Safely and Efficiently Updating References During On-line Reorganization

Chendong Zou *
921 S.W. Washington Ave, Suite 670
Portland, OR97205
email: zou@informix.com

Betty Salzberg †
College of Computer Science
Northeastern University, Boston MA02115
email: salzberg@ccs.neu.edu

Abstract

With today's demands for continuous availability of mission-critical databases, on-line reorganization is a necessity. In this paper we present a new on-line reorganization algorithm which defers secondary index updates and piggybacks them with user transactions. In addition to the significant reduction of the total I/O cost, the algorithm also assures that almost all the database is available all of the time and that the reorganization is interruptible and restartable. We believe that the technique presented in this paper could be used for improving normal database update performance as well.

1 Introduction

On-line reorganization is and will be a major problem for transaction systems of the 1990s and the 2000s. Tasks such as restoration of clustering, purging old data, compaction and data migration must be performed without interrupting service. Most of these tasks require the physical moving of data records. If there are any references to physical locations of records in other parts of the database, for example in indexes, these references must be updated when the records are moved. This paper presents a restartable, incremental, efficient and safe method for updating secondary index references to moved records during on-line reorganization in a centralized database.

The algorithm uses a deferred and incremental approach to make the changes in the secondary indexes. When a database record is moved, changes to index pages already in memory are made immediately and the pending changes are stored in some in-memory tables. Then those tables are consulted by the buffer manager when a secondary index page P is brought into the buffer by a *user* request. If P is a leaf page, changes are made at that time. If P is higher than leaf level, the relevant descendant page addresses found in P are placed in the in-memory tables.

The reason for its *efficiency* is that the changes for the secondary index leaf pages are piggybacked with user transactions, thus less I/O is caused by the reorganization process. Through both analytical and experimental study, we measure the efficiency of this technique.

The reason for its *safety* is in the interaction with the concurrency and recovery modules of the database management system. This aspect of our algorithm is important and innovative. Without correct logging, the tables of pending changes could not be reconstructed after system failure and some of the references to moved records could be incorrect.

Correct logging also enables our method to be *restartable*. If there is a system failure, reorganization does not have to start over again. No work is lost.

Because our method is *incremental*, system performance improves as reorganization progresses. In addition, as records are moved, the space they formerly occupied can be reclaimed. It is not necessary to reserve space for an additional full copy of the database for reorganization.

*The work was done while the author was a graduate student at Northeastern University.

†This work was partially supported by NSF grant IRI-93-03403.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

The rest of the paper is organized as follows: In section 2, we present the setting for our problem. In section 3, we present the algorithm without the details of logging and recovery. Recovery is discussed in section 4. Section 5 discusses correct index search protocols for user transactions during reorganization. User operations such as table scanning (without an index) and the adding and dropping of an index are addressed in section 6. We present our analytical and experimental study results in section 7. Section 8 briefly discusses related work. A conclusion is in section 9.

2 Setting

The problem we solve in this paper is how to efficiently change the references in secondary indexes (or any reference changing) during on-line database reorganization in a centralized database. We will use one reorganization problem as a case study throughout this paper. We use boldface to indicate that a technical term is being defined. After it is defined, italics are used to identify the term.

2.1 Assumptions for the Case Study

We assume one relation[1] of **RID** or **Record IDentifier** form is to be reorganized. That is, records are accessed using their RIDs, which usually are page number and slot number on that page.

We assume that originally, a "clustering" index, a B^+-tree whose leaves contain a key value and an RID for each record, was used to load the relation in key order. (For simplicity of the paper, we assume that the clustering key is unique.) Over time the clustering properties declined because of insertions, deletions and updates. In order to "recluster" the data, reorganization of the database is necessary. The reorganization process uses the clustering index to place the records in their new location. It moves records in the order of their clustering key.

We will distinguish between the "clustering" index and the other secondary indexes which are "non-clustering." We assume that the other secondary indexes are also B^+-trees. We will be referring only to the non-clustering secondary indexes as **secondary indexes** in this paper.

All access to trees will follow the Bayer-Schkolnick safe-node tree concurrency [BS77] and in particular will do latch-coupling.

A reader of an index entry will latch-couple to the leaf page, get an S latch on the leaf page, then acquire an S lock on the index key for that index entry as in ARIES/KVL [Moh90]. Notice that the S lock is first requested conditionally. If the conditional request is not granted, in order to avoid deadlock involving latches, the page latches must be dropped and the lock is requested unconditionally [Moh90]. Inserters/deleters also follow the ARIES/KVL protocol. That is, they use next-key locking to prevent phantoms.

We assume that the Write-Ahead-Logging (WAL) rule is used for logging. When reclustering an RID organization, we assume there is some space that can be reclaimed.

2.2 No-split assumption

The organization where the record resides before it is moved is called the **source organization**. After it is moved, it is in the **target organization**. In this paper, the **Old-ID** will be the identifier used in the *source* organization and the **New-ID** will be the identifier used in the *target* organization. Since we are reclustering RID organizations, the *Old-ID* and the *New-ID* are RIDs.

In this paper we will make the assumption that inserting the *New-ID* in a secondary index does not cause a split. In [ZS96a] [Zou96] we did not make this assumption and we worked out in detail how to treat the split. (If the *New-ID* were a different length than the *Old-ID* or if the *New-ID* goes in a different page from the *Old-ID* in some index, a page split is possible.)

2.3 Buffer replacement policy

We assume that the FIX-USE-UNFIX protocol [GR93] is used by the system. We assume that for each index, at least the root of the index tree is in the database buffer. We further assume that when the buffer manager chooses a page to swap out, it won't swap out a parent index page as long as there is at least one child of that parent page is still in the buffer. This implies that all ancestors of any index page in the buffer are also in the buffer. (This assumption can be easily implemented by changing the *bufferfix* [GR93] routine. We can add one extra parameter to indicate the parent page of the requested page, so that the buffer manager can chain those pages properly.) Latch coupling will assure that no child page is brought in if its parent is not already in the buffer.

We also assume that when a requester asks for pages from the buffer manager, it can specify that it only wants pages already in the buffer. That is, if the page requested is not in the buffer, the buffer manager won't do an I/O to get the page, instead, the buffer manager will return *page not found*.

[1] We shall use the word *relation* and not *table* for the collection of records we are moving, as we use the word *table* to indicate several data structures used by the system to keep track of the reorganization.

3 Towards Efficient On-line Reorganization

In this section we present the algorithm without the details of logging and recovery. First, we describe some data structures.

3.1 Data Structures

The following data structures are used in our algorithm.

- **Forward Address Table T**
  Table T is an address table which records the old and new location of the record. An entry of T has the format (*Old-ID*, *New-ID*, *Count*), where **Count** is the number of secondary references that still need to be changed.

- **Pending Changes Table Q_is**
  Tables Q_i, $i = 1, \ldots, s$, where s is the number of secondary indexes, are used to keep information about the changes that should be made to secondary indexes. An entry in Q_j has the format (*Index_value*, *Page_number*, *Old-ID*,*Flag*). The **Old-ID** field refers to the old address of the record that has been moved. The **Index_value** field is the index value of the record for that secondary index. The **Page_number** field is the page number of the secondary index page which either stores the *Index_value* (leaf page) or has a descendant which stores the *Index_value* (upper level index page, i.e., the lowest ancestor page address known when the entry is made.) The **Flag** indicates whether the pending change involves the *Old-ID* (deletion of the *Old-ID* from the original index page), the *New-ID* (insertion of the *New-ID* into a different index page), or both (change of the *Old-ID* to the *New-ID* on the same index page).

- **Boundary Value (BV)**
  The BV is a system variable that is stored in the system log. It is used to record the largest (unique) clustering index key of a record that we have moved so far. A semaphore on BV is used by index-creators and table-scanners to prevent a new reorganization unit from starting.

- **Checkpoint Semaphore (CK)**
  The CK semaphore is used to do checkpointing during reorganization. It assures that no changes are made to tables when they are to be copied to the checkpoint.

- **Reorganization Table**
  The *Reorganization Table* is a very small in-memory table which is constructed and used only during recovery.

The table T and tables Q_i are in-memory system tables. The table T is a small hash table indexed by *Old-ID*. The Q_is are small hash tables indexed by *Page_number*. The records for a Q_i that are hashed into the same bucket are sorted by their *Page_number*. We call the T table and the Q_i tables **look-up** tables. Each of the tables can only use a certain amount of memory space.

3.2 The Algorithm

In this section, we describe the actions of one **reorganization unit**. A reorganization unit moves one record, updates all relevant secondary index leaf pages that are in the buffer, modifies the look-up tables to record pending changes, and changes the BV. The log records written and the recovery scheme are discussed in section 4. We assume here that the record being moved does not have forwarding pointers. Figure 1 outlines the logic of the algorithm.

3.2.1 Obtaining initial locks

When each reorganization unit starts, it will first check to see if each of the look-up tables still has space for one entry. If there is not enough space left, the clean-up procedure, which will be discussed in section 3.4, is called to clean the space. This is a pessimistic approach, because it assumes that all the changes of the secondary indexes will be put in the look-up tables.

If there is enough space left in the look-up tables, the reorganizer will get the initial locks and latches for the record-moving as shown in steps 2,3 and 4 in Figure 1. First, to make user scans correct, explained in section 5, the BV is locked.

The reorganizer then gets an X lock on the record(*Old-ID*) and reads the content of the record to find all the secondary index key values. It will require an X lock on the new identifier, i.e., the *New-ID* of the record. All IX locks required by hierarchical locking protocols [GR93] are also obtained.

3.2.2 Secondary index key value locks

Next, the reorganizer acquires X locks on the index key value for each of the secondary indexes. Deadlocks may occur since the user transactions can lock these pages in another order. In case of a deadlock, the reorganizer will release all the locks it possesses and start over again on this reorganization unit.

3.2.3 Record moving

The reorganizer will then get an S latch on the CK semaphore. This is for correct checkpointing as dis-

1. Check to see if each of the look-up tables still has enough space left for one entry. If not, do clean up for those tables.
2. X latch the BV.
3. IX lock the relation. IX lock the leaf page of the clustering index and the old page containing the record to be moved. X lock the record (*Old-ID*) to be moved.
4. IX lock the new page and X lock the *New-ID*.
5. Read the record to be moved, and acquire X locks on the index key value for each of the secondary indexes that need to be updated to reflect the record move. If there is a deadlock, drop all the locks held by the reorganizer and start over again.
6. S latch the CK semaphore.
7. Move the record and log the move. Update the clustering index and log the change. Drop the X locks on the *Old-ID* and the *New-ID*, drop the IX locks on the relation, and the old page and the new page. Drop the IX lock on the leaf page of the clustering index.
8. For each of the secondary indexes:
   (a) Search for the page(s) where the index entries for the moved record reside. (If there are several RIDs for a given key, and they are in order of RID value, the *New-ID* may very well be placed on a different index leaf page from the *Old-ID*.) Starting from the root, X-latch-coupling down the tree until either the leaf page(s) are found or an I/O has to be done to bring in additional pages to continue the search.
   (b) If the leaf page(s) are in the buffer, we make the update and log the change.
   (c) Otherwise, for each missing index leaf page:
      i. Acquire necessary latches on corresponding look-up tables.
      ii. Put an entry in its corresponding look-up table (one of the Q_is). The *Page_number* is that of the page where an I/O has to be done in order to continue the search.
      iii. For the first Q_i updated, insert an entry in T, where the entry contains the old *Old-ID* and the *New-ID* and set the value of the *Count* field to one. For subsequent Q_i updates, increase the *Count* by one.
   (d) Release the X lock on the index key values, the X latch on the index page and the look-up table latches.
9. Update BV to the clustering index key of the record that has just been moved.
10. Drop the latches on the BV and the CK semaphore.

Figure 1: The Algorithm for One Reorganization Unit

cussed in section 4. After all the locks have been acquired, the reorganizer can move the record to the new organization, log the move, and change its reference in the clustering index. Then it releases all its locks except the X locks on all the secondary index key values and the latches on BV and CK. This is to maximize the concurrency in the system so that user transactions can read the record through the clustering index, or even update the unindexed fields. Deletion of the record and updates of the indexed fields by user transactions can not be done now because the reorganization unit still possesses X locks on the secondary index key values.

3.2.4 Secondary index leaf page updates

After the record is moved, the reorganizer then processes each of the secondary indexes that need to be modified to reflect the move. It searches each secondary index for the leaf page(s) to update the index entr(ies) for the record move. (In the secondary indexes, which are *not* assumed to be unique, many different RIDs may correspond to the same index key. These are normally listed in RID-order so that an individual entry, perhaps for deletion, can be located quickly. Thus the *New-ID* and *Old-ID* may be on different index leaf pages.)

The search will start from the root of the index tree, and X latch-couple down the tree until either the leaf page is found, or an I/O has to be done for an index page P in order to continue the search. Notice that even though the reorganizer holds an X lock on the index entry while it X-latch-couples down the tree, no deadlock occurs, because all user transactions lock requests on index entries are first made conditionally. If not granted, they would drop all the latches they hold, and request the locks unconditionally as in ARIES/KVL [Moh90]. This avoids deadlocks involving latches and locks.

If the secondary index leaf page where the change should be made is in the buffer, the reorganizer will make the change since it has an X latch on the page and an X lock on the index key values. It will also write log records about the update, which will be discussed in section 4. After finishing the change in one secondary index leaf page, it will release the X latch on the page.

If an I/O has to be done for an index page P in order to continue the search, the reorganizer will modify the corresponding table Q_i to reflect the record move, and P's page number will be in the entry of Q_i.

If this is the first pending change about the moving record that is inserted in the Q_is, an entry that contains the *Old-ID* and the *New-ID* is inserted in the table T with the *Count* value set to one. Otherwise,

there is already an entry containing the *Old-ID* and the *New-ID* in the table T. We just need to increase the *Count* value by one. Accessing and updating the look-up tables follows the standard protocol as in [GR93].

The modifications of T and the Q_is have to be done before the X lock on the corresponding secondary index key value is released. If the X lock on the corresponding secondary index key value is released before the modification on T is done, and there is a user transaction which brings in that secondary index leaf page, piggybacking can not be done. This is because the entry in T that contains the moving information hasn't been inserted yet. When both the *New-ID* and the *Old-ID* information have been either inserted in the look-up table or updated in-memory index leaf page(s), the X lock on the key value can be released.

We could have chosen to put the *Old-ID* and the *New-ID* in each entry of the Q_is. Then we do not need the T table. The reason we want to use T is to save some memory space in case there are many secondary indexes.

3.2.5 The ready state

After all the changes to secondary index leaf pages have been processed, the reorganizer then modifies the value of BV to become the clustering index key of the record just moved. We say the reorganizer is in the **READY** state at this time. That is, the record is physically moved, all the changes to the secondary index leaf pages are either already made or are inserted into the look-up tables, and the value of BV is modified. All the secondary index pages that are in the buffer either don't refer to the moved record, or already have the correct address (its *New-ID*). The reorganizer releases the X latch on the BV and the S latch on the CK semaphore when it enters the READY state. The locks on secondary leaf pages have already been released. The moving of this record is done. The reorganization process can go on to process another record.

3.3 Page-Oriented Piggybacking Changes of References

In our algorithm, the deferred reference changes are piggybacked with user transactions in a *page-oriented fashion*. Whenever an index page is brought into the buffer, we check to see if there are some changes pending in the corresponding look-up table Q_i that can be applied.

If the page is an leaf page and there are pending changes to be applied, we first make *all* changes on that index leaf page. Then we modify the *Count* fields of the corresponding items in table T. Finally we delete those items in the table Q_i. In the example above, if $P2$ is brought into the buffer, we would finish the index update recorded in Q_2, and delete that entry. All the piggybacked changes on the secondary index leaf pages should be logged as shown in section 4.

If the page is an upper level index page, then we would apply *all* relevant changes in Q_i and "propagate" them. That is, we would search the index page with the *Index_values* to find the next level of index pages where those *Index_values* could potentially be stored. Then those newly found index pages would be substituted for the old page. In the example above, if $PP3$ is brought into the buffer, a search with key value of $v3$ is done on $PP3$ to find out the next level page $P3$, and $PP3$ will be replaced by $P3$ in the look-up table entry in Q_3.

In this paper, inserting the *New-ID* is assumed to cause no page splits. Piggybacking the changes on leaf pages is straightforward. As shown in [ZS96a] [Zou96], if inserting the *New-ID* causes a page split, the piggybacking algorithm becomes more complicated.

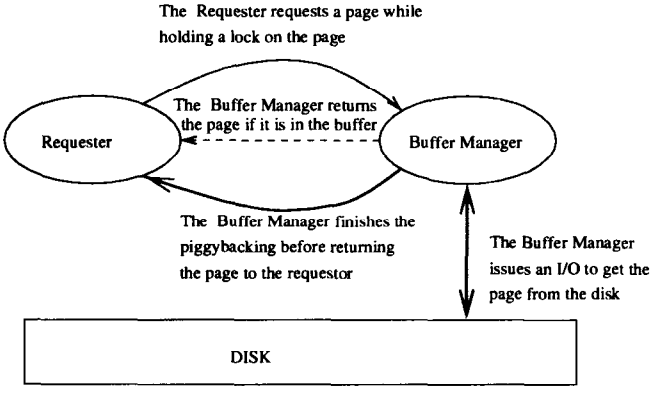

Figure 2: Interaction between the requester and the buffer manager

Piggybacking the changes should be done by the *bufferfix* routine [GR93] of the database buffer manager. When a secondary index page is requested, the buffer manager will return the page immediately if the page is already in the buffer. Otherwise, a disk read is done to bring it into the buffer. Before the buffer manager returns the page to the requester, the buffer manager first gets a shared latch on the semaphore CK, does the piggybacking(if any), then releases its latch on CK. The latch is necessary for correct recovery as explained in section 4.

Figure 2 shows the interactions between the requester and the buffer manager. The dotted line represents the buffer manager's action when the requested page is already in the buffer, the bold line represents the buffer manager's actions when the requested page is not in the buffer.

3.4 Cleaning Up Look-Up Table Space

Deferring reference changes will not save much I/O if the buffer manager must do disk accesses to find entries in the look-up tables. To avoid writing portions of the look-up tables to disk, we clean them up when there is no longer room in memory for at least one more entry for each Q_i. (That is, the clean-up process only runs when the look-up tables are full.) In this case, the clean-up process brings up those secondary index pages that have pending reference changes, and finishes the changes as described in previous sections.

When the space is full, the reorganizer has to wait for the cleaning process to clean up the space. No reorganization unit runs while the clean-up process runs. User queries can run concurrently with the clean-up process and have higher priority. When clean-up finishes, a new reorganization unit starts.

4 Recovery

Since we want to keep the look-up tables in main memory, it is essential to be able to reconstruct them after system failure. We are moving records from one physical location to another. Correctly logging the operation should prevent us from losing records in case of system failures.

4.1 Log Records and Data Structures

We use several kinds of log records. Any safe system logs updates to database and index pages. We follow current conventions in making one log record for each updated page [GR93].

- (MV, page_id, *Old-ID*, *New-ID*, record_content)
  An MV log record is the first log record written by the reorganization unit. This log record is written only after the reorganizer has physically moved the record. This record is used to REDO the delete from the page where the record resides before it is moved, if necessary. The *New-Id* is needed for recovery in case this is the first log record of a reorganization unit.

- (MV2, page_id, *Old-ID*, *New-ID*, record_content)
  The MV2 log record is written for the insert into the new page. The *Old-ID* is used to identify the reorganization unit. (Combining MV and MV2 in one log record would mean it would be more difficult to integrate our algorithm into current systems, because most systems do logging once a page is modified. Combining MV and MV2 would also make the recovery logic a little more complicated.)

- (CHANGE, clustering_or_secondary_index, page_id, index_key, *Old-ID*, *New-ID*, leaf_page)
  The CHANGE log record is written when the reorganization unit (identified by the *Old-ID*) updates a leaf page of the clustering index or one of the secondary index leaf pages in the buffer.

- (CATCH_UP secondary_index, page_id, index_key_1, *Old-ID*_1, *New-ID*_1, ..., index_key_s, *Old-ID*_s, *New-ID*_s)
  The CATCH_UP log record is written when a secondary index leaf page is brought into the buffer and there are some items in the look-up tables that need to be piggybacked. We write this one log record for all the piggybacking changes done on the same page. This is less logging than is done in other safe systems, since these changes would be done separately and would require separate log records.

During recovery, we will construct a very small *reorganization table*. At the end of the REDO pass through the log, the reorganization table will have information about the (at most one) reorganization unit which may have been active at the time of the system failure. The table has the following fields:

- ReorgID The reorganization unit identifier. It is the *Old-ID* of the MV log record.

- BeginLSN The LSN (Log Sequence Number) of the MV log record of the reorganization unit, i.e., the first log record written by the reorganization unit.

- LastLSN The LSN of the latest log record written by the reorganization unit.

4.2 Checkpointing

The following are copied into the checkpoint log record: (1) the BV and (2) the look-up tables.

In order to checkpoint the look-up tables, we need to use one semaphore for all these tables. Otherwise, there might be deadlocks between the checkpoint operation and the reorganization process. This is the CK semaphore. When the system wants to do a checkpoint, it tries to get an X latch on the semaphore, thus preventing others from accessing those tables. The reorganizer and the piggybacking processing of the tables by the I/Os of the user transactions will get an S latch on the CK semaphore whenever they want to modify those tables. The effect of this approach is that the checkpoint log record always contains the consistent tables, that is, the *Count* field of each $T[i]$ correctly indicate the number of references there are in the Q_is.

4.3 Why Not Undo?

In our recovery scheme discussed in the next subsection, we don't do undo for the reorganization process. There are two reasons that let us choose this scheme. First, undo tends to slow down the reorganization process while forward recovery speeds up the reorganization process. Second, undo sometimes does not make sense using our algorithm. Our concurrency rules were made under the assumption that we would be doing forward recovery. That is why it was safe to drop some locks before the reorganization unit finished, and thus maximize concurrency. (Transactions which drop locks before commit and then must reacquire them for UNDO are not two-phase locked and risk non-serializability.)

As an example, suppose the reorganization process has just changed $(v1, R1)$ to $(v1, K1)$ on the secondary index leaf page P and released the X lock on index key values $v1$. Now suppose there is a user transaction $T1$ which changes the values of some columns of the record with *New-ID* $K1$. One of the changes is to change $v1$ to $v2$. Suppose $T1$ commits and then the system crashes. If we undo those reorganization units that were not in the READY state when the system crashed, then we would undo the change on page P from $(v1, K1)$ to $(v1, R1)$, which can not be done since $T1$ has committed and $(v1, K1)$ can not be found on page P.

4.4 Recovery Logic

We assume the Write-Ahead-Logging (WAL) is used. We will use the ARIES [MHL+92] recovery scheme. That is, we first do an analysis pass, then we do redo all to recover the status of the system at the time of the crash, and then we selectively undo. We assume that all updates have been made to disk by the redo process. In particular, the changes made in the secondary leaf pages and the clustering index leaf page are redone if necessary. These can be done during the same pass that reconstructs the in-memory tables as described below.

Since the look-up tables in the checkpoint log record are always consistent, we only need to process those reorganization log records after the checkpoint log record during redo. First, we copy into memory all root pages of secondary indexes for the relation being reorganized. When we have a MV log record, we put a new item in T and insert corresponding items into Q_is. We will also modify the BV to be the *Old-ID* of the MV log record. This is why we don't need to log the updates of the BV by the reorganization process during normal processing. In addition, we will insert an entry in the *reorganization table* to identify the move.

When we have an MV2 log record, we update the LastLSN in the reorganization table. When we have a CHANGE log record for *the clustering index*, we will delete the corresponding entry in the reorganization table. This means that the process of deleting the record from its *Old-ID* page, inserting the record to its new page and updating the clustering index is complete. Once this happens, only the secondary index updates remain.

When we have a CHANGE log record for a secondary index or a CATCH_UP log record, we delete or modify the corresponding item(s) in the Q_i(s), and modify the *Count* field of $T[j]$(s) accordingly as described before.

If the *reorganization table* is not empty at the end of redo, this means that some reorganization unit had not finished moving the record and updating the clustering index at the time of system failure. If the LastLSN is the same as the BeginLSN, it means that the reorganization process has deleted the record it is going to move, but hasn't inserted the record in the new page. The recovery process will insert the record and write the MV2 log record. Then it will update the clustering index, write a CHANGE log record for the update and delete the corresponding entry in the reorganization table. If the LastLSN is different from the BeginLSN, it means that the record has been inserted in the new page but the clustering index has not been updated. In this case, the recovery process will only update the clustering index, log the update and delete the entry from the reorganization table.

Before the system restarts, we have to check all the secondary index pages that are in the buffer to make sure there isn't any secondary index page which is referenced in any of Q_is. This is necessary because during the recovery, some secondary index pages could be brought in the buffer which were not in the buffer at the time of the system failure.

For example, problems can arise because of redo of user transactions. Figure 3 shows an example of a possible inconsistent state. Suppose there is a secondary index leaf page, $P2$, that is in the buffer and $P2$ refers to a to-be-moved record. Before the reorganization process gets the X lock on the index value stored in $P2$, $P2$ is updated by a user transaction $T1$. The reorganization process gets the lock on $P2$ after $T1$ commits. Now the reorganization process moves the record, but before it changes the reference on $P2$, the system crashes. Suppose the reorganization process has written the MV log record and that page $P2$ hasn't been written back to disk when the system crashes, then after the redo-all pass, $P2$ should be in the buffer. If the system restarts without checking the look-up tables to see if any pending changes need to be made, then it is in an inconsistent state because $P2$ has an out-dated address of the moved record.

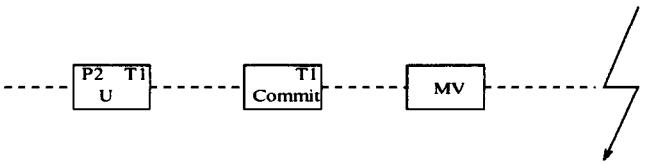

Figure 3: An example of the log

If there are any pages in the buffer at the end of the REDO and UNDO passes referred to by a Q_i entry, we make the pending changes in the look-up tables and modify the tables accordingly. At this point, the recovery process as usual makes all its updates durable and makes a checkpoint record. We also must make sure that ancestors of any secondary index pages in the buffer are also in the buffer. After some debate, we chose to do this by marking the secondary index pages without parents as empty buffer slots (swapping them out). This does not cause extra I/O as the alternative (bringing in their parents) would do.

After the last checking, we will have a consistent buffer. That is, all the secondary index leaf pages in the buffer either do not refer to the moved records, or already have the correct address. (This last checking also has the effect of "forward recovery" [ZS96b], for it can finish any reorganization unit which had not finished moving the record and updating the clustering index and putting pending changes in T and Q_is at the time of the system failure.) The system can restart. Reorganization can start from the record whose RID is next to the BV. Please refer to [Zou96] [ZS96a] for detailed discussion about the logic of the recovery scheme. It also discusses the recovery of process failure.

5 Correct Database Operations during Reorganization

In this section, we will describe the normal user transaction's behavior during the reorganization process. Additional discussion on adding an index, dropping an index and cancellation of the reorganization can be found in [Zou96] [ZS96a].

5.1 Search through Secondary Index

We assume that searchers use S latch-coupling down the secondary index tree and that when they reach a leaf, they request an S-lock on the searched index key (the leaf page is already in the buffer, as the searcher has to read the page before requesting the S-lock as in [Moh90]). After obtaining the S-lock on the searched index key, they proceed to read the content of the relevant index entries.

A reorganizing unit could already have gotten an X-lock on the index key value v when a searcher is requesting a conditional S-lock on v (and holding an S-latch on the leaf page where v index entry is stored). If this happens, the searcher will drop all the latches it holds, and request the S-lock unconditionally. This will avoids possible deadlock involving latches when the reorganizer traverses the index tree requesting latches on a page while holding an index key value lock.

If the reorganizer succeeds in obtaining all the index key value locks and finishes its processing of the updates on secondary indexes, the searcher will eventually find the *New-ID*. If the reorganizing unit gives up its locks due to deadlock, the searcher will see the *Old-ID* and run before the reorganizing unit. In all cases the search is correct. The search algorithm can be more complicated, as shown in [ZS96a] [Zou96], when the insertion of a *New-ID* causes a split and the piggybacking algorithm is different.

Notice that the requirements of the buffer page replacement policy are essential to the correctness of the searcher algorithm. Suppose that an index leaf page L and its parent page P are brought into the database buffer by a searcher, and the reorganization unit had acquired an X lock on v for record R's move, and v is on stored only on page L. When the searcher tries to get a conditional S lock on v, it will be blocked.

If page P is swapped out of the buffer before the reorganization unit processes that secondary index, an entry of (v, P, R) will be put into the look-up table. Then the reorganization unit will release its X lock on v. The searcher will get the S lock on v, and it can read the index entry on L, because the leaf page L is still in the buffer. But this will get an out-of-date address, as L hasn't been updated yet. So we can not swap out a parent page unless all its child pages have been swapped out.

In addition, the searcher has to use the *crabbing* technique [GR93] during its tree traversal. That is, the parent page won't be unfixed until the child page is fixed. For example, the searcher could read page P and get the L's address first. Suppose none of P's children is in the buffer. If the searcher unfixes P before L is fixed (brought in the buffer), P could be swapped out because at this instant, it has no children in the buffer. Then when L is brought in, we have the same problem as explained in the above paragraph. So the searcher has to use the *crabbing* technique during its tree traversal.

In addition, for *fetch_next* [Moh90] we require that when following a leaf side pointer to the next leaf, the parent must be in memory. Usually, since index nodes have on the order of 200 entries, this is the case. Occasionally, a parent (or even a higher ancestor) may have to be brought in before the next leaf can be fetched.

5.2 Scanning

When a large number of records are to be read, or when a query is made on a non-indexed attribute, *scanning* of the entire relation without using any index is often necessary. Scanning while the reorganization process is running implies that the scanner has to scan the part of the relation that is still using the *Old-IDs* as well as the part of the relation using *New-IDs*. If we do not choose the locking protocol used by the scanner carefully, the scanner can see the same record twice (before and after it is moved) or miss a record that has been moved.

One solution to these problems is to let the scanners follow the *level 3 consistency* or *repeatable read* (keep all locks until end of transaction), and let the scanners lock with page or larger granularity.

Another solution is to let the scanner get a read lock on the *BV*. Since every reorganization unit will first get an exclusive (write) lock on *BV*, if we let the scanner hold the shared lock on the *BV* until it finishes its scanning, then we are safe. The advantage of this solution is that the scanner can hold the read lock on the records or pages for only *manual duration*, thus allow more concurrency for user transactions. The disadvantage of this approach is that the reorganization is blocked by the scanner that gets the shared lock on *BV*.

For discussion on record updates, deletes and insertions, please refer to [Zou96] [ZS96a].

6 Performance Analysis

Detailed analysis and proofs can be found in [Zou96] [ZS96a]. We will summarize the results here briefly.

6.1 Properties of the Algorithm

Our algorithm has the following properties:

- The number of I/Os for the reorganization process will increase as the interval between two record moves becomes smaller. When that interval becomes smaller, the look-up tables will be filled more quickly, thus causing more clean-up processes to be run. This means that the number of I/Os (due to cleanups) will increase.

- The number of I/Os for the reorganization process will decrease as the number of user transaction I/Os between two clean-up I/Os increases. More piggybacking of changes in the look-up tables could be done between two clean-up I/Os.

- The number of I/Os for the reorganization process will decrease as the size of the look-up tables increases. If the look-up tables' size becomes larger, the look-up tables will be able to store more pending changes. This in turn means that less clean-up process I/Os are are needed and more piggybacking can be done for each secondary index leaf page brought in the buffer.

6.2 Performance Results

We compared our method with a transaction-based reorganization method [SD92]. In the transaction-based reorganization method, moving one record and changing all corresponding references are encapsulated into one transaction. Because the cost of moving one record is the same for both methods, we use the number of disk I/Os made to update the secondary indexes as the measure of efficiency. Experimental results [Zou96] [ZS96a] show that our method uses much less disk I/Os than [SD92].

7 Related Work

Recently, a number of papers on on-line database reorganization have appeared due to the resurgence of interest in this area. Algorithms for on-line construction of secondary B^+-tree indexes can be found in [MN92] [SC92]. Reclustering of records can be found in [OLS92] [OLS94]. Resequencing (compacting) of primary B^+-trees is dealt with in [ZS96b] [Smi90]. Repartitioning distributed data in Tandem's NON-STOP SQL is described in [Tro96]. Moving an RID organization to another site in a parallel database is discussed in [AON96]. Reclustering an RID organization, the problem treated in this paper, is discussed in [SI96].

In [SI96], a new utility for reclustering an RID organization which will soon be available in IBM's DB2 is described. This algorithm constructs a copy of the relation in another area, noting the LSN of the log at the beginning of the copy. A new clustering index and several new secondary indexes are built for the new copy of the relation. During the copy, users read and write to the old organization. When the copy is finished, the log is used to catch up on the updates. After catch up, the relation is frozen, and the remaining catch-ups to updates made during catch-up are performed. Then the database switches to the new RID organization. This is not restartable and does not provide incremental improvement, as the new area cannot be used during the copy and catch-up processes. If a crash occurs, one starts over. Enough disk space for two full copies of the relation is needed.

In [SD92] records are moved from one location to another location one at a time. All references to this record are changed. The record move and the reference changes are encapsulated in a database transaction. This is probably too slow for massive reorganization problems, because it makes changes to the secondary

index leaf pages immediately which would require a disk I/O for every secondary index leaf page that is not the buffer at the time of the move. This is reflected in our performance comparisons.

In [OLS92] and [OLS94], records are reclustered in place. A number of pages are read into a buffer and locked. Records are moved from one page in the buffer to another until a page is constructed where the clustering is satisfactory. Concurrency and recovery is not discussed in [OLA91] [OLS92] [OLS94]. Since records are moved and there are references to the records, the database could become inconsistent if there is a system failure. These algorithms are not safe.

In [OLS92] and [OLS94], a **differential file** [OLA91] is used so that references to the records can be changed later. The differential file is used to record the old and new key values of the moved records. It is assumed to be small enough to lie in main memory. When a user query is made through a secondary index, the differential file is searched and if there are some items in the differential file that match the *requested* keys of the *user query*, modification of the differential file and the secondary index leaf pages (which are in buffer now) are made. No changes to references are made immediately when a record is moved, even if the relevant index leaf pages are in memory.

We also use some in-memory book-keeping data structures that are similar to the differential file. We believe our algorithm has the following advantages:

- Our algorithm is safe and restartable with no loss of work. The restartability and the safeness of our algorithm makes it more practical than [OLA91].
- Our algorithm is more efficient than [OLA91] in terms of CPU costs and the time it takes to finish the entire reorganization process.
- If the differential file is too big to fit in memory, our algorithm is more efficient in I/O costs than [OLA91].

8 Conclusion

In this paper we have shown how to update secondary indexes efficiently when moving records. We have framed our algorithm in terms of reclustering an RID organization whose performance has declined, a scenario that is likely to repeat several times over the lifetime of an RID organization. Our new on-line reorganization algorithm defers secondary index updates and piggybacks them with user transactions, significantly reducing the total I/O cost. In addition, we carefully designed the logging scheme and recovery algorithm so that in case of system failure, the look-up tables can be reconstructed and reorganization units can be completed if they had been interrupted.

Analysis and performance study [Zou96] [ZS96a] show that our method uses much less I/O over the method of [SD92] which uses a single-transaction-per-record approach. These demonstrate the superiority of our method which piggybacks with user transactions some of the I/O needed for updating references to moved records.

Another advantage of our method over [SD92] is that we never UNDO any of a reorganization unit's work. In addition, we release all locks on the database pages immediately after the record is moved (before even the in-memory secondary index updates are made). We also release the lock on a secondary index leaf page immediately after it has been processed. This makes our method have more concurrency than [SD92].

Even though we discuss our algorithms in the context of a particular on-line database reorganization, our method can be used as a *general* technique for deferred secondary index updates. In [ZS96a] [Zou96], for example, shows how to use this method in moving from an RID organization to a primary B^+-tree organization. As another example, when restricted to one reorganization unit at a time (moving one record), the algorithm could be used to move an updated record which would have caused a page overflow instead of installing forwarding pointers. Since forwarding pointers make retrieval inefficient, this could be quite useful.

Our method is efficient in I/Os and interruptible without loss of work. It provides incremental improvement, guarantees consistency even if there is a failure and provides high availability through minimal locking. Since we have worked out in detail concurrency and recovery algorithms in the context of modern database systems now in use, we believe our method could be easily integrated with a commercial DBMS.

References

[AON96] Kiran J. Achyutuni, Edward Omiecinski, and Shamkant B. Navathe. Two techniques for on-line index modification in shared nothing parallel databases. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 125–136, Montreal, 1996.

[BS77] R. Bayer and M Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, Inc, 1993.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[MN92] C. Mohan and Inderpal Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 361–370, 1992.

[Moh90] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *International Conference on Very Large Data Bases*, pages 392–405, Brisbane, Australia, August 1990.

[OLA91] Edward Omiecinski, Wei Liu, and Ian Akyildiz. Analysis of a defered and incremental update strategy for secondary indexes. *Information Systems*, 16(3):345–356, 1991.

[OLS92] E. Omiecinski, L. Lee, and P. Scheuermann. Concurrent file reorganization for record clustering: A performance study. In *International Conference On Data Engineering*, pages 265–272, 1992.

[OLS94] E. Omiecinski, L. Lee, and P. Scheuermann. Performance Analysis of a concurrent File Reorganization Algorithm for Record Clustering. *IEEE Transactions on Knowledge and Data Engineering*, 1994.

[SC92] V. Srinivasan and Michael J. Carey. Performance of on-line index construction algorithms. In *International Conference on Extending Database Technology*, pages 293–309, 1992.

[SD92] B. Salzberg and A. Dimock. Principles of transaction-based on-line reorganization. In *International Conference on Very Large Data Bases*, pages 511–520, 1992.

[SI96] Gary Sockut and Balakrishno R. Iyer. A survey of online reorganization in IBM products and research. In *Data Engineering Bulletin*, pages 4–11, June 1996.

[Smi90] Gary Smith. Online reorganization of key-sequenced tables and files. *Tandem System Review*, 6(2):52–59, October 1990. Describe algorithm of Franco Putzolu.

[Tro96] Jim Troisi. NonStop SQL/MP availability and database configuration operations. In *Data Engineering Bulletin*, pages 12–18, June 1996.

[Zou96] Chendong Zou. *Dynamic Hierarchical Data Clustering and Efficient On-line Database Reorganization.* PhD thesis, College of Computer Science, Northeastern University, 1996.

[ZS96a] Chendong Zou and Betty Salzberg. Efficiently Updating References During On-Line Reorganization. Technical Report NU-CCS-96-08, College of Computer Science, Northeastern University, 1996.

[ZS96b] Chendong Zou and Betty Salzberg. On-line reorganization of sparsely-populated b+trees. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 116–125, Montreal, Canada, 1996.