Online Index Rebuild

Nagavamsi Ponnekanti Hanuma Kodavalla

Sybase Inc.

1650, 65th street, Emeryville, CA, 94608

vamsi@sybase.com,hanuma@ieee.org

ABSTRACT

In this paper we present an efficient method to do online rebuild of a B+-tree index. This method has been implemented in Sybase Adaptive Server Enterprise (ASE) Version 12.0. It provides high concurrency, does minimal amount of logging, has good performance and does not deadlock with other index operations. It copies the index rows to newly allocated pages in the key order so that good space utilization and clustering are achieved. The old pages are deallocated during the process. Our algorithm differs from the previously published online index rebuild algorithms in two ways. It rebuilds multiple leaf pages and then propagates the changes to higher levels. Also, while propagating the leaf level changes to higher levels, level 1[1] pages are reorganized, eliminating the need for a separate pass. Our performance study shows that our approach results in significant reduction in logging and CPU time. Also, our approach uses the same concurrency control mechanism as split and shrink operations, which made it attractive for implementation.

1 INTRODUCTION

B+-trees [Comer79] are one of the main indexing methods used in commercial database systems. A primary B+-tree index has data records in the leaf pages while a secondary B+-tree index has only the index ***keys*** in the leaf pages, where a key consists of a key value and the ROWID of the data record. In this paper, we consider the rebuild of a secondary index[2]. We assume that the leaf pages of the index are doubly linked and the non-leaf pages are not linked.

Over time, insertions and deletions may cause allocations and deallocations of index pages. As mentioned in [GR93], most practical implementations of B-trees do not merge index nodes upon underflow and the same is true for ASE. Index pages may become less than half full causing a drop in the space utilization and also an increase in the number of disk reads required to read the same number of index keys. Further, the index may become declustered (i.e. index keys within a key range may not be in contiguous disk space) thereby degrading the performance of range queries. To restore the clustering, users can drop and recreate the index. However, that typically requires holding a shared table lock on the table thereby making the table inaccessible to OLTP transactions, which may not be acceptable.

Online index rebuild restores the space utilization and clustering of the index with minimal blocking of readers and writers. It copies the index keys to fresh pages and deallocates the old pages. In this paper, we present an algorithm for online index rebuild that provides high concurrency, does minimal amount of logging, has good performance and does not deadlock with other index operations. Section 2 describes the concurrency control mechanisms in the index manager. Section 3, Section 4 and Section 5 describe the online index rebuild algorithm. Section 6 evaluates the algorithm with respect to some desirable properties. Section 7 compares it with related work in this area and finally, Section 8 gives the conclusions.

2 INDEX CONCURRENCY CONTROL

In this section, we describe the concurrency control mechanisms in the index manager.

We assume ***row level locking***. Insert, delete and scan operations acquire logical locks on rows as needed. Logical locks are meaningful only on rows at the leaf level. We do not discuss logical locking further, as split, shrink and rebuild operations do not acquire logical locks.

1. Level 1 is the level immediately above the leaf level

2. However, if the primary key value is used as data ROWID in the secondary indices, then the same algorithm can be used to rebuild a primary index as well.

MOD 2000, Dallas, TX USA

Latches are used for physical consistency at the page level. To read or modify a page, an S (shared) or X (exclusive) latch is acquired on the buffer that contains the page. Latch deadlocks are prevented by requesting the latches in top down order or left to right order.

An insert may cause a split operation which consists of adding a new leaf page to the chain, possibly moving some keys to it and updating the parent and possibly higher levels. Shrink operation consists of removing a leaf page from the chain and updating parent and possibly higher levels. A page is shrunk when the last row is removed from it. Split and shrink operations are performed as ***nested top actions*** [GR93], which means that once the operation is complete, it is not undone even if the transaction performing it rolls back.

Now, we give an overview of insert, delete, split, shrink and scan operations. We also present the pseudocode for tree traversal routine. The deadlock issues are discussed in Section 6.5.

2.1 Insert and Delete

Insert and delete call traversal module to retrieve the appropriate leaf page X latched. If no split or shrink is needed, the latch is released after performing the insert or delete.

Traversal uses the familiar ***crabbing*** strategy [GR93] with latches. An S latch is sufficient, except at the leaf level, where an X latch is acquired. However, if a page in the path traversed is undergoing a split or shrink by another transaction, traversal *may* need to release its latches and block for the split or shrink to complete, as explained in the following sections (Section 2.2, Section 2.3 and Section 2.6).

2.2 Leaf Split

To split a leaf page P_0, both P_0 and the new page, say N_0, are *X latched* and ***address-locked*** *in X mode.* In addition, ***SPLIT bits*** are set on both of them. The X lock acquired by the split is called an ***address-lock*** to distinguish it from logical locks. For the rest of this paper, unless specified otherwise, a lock refers to an address lock. While the X latches are released soon after the modification of P_0 and N_0, the X locks and the SPLIT bits are retained till the end of the top action. The purpose of setting the SPLIT bit on a page is to block writes to that page by concurrent transactions after the splitter has released its X latch[3]. The writers block by releasing any latches held and requesting an unconditional instant duration S lock on the page[4]. Thus the writers are blocked till the top action is complete. However, readers can still access P_0 or N_0, if they have successfully acquired an S latch on it.

3. SPLIT bit does not block a writer that just wishes to modify its previous page link. This optimization allows two adjacent leaf pages to be split concurrently.

No locks or latches are held on higher level pages by the splitter when it is splitting the leaf page. They are acquired during the propagation phase, as explained below.

2.3 Propagation of Split to Higher Levels

The split is propagated *bottom up*. The latches held on the pages at the current level are released before moving to the next higher level. To propagate the split to level i, split calls traversal to retrieve the appropriate non-leaf page P at level i latched in X mode. (However, traversal may not start from root in this case. See Section 2.6.1.) The page returned by traversal is guarenteed not to have SPLIT (or SHRINK) bit set on it. Here is the action to be taken on P:

- If P needs a split, both P and the new page, say N, are X latched, X locked and SPLIT bits are set on them(just as in leaf split). Suppose that keys >= K are moved to N. The page P is also marked with OLDPGOFSPLIT bit and entry [K, N] is stored on page P as a ***side entry***[5]**.** Once the side entry is established, both P and N are unlatched and the propagation continues to the next level. In case a concurrent traversal visits page P from its parent before the split propagates to the parent, the traversal uses the side entry to decide which of P or N is the correct target page.
- If no split is needed, no X lock or SPLIT bit is needed on P. The insert is performed, and the top action is completed and P is unlatched. The SPLIT bits and the OLDPGOFSPLIT bits are cleared and the X locks are released.

Recall that setting SPLIT bit on a page blocks writes to that page but not the reads. Thus a concurrent insert, delete, split or shrink operation that wants to traverse through P (or N) to a lower level page can access P (or N) after splitter has released its X latch on P (or N).

2.4 Shrink

Shrink is also performed as a nested top action and is propagated quite similar to split operation, except that SHRINK bits are set on the affected pages instead of SPLIT bits. Also, note that setting SHRINK bit on a page blocks both read and write operations on the page.

2.5 Scan

Scan calls traversal module to retrieve the starting page for the scan S latched. The scan qualifies the index keys under S

4. The SPLIT bit is similar to the SM bit in [MF92]. However, SM bit is accompanied with a tree latch (rather than an X lock on the page), which increases the likelihood of blocking. Also, in our approach, the bit is only an optimization of calls to the lock manager (checking for the bit can be replaced with a request for a conditional instant duration S lock).

5. Although the sidekey is similar to side pointers in B-link trees [LY81], it is valid only as long as the OLDPGOFSPLIT bit is set.

latch. The page is unlatched before returning a qualifying key to query processing layer and is latched again to resume qualification. Also, note that depending on the isolation level, the scan may need to acquire logical locks on qualifying keys.

2.6 Traversal Pseudocode

Here is the pseudocode for traversal. Note that a page is latched in X mode only if it is at the target level and the traversal was called in writer mode. In all other cases, the page is latched in S mode.

```
traverse(searchkey, searchmode, targetlevel)
{
retraverse:
    p = get root page latched;
    while (level of p > target level)
    {
        Search p to identify the child to chase;
        c = get child page latched;
        if (c has SHRINK bit set)
        {
            Unlatch c and p;
            Wait for instant duration S lock on c;
            goto retraverse;
        }
        if (OLDPGOFSPLIT bit is set in c)
        {
            if (searchkey >= key in side entry)
            {
                sibling = Get right sibling latched;
                Unlatch c;
                c= sibling;
            }
        }
        /* Now we are on the correct child */
        Unlatch p;
        p = c;
    }
    /* Target level is reached */
    if ((searchmode == writermode) and
       (p has SPLIT bit set))
    {
        Unlatch p;
        Wait for instant duration S lock on p;
            goto retraverse;
    }
    return p;
}
```

2.6.1 Retraversing

In the above algorithm, retraversal starts from the root page. However, ASE actually uses a more efficient strategy. While traversing down the tree, the pages encountered in the path are remembered. When there is a need to retraverse, rather than starting from the root, it starts from the lowest level page in the path that is *safe*. A page is safe if it is still at the same level as expected and the search key is within the range of key values on it. Same strategy is used by traversal during the propagation of split and shrink to avoid starting from root. Later, in Section 5.4.1, we will see that the propagation phase of online index rebuild also uses traversal and benefits from this strategy.

3 ONLINE INDEX REBUILD OVERVIEW

Online rebuild runs as a sequence of transactions, with each transaction performing a series of nested top actions and each top action rebuilding multiple contiguous leaf pages in the page chain. The top actions are called ***multipage rebuild*** top actions. The number of pages to rebuild in a single top action is denoted by **ntasize** and the number of pages to rebuild in a transaction is denoted by **xactsize.** Rebuilding multiple pages in a single top action reduces logging and CPU time. We chose an ntasize of 32 based on our performance study (Section 6.4). The significance of xactsize is explained below.

At the end of each transaction, the new pages generated in the current transaction are flushed to disk and then the old pages that were removed from the tree are made available for fresh allocations.

Flushing new pages to disk before making old pages available for fresh allocations allows rebuild not to log full keys during the key copying. Instead, the log records contain only the PAGEIDs and the timestamps of the source page and the target page and just the ***positions*** of the first and the last key that were copied. Redo may have to read the source page to redo the key copying. On the other hand, if the source page is made available for allocation before the target page is flushed to disk, then the new contents of the source page could reach the disk before the target page reaches the disk. If a crash occurs after the new contents of the source page reach the disk, but before the target page reaches the disk, the target page cannot be recovered.

While rebuilding several pages in a transaction has the advantage of delaying the forced write of new pages, it also delays the availability of the old pages for reuse. It is desirable to rebuild a few hundred pages in a transaction.

4 MULTIPAGE REBUILD TOP ACTION

Consider the rebuild of contiguous pages P_1, P_2,..., P_n in a single nested top action. Suppose that PP is the previous page of P_1 and NP is the next page of P_n. The top action involves a copy phase and a propagation phase, which are explained below:

4.1 Copy Phase

The index keys are copied from P_1, P_2,..., P_n to PP and zero or more newly allocated pages, say N_1,N_2,..., N_k, where k >= 0. Note that k could be > n if the user has specified that the new leaf pages be filled only upto a desired ***fillfactor***, so that some space is left free for future inserts. Copy phase also includes fixing page linkages and deallocating the old pages.

4.1.1 Locking

X locks are acquired and SHRINK bits are set on PP, P_1, P_2,..., P_n in that order. For i > 1, if P_i has SPLIT or SHRINK bit set on it, rebuild does not wait for lock. Instead, only pages P_1, P_2,...,P_{i-1} are rebuilt in the current top action. On the other hand, if PP or P_1 has SPLIT or SHRINK bit set, then rebuild waits for the split or the shrink to complete.

4.1.2 Logging

Copy phase generates a ***single keycopy*** log record to capture ***all*** the key copying that has occurred from pages P_1, P_2,..., P_n to PP and the newly allocated pages. It has multiple entries of the form [source pageno, target pageno, position of the first key copied, position of the last key copied]. It also generates allocation and deallocation log records and *changeprevlink* log record for NP.

4.1.3 Page Deallocations

A page can be in one of allocated, deallocated or free states. Only a page in free state is available for fresh allocations. When the page manager is called to deallocate a page, it logs a deallocation record and takes the page to deallocated state. The page manager has to be called again to free the page. The transition from deallocated state to free state is not logged by the page manager and it cannot be undone. In the event of a crash, after the redo and undo phases, recovery frees up pages that are still in deallocated state.

In the case of a shrink top action, deallocated pages are freed when the top action commits. However, in the case of multipage rebuild topaction, the deallocated pages are freed only when the current *transaction* commits. It uses log scan to determine what pages need to be freed up. Also, note that if rebuild needs to abort due to lack of resources or internal error or a user interrupt, during rollback, it needs to free up the pages deallocated in completed top actions. Before freeing up the old pages, the new pages need to be flushed to disk.

4.2 Propagation Phase

The changes are propagated to level 1 by deleting the entries for P_1, P_2,..., P_n and inserting the entries for N_1, N_2,..., N_k in the parent(s) of P_1, P_2,..., P_n. The propagation may continue above level 1. The propagation of split (shrink) can be thought of as passing of an insert (delete) command from one level to the next. The propagation of rebuild top action can be thought of as passing multiple commands from one level to the next, where each command could be an insert, delete or an update. At each level several pages could be affected. At a given level, the affected pages are modified in left to right order. Also, all modifications at the current level are finished before moving to the next higher level. For each affected non-leaf page, no more than one ***batchdelete*** log record and one ***batchinsert*** log record are generated. These log records contain the entire keys that were inserted or deleted. The propagation phase is described in detail in Section 5.

4.3 Advantages of Rebuilding Multiple Pages in a Single Top Action

Insert and delete log records in ASE have not only the key being deleted or inserted but also a lot of additional information such as transaction ID, old and new timestamps for the page, position of delete or insert etc. The amount of such additional information is as high as 60 bytes and is amortized by batching multiple inserts or deletes in a single batchinsert or batchdelete log record. Similarly, the overhead in other log records is amortized by rebuilding multiple pages in a single top action. Besides saving log space, rebuilding multiple pages in a top action reduces the number of visits to level 1 pages significantly, reducing the calls to lock manager, latch manager etc. Our performance study reflects this (Section 6.4).

5 PROPAGATION PHASE OF REBUILD

In this section, we discuss how the rebuild of multiple leaf pages is propagated to higher levels. The propagation is bottom up and the modifications to be done at the next higher level are specified in the form of ***propagation entries***. Before describing propagation entries, we explain what an index entry is.

We assume that a nonleaf page in the B+-tree that has n child pointers has only n-1 key value separators. An index entry is of the form [key value, child pageid], except for the index entry for the first child, which does not have the key value. An index page having n children has n index entries C_0, [K_1, C_1], [K_2, C_2],...,[K_{n-1}, C_{n-1}]. For 0 < i <= n-1, C_i has index entries greater than or equal to K_i and for 0 <= i < n-1, C_i has index entries less than K_{i+1}.

Now, we define propagation entries and explain what propagation entries are passed from the leaf and the nonleaf

pages. Then we describe how the propagation phase proceeds from one level to the next.

5.1 Propagation Entries

A ***propagation entry*** specifies the following:

- the page P that is sending the propagation entry.
- operation that must be performed at the next higher level. The possible operations are DELETE, UPDATE or INSERT of an index entry.
- INSERT propagation entry specifies the entry to be inserted at the next level. UPDATE propagation entry specifies the entry to replace the existing entry for that page. UPDATE and DELETE propagation entries do not specify the contents of index entry to delete (pageid P uniquely identifies the index entry).

5.2 Propagation Entries Passed From a Leaf Page

Consider the rebuild of leaf pages P_1, P_2,..., P_n in a single top action. Let PP be the previous page of P_1 and NP the next page of P_n. Here are the rules that determine what propagation entries are passed from a *single* page P_i:

- Suppose that k, where *k > 0*, new allocations are needed to accommodate the keys from P_i. The entry for P_i needs to be deleted from parent and entries for the k new pages need to be inserted in the parent. So an UPDATE propagation entry followed by k-1 INSERT propagation entries are passed.
- If all the keys from P_i could be copied into the last newly allocated page (i.e. *no new allocation* was needed to accommodate the keys from P_i), it passes DELETE propagation entry.

Thus, each page that was rebuilt passes one or more propagation entries. All the propagation entries from P_1, P_2,..., P_n are accumulated before the propagation proceeds to level 1.

5.3 Propagation Entries Passed From a Non-leaf Page

A non-leaf page P passes propagation entry(s) in the following cases:

- P is becoming empty (in this case P needs to be shrunk)
- P is split
- P is not becoming empty but there was some key movement from the subtree under P to the subtree under its left sibling.

These three cases are discussed in more detail below. Note that the last two cases are *not* mutually exclusive.

5.3.1 Shrink of P

If all children pass DELETE propagation entries, then page P needs to be shrunk[6]. It passes DELETE propagation entry. This means that *all* the leaf rows in the subtree under P have been moved to the subtree under its left sibling.

5.3.2 Split of P

The inserts to be performed on page P (as a result of UPDATE/INSERT propagation entries coming from children of P) may cause P to be split. If so, P is split in such a manner that all the remaining inserts go to the old page or all of them go to the new page. Note that one split may not be sufficient to accommodate all such inserts. If the insertions cause k splits, then k siblings are generated for P and k INSERT propagation entries are setup for inserting entries for these new pages at the next higher level.

5.3.3 Key Movement Across Subtrees

Consider figure 1 shown below. P' is the parent of P and L is the left sibling of P. [K1, L] and [K2, P] are the entries for L and P in P'. Consider some key movement from the subtree under P to the subtree under L. If keys up to (but not including) K are moved to the subtree under L, then the entry for P in P' needs to be changed from [K2, P] to [K, P] to keep the index consistent. So P needs to pass an UPDATE propagation entry [K, P] to P'. Now let us look at how to detect such key movement and how to find the value of K.

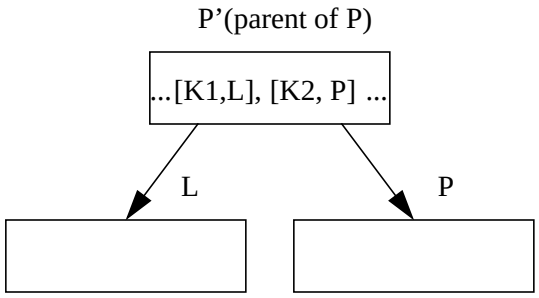


Figure 1: Key Movement Across Subtrees

Let C_0, C_1,...,C_n be the children of P.

If C_0 did not pass DELETE or UPDATE propagation entry, then *no* key movement has occurred from the subtree under P to the subtree under L.

Otherwise, let C_i, where $0 <= i <= n$, be the leftmost child of P that did not pass DELETE propagation entry. (Such a child must exist. Else, all children must have passed DELETE propagation entries and it is the shrink case discussed in Section 5.3.1). Since the children C_0, C_1,...,C_{i-1} have passed DELETE propagation entries, it means all the keys in the subtrees under them have been moved and they have become empty. The entries for all of them on P need to

6. In this case, there is no need to perform the deletes. Page can directly be deallocated.

be deleted and C_i needs to become the first child of P.

- If C_i has passed an UPDATE propagation entry, say $[K_u, C_i]$, then keys < K_u may have been moved from subtree under C_i to that under its left sibling. So, P passes UPDATE propagation entry $[K_u, P]$ i.e. $K = K_u$.
- Otherwise, C_i must have passed INSERT propagation entry(s) or no propagation entries. In either case, *no* key movement has occurred from the subtree under C_i to a subtree under its left sibling. If $[K_i, C_i]$ is the entry for C_i on P, then P passes UPDATE propagation entry $[K_i, P]$ to its parent i.e. $K = K_i$.

5.4 Propagation From Level i to Level i+1

An algorithm to apply a list of propagation entries passed from level i to level i+1 is described below.

5.4.1 Algorithm Propagate_to_level

Input: List $L = [E_1, E_2, ..., E_m]$ of all propagation entries to be applied to level i+1 pages (these were passed from level i pages)

Output: List L_1 of propagation entries passed to the next higher level from level i+1 if any

Side Effect: The modifications specified by the input propagation entries are applied on level i+1 pages

```
propagate_to_level(L, i+1)
Initialize L1 to empty list;
while (L is not empty)
{
    e = first propagation entry in L;
    C = page that propagated e;
    K = Any key from page C;
    /* Get the parent of C X latched. Note that
    ** traversal uses same strategy as described in
    ** retraversal section earlier to avoid starting
    ** from root (See Section 2.6.1).
    */
    P = traverse(K, writer, i+1);
     /*  identify all the propagation entries that
     ** were sent by children of P (they are
     ** guaranteed to be contiguous in L).
     */
     e' = last propagation entry in L that was
          passed by a child of P;
     Delete propagation entries e through e'
        from L;
    /* apply the propagation entries e through
    ** e' on P(See Section 5.4.2).
    */
    Modify P;
    Append the propagation entries passed by P
          if any to L1
    Release any latches held;
}
```

5.4.2 Modification of Page P

The propagation entries passed by the children of P are applied on page P in two phases, the delete phase followed by the insert phase. In the delete phase, the index entries for all the children that passed DELETE or UPDATE status are deleted. All such index entries will be contiguous. In the insert phase, the index entries specified by the INSERT/UPDATE propagation entries coming from children of P are inserted. The index entries inserted will also be contiguous.

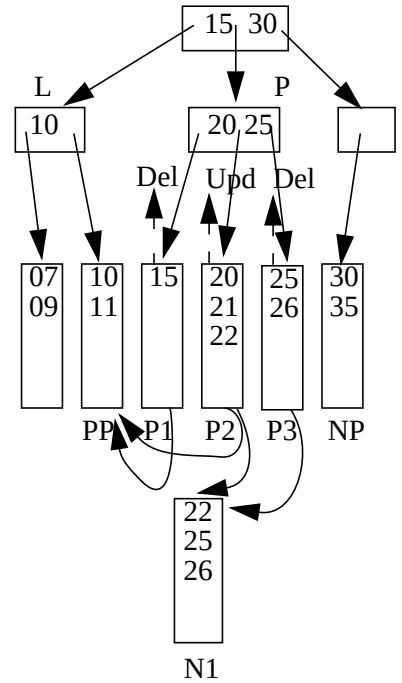
Traversal would have retrieved page P latched in X mode. However, latch alone is not sufficient. The address locking mechanism used by split or shrink top actions is used here and the SPLIT and SHRINK bits are overloaded. P is locked in X mode. A SHRINK bit is set on P if traversals through P need to be blocked. If modifications to P need to be blocked but not the traversals through P, a SPLIT bit is set on it. The rules for deciding which bit needs to be set are mentioned below.

- If any delete is performed on a page (i.e. atleast one child passed a DELETE or UPDATE status), SHRINK bit is set.
- If only inserts are performed on a page (i.e. no deletes and no splits), then SPLIT bit is set.
- If P needs to be split, a SHRINK bit is set on it. The new page is also X locked and SHRINK bit is set on it. There is no need to establish a side entry as traversals through P are being blocked anyway. X latch needs to be retained only on the page where the rest of the inserts in the insert phase need to be performed.

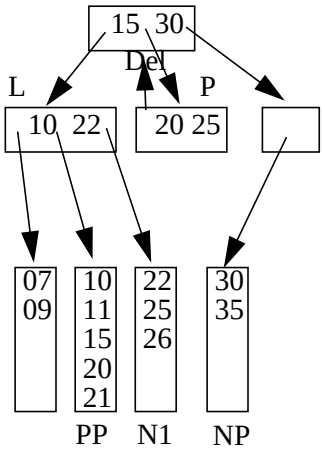
These rules are very conservative. Traversals are being allowed through the page only in the insert-only case, as no keys in the subtree under the page would have been moved to the subtree under its left or right sibling in that case. (See Section 6.2 for a possible improvement).

5.5 Reorganizing Level 1 Pages

Consider the propagation from leaf level to level 1. In the propagation algorithm that has been described, while applying propagation entries on a level 1 page P, the insert phase inserts the index entries specified in UPDATE/INSERT propagation entries sent by the children of P. However, it is better to perform as many of those

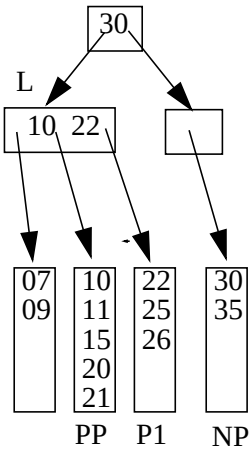


Leaf pages P1, P2 and P3 are reorganized. Assuming five rows fit into a leaf page, all of P1's rows and some of P2's rows are moved to PP. The remaining rows from P2 are moved to the new page N1. All of P3's rows are moved to N1. P1 passes a DELETE propagation entry as it did not cause allocations. P2 passes UPDATE propagation entry with [22, N1] as the index entry to replace the entry for P2 at the next higher level. P3 also passes DELETE propagation entry.



Propagation to level 1:

The three reorganized pages P1, P2, P3 all have the same parent P. In the delete phase, the entries for all these pages need to be deleted. In the insert phase, [22,N1] needs to be inserted. However, this insert is performed on the previous page L. Since all the entries in P need to be deleted and there are no inserts to perform on P, DELETE propagation entry is passed.



Propagating to level 2:

The entry [15, P] is deleted. The propagation is complete and the top action is committed.

Figure 2: Multipage Rebuild Top Action

inserts as permitted by space on the immediate left sibling of P that is not being shrunk in the current top action. Note that this can only be done if the first child of P is getting deleted in the delete phase (i.e. it passed a DELETE/UPDATE status). Otherwise, it would violate the index key ordering at level 1.

With this enhancement, level 1 pages are filled as much as possible without requiring a separate pass. An example of multipage rebuild top action with this enhancement is shown in figure 2.

6 EVALUATION

Here, we evaluate the algorithm with respect to some important metrics.

6.1 Restoration of Clustering

When online index rebuild begins, the page manager tries to allocate a new page from a chunk of large contiguous free disk space. After all the pages in the chunk are used up, it again looks for a chunk of large contiguous free disk space. As the index keys are moved to the newly allocated pages in the increasing key order, the new leaf pages are expected to be well clustered.

6.2 Concurrency

Although rebuilding multiple pages in a top action has the disadvantage of keeping many leaf pages locked at a given time, it significantly reduces the number of visits to a level 1 page and the total duration of exclusive access to it. It also significantly reduces the CPU time for the rebuild operation which in turn reduces the negative impact of the operation on the throughput of the system.

Here are some possible enhancements to reduce the impact on concurrent index operations:

- In the propagation phase, setting SHRINK bit on all nonleaf pages on which a delete was performed is pessimistic. Rebuild deletes contiguous index entries on nonleaf pages. Suppose that all index entries between [K_i, C_i] and [K_j, C_j] are deleted. There is no reason to block traversals through the page that are looking for < K_i or >= K_j. Thus the *positions* of these index entries could possibly be established on the page (just as a split establishes a side entry) to benefit concurrent traversals. This enhancement only helps in those cases where the propagation continues above level 1.
- Consider the rebuild of P_1, P_2,..., P_n in a single top action. Let PP be the previous page of P_1 and NP be the previous page of P_n. As the address locks are acquired on the pages being rebuilt, SPLIT bits (rather than the SHRINK bits) could be set on them (except on PP) so that only writers are blocked and not the readers. Once the contents of *all* the n pages have been copied to PP and possibly one or more newly allocated pages, the SPLIT bits could be modified to SHRINK bits (under an X latch). Now the next page pointer of PP and previous page pointer of NP can be set so that the old pages are effectively unlinked and new pages are linked into the chain.

6.3 Disk I/O

One scan of the old index is performed in the page chain order and the new pages are written out to disk once. While the page size is 2KB, the buffer manager allows the user to configure buffer pools with 4K, 8K or 16K buffer sizes. Online rebuild requests buffer manager to use the largest size buffers available for reading old pages and for writing new pages to reduce disk I/O.

6.4 Logging and CPU Time

We performed some experiments to see how the log space used and the CPU time consumed vary with *ntasize*. Our experiments are performed under the following conditions

- The space utilization in the index being rebuilt is about 50% and the rebuild specified a fillfactor of 100%.
- The cache is cold (i.e. all pages had to be read from disk).
- The page size is 2KB but the buffer pool is configured with 16KB buffers so that 16KB I/O size is used for index page reads and writes as well as log writes.
- Sun Ultra-SPARC machine running SunOS 5.6 is used.

For a given number of leaf pages in the old and the new index, the log space required varies primarily with the average nonleaf row size. The index manager in ASE uses suffix compression which reduces the nonleaf row size especially when the index is on multiple columns or on wide columns. We experimented with index key size (i.e. sum of maximum column lengths of all index columns) of 4 bytes and 40 bytes and the results are shown below. L_{ratio} is the ratio of log space required when ntasize of 1 is used to the log space required at the specified ntasize. C_{ratio} is defined similarly for CPU time. Although our experiments were performed with 2K page sizes, speaking analytically, the numbers for log space are expected to be valid for a wide range of page sizes. However, the ratio of log space required to that of the index size is expected to be inversely proportional to index page size. From Table 1, it is desirable to choose a large number for ntasize (32 to 64 pages).

key size	avg non-leaf row size	nta-size	L_{ratio}	C_{ratio}
4	10	32	7.3	2.4
4	10	64	8	2.4
40	20	32	4.9	3.7
40	20	64	5.4	4

Table 1: Log Space and CPU Time

6.5 Deadlocks

Our concurrency control protocols are such that the index operations never get into a deadlock involving latches or address locks or both. The only possible deadlock is one that involves only logical locks. The following rules ensure this:

- While holding a latch, unconditional logical lock is never requested and an unconditional address lock is requested only on a page that is being allocated (and hence not accessible from the tree) or a page that does not have SPLIT/SHRINK bit set.
- Latches are requested only in left to right order at a given level and top down order across levels.
- Address locks are requested only in bottom up order across levels.
- Address locks within a nonleaf level are acquired only in left to right order.
- Address locks within leaf level: Shrink acquires address locks on two pages and they are acquired in *right to left* order. Split acquires address lock on the old page and then the new page. But since new page is not yet part of the tree, this sequence does not cause a deadlock with shrink. Rebuild acquires address locks in left to right order. However, as mentioned before, if rebuild needs to wait, it releases all the locks that are acquired already before waiting. After wakeup, it retries for all the locks again.

7 COMPARISON WITH RELATED WORK

The first published article on online rebuild is from Tandem [Smi90]. Our approach has the following advantages:

- In Tandem's approach, when the page split and merge operations are performed, the entire file is made inaccessible to the OLTP transactions where as in our method only access to the affected pages is restricted.
- Further, in Tandem's approach, although it is not explicitly stated, it seems all the moved keys are logged where as in our approach the key contents themselves are not logged.

A more recently published work in this area is [SBC97]. This paper describes a comprehensive scheme to reorganize a table and rebuild the associated indexes. That scheme has the following drawbacks:

- A separate copy of the table is made and the associated indexes are rebuilt thereby doubling the storage requirement.
- User transactions must be directed to use the new copy. If there are long-running user sessions (with opened cursors), reorg waits for them to complete.
- For the duration of the reorg, the log should not be truncated because the reorg relies on the log for any changes that need to be applied to the new copy.
- Incremental reorganization is difficult.

By doing inline reorganization, our scheme avoids the above problems.

[ZS96] gives a detailed description of an algorithm for rebuilding an index. We believe our algorithm has the following advantages over it:

- Our algorithm reorganizes level 1 pages without requiring a sidefile. The sidefile mechanism adds a lot of implementation complexity. It also adds overhead to splits and shrinks happening in the index during the rebuild of non-leaf levels.
- Logging is reduced in [ZS96] by assuming "careful writing" mechanism in the buffer manager. Our algorithm does not require such a mechanism in the buffer manager[7].
- Unlike [ZS96], our algorithm does only one pass of the index.
- In [ZS96], only one new page is rebuilt in each reorganization unit. However, we believe that it is important to build multiple new pages in each reorganization unit to reduce logging overhead and CPU time.
- In [ZS96], switching to the new B+-tree requires an X lock on the tree which may cause unbounded wait. It is suggested that the transactions active in the tree be aborted if lock cannot be acquired after certain timeout interval. User transactions are never aborted in our algorithm.

Our algorithm has following drawbacks compared to [ZS96].

- During the propagation phase of multipage rebuild, pages above level 1 may need to be modified in which case X lock is acquired on the page being modified. [ZS96] does not X lock pages above level 1 in X mode (except for the X lock on the tree in the switching phase). However, since propagation is bottom up (as opposed to top down), the duration of X lock on non-leaf pages is expected to be small. This is because most of the time in the topaction is spent in reading old leaf pages and moving rows from old leaf pages to new pages.
- As mentioned before, to achieve good clustering, our algorithm needs a large chunk of contiguous free space on disk to begin with. However, since the amount of contiguous free space needed is small compared to the size of the index, this is not a significant problem.
- At the end of each transaction, new pages need to be flushed to disk. This disadvantage is alleviated to some extent by using large buffers and building a few hundred new pages in each transaction.

7. Note that we just assume "forced write", which is different from "careful writing". The former just requests the buffer manager to force a page to disk(without violating WAL), while the latter assumes a more involved mechanism of tracking the relative order in which a certain set of pages need to be written to disk.

8 CONCLUSIONS

We have presented an industrial-strength algorithm for online index rebuild that provides high concurrency, does minimal logging and has good performance. By rebuilding multiple leaf pages in each top action, the updates to level 1 pages can be batched resulting in significant reduction in logging and CPU time. The level 1 pages are reorganized while propagating the leaf level changes thereby eliminating a separate pass for reorganizing level 1 pages.

9 ACKNOWLEDGMENTS

We thank our colleagues, Yang Wang, Sangeeta Doraiswamy, T. E. Raghavan and Shampa Chakravarty for reviewing the design and the implementation of this feature. We also thank Nageshwara Rao and Gopinath Chandra for their help in testing this feature.

10 REFERENCES

[Com79] Douglas Comer. The Ubiquitous B-Tree. *Computing Surveys*, Vol. 11, No. 2, June 1979.

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

[LY81] Lehman, P. L., and S. B. Yao. Efficient Locking for Concurrent operations on B-Trees. *ACM TODS. Vol. 6, No.*

4, pages 650-670, December 1981.

[MF92] C. Mohan and Frank Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method using Write-Ahead Logging. *Proc. of ACM SIGMOD Conf*, pages 371-380, 1992.

[SBC97] Gary H. Sockut, Thomas A. Beavin and Chung-C. Chang: A Method for On-Line Reorganization of a Database. IBM Systems Journal. Vol. 36, No. 3, pages 411-436, 1997. Available at http://www.research.ibm.com/journal/sj/363/sockut.html

[Smi90] Gary Smith. Online Reorganization of Key-Sequenced Tables and Files. *Tandem Systems Review*, October 1990.

[ZS96] Chendong Zou and Betty Salzberg. On-line Reorganization of Sparsely-populated B+ trees. *Proc. of ACM SIGMOD Conf*, pages 115-124, 1996.