Principles of Transaction-Based On-Line Reorganization

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

Abstract

For very large databases such as those used by banks and airlines, cost considerations may forbid shutting down the service for a long period of time and reorganizing off-line. Similarly, the size of the database may preclude constructing another copy with the desired organization on another disk collection. Such databases need incremental on-line reorganization. References to records occur in many places in the database. If the identifier used for the record changes due to reorganization, all of these references must be changed. This paper concentrates on the problems of updating references to enable on-line parallel incremental reorganization to be correct while reusing existing code and making minimal changes to underlying transaction processing software.

1 Introduction

In many large database installations, down time of only a few minutes can cause a loss of millions of dollars. Yet currently, the only way to reorganize the database is to shut it down and spend hours or days reconstructing it. Buying enough disk space for a duplicate of the database may be too expensive. As databases are growing to the size of several terabytes, this has become a pressing problem.

For this reason, there has been a resurgence of interest in on-line incremental reorganization algorithms. Papers on on-line construction of secondary B^+-tree indexes [Mohan1992a, Srinivasan1991, Srinivasan1992], conversion from B^+-trees to hashing [Omiecinski1988], reclustering of records [Omiecinski1992], resequencing of primary B^+-tree leaves [Smith1990], and use of partial indexes[Stonebraker1989] have recently appeared.

In this paper, we consider only one aspect of the on-line reorganization problem, an aspect which has been ignored in the other papers. This is the changing of all references to a record when its primary identifier is changed. For some reorganizations, such as the construction of new secondary indexes, this is not relevant. However, for other reorganizations, such as reclustering records in a database where page IDs are part of the identifier for a record, or construction of a new *primary* B^+-tree index, this is crucial.

Although we do not consider object-oriented databases *per se*, if physical addresses are used for object identifiers as in O_2 [Velez1989], the principles outlined in this paper apply. Reclustering of objects in such databases will require updating of references in indexes and in containing objects.

We assume there are several secondary indexes referring to the record and possibly some foreign key references. We shall look at the problem in terms of "records moved." For each record moved, several pages of the database must be updated. If a system failure occurs when some of the references have been changed and others have not, the database will be inconsistent. For consistency, we suggest all changes for a given moved record be encapsulated within a transaction. In section 2, we describe the problem in more detail and outline the basic record-moving transaction. We explain why we do not choose to keep a differential table of old and new addresses. We show how some but not all of the code for deleting and inserting records can be reused.

*This work was partially supported by NSF grant IRI-88-15707 and IRI-91-02821.

**Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992**

When a record is moved, its address changes. What prevents a database transaction from obtaining an outdated address? In section 3, we will explain how this situation is in some cases already handled by existing database code which was written to prevent access to deleted records. We show how code must be modified to handle record moving as well.

When new primary B^+-trees are constructed, one must keep track of how far one has got in the reconstruction process, so that it can be resumed after a system failure. Further, on encountering a reference, one must be able to decide whether it refers to the new or the old organization. For reclustering in databases where page IDs are used in the primary identifier for records, one may wish to keep track of requests to recluster a group of records. These are all problems of "transaction context," information about the reorganization process which must persist between record-moving transactions. We shall discuss transaction context and scheduling in section 4.

2 Transaction Protection

2.1 Discussion

There are two main referencing mechanisms in use today. DEC's Rdb and IBM's DB2 refer to records using their *Record ID* or *RID*, a page address and a slot number within that page. Tandem's default organization is the primary B^+-tree and records are identified elsewhere in the database by their key in that B^+-tree. All these commercial products support secondary indexes. A secondary B^+-tree will have <secondary key, primary key> or <secondary key, RID> in its leaves. Secondary indexes based on hashing are also available.

For the fixed-page databases used by IBM and DEC, there are compelling reasons to move records in spite of the underlying organization. First, one may wish to cluster records by the values of one attribute. One can request such clustering in these products when designing the schema and loading the database. But clustering is not dynamic. Space can be allocated for a cluster, but it can always be outgrown. After a period of database growth, queries related to the desired clustering are no longer efficient.

Second, variable length records can grow. After an update to a variable length record causes it to grow, it may not fit on the page to which it was originally assigned. In this case, forwarding addresses are attached and extra disk accesses are necessary for all queries related to the expanded record. (We do not consider the special problems of records which are so large that they do not fit on one page.)

Using primary B^+-trees makes certain types of reorganization unnecessary. Clustering is automatic. Variable length records do not need to be referred to by forwarding addresses when they no longer fit in the page to which they are originally assigned. In a B^+-tree, when a record update makes the record too large to fit in a leaf, the leaf is split. Thus, from a reorganization perspective, primary B^+-trees are better than RIDs.

It is true that if one wishes key-consecutive B^+-tree pages to be physically consecutive for fast range searches, one must reorganize. This is nicely treated in [Smith1990]. The only reorganization we discuss for the primary B^+-tree database is changing which key should be the primary one or switching from a fixed page organization to a primary B^+-tree. These are both rare events.

The basic act in our reorganization process is the moving of a single record from one physical location to another. (When forwarding addresses are used for variable length records, the actual record may have been moved before; we would be concerned with erasing the forwarding address—*effectively* moving the record.) For consistency, all references to this record in the database must be updated. These references will be on different pages in the database, in secondary indexes and in foreign key entries in other records. To keep the database consistent, we suggest record-moving and the related updating of references be encapsulated in a database transaction.

One transaction may contain all the changes for several records, however. That is, record moves may be batched. The more record moves in one transaction, the longer locks are held and the less control one has on interference with ongoing database work. On the other hand, setting up transaction table entries for each record moved may prove to be too much overhead. Therefore the length of the batch transaction is a parameter which can be tuned to system requirements. We do not discuss batch size further; in the remainder of the paper, we assume only one record is moved per transaction. The extension to batching is straightforward.

An alternative approach, as in [Omiecinski1992], would be to make a differential table or look-aside table correlating new and old addresses after a record is moved. We believe there is little advantage to postponing the updating of references and maintaining a differential table. While there are still entries in the table, every database transaction using an index or foreign key reference to the records in the relation with moving records must consult this table. If this table is not always completely memory-resident, look-up can be a serious performance drain.

In addition, there is still a need for transaction protection for the moving of the record and the creating of the table entry. Other transactions must read the table entry if and only if the move has occurred. Also, if a record is updated by another transaction and then the record is moved, it may not be possible to UNDO the update. Thus one must lock the record when it is moved to cause an updater to either commit before the move or do the update after the move. One must also log the move so REDOing the updates before and after the move is possible.

Also, an entry in a differential table cannot be deleted until it is determined that all references to the old address in the database have been changed. If all reference changes and the deletion of the table entry are encapsulated in a transaction, table entry deletion is safe. The advantage to a differential table is in being able to postpone the work of changing references to another time. However, the scheduling algorithms in section 4 should keep reorganization from overloading the system, so this does not seem to be an important advantage.

The transaction paradigm has the advantage of clear semantics and existing implementation. For the time being, this is the simplest way to guarantee eventual consistency in a database. But a non-transactional "mini-batch" approach as in [Lomet1991] can also be applied to reorganization.

For example, a system table could be maintained listing which references of an ongoing move operation remain to be changed. This table would be placed in log checkpoint records. Each new reference change would be logged and appropriate locks or semaphores used to guarantee isolation. In case of system failure, the recovery manager could reconstruct the table from the checkpoint record and the subsequent log records. This would guarantee eventual consistency without using a transaction. The advantage is that no reference change need be undone after a system failure. The disadvantage is that recovery code must be rewritten to handle special log records referring to reorganization reference changes. We choose to aim for minimal rewriting of transaction processing software. We thus opt for transaction-based reorganization.

2.2 A Record-Moving Transaction

Each record-moving transaction must lock the record to be moved until end of transaction to properly serialize with other transactions which may wish to update (or delete) the same record. Some systems allow record level locking; others may require that the page be locked. All systems lock by name. We shall use the word *Identifier* to denote the name which is locked. Thus the identifier is the RID of the record or the primary key of the record or the identifier of the page the record is in. The identifier is *X-locked* (exclusively locked) for the duration of the transaction. In addition the new identifier of the record (after moving) is locked.

The record-moving transaction performs at least the following steps (in section 4, we shall consider adding operations to establish transaction context as well):

- X lock the old IDENTIFIER.
- X lock the new IDENTIFIER.
- Move the record or remove the forwarding address. (For forwarding address clean-up we assume the record has already been moved.) As necessary, adjust the old and new primary B^+-tree indexes when moving from one primary B^+-tree to another. Log these updates.
- Update any uses of the old IDENTIFIER as a foreign key. Each record which is updated must be X-locked. Log all updates.
- For each secondary index, update the IDENTIFIER in the index. Use the concurrency algorithms required for isolation on the indexes. Log all updates.
- Commit transaction; release locks.

2.3 How delete then insert differs from moving a record.

Part of the code for reorganization already exists in the database system. For example, one has routines for finding all index entries for a given record so that the record can be deleted. This code can be used to find the entries which must be updated when a record is moved. Also, to enforce referential integrity, one must be able to decide whether or not there are foreign key references to a given record. Probably this is enforced using an index on the foreign key field for the relation which references this record. This index can be used to find the places where foreign key references must be changed when a record is moved. But using the delete operation followed by an insert operation is not the same as executing a record-moving transaction.

Deletions and insertions may cause triggers to be executed. For example, ANSI standard SQL requires that foreign keys refer to existing records. If a record which has a reference to it is deleted, some action must be taken. Often, the DBMS will make a check at commit time to see if there are dangling references to a deleted key. Some implementations simply abort the

transaction if this is the case (called RESTRICTED in [Date1990]). Another solution might be to automatically delete all the records referring to the deleted record (called CASCADES in [Date1990]). (CODASYL systems are able do this.) But if an IDENTIFIER is changed due to reorganization, aborting the record-moving transaction or deleting other records is not acceptable.

When a record is moved, the references to it in other records must be changed. But a delete operation does not automatically erase foreign key references and an insert operation does not automatically create them.

Further, more general triggers may be specified on insertion or deletion of a record. These should not be executed when a record is moved.

If a delete is followed by an insert, each index is visited twice. This is inefficient and so should not be allowed to occur. In addition, index page consolidation might take place if an entry is removed from the index. Except in the case where the same key has multiple entries extending over several index pages, the new entry will be put in the same page and there will be no reason to consolidate.

So while there are some similarities between moving a record and doing a delete followed by an insert, the two operations are not the same. Some of the code which already exists can be reused (such as for finding all foreign key references to a record), but one must be careful not to execute triggers or visit indexes more than once, and one must code the updating of foreign key references.

3 Correct Search During Reorganization

In this section, we will look at the treatment of outdated references in indexes and foreign keys, searching using primary keys, and scanning during reorganization. We will show how some mechanisms in place for detecting or preventing access to deleted records can be used to provide correct search for moved records.

3.1 Outdated IDENTIFIERS

While a record is being moved, a database transaction T may obtain the old IDENTIFIER in an index or a foreign key reference before it can be changed. For example, T may use a secondary index before the index entry is changed but after the record-moving transaction has begun. T will read the old IDENTIFIER in the index and request a lock on the old IDENTIFIER. After the record has been moved and this IDENTIFIER is unlocked, T obtains the lock and looks for the record. However, this IDENTIFIER will not lead to the correct record.

Every DBMS must assure that transactions do not use addresses of records which have been deleted, or if they do, that they can detect that the record has been removed. This can be accomplished in a variety of ways. Any of these mechanisms can be used to assure that a transaction either does not obtain a lock on a changed IDENTIFIER or can detect when a record IDENTIFIER has been changed. Existing DBMS system code must be modified to handle detected missing records now that they may be missing because they have been moved as well as because they have been deleted from the database.

3.1.1 Holding all locks to end of transaction

If all locks (including locks on index pages and "key-value" locks as in [Mohan1990]) are held to end of transaction, any index page which is locked to read an index entry or key-value lock held to access all references with this key or any record which contains a foreign key will be locked until the transaction commits or finishes abort processing. Then if a delete or a reorganization of a record is in progress, a deadlock, detectable by the lock manager, will occur. The deleter or reorganizer will hold a lock on the record to be moved or deleted. The other transaction will wait for this lock. The deleter or reorganizer will need to wait for the page or record containing the IDENTIFIER or key-value lock in order to change the IDENTIFIER. One of the two transactions will be aborted by the lock manager. This provides the desired correctness.

3.1.2 Crabbing

Holding locks to the end of the transaction is more than is necessary for correctness. A weaker protocol, merely holding them long enough to obtain the lock on the IDENTIFIER to which they refer, is sufficient. This technique is called *crabbing* or *lock coupling*.

We are not discussing tree traversal, which is often associated with lock coupling. We simply suggest (leaf) index page locks or locks on records with foreign keys or key-value locks would be kept until the IDENTIFIER locks were obtained. This will be enough to cause a detectable deadlock if a delete or reorganization is in progress. It may not be enough for other consistency requirements of the database.

3.1.3 Checking references again after waiting for a lock

Another technique for safety in using references is to recheck the information in the access path after the lock on the reference has been obtained. For example, it is proposed in [Mohan1992b] to increase concurrency in indexes by briefly holding "latches" on pages and releasing the latches as soon as the page is no longer of immediate interest. Latches are semaphores and the partial ordering of index structures is used for deadlock avoidance between latches. Latches are not visible to the database lock manager.

This creates problems in going from the latched index structure to a locked IDENTIFIER: if a latch is held while waiting for a lock then undetectable deadlocks may occur. Undetected deadlocks are avoided and the reference is checked as follows:

- While holding the leaf page latch attempt to lock the IDENTIFIER, but instruct the lock manager to fail rather than wait for a lock (conditional locking).

- If the lock is granted immediately then the IDENTIFIER is still valid, and the latch may be released and the record accessed.

- If it is necessary to wait for a lock, remember the SI in the index page, unlatch the page, and re-request the lock in the normal fashion. (The SI, or *State Identifier*, is changed when anything in the page is updated. Usually the log sequence number (LSN) of the corresponding log record is used as a state identifier.)

  Once the lock is eventually granted, relatch the index page. If the SI is unchanged then the locked IDENTIFIER is valid, so unlatch the index page and access the record.

- If the SI of the index page has changed, keep the IDENTIFIER lock and re-search the index.

3.1.4 Deletion marks

If access to an outdated IDENTIFIER is not prevented, it may be detected. A slot in a page can be marked as deleted. This has been used in a number of DBMSs. When encountering a deletion mark while reorganization is in progress, one must search again. It cannot be assumed that the record has been deleted; it might have been moved.

3.1.5 Missing keys or deletion marks when constructing primary B^+-trees

There is a special case, where detection of a missing record does not require searching again. This case occurs when we know from the position of an RID in the database, while converting from an RID organization to a primary B^+-tree, that the record cannot have been moved. We are assuming that whether converting from one primary B^+-tree to another or from a fixed record organization to a primary B^+-tree, one proceeds in physical (not key) order in the original organization. This will free contiguous space to be used later for the new organization. We also assume (see section 4.3) that we can tell how far reorganization has gone in terms of physical location of the last record moved.

In changing from one primary B^+-tree to another, since we proceed in physical order, a missing primary key IDENTIFIER requires that the search though the secondary index or foreign key be made again. We believe this is less important than losing contiguous free space by proceeding in key-order.

If we had obtained an RID, however, during a reorganization from a fixed-page to a primary B^+-tree structure, and there is a deletion mark for that RID, and the RID was past the last record moved, then the record has been deleted, not moved. If the RID was in the area where records had already been moved, we must search again whether there is a record in place or not, so there is no need to attempt to actually access the record with this RID.

3.1.6 Summary of dealing with outdated IDENTIFIERS

To summarize, when record-moving transactions are possible, a transaction which uses foreign key references or secondary indexes must:

- hold locks on index pages (or key values or records with foreign key references) to end of transaction

- or lock-couple (i.e. hold locks only until the IDENTIFIER lock is obtained)

- or check the State Identifier in an index page after waiting for a lock on an IDENTIFIER and search again if it has changed

- or search for the record again if a deletion mark or missing source key is encountered. The one exception, when one can tell that a missing record means that deletion has occurred, is in switching from a fixed-page organization to a primary B^+-tree, where the RID is in the area where records

have not yet been moved. If the RID is in the area where records *have* been moved from a fixed-page organization to a primary B+-tree, the search must take place again whether or not a record is present.

3.2 Search using primary key

A search can be made directly using a primary key value, without going through a secondary index or a foreign key reference. This would happen when a query is made using a value of the primary key for selection. We assume that when the search begins, the searcher has a lock on the IDENTIFIER, so that if the record exists, it cannot be moved during the search. We assume there is a source (old) primary B+-tree and a *target* (new) primary B+-tree. We assume all insertions are made to the target tree. Existing tree concurrency protocols are followed. When scans are required, scanning locks (next section, 3.3) are made.

First we shall look at the case where the source tree is not converted to a secondary tree by the reorganization; it simply shrinks as records are moved. Here a missing source key value can mean that a record does not exist (might have been deleted or never was in the database) or the record has been moved or inserted in the target tree. A search on such a value must include a scan on the target primary B+-tree. For range searches, key range locks ([Gray1991, Mohan1992b, Mohan1990]) on the source tree and scanning locks (section 3.3) on the target tree must both be made.

Symmetrically, if a query is made on the target key, and there is no secondary index on the target key, and the record is missing from the target primary B+-tree, a scan must be made on the source organization.

In switching from a source primary B+-tree to a target primary B+-tree, one may wish to convert the source B+-tree to a secondary index. As records are moved, entries with <source key, target key> can be placed in a new secondary B+-tree. When a new record is inserted in the relation, a new entry is inserted in the secondary B+-tree. (The record itself is inserted in the target B+-tree.) Both the secondary and the primary B+-trees on the source key may need to be searched for a given value. Here a missing source key means that the record does not exist. Thus, searches on source key do not need to scan the target tree in this case. Key-range locking must be extended to cover both the secondary and the primary trees on the source key.

In the case where there is already a secondary B+-tree with the target key, a record-moving transaction removes entries from the secondary B+-tree as the records are inserted in the target primary B+-tree. Searches on target keys may have to use both indexes. Again, key range locking needs to involve both indexes. No scans on the source organization are needed.

3.3 Scanning

Suppose that a transaction T is scanning the entire relation without using an index. This would be the method of choice when a large number of records are to be read and/or updated, or when a query making a selection on a non-indexed attribute is made. If a record-moving transaction is in progress and if strong enough locking protocols are not used, anomalies can occur. T could see the same record twice (before and after it is moved) or miss a moved record because it is moved to a location the scan has already passed.

Suppose, for example, that T locks records or pages, reads them, and then releases the read (share) locks. (This occurs in *level 2 consistency* or *cursor stability*, a locking option provided by many commercial systems.) Then T can read the record to be moved, unlock the record, then encounter it again after it has been moved and after the record-moving transaction has dropped its locks.

It is simple to assure that records are not read twice by scanners. Simply require that scanners keep all locks to end of transaction (*repeatable read* or *level 3 consistency*) at least while reorganization is in progress. In this case, once T reads a record, the record cannot be moved until T completes.

It is more subtle to assure a moved record is not missed by a scanner. Here we must deal with the phantom problem. Say T is making record-level locks only. (That is, the IDENTIFIER is for one record, not for a larger granularity such as a page.) T reads through page P_1 and locks each record in it. T has not yet reached page P_2. Then a record can move from P_2 to P_1. The locks held by T do not conflict with the new or the old IDENTIFIER of the moved record. T misses the moved record, for when T reaches P_2, the record is no longer there.

Similarly, suppose a transaction begins to move a record from P_1 to P_2. When T reaches P_1, the record is no longer there. If the record moving transaction aborts, and the record is replaced in P_1 after T has passed through, T never sees the record.

The over locking or next-record locking techniques of [Mohan1992b, Mohan1990] used for range searches could be applied to the phantom problem. However, the phantom problem is usually not relevant for scanning entire relations because scanners usually lock with larger granularity—pages, or collections of pages, or even the whole relation. The real controversy (with

lock granularity smaller than the whole relation) is whether or not level 2 (cursor stability) should be an option. One must be aware that inconsistent views of the database can result.

3.3.1 Mixed Multiversion Concurrency

One solution to the scanning problem is to use mixed multiversion concurrency as implemented in DEC's Rdb [Joshi1989]. One marks versions of records with commit-time timestamps of the transactions which created them. Read-only transactions read the most recently committed version with a timestamp before their own start time. Read-only transactions do not set locks. The record-moving transaction would put its timestamp on the moved ("new version") record, and the read-only scanner would read the old "version" which had not been moved.

4 Transaction Context and Scheduling

The problem of when to systematically reorganize a database was extensively written about in the late 1970s and early 1980s. [Yao1976] was one such paper. The issue considered in most of the literature concerned deciding when the DBA should shut down the database for offline reorganization. Soderlund [Soderlund1981] however suggested that online reorganization techniques were practical for CODASYL databases so long as the size of any individual reorganizing transaction was kept small so that "real" work would be minimally delayed. This is the scheduling principle we adopt.

4.1 Scheduling

We also borrow some suggestions from [Smith1990], which describes a B^+-tree reorganization designed and implemented by Franco Putzolu. Here, the problem is to resequence B^+-tree leaves so that leaves which are consecutive in key order will be consecutive on the disk. Putzolu's algorithm runs through the leaves in key order performing short transactions such as interchanging two blocks (containing B^+-tree leaves) or splitting one leaf. (Behind the current leaf under treatment, the tree may lose physical contiguity.) This follows Soderlund's principle of isolated small transactions.

One can issue requests for the next transaction in the Putzolu algorithm with a parameterized time delay

$$(\frac{100 - \text{RATE}}{\text{RATE}})\text{TIME}$$

where RATE can be any positive number up to 100 specified by the "user" (DBA) and TIME is the time used to complete the previous requested reorganization transaction. A rate of 100 means reorganization has as high priority as other ongoing transactions; consecutive requests are not delayed. Otherwise, the delay depends both on the priority assigned to the reorganization and the load on the system. [Sm90] also notes that extra log records are generated by reorganization and that if the log record generation is too fast to be handled by the recovery system, the priority (RATE) can be adjusted downward by the user. That is, delay between transactions can be used not only to minimize interference when there is heavy demand on the system, but also to prevent the extra log records generated by reorganization from clogging the system.

4.2 Context for forwarding address removal or clustering

For removal of forwarding addresses, or for reclustering, one may wish to keep a request queue. One does not wish the queue to disappear if there is a system failure. So this should be a recoverable queue. Many transaction processing systems implement recoverable queues [Bernstein1990a] [Bernstein1990b].

The record-moving transaction would begin by removing a request from the queue. If a record-moving transaction aborts, or if a system failure occurs before the record-moving transaction can complete, the UNDO of this step would place the request back in the queue. Since requests can be repeated, the record-moving transaction must check to see if the move still needs to be made.

Transactions may request removal of forwarding addresses if (1) they create the forwarding address while updating a database record or (2) they follow a forwarding address left by another transaction. This requesting ability can be suppressed until the performance of the system declines to a certain point. Requesting reclustering is similar. When a cluster no longer fits in its assigned space, a request for reclustering can be made. This can happen at the time a new member of the cluster is entered in the database, or when a query has to make too many disk accesses for a given cluster. This can also be suppressed until performance falls below some threshold.

As an optimization, we make two suggestions: (1) Only make requests when a forwarding address or wide-spread cluster is encountered, not when it is created. These constructs cause no performance decline if they are not accessed. (2) Let the length of the request queue influence the reorganization priority. A long queue means many extra I/Os have been made

and reorganization is more important.

Other optimizations seem more questionable. For example, one could count occurrences of the same request and make the queue a priority queue. But then one must search the queue each time a request is made. Even more complicated, with less benefit, is to have a threshold count for reorganization. This requires also keeping a timestamp to discard old requests with low counts.

4.3 Context for new primary B$^+$-trees

For constructing a new (target) primary B$^+$-tree, context is more subtle. One must keep track of which parts of the old (source) organization have already been processed. We assume that we process a source organization in physical order whether it is a primary B$^+$-tree or a fixed-page organization. This will free contiguous space to be used later for the target organization.

A record-moving transaction must include an update of status in a "status" relation. This is logged as a record update. If the record-moving transaction fails, the previous status record contents is replaced. Such a record-moving transaction must read and X-lock the status record as its first operation. Parallelism can be supported by having separate status records for each partition of the source organization. This solves the problem of how to restart, or what the next record-moving transaction for a given partition of the source organization should be.

There is another context problem with new primary B$^+$-trees. Suppose that a record has been moved and its IDENTIFIERS changed in all the relevant secondary indexes and all the foreign keys. Suppose that a search for this record obtains the correct IDENTIFIER and locks it. How does the searcher know whether this is an IDENTIFIER of the source B$^+$-tree (or in the case of conversion from fixed location to B$^+$-tree, the source fixed location IDENTIFIER), or the IDENTIFIER of the target B$^+$-tree?

In the case where one is converting from a fixed-page organization to a primary B$^+$-tree, the status record can be used to determine that an IDENTIFIER is not in the source organization, if its value is less than the "where am I?" marker. But if the value is greater than the marker, one has no information.

Perhaps the source IDENTIFIERS are all a different length from the target ones. Then nothing more is needed. But if they are the same length or variable length, one cannot tell from the IDENTIFIER which one it is. We propose using a code bit for each entry in a secondary index and for each record in a relation referencing the moved records as foreign keys. The code bit tells us if this is an old (source) key or a new (target) key.

There are two possibilities: store the code bits in the pages of the index (or the pages of the relation with the foreign keys) or store them separately. Storing them separately has the advantage that after the reorganization is finished, they can be instantly discarded. But storing them in the pages of the index or relation has advantages which we believe are more important.

When a reference is changed, one already has in memory the page where that reference is. So only one page fetch is needed to change both the reference and the bit if the bit is stored in the page.

Log records in most systems are page-based [Gray1991]. That is, for each page modified, one log record is written. Thus, changing the reference and changing the bit vector for the page requires only one log record if the code bit is stored in the page. If the code bits are stored separately, reorganization requires approximately twice the number of I/Os (unless the bit vectors are all memory resident throughout the reorganization) and twice the log records.

Each secondary index has leaf nodes with <secondary key, primary key > entries. If each key is about 8 bytes, there are 128 bits in each key. So the bit vector for an index page is less than 0.01 of the page capacity. For pages with database records in them (for relations having foreign keys) it is more like 0.001 of the page capacity. (There are less records in a database page than entries in an index page, so the bit vector is smaller.) So space for the bit vectors should not be a big problem. As each page is first updated for reorganization, the bit vector is inserted. The bits are changed with each further update. When reorganization is over, the bit vectors can be removed lazily as the pages are accessed for other purposes.

One should keep in mind that the construction of a new primary B$^+$-tree is a rare event. It is the price one pays for not choosing the correct primary key in the original schema—or for switching from a fixed-page architecture to a primary B$^+$-tree architecture.

4.4 Parallelism

We have alluded to parallelism above. We do not assume any particular architecture such as shared nothing or shared disks. Thus a "partition" can be on a separate disk, at another node of a network, and so forth.

For removal of forwarding addresses or for clustering, any number of processors can access the request queue. If processors are associated with partitions of the data, there can be separate queues for each parti-

tion. This is a naturally parallel sort of reorganization, since each record-moving transaction is independent of the next.

For creation of new primary B^+-trees, the source file can be partitioned with separate status records for each partition. Different processors can work on different source partitions. The target can be partitioned also, so that insertions in the target can be handled in parallel. For example, different horizontal partitions based on the target key might be at different nodes of a network in a shared-nothing architecture.

5 Conclusion

Our goal in this paper has been to present principles which will allow many types of on-line reorganizations to be incremental, parallel and correct. We have not considered criteria for deciding when reorganization is needed. We have not looked at the question of finding space for the reorganized records, or how to rearrange the space one has. We have tried to point out what pieces of already existing code can be reused, and how they must be modified. We have tried to indicate where to be careful so that search will be correct and the database will remain consistent.

We have argued that the moving of a record and the changing of all references to that record in the database should be encapsulated in a transaction. This was motivated by the desire to keep system software modifications to a minimum. Regulating the delay between transactions can mitigate interference with ongoing work and can limit the rate of reorganization log record generation.

Although the full deletion and insertion routines available in the DBMS cannot be used as is, many lower level subroutines can be used. Deletion or insertion might set off triggers which should not be executed when a record is moved. In addition, if a deletion routine is followed by an insertion routine, all indexes are visited twice, unnecessary node consolidation in indexes may take place and there is no provision for finding the places to reinsert foreign keys.

We have noted how policies which prevent the access of deleted records also prevent the possibility of obtaining outdated addresses for records which have been moved. If access to deleted records is not prevented, one must be able to detect that they are missing. Code must be changed to allow for the fact that a record which is missing may have been moved, not deleted.

To obtain a consistent view of the database in a scanning operation, even read locks must be held to end of transaction, and granularity of locks must be large enough to prevent phantoms. If these rules are not followed, a moved record may be seen twice or not at all.

For reclustering or removing forwarding addresses, the reorganization process can be an ongoing background process or it can be invoked when performance declines. A recoverable queue of requests for record-moving transactions is kept. Transactions encountering forwarding addresses or wide-spread clusters can place requests on the queue.

For creating new primary B^+-trees, status records for each partition of the original organization must be kept to resume action after a system failure, or just to start up the next transaction. In addition, bit vectors must be kept to indicate which references have been changed to refer to the new B^+-tree.

If these principles are followed, reorganizations can be incremental, on-line, correct, and parallel.

Acknowledgements We would like to thank David Lomet, Michael Carey, Georgios Evangelidis and the referees for their comments on this manuscript. Thanks also to Liz Chambers of Tandem for sending us a copy of [Smith1990].

References

[Bernstein1990a] P. Bernstein. *Transaction Processing Monitors.* Communications of the ACM Vol. 33 No. 11. 1990

[Bernstein1990b] P. Bernstein, M. Hsu, B. Mann. *Implementing Recoverable Requests Using Queues.* Proc. ACM SIGMOD Conference on Management of Data. 1990 pp.112-122.

[Date1990] C. J. Date, *An Introduction to Database Systems, Vol. 1, Fifth Edition.* Addison-Wesley, Reading, Massachusetts, 1990.

[Gray1976] J. Gray, R. Lorie, G. Putzolu, I. Traiger. *Granularity of Locks and Degrees of Consistency in a Shared Data Base.* IFIP Working Conf on Modeling of Data Base Management Systems. 1976

[Gray1991] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann 1991. (draft)

[Joshi1989] A. Joshi, K. Rodwell. *A Relational Database Management System for Production Applications.* Digital Technical Journal, No 8 (Feb 1989) pp. 99-109.

[Lomet1991] D. Lomet, B. Salzberg, *Media Recovery with Time-Split B-trees.* Digital Equipment Corp. TR Cambridge Research Lab 91/9.

[Lomet1992] D. Lomet, B. Salzberg. *Access Method Concurrency with Recovery.* to appear in SIGMOD 1992 (San Diego).

[Mohan1990] C. Mohan *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes.* Proceedings of the 16th VLDB Conference. Brisbane. pp. 392-495.

[Mohan1992a] C. Mohan, I. Narang. *Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates.* to appear in SIGMOD 1992 (San Diego).

[Mohan1992b] C. Mohan, F. Levine. *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging.* to appear in SIGMOD 1992 (San Diego).

[Omiecinski1988] E. Omiecinski. *Concurrent Storage Structure Conversion: From B Plus Tree To Linear Hash File.* Proceedings of the Fourth International Conference on Data Engineering.

[Omiecinski1992] E. Omiecinski, L. Lee and P. Scheuermann. *Concurrent File Reorganization for Record Clustering: A Performance Study.* Eighth International Conference on Data Engineering, 1992. pp. 265-272.

[Smith1990] G. Smith. *Online Reorganization of Key-sequenced Tables and Files.* Tandem Systems Review, October 1990. (A description of software designed and implemented by F. Putzolu.)

[Soderlund1981] L. Soderlund. *Concurrent Database Reorganization - Assessment of a Powerful Technique Through Modeling.* Proceedings of the Conference on Very Large Databases, 1981.

[Srinivasan1991] V. Srinivasan, M. Carey. *On-Line Index Construction Algorithms.* University of Wisconsin-Madison Computer Sciences TR March 1991 and High Performance Transaction Systems Workshop, Asilomar, September 1991.

[Srinivasan1992] V. Srinivasan, M. Carey, *Performance of On-Line Index Construction Algorithms.* to appear, International Conference on Extending Database Technology, Vienna Austria 1992.

[Stonebraker1989] M. Stonebraker, *The Case for Partial Indexes.* SIGMOD RECORD, vol. 18, no. 4, Dec. 1989.

[Velez1989] F. Velez, G. Bernard and V. Darnis, *The O_2 Object Manager: An Overview.* VLDB 1989 (Amsterdam) pp. 357-366.

[Yao1976] S. Yao, K. Das, T. Teory. *A Dynamic Database Reorganization Algorithm* ACM Transactions on Database Systems 1,2 1976.