# On-line Reorganization of Sparsely-populated B$^+$-trees

**Chendong Zou**
College of Computer Science
Northeastern University
email: zou@ccs.neu.edu

**Betty Salzberg** *
College of Computer Science
Northeastern University
email: salzberg@ccs.neu.edu

## Abstract

In this paper, we present an efficient method to do on-line reorganization of sparsely-populated B$^+$-trees. It reorganizes the leaves first, compacting in short operations groups of leaves with the same parent. After compacting, optionally, the new leaves may swap locations or be moved into empty pages so that they are in key order on the disk. After the leaves are reorganized, the method shrinks the tree by making a copy of the upper part of the tree while leaving the leaves in place. A new concurrency method is introduced so that only a minimum number of pages are locked during reorganization. During leaf reorganization, *Forward Recovery* is used to save *all* work already done while maintaining consistency after system crashes. A heuristic algorithm is developed to reduce the number of swaps needed during leaf reorganization, so that better concurrency and easier recovery can be achieved. A detailed description of switching from the old B$^+$-tree to the new B$^+$-tree is described for the first time.

## 1 Introduction

B$^+$-trees are one of the main indexing methods used in commercial database systems. After a B$^+$-tree has been used for a while, its performance will often degrade. The degradation could be caused by both insertions and deletions. Large numbers of insertions often cause page splits in the B$^+$-tree, with the result that the leaf pages within a key range in the B$^+$-tree are not in contiguous disk space This will require more disk read time for a range query. Large numbers of deletions will cause the pages (mostly leaf pages) in the B$^+$-tree to be sparse, and this also makes reads inefficient, because for the same amount of data, it will take more page reads for a sparsely populated B$^+$-tree than for a normal(unsparse) one.

In this paper, we show how to do on-line reorganization of sparsely populated B$^+$-trees with good concurrency, little loss of work in case of system failure and a minimal amount of logging. In section 2, we define the problem. We outline our solution in section 3. Section 4 contains the concurrency algorithm used for reorganization of groups of leaf pages with the same parent and for swapping of leaf pages. Section 5 describes our forward recovery algorithm used for leaf page reorganization. Section 6 presents the overall reorganization algorithm. Section 7 discusses the algorithm for reorganizing the upper levels of the tree, and the concurrency and recovery problem associated with it. Section 8 describes related results. Section 9 contains a summary and a discussion of future work.

## 2 The Problem

The B$^+$-tree was introduced by Wedekind in [Wed74]. It is a balanced tree where the distance between the root and any leaf is the same. In the variation in this paper, a B$^+$-tree internal node with $n$ keys has $n$ children.

The B$^+$-tree we discuss here is used as the primary index for the database, where leaf pages contain the data records. We assume that the B$^+$-tree is *sparsely populated*, that is, a large portion of many leaf pages is unused. There are also some free pages available in the database, which are not connected to the B$^+$-tree.

We assume that there is no ongoing node consolidation in the B$^+$-tree, that is, the free-at-empty [JS93] policy is adapted for leaf pages. In the free-at-empty policy, non-empty sparse nodes are never consolidated, but when a node becomes completely empty, its page is deallocated and its parent is updated to reflect this. Most commercial systems adapt this policy.

Since availability of the database system is essential, it is important that the reorganization process does not affect the system performance very much. So solutions which require locking up the entire B$^+$-tree to do reorganization are out of question. In our solution, we are very careful to lock as little of the database as

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



SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

possible. In addition, we log some information so that reorganization work already done is not lost in case of system failure. Since log size is a concern, we try to make the amount of information logged small.

## 3 The Solution

There are basically two ways to perform on-line reorganization. One is to concurrently create a new structure somewhere else in the disk, and switch to the new structure when it is built. We call this *new-space reorganization*. The other method is to do the reorganization in-place, that is, to try to move data(records) around disk pages without using new disk space. We call this *in-place reorganization*.

Our approach is a combination of both methods. We reorganize the leaf pages of the $B^+$-tree using both the in-place method and the new-place method. Since the $B^+$-tree is used as the primary index of the database, and the size of the database could be very large, perhaps several terabytes, it is very unlikely that there is enough space in the disk to create the new $B^+$-tree. So an in-place method has to be used in order to reorganize the leaf pages. However, as we expect the tree to be sparsely populated and as we expect to have some free space not used by the tree, we are able to mix the in-place operations with some new-space operations.

After the leaves are compacted, they still may not be in key order on the disk, and there may be empty pages scattered among the new compacted leaf pages. At this stage, we may choose to do some swapping of leaf pages and moving of leaf pages to empty pages to put the pages in order. (Swapping two leaf pages is an in-place operation and moving to an empty page is a new-place operation.) We use a separate pass for swapping and moving after the compacting pass because (1) sometimes this pass could be omitted (leaving the pages out of order) if the performance of the $B^+$-tree is not so bad; (2) this two-pass method has some advantages in concurrency and log size as we shall see later.

We use only the new-place method to reorganize the internal pages of the $B^+$-tree. Even though the database could be very large, the space needed for the internal pages of the $B^+$-tree is still relatively small, so there should not be any problem of finding disk space to create the new internal pages. Another important concern here is the availability of the $B^+$-tree. If we reorganize the internal pages of the $B^+$-tree using an in-place method, then we have to lock part of the $B^+$-tree. Since we might have to work on the root, this would force us to lock up the entire $B^+$-tree, which would make it unavailable. Our method using the new-place reorganization for internal pages will guarantee that the reorganizer will at most lock one internal page (and only with a share lock) until it is time to switch to the newly created tree.

Our algorithm is a three-pass algorithm. We first compact the leaf pages using both in-place and new-place operations. In the second pass, we swap leaf pages and move to empty pages to make the disk space used by the leaves contiguous and in key order. (During the first and second passes, ongoing database activity may split leaves in areas that have already been reorganized. We do not try to clean this up, so the result at the end of the first two passes is not necessarily a *perfectly* ordered compacted set of leaves.) In the third pass, we reorganize the internal pages using a new-place algorithm to make the internal pages more compact so that it might reduce the height of the tree. Our three-pass algorithm is illustrated in Figure 1.

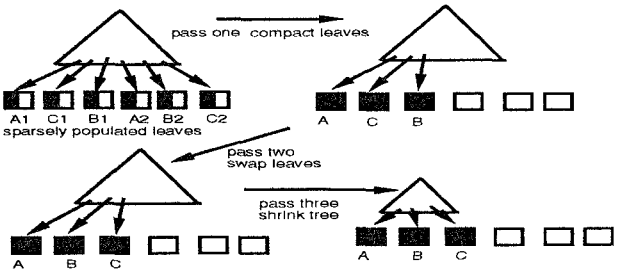


Figure 1: Three-pass Algorithm

For convenience, we will call $B^+$-tree internal pages that are right above the leaf pages in the $B^+$-tree *base* pages, since these are the pages where we get the information about leaf pages. The base pages are at the parent-of-leaf-level in the $B^+$-tree.

In the first pass, each separate operation on the leaves involves only one base page. This enables us to use locking protocols that affect at most a part of the tree which is below that one base page. (An exception occurs if there are side pointers to be maintained. In this case, a child of a neighboring base page may be updated to change the side pointer.)

In the next section, we will show how to reorganize several children of one base page in an in-place operation. We will explain in detail the locking protocol for this operation.

In section 5, we will look at the recovery problem when reorganizing the leaf pages. In section 6, we will put together the operations and show how they are orchestrated to reorganize the whole tree.

## 4 Concurrency for Leaf Page Operations

In this section, we will discuss the concurrency problem for the leaf page operations during our reorganization. We assume there is a "tree" lock which all operations must obtain first. This is a large granularity lock and

Table 1: Lock Compatibility

| Granted | Requested | | | | | | |
|---|---|---|---|---|---|---|---|
| | *IS* | *IX* | *S* | *X* | *R* | *RX* | *RS* |
| *IS* | Yes | Yes | Yes | No | | No | |
| *IX* | Yes | Yes | No | No | | No | |
| *S* | Yes | No | Yes | No | Yes | No | Yes |
| *X* | No | No | No | No | No | No | No |
| *R* | | | Yes | No | | | No |
| *RX* | No | No | No | No | | | |

is usually requested in IX or IS (intention exclusive or intention share) mode.

### 4.1 In-Place Compacting and Swapping of Leaf Pages

For each in-place compacting operation at the leaf level, we move the contents of several leaf pages, all of which are children of the same base page, to one of them and we update the base page. For a swap operation, we swap the contents of two leaf pages, and update both their parents to reflect the change. In this section, we discuss how to do this with high concurrency.

A simple way of dealing with this problem would be to get a $X$ lock on all the pages involved in one *unit* of reorganization. This includes both the base pages and the leaf pages. Since each base page might contain pointers to around two hundred leaf pages, and one unit might involve several base pages, this method could block a lot of readers and updaters.

Instead, we propose a fine granularity locking method which would provide better concurrency for both readers and updaters. Our new locking protocol only holds an X lock on base pages for a short period of time, after the records in the leaf pages have been reorganized.

We introduce three new lock modes: $R$, $RS$ and $RX$. Table 1 shows the compatibility of the locks. The left column represents the lock mode that is already held by other transactions. The top row shows the new requesting lock mode. The blank in the table means that the two lock modes won't be requested together by different requesters. (This happens when, for example, one lock mode is only used on leaf pages and another only on base pages.) We assume readers and updaters may request or hold intention locks ($IX$ or $IS$) (*on leaf pages only*) if they are doing record-level locking [GR93]. ($RS$ is not listed in granted mode in Table 1 because as an "instant duration lock", it is never actually granted.)

The $R$ mode is used by the reorganizer to lock the base pages which contain pointers to those leaf pages that are in one reorganization unit. It allows the reorganizer to read the content of the page. (This is before the reorganizer modifies the keys in the base page.) It is compatible with the $S$ lock. This means that when the reorganizer holds an $R$ lock on the base page, a reader could get an $S$ lock on that base page and read. Similarly, when the reader gets an $S$ lock on a base page first, the reorganizer can also get its $R$ lock on that page and read.

The reorganizer will hold $RX$ locks on the leaf pages in a reorganization unit. The $RX$ mode is not compatible with any lock mode. $RX$ is not the same as $X$, because the action of the lock manager when a conflicting request arrives is different. With a $X$ lock, a conflicting request causes the requester to wait. With an $RX$ lock, a conflicting request causes the requester to forgo the conflicting request, release its lock on the base page and request an $RS$ lock on the base page.

The $RS$ mode is not compatible with $R$. It is used to block the reader from reading the leaf pages that are being reorganized. During reorganization, records are moved between leaf pages, so the keys in the base pages have to get changed to maintain search correctness in the $B^+$-tree. Since the old path leading to that leaf page might get changed after the reorganization, we have to force readers to read the base page after the reorganization to get the new path.

The solution we propose here is that the reader request an *unconditional instant duration* [Moh90] $RS$ lock instead of a normal lock request after it is blocked by the reorganizer. An *unconditional instant duration* lock means that the lock is not to be actually granted, but the lock manager has to delay returning the lock call with the success status until the lock becomes grantable. This will achieve the goal of blocking the reader from reading the reorganizing data. After the success status is returned for the $RS$ lock request, (but the reader doesn't actually hold the lock), the reader will request a $S$ lock on the base page and proceed. This is a good solution because the cost of an *instant duration* lock is small, and the protocol can block the reader from reading any reorganizing data.

Since the reorganizer is locking more than one page, deadlock can occur. For example, suppose a reader gets an $S$ lock on page $A$ first, and then attempts to get an $S$ lock on page $B$, where $B$ is going to be swapped with $A$ or compacted with $A$. Suppose that the reorganizer has already got an $RX$ lock on $B$ and is attempting to get an $RX$ lock on $A$. The reader will give up its locks on $B$'s parent, $P_B$ and wait for reorganization to finish by requesting an $RS$ lock on $P_B$. So the reader is blocked. Also the reorganizer is blocked waiting for the lock on $A$.

Whenever the reorganizer gets in a deadlock, we always force the reorganizer to give up its lock. Note that we always require the reorganizer to get all the $RX$ locks and $R$ locks before it starts to move data, so that by forcing it give up its locks, it usually won't have to

roll back a lot of work.

However, once it has obtained its $R$ locks and all its $RX$ locks, the reorganizer must still convert its $R$ locks to $X$ locks to update the base pages. Then there can still be a deadlock. However, more then one user transaction has to be involved, producing a deadlock cycle of length at least three.

#### 4.1.1 Reorganizer Protocol

Here is how the reorganizer works on each reorganization unit:

- $IX$ lock the tree lock.
- $S$ lock-couple down the tree until it reaches the base pages.
- $R$ lock the base page(s) and then $RX$ lock the leaf pages that are going to be reorganized. (This will block readers of the pages that will be reorganized.)
- Move records between leaf pages.
- Upgrade its lock on base pages to $X$ mode. This would prevent anyone from reading it for a short period of time.
- Modify necessary keys and pointers in the base pages.
- Release locks.

#### 4.1.2 Reader Protocol

The reader will work as follows:

- $IS$ lock the tree lock.
- $S$ lock-couple down the tree. If it can't get an $S$ (resp. $IS$) lock on the leaf page, and the conflicting lock is $RX$, it releases its lock request on the leaf page and the $S$ lock it has already held on the base page. Then it tries to get an *unconditional instant duration* $RS$ lock on the parent base page. This will guarantee that it will be blocked until the reorganizer finishes. After the success status is returned from the $RS$ lock request call, the reader will request a $S$ lock on the base page.
- $S$ (resp. $IS$) lock the leaf page and read.(Often an $S$ lock is first requested on the page then the read takes place, then the $S$ lock on the page is downgraded to $IS$ lock while an $S$ lock on the read record is held to the end of transaction.)
- Drop all locks at end of transaction.

#### 4.1.3 Updater Logic

Updaters which cause splits or node consolidations requiring base page modifications have a special interaction with the *Switch* operation explained in the next section. Otherwise, updater logic is very similar to reader logic. The updater will act as follows:

- $IX$ lock the tree lock.
- $S$ lock-couple down the tree. If it can't get an $X$ (resp. $IX$) lock on the leaf page, and the conflicting lock is $RX$, it releases its lock request on the leaf page and the $S$ lock it has already held on the base page. Then it tries to get an $RS$ lock on the parent base page. This will guarantee that it will be blocked until the reorganizer finishes.
- $X$ (resp. $IX$) lock the leaf page. If no split (or consolidation) is needed, then do the update.
- If a node split or consolidation is needed and Bayer-Scholnick safe node [BS77] is used, all locks are released and the operation starts again. It uses lock coupling with X-locks until a safe node is reached and then releases ancestors of the safe node. This will wait for a reorganizer when it attempts to get an X-lock on a base page.
- If [LS92] (the Π-tree) is being used, a split can be made once an X-lock is made on the leaf page to be split. The updating of the base page can be done later. In this case, side pointers are necessary.
- When updating on a base page is done, the operation doing the updating must also first test if there is an internal(index) page reorganization process running. If so, there will be some extra testing and locking which will be explained in section 7
- Drop all locks at end of transaction.

In the above description, latches could be used instead of locks to improve the efficiency. But since there is no deadlock detection for latches, we would have to force the reorganizer to release all its latches when it cannot get a latch on a page.

### 4.2 New-Place Copying and Switching of Leaf Pages

A copying and switching operation can be used in the compacting process, where we copy the content of several leaf pages to one empty page and make that page the new leaf page. It can also be used in the swapping phase, where we copy one already compacted leaf page to an empty page.

There are several options for the concurrency protocol to be used in this operation. One way is to use the locking protocol for the in-place compacting operation described above. We put an $R$ lock on the base page and an $RX$ lock on each leaf page that is to be copied. All the locks should be acquired before the operation as discussed in the above section. This protocol is simple to use, and easy to implement. However, since we put an $RX$ (Reorganizer eXclusive) lock on each involved leaf page, we are actually preventing readers and updaters from accessing the involved leaf pages. Alternative protocols for new-place leaf operations are described in [ZS95].

### 4.3 Side Pointers

Many B$^+$-trees have side pointers at the leaf level to make searching in key order more efficient. If leaves are moved, these side-pointers must be adjusted. Depending on the type of side-pointer (one-way side-pointer or two-way side-pointer) the system has, and the type of operation (in-place compacting, copying and switching, or swapping) involved in the reorganization unit, the reorganizer has to $RX$ lock some number of leaf pages ($X$ lock for those leaf pages that are not children of the same base page as the leaf pages being reorganized) to make the side-pointer changes in order to keep the B$^+$-tree consistent. Since acquiring $RX$ or $X$ locks after the records have been moved can result in deadlock, and aborting the reorganizer will undo a lot of work, we will let the reorganizer acquire all the necessary locks before it starts moving records. This includes locks that are necessary for updating the side-pointers.

## 5 Logging and Recovery for Leaf Page Reorganization

When doing leaf page reorganization, there is a danger of losing records if no recovery mechanism is provided for the reorganizer. The method we propose is to log each action of the reorganizer. We assume that WAL is used (Write Ahead Logging).

There are several kinds of log records. Each corresponds to one type of operation by the reorganizer.

- Begin record.
  (BEGIN, Unit_$m$, Type, base_page1, ... , base_page$i$, leaf_page1, ... , leaf_page$k$)
  This log record is only written after all leaf page locks for the reorganization unit are acquired. It lists all the base pages and leaf pages in the unit. Unit_m is a monotonically increasing integer to identify the unit. The field *Type* is to identify what type of reorganization of this unit is. Its possible values are:
  - *Compact*: compacting the leaf pages under the same base page.
  - *Swap*: swapping two leaf pages under one or two base pages.
  - *Move*: moving one leaf page under one base page to an empty page.
- Moving Records.
  (MOVE, record_contents, org_page, dest_page, prev_LSN)
  This log record says that we are moving some records with *record_contents* from org_page to dest_page. This only involves leaf pages. We will always write the MOVE log record for the org_page page first, then write the MOVE log record for the dest_page. Instead of record content, we could use only the keys of records if "careful writing" by the buffer manager is enforced [LT95]. With careful writing, a page which will be deallocated cannot be deallocated until the new page where its contents was copied is on disk. A page which had only a part of its contents copied to a new page cannot be written to disk until the new page is written to disk. (When we do swapping of leaf pages there is no way to avoid logging at least one of the full page contents.) Prev_LSN is the LSN of the previous log record for this same reorganization unit.
- Modifying the keys in the base page.
  (MODIFY, base_page, org_key, org_pointer, new_key, new_pointer, prev_LSN)
  This describes the modification of the base key and base pointer after moving the records, . This is necessary to keep the B$^+$-tree consistent.
- End record.
  (END, Unit_m)
  When the reorganization of the unit is done, we write this log record.

In case of system failure, we want to know if there is any on-going reorganization unit that hasn't completed yet. We also want to know where to restart the reorganization in case of system failure. We keep an in-memory table to record the minimum LSN of the current reorganization unit. (It should be the LSN of the BEGIN log record of the reorganization unit.) We keep the most recent LSN of the unit. We also record the largest key($LK$) of the last finished reorganization unit processed. This table should be a system table. It should be very small. It will be copied to the log checkpoint record.

Whenever a new reorganization unit starts, it puts the LSN of its BEGIN log record into this table. Whenever it writes a log record, it puts its most recent LSN into the table, and this is used for the prev_LSN field of the next log record. When the reorganization unit finishes, it deletes its entry in the table, and changes the value of $LK$ to the largest key it has just processed. (This information, together with the transaction *low-water mark* [GR93], can be used to calculate the low-water mark for system recovery—i.e., the lowest LSN that must be kept available for recovery.) So far, we are treating a reorganization unit very much like an individual database transaction, in terms of logging.

Since we are doing reorganization using one process, the table should have one (only $LK$) or two ($LK$ and LSN of the BEGIN log record) or three ($LK$ and LSN of BEGIN and most recent LSN) items at any time. If it has only $LK$, then it means that the last reorganization unit has finished and the new one hasn't started yet.

We can use $LK$ to find the correct position to restart the reorganization in case of system failure. If there are LSNs in the table, we can look at the log records and do *Forward Recovery* as explained in the next section.

### 5.1 Forward Recovery

We assume as in [GR93] that a redo pass is run first from the earliest LSN of a buffer page with data from that log record which was not written to disk at the time of the last checkpoint. This bookkeeping during forward processing and redoing at recovery is not affected by the fact that some of the actions logged are those of the reorganization process. After the redo pass, all forward operations from the log will have been installed in the database and from the information in the most recent log checkpoint record and the log records that come after it, the list of incomplete transactions (to be aborted) and possibly an incomplete reorganization unit, and their most recent log record LSNs will be known.

As the transaction undo progresses going backwards in the log looking at prev_LSNs, so also can all the information about the one possible incomplete reorganization unit be gathered. One finds out what remains to be done and what locks must be obtained to do it. (The undoing of database transactions could also be finished after restart as an alternative—in this case, they also must reobtain locks to undo their actions.)

The reorganization unit will be able to finish the work instead of rolling back and wasting the work that has already been done. This usually isn't an option for normal transaction processing, because you don't know what the user transaction wants to do next. It is a special case here, since we know this is a reorganization processing unit, and we know what type it is by looking at the *Type* field of the *BEGIN* log record.

Thus in our system, recovery for the reorganization process is special. Not only does it not do undo, it also goes forward to finish the unfinished work. We call this *Forward Recovery*. Using this technique, we would acquire locks during system recovery, then go forward during normal processing after recovery.

Using *Forward Recovery*, an interesting observation is that the reorganization process gains some leverage after the system failure, assuming that transaction undos are done before restart. In this case, reorganization could acquire some locks on base pages that had been previously held by user transactions which were aborted because of the system failure.

### 5.2 Undo at Deadlock

Although no work from a reorganization unit is undone at system failure, work must be undone if the reorganizer has already moved records and gets into a deadlock situation. Our concurrency algorithm makes this unlikely, but not impossible. In case of deadlock, the chain of prev_LSNs can be used to find log records to undo a reorganization unit.

## 6 The Full Algorithm for Leaf Page Reorganization

In this section, we put together the in-place and new-place compacting and switching algorithms for leaf pages. We assume that before the reorganization, the page fill factor of the leaf pages of the B$^+$-tree is $f1$, and we want the page fill factor to be $f2$ after the reorganization, where $f2 > f1$.

We are going to do the reorganization of leaf pages by order of the key values in the leaf pages. That is, we start from the leaf page associated with the smallest key in the base pages. For each leaf not yet compacted, we decide whether to do in-place compacting or new-place copying and switching.

We can either use the free space generated by the in-place compacting or existing empty space to do copying and switching, or we can do in-place compacting again depending on the location of the empty space. Finally we are going to swap leaf pages to make them contiguous in the key order. We elect to separate the compacting of leaf pages and swapping of leaf pages to two passes. We want swapping to be optional, that is, the user can decide not to do swapping to make the leaf pages in exact key order. Swapping tends to lock more base pages than compaction and thus blocks more user transactions. One scenario we envision is choosing to do swapping only when range query performance falls below some acceptable level.

```
While(more leaves) {
  Find-free-space;
  If there is appropriate free space
      Copying-Switching;
  Else
      In-Place-Reorg;

}/*end while more leaves*/
Swapping_Moving;
```

Figure 2: Algorithm for leaf reorganization.

In Figure 2, we show the outline of the main program for reorganizing the leaves. In it, we call four subroutines: Find-Free-Space, Copying-Switching, In-Place-Reorg and Swapping-Moving. Find-Free-Space will see if there is a "good" (i.e. well-placed) empty page we can use. If so, we call Copying-Switching, which will copy data into the good empty page using new-place concurrency. If not, In-Place-Reorg is called. Finally, if we want the pages to be in key order and not to be mixed with empty pages, we call Swapping-Moving to sort the compacted leaf pages.

We assume that the leaf pages and internal pages are in a different part of the disk or in different disk. So when we read the leaf pages in the physical order, we only encounter either leaf pages or empty pages.

We choose to construct one new leaf page at a time for the leaf page reorganization. While we could construct more then one page, it would require the reorganization unit to hold locks longer, thus it will block more user transactions. So on average, $d$ ($d = \lceil f2/f1 \rceil$) pages get compacted in each reorganization unit.

### 6.1 Finding a Good Empty Page

We have to be careful in choosing an empty page to construct a new compacted leaf page. Our goal is to minimize the amount of swapping (as opposed to moving to an empty page) done in the second pass. Moving to an empty page allows more concurrency than swapping since swapping usually involves two distinct base pages. In addition, swapping cannot take advantage of careful writing, so the entire contents of at least one of the pages being swapped must be written to the log. Since log size is a significant factor in reorganization methods, this is important.

In our algorithm, we choose the first empty page which is in front of the leaf page that is going to be reorganized, C, and after the largest finished leaf page ID, L. This forces C always to move to the "left" or towards the beginning of the data collection. Since the total number of leaf pages after reorganization is going to be smaller, this is the correct direction. Requiring that the empty space be after the largest reorganized page L means that the new page constructed will be in the correct relative order with all the leaf pages that have already been compacted and that C will be moved closer to L. Initial experiments showed that our algorithm can greatly reduce the number of swaps needed at the second pass [ZS95].

## 7 Reorganizing the Internal Pages

We choose to reorganize the internal pages using the new-place method because it avoids the possibility of locking up the entire B$^+$-tree. The idea is simple. Since the keys in the *base* pages are already sorted, we start by reading the *base* pages from left to right, that is, we read the keys in ascending order. We start building a new B$^+$-tree in a bottom-up fashion.

Constructing a B$^+$-tree from sorted records in a bottom-up fashion is described in chapter 5 section 5 of [Sal88]. Essentially, the records are copied to newly allocated empty pages as they arrive. When a new page is added, no splitting is necessary. The first page is filled to a pre-assigned fill factor, and then the next records go in the next page. Each new page requires a new entry in the level above. At all levels, when a page is filled to the fill factor, a new empty page is allocated and the next record or pointer to a record is entered there.

The reorganizer only holds an $S$ lock on the *base* page that it is reading, so other readers could also access that page. In order to deal with possible updaters (this happens when there is a page split at leaf level or when deallocating an empty leaf page), we keep track of where we are in terms of the position in the *base* pages. If the update is on a *base* page that has already been read by the reorganizer, we make the update in the old tree, then we record that update in an append-only *side-file*.

If the update is on a *base* page that has not been read yet, then we do nothing, since that update will be read and we won't miss it. When the reorganizer finishes building the new B$^+$-tree, we will apply the updates in the *side-file* to the new B$^+$-tree to catch up. While the reorganizer is doing catch-up, some more updates may be appended to the *side-file*. Since leaf page splits don't happen very often, we will eventually catch up all the changes. After that, we will switch over from the old tree to the new tree. Details of concurrency control on the side file and during switching are below.

### 7.1 Building the new upper-level of the B$^+$-tree

We assume that the internal pages above the base page level should be in memory. This is a reasonable assumption because they don't take up much total space.

We get the first base page by following the leftmost pointers at each upper level of the B$^+$-tree. We assume that there is a low mark on each base page when the page was first created. This low mark tells you the smallest key on this page. We have a *Get_Next(k)* utility. This will get you the next base page whose low mark is the smallest one that is greater than the key $k$. We will use the low mark key as the indication of where we are. We will store $CK$ (Current Key), the low mark of the base page that is currently being reorganized, in the system table. We have a utility *Get_Current()*, which will return the value of $CK$. The value of $CK$ is changed by reorganizer to *Get_Next(CK)* before it gives up the $S$ lock on the base page it just finished reading.

If a leaf page split causes a new item to be inserted into one of the base pages, we can tell if that insertion needs to be appended into the side-file (in addition to being carried out on the old copy of the tree) by comparing the inserted key with the current low mark key. If it is greater, then we don't need to append it, because it must have been inserted in a base page we haven't read yet. It can not be inserted to the current page, because the reorganizer is holding a $S$ lock on the base page. If it is smaller, then we know it has been inserted into one of the base pages that we have already read. So we have to record that insertion into the side file. We follow the same logic for leaf node deletion.

After building the new upper levels of the B$^+$-tree by reading the old B$^+$-tree's base pages, we apply the contents of the side-file to the new B$^+$-tree's base page. This will make the new B$^+$-tree up to date.

### 7.2 Concurrency Protocol for the Side File

When the internal node reorganization begins, the side file is created and a *reorganization-bit* is set to one. The side file is a system database table. Putting an entry in the side file is similar to making an insertion in a table. Record-level locking is used, based on the key of the entry being made. The insertion to the side file is logged by the transaction which makes the insertion.

When an updater wants to update a *base* page, it first gets the $X$ lock on the base page. Then it checks the *reorganization-bit*. If the bit is one (which means the reorganization of internal pages is going on), it calls *Get_Current()*, and gets the low water mark of the base page that is currently being reorganized. Suppose *Get_Current()* returns $k1$. If the key $K$ that the updater wants to update in the base page is less then $k1$, this means the reorganizer has already passed that page. So key $K$ has to be inserted to the side file. If $K$ is greater then or equal to $k1$, the reorganizer hasn't read that base page yet, so no special action is needed. Since the updater has the $X$ lock on the base page while updating it, the reorganizer cannot read that page and change the current key until the updater finishes.

After determining that it is necessary, the updater requests an $IX$ lock on the side file table, then an $X$ lock on the key it wants to insert. If it can't obtain the $IX$ lock, this means *switching* is in progress. In this case, it requests an instant duration $IX$ lock. When the success status is returned (switching is finished), the updater must search in the new tree for the correct base page on which to make its update.

When the reorganizer has read all the base pages and constructed the complete (modulo changes from the side file) tree and all internal nodes of the new tree have been written to disk, it starts to process the side file. As each side file record is applied to the new tree, that record is deleted from the side file. The actions of changing the new base page and of removing the side file record are logged.

### 7.3 Making the new B$^+$-tree durable

After catching up all the updates, we have to make the new B$^+$-tree durable before we make the switch. The simplest way is to force all the new B$^+$-tree internal pages to disk after the new B$^+$-tree has been built. But this would require restarting the whole process in case there is a system failure, since no logging is done.

In order to improve the efficiency, an optimization would be force write after a certain number, say 5, of new pages has been built. At each of these "stable points", we would also force to disk any upper level nodes which had been changed since the last stable point. This would normally only be one parent. This would guarantee a path on disk from the root to the pages which had been constructed up to this point Some of these pages may have already reached the disk due to the buffer replacement policy, so "force writing" a group of $N$ base pages and their ancestors may actually involve less than $N$ I/Os.

After these pages are forced, only the key of the next page to be read need be recorded in the log. This key (last stable key) is also recorded in the Database Log Checkpoint. The location of the concurrent root of the new B$^+$-tree is also put into the log checkpoint. Then after recovery, when reading from the Database Log Checkpoint, and reading forward in the log, the most recent stable key can be determined. Either it is the one in the Database Log Checkpoint, or in the most recent log record after the checkpoint containing a stable key. (This is another case of careful logging. After information is in the stable database, the amount of information needed in the log is reduced.)

Normal Database recovery also recovers the insertions made in the side file by database transactions. It can happen that side file entries were made for leaves which had been reorganized but had not yet become stable at the time of system failure. These entries will refer to changes in base pages which will be found by the reorganizer on restart. During recovery, entries in the side file which refer to records which come after the most recent stable key can be removed from the side file. Space allocation and force writing are also logged. Space which is allocated after the most recent force-write log record can be deallocated during recovery. See [ZS95] for details.

### 7.4 Switching to the New B$^+$-tree

After finishing applying all the records in the side file to the new B$^+$-tree, in order to make the switch to the new B$^+$-tree, the reorganizer first requests an X-lock on the side file. This will prevent any further updates on base pages of either the new or the old tree. But other ongoing readers and updaters that are not going to affect the structure of the tree can still finish.

After we get the $X$ lock, we catch up the changes to the base pages again by applying the contents of the side file, and log those changes. Usually there will only be a small number of such changes since these are the ones made while the reorganizer is waiting for the $X$ lock.

At this point, we change the information about the location of the root of the old B$^+$-tree to that of the new B$^+$-tree. This information is usually on a special place on the disk. The new B$^+$-tree also has a lock name which is distinct from the old B$^+$-tree. After this, the new B$^+$-tree can be used. Changes to the base pages are still prevented until current readers using the old

tree are finished by continuing to keep the $X$ lock on the side file.

After the reorganizer declares the new tree is available (while maintaining the $X$ lock on the side file), it requests an $X$ lock on the old tree. This is used to decide when to discard the upper level nodes of the old tree. Since each transaction using the old B$^+$-tree would have an intention lock ($IX$ or $IS$) on the old tree, when the reorganizer gets the $X$ lock on the old tree, it means that all the on-going transactions using the tree have finished. So we can discard the upper level nodes of the old tree and reclaim the disk space. Then we set the *reorganization-bit* to zero, release the $X$ lock on the side file and on the old tree. The switching process is complete.

Since there might be some on-going long transactions after we begin to switch, we might have to wait for a long time before we can get the $X$ lock on old tree. We cannot release the $X$ lock on the side file before we know all the on-going transactions using the old tree have terminated because updates on the base pages of the new tree would make the old tree's leaf page addresses obsolete. Search would become incorrect in the old tree when, for example, a leaf page splits and there are no side pointers.

Since we don't want to block structural updates to the new tree for too long, we might set a time limit that the reorganizer can wait for the $X$ lock on the old tree. If the reorganizer cannot get the $X$ lock within the time limit, then it will force the on-going transactions that use the old tree to abort. Then it can get the $X$ lock.

(If Π-tree concurrency is used [LS92], all updating can be done without making changes above the leaf level. Base page updates can be postponed. This could allow the reorganizer to avoid aborting ongoing transactions without preventing new work. In fact, due to the existence of side-pointers and sibling terms in the Π-tree, updates could be made in the new tree's base pages without affecting search correctness in the old tree. The reorganizer would only need an instant duration $X$ lock on the side file. Only deallocation of pages of the non-leaf levels of the old tree would need to be postponed until the old transactions left the system.)

### 7.5 Summary of Internal Page Reorganization

While the reorganizer is reading the *base* pages, building the B$^+$-tree, or doing catch-up, all the insertions, deletions, and searches still use the old B$^+$-tree. Since the reorganizer only holds an $S$ lock on one *base* page at a time, all the readers can still access the B$^+$-tree. It will only block insertions that possibly will cause a split in the leaf pages, and thus update the *base* page that is being read by the reorganizer. This will provide very high concurrency. Only during the switch are updaters prevented from making changes on all base pages and even then, if a Π-tree is used, these updaters need only wait for possibly a few side file changes to be installed in the new tree—most of the catching up is done before the reorganizer requests an $X$ lock on the side file.

## 8 Comparison with Related Works

Most of the previous work on on-line reorganization does not deal with reorganizing primary B$^+$-trees. For example, in [SD92], a method is described which deals with moving records from one physical organization to another physical organization, RID to B$^+$-tree for example. This method encapsulates all the necessary changes relating to moving one record into one transaction.

In [Omi88], records are moved from a primary B$^+$-tree to a linear hashing organization, one leaf at a time. The same space is used. (This is an in-place reorganization.) Logging and recovery is not discussed. In [OLS92], records are reclustered in-place. "Logical pages" are sets of records one wishes to place in one physical page. One by one, all the logical pages are transformed to physical pages by bringing all their records into the buffer and moving them. References to moved records are changed later. A list of moved records called a "differential file" is used to redirect searches to the new addresses. Logging and recovery is not discussed.

[MN92] [SC92] describe some on-line algorithms for constructing *secondary* B$^+$-tree indexes. Our new-place reorganization method for the upper level of the B$^+$-tree has some points in common. In particular, this is where we got the idea of using a side file to catch up updates. [MN92] discusses logging and recovery for secondary index construction and also includes a restartable sorting algorithm.

[Smi90] describes the Tandem on-line reorganization product. It deals with reorganizing a primary B$^+$-tree. However, we believe our method has several advantages:

- Better concurrency. For each of the four operations they perform, (swapping two leaf pages, merging the contents of two leaf pages, moving the contents of a leaf page to an empty page, or splitting the contents of one leaf page into two to obtain a desired fill factor), [Smi90] prevents user transactions from accessing the entire file (B$^+$-tree). In our method, while doing leaf page operations , we only $RX$ lock the relevant leaf pages while moving or compacting leaf page data. During this part of a leaf page operation, readers and updaters can even access other children of the same base page. Only after moving the records, for the short time needed to post information in the base page, is the base page locked with an exclusive lock. During reorganization of internal pages, we only $S$ lock one base page at a time until the switching operation. This increased concurrency is the most important advantage our method has over [Smi90].

- Better recovery method. We introduce a new recovery method: *forward recovery*. It will resume the work instead of aborting the work as a normal recovery method will do. This will make reorganization faster in case of system failure. [Smi90] treats each leaf page operation as a database transaction, so it is rolled back if interrupted.
- Better granularity. No matter what the new page fill factor is, each transaction in [Smi90] will only deal with two blocks (pages). So in order to fill one new page, it might require running several transactions. These will cause more transaction overhead and locking overhead. In our method, if we do in-place compaction, we may compact several pages into one.
- Less transaction overhead. [Smi90] uses one transaction for each reorganization operation: block move, block merge, block swap and block split. In our method, the reorganizer runs in the background as one process. So there is less transaction overhead.

Also, how to switch from the old $B^+$-tree to the new one is discussed in detail for the first time.

## 9 Conclusion

In this paper, we present the principles of on-line reorganization of sparsely populated $B^+$-trees. We introduce a new locking protocol which provides high concurrency during reorganization. Compacting and moving only affects small collections of leaves under one parent and only locks that parent against readers for a short part of the compacting operation. Swapping is similar but sometimes involves two parents. Internal node reorganization only S-locks one base page at a time until the switching operation.

We describe for the first time how to switch from an old $B^+$-tree to the new one when we finish reorganizing the internal pages. We also introduce the *forward recovery* method which will make reorganization faster in case of system failure. Future work includes implementation of a prototype system, and exploration of parallelism in reorganization.

### Acknowledgements

We would like to thank Dr. Svein-Olaf Hvasshovd of Telenor who led us to this problem, and asked some good questions during our discussions.

## References

[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.

[JS93] Theodore Johnson and Dennis Shasha. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *Journal of Computer and System Sciences*, 47(1):45–76, 1993.

[LS92] D. Lomet and B. Salzberg. Access method concurrency with recovery. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 351–360, 1992.

[LT95] David Lomet and Mark Tuttle. Redo recovery after system crashes. In *International Conference on Very Large Data Bases*, pages 457–468, 1995.

[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.

[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

[Omi88] E. Omiecinski. Concurrent storage structure conversion: From B+Tree to linear hash file. In *International Conference On Data Engineering*, pages 589–596, 1988.

[Sal88] Betty Salzberg. *File Structures: An Analytic Approach*. Prentice Hall, 1988.

[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.

[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.

[Wed74] H. Wedekind. *On the selection of access paths in a data base system*, chapter Database Management. North Holland Publishing Company, 1974.

[ZS95] Chendong Zou and Betty Salzberg. On-Line Reorganization of Sparsely-Populated B+trees. Technical Report NU-CCS-95-18, Northeastern University, College of Computer Science, Boston, MA (USA), 1995.