A RELIABLE AND DEADLOCK-FREE MULTI-INDEXED B^+-TREE

Arthur M. Keller

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-86-19 July 1986

A Reliable and Deadlock-Free Multi-Indexed B^+-Tree

Arthur M. Keller
University of Texas at Austin

ABSTRACT. B-trees are a mainstay of relational database implementation. Because multiple indexes may be desired on a single relation, a multi-indexed B-tree is needed. Our goal was to design a disk data structure and associated algorithms for a multiple B-tree structure that was reliable, deadlock-free, and reasonably efficient. Our system also supports variable-length keys and variable-length records. An arbitrary number of B-tree indexes can be defined, each specifying one or more fields; a field can be part of more than one index. We separated the dichotomy between primary and secondary indexes into the choice of at most one clustered index and the ability to decide whether any particular index is unique or allows duplicates. This requires a careful balance of these conflicting needs in the design of a synergistic system. Often design decisions made originally for one goal were found to be beneficial for another goal, or even had unexpected benefits. Our design engineers the combination of several techniques to create a workable solution to the problem of multiple B-tree indexes for a single file of data.

1 Introduction

B-trees present sequential and keyed access to a collection of data [Bayer 72, Comer 79]. Records may be accessed in order based on the values of some key or a specific record may be found that has a particular key value. Hashed data structures may provide faster access to individual records by key, but they typically do not provide ordered sequential access.

A multiple B-tree index structure permits access to records based on the values of multiple keys. This approach is prone to deadlock because the data structure may be entered through any index and updating a record may require updating several, maybe even all, indexes. Through a combination of techniques, we have designed data structures and algorithms that overcome this problem. An implementation, FLASH, has been built that includes most of the features described in this paper [Allchin 80]. We will call a package supporting these algorithms IFAP: Indexed File Access Package. A formal specification exists for this as an Ada package [Keller 86].

There has been much interest in concurrent operations on B-trees [for example, Comer 79, Lehman 81]. Our work differs from previous work in that we consider multiple B-trees referencing the same collection of data. We use the pre-splitting scheme to split index blocks in anticipation of subsequent need.

We were particularly interested in reliability. In particular, the structure had to be crash-proof. Based on the assumption that a disk block is either correctly and completely written or not at all, our algorithms cannot corrupt a single B-tree. They can cause blocks to be lost rather than returned to the free block pool or one or more B-trees to become unsynchronized from the data. This last issue can be fixed during system restart if we maintain a list of such update operations pending.

2 Protocol for Access

We access blocks using a protocol that prevents deadlock. The principles of this protocol are as follows.

1. All locks on blocks are for exclusive access. Consequently, we never need to promote locks on blocks from shared to exclusive.

This work was started while the author was at the Computer Science Department of Stanford University. This work was supported in part by contract N39-84-C-0211 (the Knowledge Base Management Systems Project, Prof. Gio Wiederhold, Principal Investigator) from the Defense Advanced Research Projects Agency of the United States Department of Defense, and by the Computer Sciences Research and Development Fund and the University Research Institute of The University of Texas at Austin. The views and conclusions contained in this document are those of the authors and should not be interpreted as representative of the official policies of DARPA, the US Government, or the State of Texas.

Author's address: The University of Texas at Austin, Department of Computer Sciences, Austin, TX 78712-1188.

2. Moving a block requires that its parent block remain locked during the move. When a block is only reachable through its parent, locking the parent block prevents following a dangling pointer.
3. When following a pointer to a block that may move, we retain the lock on the parent block until the child block is locked. Then, if it is not necessary to move the child block, we can release the lock on the parent block.
4. When following a pointer to an anchored block—a block that is not allowed to move—the parent block may be unlocked before the child block is locked.
5. You can wait for a lock only if it is later in the list than all the locks being held.
6. If you cannot acquire a lock, and are not allowed to wait for it, you must restart your operation at an anchored block, such as the root block of an index.
7. If a block that is moved has multiple pointers to it, it must remain locked until all pointers to it have been updated.

These principles are used to access the data structures. The data structures and the algorithms have been specifically designed to maintain a consistent data structure that survives system failure and is deadlock-free for physical block access. In particular, we avoid the circular wait condition for deadlock.

3 Individual B$^+$-Tree Indexes

Each B-tree index has the same structure (see Figure 1) and protocol for access. When reading the B-tree to find a record, we use the following algorithm.

```
Set current block to root of index
Lock root block
WHILE not leaf index block
    Read current block
    Set next block to address of desired child block
    Lock next block
    Unlock current block
    Set current block to next block
Read data block (which is now the current block)
```

Algorithm 1. Find Data Block for Reading

Each block is locked before it is read. Because a block may be moved, we keep the lock on the parent block until we have locked the child block. This is compatible with our protocol for moving blocks, which involves locking both the block being moved and the parent block which points to it. Because we are acquiring locks in a defined order, we can assure deadlock-free access.

When updating records, especially when inserting new records, it may be necessary to insert new pointers in index blocks. The potential exists to cascade index block splits up the tree all the way to the root if each block does not have room for an additional pointer. Not only does this occasionally involve bad performance as index blocks are split, but it also incurs the possibility of deadlock when a writer splitting block marching up the tree encounters another process marching down the tree. To avoid the march up the tree when splitting index blocks, we presplit index blocks that are full on the way down. This is illustrated in Figure 2. This algorithm follows.

```
Lock root block
Read root block
Set parent block to root block
Set current block to desired child block
WHILE current block not a data block
    Lock current block
    Read current block
```

```
   IF there is not enough room in current block for a new pointer
      THEN Obtain two new blocks
           Place half of current block in each new block
           Write both new blocks (Figure 2, Step 1)
           In parent blocks, replace pointer to current block with
              pointers to new blocks
           Rewrite parent block in place (Figure 2, Step 2)
           Place current block on the free block list (Figure 2, Step 3)
           Set current block to the new block with the correct key range
           Unlock other new block (Figure 2, Step 4)
      Unlock parent block
      Set parent block to current block
      Set current block to address of desired child block
Lock current block (the data block)
Read data block
```

Algorithm 2. Find Data Block for Updating

There are several important observations about this algorithm. First, it preserves the ordering of accessing blocks that prevents deadlock. Second, the order in which blocks are locked, read, and unlocked means that the splitting of blocks does not endanger the reading of blocks as no wild pointers are followed. Third, when a block is split, the blocks are written in an order that preserves the integrity of the tree structure even if the operating system crashes during the process, provided that a block is either entirely written or not changed at all. In case of a crash, the worst that could happen is that there are blocks in the file that are not referenced by the index structure, but all the data blocks are still reachable and the index structure is still intact. The non-referenced index blocks can be reallocated by a clean-up utility that audits the data structure.

A different protocol is needed when splitting the root index block of a B-tree so that the root doesn't move. This is illustrated in Figure 3. This algorithm follows.

```
Lock index block
Read index block
Set parent block to root index block
IF there is insufficient room for another pointer in root index block
   THEN Obtain two new blocks
        Copy half of root index block into the two new blocks
        Write the two new blocks
        Change root index block to point only to the two new blocks
        Rewrite root index block
        Set parent block to the new block with the correct key range
        Lock parent block
        Unlock root index block
Set child block to desired child block of parent child
Proceed with WHILE loop of Algorithm 2
```

Algorithm 3. Split Root Index Block

The split root index block algorithm (Algorithm 3) entails the same three observations as for Algorithm 2. The algorithm for splitting data blocks will be covered in the next section.

4 Clustered vs. Non-Clustered Indexes

While there are a lot of similarities between treating clustered and non-clustered indexes, there are important differences. There may be at most one clustered index per file. Any index may be unique (require distinct data values) or non-unique regardless of whether the index is clustered.

Since clustering indexes point directly to data blocks, finding a data record by key to read or write requires access to $O(\log n)$ blocks; however, several alternatives exist for non-clustered indexes. The operations on data blocks are: find data block through index (clustered or non-clustered), add new record to data block, rewrite or delete record from data block, and split data block.

We will compare three alternatives for referencing data records from non-clustered indexes. The index may point directly to the data block; the index may refer to the key value of the clustered index (if it is unique); or the non-clustered index points to an indirect-block which points to the data block. For the specified operations, Table 1 gives the number of blocks read or changed for each alternative, where n is the number of data records, m is the number of data records moved, i is the number of indexes, and j is the number of index entries affected.

Operation	Direct Pointer	Indirect Pointer	Pointer Through Clustered Index
Find data block through clustered index	$O(\log n)$	$O(\log n)$	$O(\log n)$
Find data block through non-clustered index	$O(\log n)$	$O((\log n)+1)$	$O(2\log n)$
Add new record to data block	$O(i \log n)$	$O(i \log n)$	$O(i \log n)$
Rewrite or delete data record	$O((j+1)\log n)$	$O((j+1)\log n)$	$O((j+1)\log n)$
Split data block	$O(m \log n)$	$O(m)$	$O(1)$

Table 1. Comparison of Costs of Some Operations on Data Blocks and Data Records

We observe that using an indirect pointer is only slightly more expensive for finding a data block through a non-clustered index than a direct pointer while being considerably cheaper than a pointer through the clustered index. We also observe that using an indirect pointer is intermediate in cost for splitting a data block. The indirect pointer method has advantages not only in cost but also in reliability and freedom from deadlocks, as we will soon see. We will proceed to describing the data structures and algorithms for connecting data blocks to the index structures.

Figure 4 illustrates the data structures for data blocks. The data blocks are directly referenced by the leaf index pages of the clustered index. The leaf index pages of the non-clustered indexes point to blocks containing one record per key value, where the record contains a list of tuple IDs for the records with that index value. These are called record list blocks and their format is similar to that of data blocks. Tuple IDs are permanently associated with records and do not change when records are updated, even when the clustered index value changes. Physically a tuple ID consists of two parts: the block number of the tuple ID block and the slot number within that block. Tuple IDs are assigned sequentially from tuple ID blocks. Thus, to assign a new tuple ID, it is only necessary to know the last tuple ID assigned. When a record is deleted from the file, its tuple ID is turned into a tombstone for the record; rather than being reused, it contains a value that indicates the record was deleted. Ordinarily, tuple ID pointers in the tuple ID block contain the address of the block containing the record. Since data records are preceded by a header that contains the tuple ID, the data block may be searched for the desired record. Since tuple IDs are essentially physical pointers, tuple IDs may be used as a fast access path for records; they are suitable for use as links between relations in a relational database. Also because tuple ID blocks never move, it is not necessary to hold onto the lock of a leaf index block when reading a tuple ID block. (Recall that we held onto the lock of a parent block until we have locked the child block when we cannot otherwise guarantee that the child block will not move.)

Now that we have explained the data structures connecting data blocks to the indexes, we proceed to algorithms for the operations on data records. Algorithm 4 describes the process of reading a data record

using the clustered index.

```
Perform Algorithm 1 on clustered index
(The parent block is not now locked)
Search data block sequentially for desired record
```

Algorithm 4. Reading Data Record Through Clustered Index

Algorithm 5 describes the process of reading a data record through a non-clustered index. There are several important observations about this algorithm. We have already observed that we may unlock the record list block once we have obtained the tuple ID. We do not keep any locks while waiting for another lock. This is how we prevent deadlock in this portion of the data structure. When we get to Algorithm 6, we will see that updates may require traversing from the data block to the tuple ID block, the opposite direction from Algorithm 5. Also note that if the record has moved to another data block while we were waiting for its old block, we repeat the loop still holding onto the lock of the tuple ID block. If the record was deleted while waiting for its old block, we need to retry the entire operation at the beginning, probably waiting a short time for the index to become consistent by a concurrent writer.

```
Perform Algorithm 1 on non-clustered index
(The leaf index block is not now locked)
Search record list block for desired key value
Obtain tuple ID
Release lock on record list block
Perform Algorithm 6 to read data block using tuple ID
Release lock on tuple ID block
Scan data block for record with correct tuple ID
```

Algorithm 5. Reading a Data Record Through a Non-Clustered Index

Reading a data record given its tuple ID is described in Algorithm 6.

```
REPEAT
    Obtain lock on tuple ID block (wait if necessary)
    Read tuple ID block
    Find address of data block
    Try to lock data block
    IF cannot lock data block
       THEN Release lock on tuple ID block
            Wait for lock on data block
            IF Succeeds
               THEN Check that tuple ID still refers to same data block
                    IF tuple ID is a tombstone
                       THEN release locks and GOTO start of B-tree
                            package (retry operation)
                    IF Same
                       THEN Continue below after UNTIL
                       ELSE Release lock on data block
               ELSE Release lock on data block
UNTIL both tuple ID block and data block are locked
```

Algorithm 6. Reading a Data Record Using Its Tuple ID

5 Updating Records

There are three operations for updating records: replace, delete, and insert. For replace and delete, we require that the record have been read with intention to modify. The tuple ID is used to identify the record. For replace, all fields may be changed, including those associated with the clustered index. Records may also be inserted, and a new tuple ID is assigned unless one for a deleted record is supplied. (This allows rollback of a deletion performed by a transaction subsequently aborted.) Insertions are done by traversing the clustered index to place the record and then the other indexes are corrected. Deletions are done by locating the record using the tuple ID, removing it from its data block, and then fixing the indexes. Replacement is done by using the tuple ID to locate the record, updating it, and then fixing the indexes. However, if the clustered key changes, it is deleted from its block using the tuple ID, inserted using the new cluster key, and then the indexes are updated. Replacement and insertion may require a data block to be split to hold the new record. This would require the leaf block of the clustered index to be updated. If this block is already locked, we can update it. Otherwise, locking it would involve traversing the clustered index the wrong way with the potential for deadlock. We use a different algorithm when the clustered index leaf block is not locked. Algorithm 7 describes the process of splitting a data block when the block and its clustered index leaf block are both locked.

```
Obtain 2 new data blocks (or as many as it takes to hold all the records)
Copy half of the records including new record into each new block
Write the new blocks
Rewrite the leaf index block so that it points to the new data blocks
Make a list of tuple ID pointers that need to be changed
Release locks on new data blocks as well as index block, but retain
    lock on old data block
FOR each tuple ID pointer that needs to be changed
    Lock tuple ID block (waiting if necessary)
    Read, change, and rewrite tuple ID block
    Release lock on tuple ID block
```

Algorithm 7. Splitting a Data Block Through Clustered Index

When the clustered index leaf block is not locked or when a record is too big to fit in one block, we create a train of blocks. This is shown in Figure 4. The train of blocks is treated as one large block. Only the first block in the train needs to be locked; locking it locks the entire train. Successive blocks in the train are linked. Either the first block points to all the other blocks, or they are in a singly linked list. This approach is also used for the list of records for non-clustered indexes when they do not fit in one block. Algorithm 8 describes the processes of splitting a data block when the clustered index leaf block is not locked. We assume the data block is locked. There is no need to update the tuple ID blocks as all records are still reachable through the first block in the train.

```
Obtain a new data block
Copy records into new data block
Write new data block
Remove duplicate records from old data block
Rewrite old data block
```

Algorithm 8. Splitting a Data Block Not Through Clustered Index

Inserting a new record is described in Algorithm 9.

```
Perform Algorithm 2 on clustered index
Obtain tuple ID
IF there is enough room in data block for new record
   THEN Add record to data block
        Rewrite data block
```

```
   ELSE Perform Algorithm 7 to split data block
Unlock parent block
Lock tuple ID block (waiting if necessary)
Place pointer to data block in tuple ID slot
Rewrite tuple ID block in place
Unlock data block and tuple ID block
FOR each non-clustered index
    Perform Algorithm 10 to insert new key in index
IF uniqueness test fails on any index
   THEN Delete record using Algorithm 12
        Return with error status
```

Algorithm 9. Insert New Record

There are several interesting observations about Algorithm 9. Before accessing any of the non-clustered indexes, we release all physical locks. This prevents deadlock. Also, we may not discover a duplication in a unique index until after we have inserted the record. As we have not yet returned to the caller, we may delete the record using Algorithm 12. To prevent another user from finding that record, we have to use logical locks on records. The distinction between logical locks and physical locks is described in Section 7 below.

Algorithms 10 and 11 maintain non-clustered indexes. Algorithm 10 inserts a key into a non-clustered index and Algorithm 11 removes a key from a non-clustered index.

```
Perform Algorithm 2 on non-clustered index
IF key and tuple ID are already in index
   THEN that is ok; proceed to next index
IF key already exists in index
   THEN IF index is unique
           THEN Proceed to undo request
           ELSE Add new tuple ID to end of list
                IF it fits
                   THEN Rewrite record list block
                   ELSE Perform Algorithm 7 to split record list block
   ELSE Create a new list
        Assign a new tuple ID to the list
        IF the new list fits
           THEN Rewrite record list block in place
           ELSE Perform Algorithm 7 to split second list block
        Release all physical block locks
        Lock tuple ID block
        Read tuple ID block
        Set tuple ID slot to point to new record list
        Rewrite tuple ID block in place
        Release lock on tuple ID block
 Release all remaining physical locks
```

Algorithm 10. Insert a Key and Tuple ID Into Non-Clustered Index

```
Perform Algorithm 2 on non-clustered index
Find key and tuple ID in record list
IF either does not exist
   THEN that is ok; proceed to next index
IF only one tuple ID in record list
   THEN Remove record list
        Rewrite record list block
        Release all physical block locks
        Lock tuple ID block for record list (wait if necessary)
        Read tuple ID block
        Mark tuple ID for record list deleted
        Rewrite tuple ID block in place
        Release lock on tuple ID block
   ELSE Remove tuple ID from record list
        Rewrite record list block
Release all remaining physical block locks
```

Algorithm 11. Remove a Key and Tuple ID From a Non-Clustered Index

There are several reasons that we assign tuple IDs to record lists. The format of the record list blocks are now identical to that of data blocks. Also if we are reading the records sequentially through a secondary index, we can find the next record with the same key by using the tuple ID of the record list rather than by searching the non-clustered index again.

Algorithm 12 describes deletion of a record using its tuple ID.

```
Perform Algorithm 6 to read data block
Remove record from data block
Rewrite data block in place
Unlock data block
Mark tuple ID deleted in tuple ID block
Rewrite tuple ID block in place
Unlock tuple ID block
FOR each non-clustered index
    Perform Algorithm 11 to remove key and tuple ID from index
```

Algorithm 12. Deletion of a Record Using Its Tuple ID

The algorithm for replacing a record depends on whether the clustered index key changes. Algorithm 13 describes the process if the clustered index key does not change; Algorithm 14 describes the process when it does. Note that we add the new index entries before deleting the old ones so that we will have less work to undo if we find a duplicate in a unique index.

```
Perform Algorithm 6 to read data block by tuple ID
Unlock tuple ID block
Remember old record
Replace it with new record, performing Algorithm 8 if necessary
Rewrite data block
Unlock data block
FOR each non-clustered index that changed
    Insert new key using Algorithm 10
IF failure due to duplicate entry in a unique index
   THEN Perform Algorithm 6 to read data block by tuple ID
        Unlock tuple ID block
        Restore old record
```

```
        Rewrite data block
        Unlock data block
        FOR each non-clustered index we have changed
            Remove new key by using Algorithm 11
        Return with error status
FOR each non-clustered index that changed
    Remove old key by Algorithm 11
```

Algorithm 13. Replace Record When Clustered Key Does Not Change

```
Perform Algorithm 6 to read old record by tuple ID
Remember old data record and tuple ID
Read old tuple ID block
Mark old tuple ID deleted
Rewrite old tuple ID block
Unlock old tuple ID block
Obtain new tuple ID for old record
Lock new tuple ID block
Change tuple ID of old record to new tuple ID
Rewrite old data block in place
Read new tuple ID block
Set tuple ID pointer to refer to old data block
Rewrite new tuple ID block in place
Release all physical block locks
Insert replacement record using Algorithm 9 and old tuple ID
IF insert fails due to duplicate entry in clustered index
   THEN Perform Algorithm 6 to read old record using new tuple ID
        Mark new tuple ID as deleted
        Rewrite new tuple ID block
        Unlock new tuple ID block
        Lock old tuple ID block
        Change tuple ID of old record to old tuple ID
        Rewrite old data block in place
        Read old tuple ID block
        Set old tuple ID to old data block
        Rewrite old tuple ID block
 Release all physical block locks
        FOR each non-clustered index changed (only for changed fields)
            Remove new key by Algorithm 11
        Return with error status
 Perform Algorithm 6 on old data record using new tuple ID
 Remove record from data block
 Rewrite data block in place
 Unlock data block
 Mark new tuple ID deleted in new tuple ID block
 Rewrite new tuple ID block in place
 Unlock new tuple ID block
 Release all remaining physical block locks
 FOR each clustered index key that changed
     Perform Algorithm 11 to remove key and old tuple ID from index
```

Algorithm 14. Replace Record While Changing Clustered Index Key

There are several important observations about Algorithm 14. To prevent deadlock, we do not hold any physical locks when we access a non-clustered index. We also do not remove the old record nor the old index entries until the insertion of the new ones succeed. This reduces the amount of undo/redo work needed when a replacement fails due to a duplicate entry in a unique index. Were the old index entries to be prematurely deleted, it might be impossible to restore them if another conflicting index entry was inserted in the interim. We use a temporary tuple ID for the replaced record so we can find it again in case the data block is split and the record is moved. Since the tuple IDs for replaced records are short-lived and internal, they may be assigned from a special pool of tuple IDs that are reused.

When a train of blocks is formed by Algorithm 8, we spawn a process to perform Algorithm 15 to convert the train into individual blocks.

```
Perform Algorithm 2 to find train through clustered index
Perform Algorithm 7 to split train into ordinary data blocks
```

Algorithm 15. Clean Up Trains Formed by Splitting Data Blocks not Through Clustered Index

6 Files Without Clustered Indexes

The algorithms presented in preceding sections assume that there is a clustered index. If the file does not have a clustered index, several of the algorithms change. The revised data structure is shown in Figure 5.

The primary differences affect insertion and block splitting algorithms. The insertion algorithm (Algorithm 9) now uses any index, preferably a unique one, and places the new record in any data block with sufficient room. This is shown in Algorithm 16. The block splitting algorithms (7 and 8) no longer need be concerned with the clustered index leaf block. This is shown in Algorithm 17. We will not list the other algorithms that require adaptation.

```
Perform Algorithm 10 on an index
Obtain tuple ID
Obtain a data block for new record
Lock data block
Read (if necessary), add record to, and rewrite data block
Lock tuple ID block (waiting if necessary)
Set tuple ID slot to point to data block
Rewrite tuple ID block in place
Unlock all physical block locks
FOR each other index
    Perform Algorithm 10 to insert key and tuple ID
IF uniqueness test fails on any index
   THEN Delete record using Algorithm 12
   Return with error status
```

Algorithm 16. Insert Record When There is no Clustered Index

```
Obtain new data block for new record
Move new record into new data block
Write new data block
Lock tuple ID block for new record
Read tuple ID block
Set tuple ID slot to point to new data block
Rewrite tuple ID block in place
Unlock tuple ID block
Remove old record from old data block if necessary
Rewrite old data block in place if necessary
```

Algorithm 17. Splitting a Data Block When There is no Clustered Index

7 Physical vs. Logical Locks

We use physical locks on blocks and logical locks on records. The physical locks are by block address; the logical locks are by tuple ID. We will first show that concurrent operations involving physical locks is deadlock free. We will then consider logical locks.

THEOREM. The algorithms described above for physical locks are deadlock free.

PROOF. Each index is treated separately. We never have blocks locked from more than one index at a time for any one request. The blocks are locked in indexes according to a partial order that may be extended to a total order. With the exception of tuple ID blocks, there is no circularity in lock acquisition, which is one condition for avoiding deadlock.

There is some circularity involving data blocks and tuple ID blocks. However, the following observations about our locking discipline will show that there is no possibility of deadlock on physical locks. Only one tuple ID block may be locked at a time. We can wait for a tuple ID block. But if a tuple ID block is locked, we can attempt to acquire but cannot wait for a lock on a data block. Therefore, tuple ID blocks appear later in the ordering than data blocks, and we are not allowed to wait for a lock earlier in the ordering than locks we hold. If we can acquire a data block lock while holding a tuple ID block lock, we can proceed with impunity, but if the data block is already locked, we must give up our lock on the tuple ID block in order to wait for the lock on the data block. Since there can be no circularity in the 'wait-for' graph, no deadlock on physical locks is possible. ∎

Locks on physical blocks are held only while manipulating the disk data structure. In between calls to the IFAP, no physical locks are held. Instead, logical locks on records may be held. Much literature exists on locking approaches and strategies. We have not described when logical locks are acquired on records but we shall be content to make several observations.

Our physical locks do not obey a two-phase locking protocol. If it is necessary to abort an operation, we perform a compensating operation which ned not leave the file in its original internal state but only in the same state as visible from the outside.

The protocol implemented on records may be two-phase locking [Eswaran 76, Gray 78], optimistic concurrency control [Kung 81], or some other approach [Buckley 85]. We aim to guarantee the consistency of the data structures. In the event of a crash, the integrity of the data structure is maintained. We are not concerned with whether the old record is there or the new one, but only that only one record be reachable through the index structures correctly. Unreachable blocks are not as serious a problem as unreachable or duplicate records. File recovery techniques are described in Section 8. Use of this IFAP by systems requiring atomic transactions, such as database systems, need to implement their own logging and recovery of records. We will guarantee that once records are stored they remain in the file intact until replaced or deleted.

The separation of locking into logical and physical locks solves many problems as we have seen, but it introduces some new ones. If a record is deleted by a transaction that subsequently aborts, it may not always be possible to restore the deleted record. In particular, if in the interim another record is inserted with the same values for some unique index as the deleted record, the attempt to reinstate the deleted record will fail. We do not lock phantom records. We also keep indexes up to date. If we deferred updating the index, another insert request by the same transaction with a matching unique index key value might unnecessarily fail.

For the benefit of systems using two-phase locking [Eswaran 76, Gray 78], the IFAP should automatically acquire locks as records are returned to the caller. We suggest that the intention to modify the record be specified when the record is read so that an exclusive lock on the record may be obtained. Locks may be released explicitly, at end of transaction, or when the record is modified (rewritten or deleted). Therefore, with the exception noted earlier, a scheme for logical locks on records could be included (indeed, it exists in our implementation) that would support two-phase locking. Since no physical locks are held between calls, any locking scheme may be used on logical records, including non-two phase protocols [Buckley 85].

8 File Recovery Techniques

The algorithms presented above are specially designed to maintain the integrity of the disk data structure in the event of a crash. In particular, we use leaf-first updating. We do require that disk blocks are either completely and legibly written or not altered at all. Uninterruptable power supplies to disk drives can guarantee

that writes complete when there is a power failure. However, during some operations, inconsistencies may arise due to failure, such as indexes may not reflect the latest changes to the data. This section describes how to limit these problems and correct them when they arise.

We assume that physical locks apply to all processes within an operating system environment. We also assume that the IFAP is allowed to complete the current request if the caller is cancelled at user or operating system request. Consequently, we can assume that if any IFAP fails during a request, the cause is an unexpected shutdown affecting all processes in the operating system environment. Thus, the integrity of a file can be restored by the first process to attempt to access that file after the crash.

IFAP stores in a known location in the file the status of each request in progress that may result in inconsistencies of the disk data structure. When the request is started the relevant information is stored and when the request is completed, the information is deleted or marked completed.

Each time IFAP opens a file, it attempts to acquire exclusive access to the file. If it can, it is the sole user of the file. It will then check for operations in progress and complete them as necessary. When this is complete, if shared access to the file is desired, IFAP will demote its lock on the file to be shared. The next IFAP instantiation will then not be able to acquire an exclusive lock on the file and it will assume that the file has been recovered if necessary. This assumes that physical locks do not transcend crashes of the operating system environment, or at least that they can be acquired wholesale by an exclusive accessor of the file.

The user of IFAP, typically a database management system, needs to be concerned with logging and recovery of a record [Haerder 83, Kohler 81]. IFAP ensures that record operations are completed; the user must ensure that the correct records are stored.

9 Efficiency Considerations

We have decided not to link the leaf pages of the index together, for example, as in the sequence set in IBM's VSAM. There are several reasons for this decision. First, there are no algorithms for maintaining a B-tree with a sequence set that avoid violation of the integrity of the pointer structures in the event of a crash between the writing of blocks for an operation. Second, concurrency control considerations are exacerbated by the additional access path provided by the sequence set. Third, presplitting of index blocks requires having the parent block locked when determining whether child index block should be split; this is not done when using the sequence set. The algorithms are considerably simplified when all access to the B-trees involves traversing a tree from its root.

Since every access to the file except by tuple ID involves searching a B-tree from the root, special care must be taken to make this access efficient. We suggest use of a buffer management facility that maintains a cache of all disk blocks read or written recently. When our instantiation of IFAP reads an index block from a file, if the block has been accessed recently, the buffer manager will return it without having to read from disk. If another instantiation of IFAP has updated it since we last read it, we will get the latest version without much additional overhead. Thus even though complete traversal of a B-tree will require many repeated requests for the root index block, few disk reads for the root index block will be required if sufficient buffer memory is available. This facility is provided by the PMAP input-output interface to TOPS-20 adapted from Tenex.

If main memory is plentiful, the tradeoff of using the indirect pointer (tuple ID) rather than clustered index key may have different characteristics, especially if the entire clustered index fits in memory. This requires that the clustered index key be unique or at least extended internally to make it unique. The algorithms presented here can be adapted to use clustered index key.

10 Conclusion

We have described a multi-indexed B^+-tree that is reliable, deadlock-free, and reasonably efficient. We wanted the B-tree structure to be reliable. By this we mean that a crash cannot result in corruption of the B-tree structure. The worst that can happen is the B-trees lose synchrony with the data, which is easily fixed in restart, or disk blocks are lost, but they are easily returned to the free block pool by scanning the file.

We wanted the B-tree structure to be deadlock-free. Obviously this is not possible on the record level if multiple record locks can be held by any user. Rather, manipulation of the B-tree structure requires that

block locks are held during a B-tree operation, and it is the protocol on these block locks that is deadlock-free. As a replacement may enter through any of the B-trees and, by changing multiple attributes, cause any collection of B-trees to change, a single partial ordering applied to all block locks could not alone prevent deadlock.

We wanted the B-tree structure to be reasonably efficient. But we were also interested in low variance in response. For example, the use of indirect pointers between non-clustered indexes and the data is a compromise between using direct pointers and using the clustered index key.

Other requirements included variable-length keys, variable-length records, separation of clustering (an efficiency issue) and unique index (a data dependency issue), as well as minimal assumptions about the capabilities of the underlying operating system.

These conflicting needs had to be carefully balanced in the design of a synergistic system. Often design decisions made originally for one goal were found to be beneficial for another goal, or even had unexpected benefits. For example, we used an anchored pointer for indirect pointers from non-clustered indexes to the data. The indirect pointer was created for efficiency in moving clustered records. Anchoring it allowed us to release all locks before waiting for a new lock when it would violate the partial order. This permits the use of a non-preemptive deadlock prevention algorithm.

Our design is a solution to an important database implementation problem. But it also illustrates clearly some tradeoffs made in design decisions. We have combined an appropriate collection of techniques to obtain a synergistic whole. The whole is greater than the sum of its parts, for it says how they fit together. The choice of these parts and the process of fitting them together is engineering.

11 Acknowledgements

Gio Wiederhold suggested the problem. The first implementation of the system, called FLASH, was by Jim Allchin. Further implementation work was done by Xiaolei Qian and Arun Swami. The design of the system and the refinement of the algorithms were done by Arthur Keller. This manuscript was prepared with the helpful assistance of K. F. Carbone. A functional description and an interface for Ada is also available [Keller 86]. George Copeland suggested considering the issue when large main memory is available. Bruce Shriver suggested some improvements in the presentation, more of which will be incorporated in the published version.

12 Bibliography

[Allchin 80] Allchin, J. E., Keller, A. M., and Wiederhold, G. "FLASH: A Language Independent, Portable File Access System," *Int. Conf. on Management of Data*, Santa Monica, CA, ACM, May 1980, pp. 151-156.

[Bayer 72] Bayer, R., and McCreight, C. "Organization and Maintenance of Large Ordered Indexes," *Acta Inf.* 1, 3 (1972), pp. 173-189.

[Bayer 77] Bayer, R., and Schkolnick, M. "Concurrency of Operations on B-trees," *Acta Inf.* 9, 1 (1977), pp. 1-21.

[Buckley 85] Buckley, G. N., and Silberschatz, A. "Beyond Two-phase Locking," *Journal of the Assoc. for Comput. Mach.*, Vol. 32, No. 2, April 1985.

[Comer 79] Comer, D. "The Ubiquitous B-tree," in *Comput. Surv.* 11, 2 (June 1979), pp. 121-137.

[Eswaran 76] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. "The Notions of Consistency and Predicate Locks in a Database System," *Comm. of the Assoc. for Comput. Mach.*, Vol. 19, No. 11, November 1976.

[Gray 78] Gray, J. "Notes on Data Base Operating Systems," in *Operating Systems*, Lecture Notes in Computer Science, Vol. 60, Springer-Verlag, New York, 1978.

[Haerder 83] Haerder, T., and Reuter, A. "Principles of Transaction-oriented Database Recovery," *ACM Comput. Surv.*, Vol. 15, No. 4, December 1983, pp. 287-213.

[Keller 86] Keller, A. M. "Indexed File Access for Ada," submitted for publication.

[Kohler 81] Kohler, W. H. "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Comput. Surv.*, Vol. 13, No. 2, June 1981, pp. 149-183.

[Kung 81] Kung, H. T., and Robinson, J. T. "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems*, Vol. 6, No. 2, June 1981, pp. 213-226.

[Lehman 81] Lehman, P. L., and Yao, S. B. "Efficient Locking for Concurrent Operations on B-trees," *ACM Trans. on Database Systems*, Vol. 6, No. 4, December 1981, pp. 650–670.

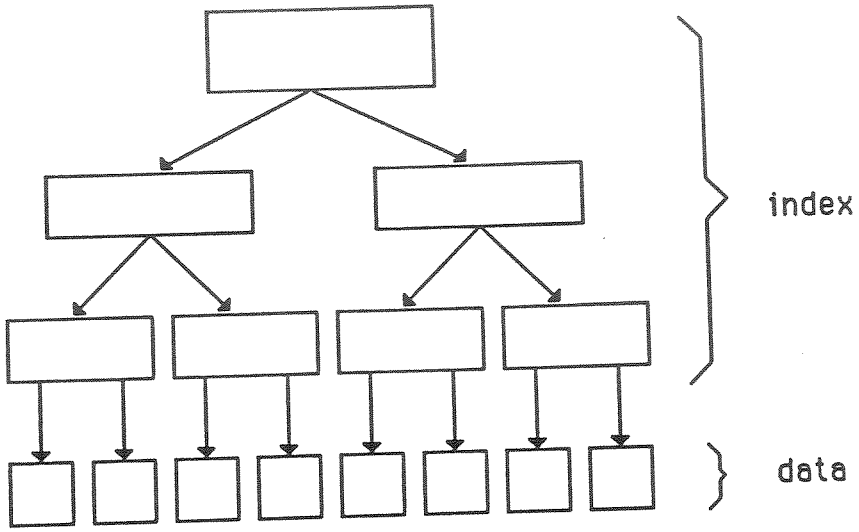
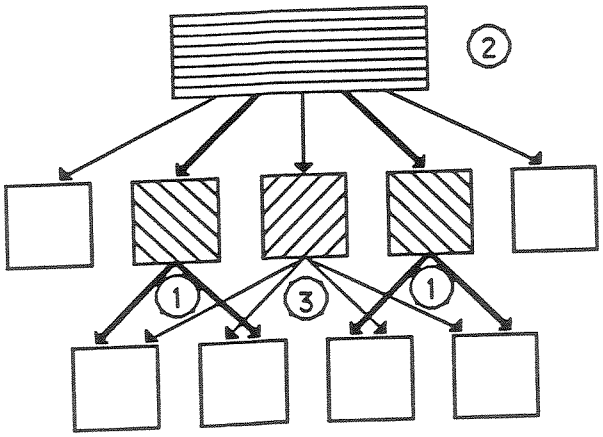


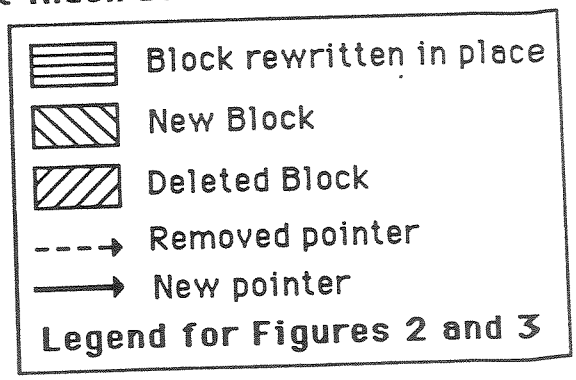
Figure 1. Index Data Structures

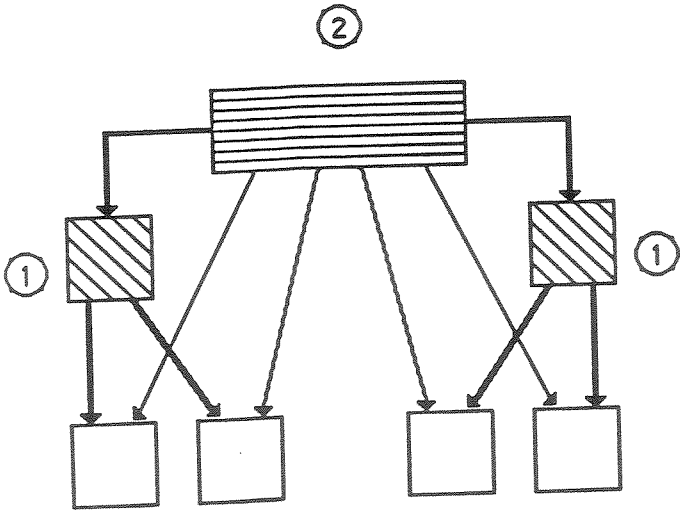


Parent and child blocks both locked

1. Write new child index blocks
2. Write parent with new pointers
3. Add old child index block to available list
4. Release locks on old child, parent, and one child block

Figure 2. Splitting a non-root index block





1. Write new child index blocks
2. Write root index block with pointers only to new child index blocks.

Figure 3. Splitting a root index block

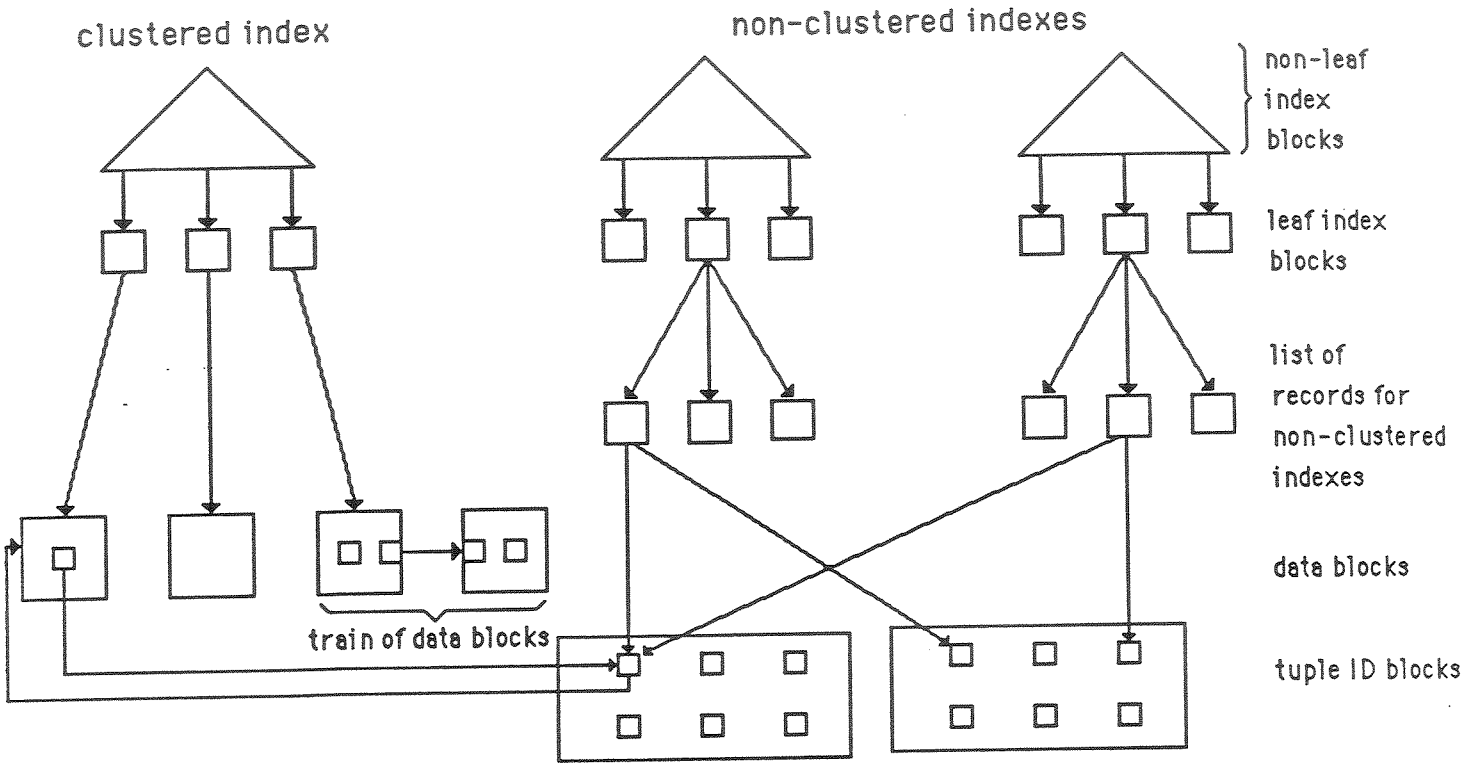


Figure 4. Disk Data Structures

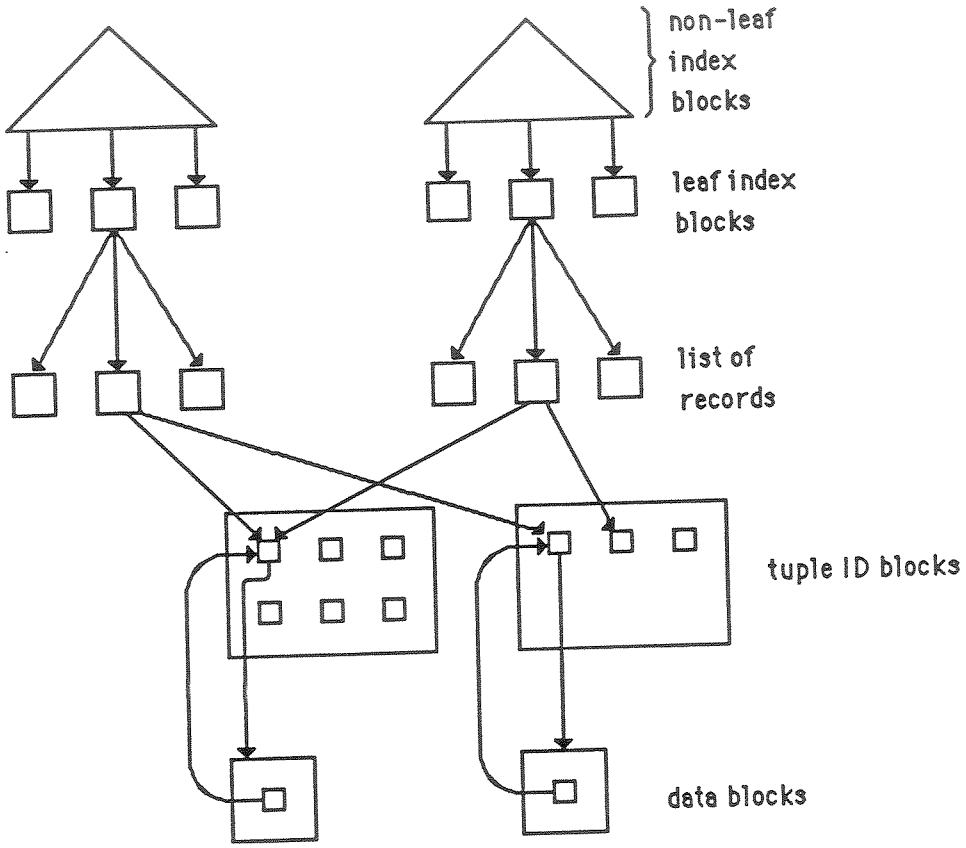


Figure 5. Disk data structures for files without clustered indexes