shown in **Figures 6** and **7**. In these graphs, the nodes having two descendents are shown as boxes.

A node having a single descendent is called a *1-node*. A node having two descendents is called a *2-node*. Let K be the number of 1-nodes and let N be the number of 2-nodes. The sequence of 1-nodes that precedes a 2-node, along with the 2-node, is called a *branch*. Each branch in Figures 6 and 7 is encircled with a dotted line. A node with no descendents is a *leaf* node. Let L be the number of leaf nodes.

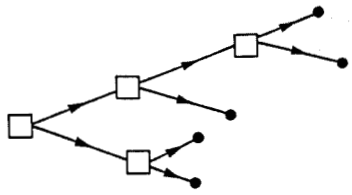

Figure 8

Condensed graph of Figure 7.

Consider first the case in which no 2-nodes appear in the graph. This means that after every reorganization there are at least $m - TD + 1 \geq (m/4) - 1$ empty entries in the one new bucket written. Then at least $m/4$ new entries are included in the $m + 1$ entries that take part in the later reorganization of this new bucket. Thus, by induction, every reorganized bucket accounts for at least $m/4$ new entries. This completes the proof for the case of no 2-nodes in the graph.

Now assume that there is at least one 2-node in the graph. If we sequentially remove each 1-node and connect its descendent node, if any, to its ancestor node, then we end up with a nonempty binary tree, the *condensed* graph. (We also remove the root node if it is a 1-node.) **Figure 8** shows the condensed graph resulting from the graph of Figure 7. The leaf nodes represent all the data buckets of the WOBT that are "active." A simple inductive argument for binary trees shows that

$$L = 1 + N. \tag{1}$$

The contents of every 2-node bucket includes the records passed to its branch by its ancestor 2-node bucket (if one exists). By definition of reorganization, the keys of these records are distinct. Let A be the number of such keys in all 2-nodes. Every 2-node bucket also contains initially the new keys stored into its branch buckets (excluding the branch buckets' initial contents) or added to branch buckets during reorganization. Let S be the number of such keys in all 2-nodes. Let U denote the number of new keys (excluding initial contents) stored into all 2-nodes or added to 2-node contents before reorganization. (Reorganization occurs when an entry sent to a bucket finds the bucket full. The new entry is included in the reorganization.) Let R be the number of repeated (update) keys stored into a 2-node or added to 2-node contents before reorganization.

By definition, every 2-node has been reorganized. An entry typically is first stored in some bucket, then copied several times as each successive bucket is reorganized. Each of the $m + 1$ entries associated with 2-node X's reorganization satisfies one and only one of the following conditions:

1. The entry was first stored prior to creation of X's branch.
2. The entry was first stored in one of X's branch 1-nodes (or added to such a node before its reorganization).
3. The entry was first stored into X itself (or added to X before its organization) and was the first entry to appear in X with that key.
4. The entry was first stored into X (or added to X before its reorganization) and repeated a key that already appeared in X.

From this it follows that

$$N(m + 1) = A + S + U + R. \tag{2}$$

The contents of a reorganized bucket may include as many as $m + 1$ distinct keys. Because of the even splitting, at most $1 + m/2$ keys are passed from an ancestor 2-node bucket to the branch. Therefore, since one 2-node is the root of the tree, $A \leq (N - 1)(1 + m/2)$ and hence, by (2),

$$N + 1 < 2(S + U + R)/m. \tag{3}$$

Let R' be the number of repeated keys that are stored into 1-node buckets or added to 1-node bucket contents when reorganized. A 1-node bucket, by definition of the copy rule, holds at most $TD - 1$ distinct keys when it is reorganized, and so at least $m - TD + 2$ repeated keys are eliminated when a 1-node is reorganized. Therefore,

$$K \leq R'/(m - TD + 2). \tag{4}$$

Applying (1)–(4), we find that the total number of nodes in the original graph is $K + L + N = K + 2N + 1 < R'/(m - TD + 2) + 4(S + U + R)/m$. By assumption $TD \leq 3m/4 + 2$, that is, $m - TD + 2 \geq m/4$, and therefore

$$K + L + N < 4(S + U + R + R')/m \leq 4E/m. \tag{5}$$

This completes the proof.

• *Comment 1*

With the substitution of TI for TD, the result applies to any non-root index level of the tree. In this case, E refers to the number of index entries inserted including the dummy entry.

• *Comment 2*

We briefly discuss the tightness of the bound and worst cases. Consider the last inequality in (5). There are certain entries in the WOBT not counted in the sum $S + U + R + R'$. Equality in (5) occurs if and only if no entries have been made in any leaf node since that node's creation and no new key has been stored in any 1-node that does not belong to a branch. In practice, however, leaves of the tree will contain new entries so the bound will be pessimistic.

For example, suppose there are no repeated keys in the entries (that is, no updates). Then, just after reorganization, a node bucket holds about $m/2$ entries. If there is a random

distribution of incoming keys into leaves, then the average number of entries in a leaf bucket is about $3m/4$. If L is the number of leaves, then $3mL/4$ is approximately the number of entries made to the tree. By (1), approximately $2L$ data buckets are consumed. For this case, about $2.67E/m$ data buckets are consumed if E is the number of entries to the tree.

On the other hand, suppose that the sequence of inserted keys arrives in reverse sorted order and there are no updates. Then, after reorganization of a bucket, the new bucket holding the higher keys from the old bucket will never again have anything stored into it. Moreover, this bucket will be half full and will remain active. The new bucket holding the lower keys from the reorganization will ultimately be filled and discarded. Thus, approximately 1/4 of the space allocated will contain active data and the worst-case bound will be approached.

The conclusion from this discussion is that the bound can be approached, but for randomly arranged keys with no updates, the average consumption will be approximately $2.67E/m$.

References

1. A. E. Bell, "Optical Data Storage Technology Status and Prospects," *Computer Design*, pp. 133–146 (January 1983).
2. G. Copeland, "What If Mass Storage Were Free," *Computer*, pp. 27–35 (July 1982).
3. L. Svobodova, "Management of Object Histories in the Swallow Repository," *Report No. MIT/LCS/TR-243*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 1980.
4. D. Maier, "Using Write-Once Memory for Database Storage," *Proceedings of the ACM Symposium on Principles of Data Base Systems*, March 29–31, 1982, pp. 239–246.
5. R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Inform.* **1,** 197–189 (1972).
6. J. S. Vitter, "An Efficient I/O Interface for Optical Disks," *Technical Report No. CS-84-15*, Department of Computer Science, Brown University, Providence, RI, June 1984.
7. M. C. Easton, "Method for Dynamic Data Management on Write-Once Disk," *Invention Disclosure No. 8830109*, IBM Research Division, San Jose, CA, March 1983.
8. P. Rathmann, "Dynamic Data Structures on Optical Disks," *Proceedings of the IEEE Data Engineering Conference*, Los Angeles, CA, April 1984.
9. R. L. Rivest and A. Shamir, "How to Reuse a 'Write-Once' Memory," *Proceedings of the 14th Annual STOC Conference*, San Francisco, May 1982. See also *Computer*, pp. 27–35 (1982).
10. D. E. Knuth, *The Art of Computer Programming*, Vol. 1 (2nd ed.), Addison-Wesley Publishing Co., Reading, MA, 1973.
11. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing—A Fast Access Method for Dynamic Files," *ACM Trans. Database Syst.* **4,** No. 3, 315–344 (September 1979).
12. G. G. Langdon, Jr. and J. J. Rissanen, "A Double-Adaptive File Compression Algorithm," *IEEE Trans. Commun.* **COM-31,** No. 11, 1253–1255 (November 1983).
13. D. E. Knuth, *The Art of Computer Programming*, Vol. 3, Addison-Wesley Publishing Co., Reading, MA, 1973.
14. G. K. Zipf, *Human Behavior and the Principle of Least Effort*, Addison-Wesley Publishing Co., Reading, MA, 1949.

Received August 2, 1985; accepted for publication December 26, 1985

Malcolm C. Easton *IBM Research Division, 5600 Cottle Road, San Jose, California 94304.* Dr. Easton is a Research staff member in storage systems and technology. He received a B.S. from Massachusetts Institute of Technology, Cambridge, in 1964, and a Ph.D. in applied mathematics from the State University of New York, Stony Brook, in 1973. From 1973 to 1980, he was with the Department of Computer Science, IBM Thomas J. Watson Research Center, Yorktown Heights, New York. For the 1979–1980 academic year, he was on sabbatical leave to Stanford University, California, where he was Consulting Associate Professor of Electrical Engineering. Dr. Easton received an IBM Outstanding Technical Achievement Award in 1982 for his development of novel data analysis techniques.