Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates

C. Mohan
Inderpal Narang

Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120, USA
{mohan, narang}@almaden.ibm.com

Abstract As relational DBMSs become more and more popular and as organizations grow, the sizes of individual tables are increasing dramatically. Unfortunately, current DBMSs do not allow updates to be performed on a table while an index (e.g., a B^+-tree) is being built for that table, thereby decreasing the systems' availability. This paper describes two algorithms in order to relax this restriction. Our emphasis has been to maximize concurrency, minimize overheads and cover all aspects of the problem. Builds of both unique and nonunique indexes are handled correctly. We also describe techniques for making the index-build operation restartable, without loss of all work, in case a system failure were to interrupt the completion of the creation of the index. In this connection, we also present algorithms for making a long sort operation restartable. These include algorithms for the sort and merge phases of sorting.

1. Introduction

This paper describes two algorithms which would allow a data base management system (DBMS) to support the building of an index (e.g., a B^+-tree) on a table concurrently with changes being made to that data by (ordinary) transactions. Current DBMSs do not allow updates to a table while building an index on it. Eliminating this restriction in the context of very large tables has been identified as an open problem [DeGr90, SiSU91]. As sizes of individual tables get larger and larger (e.g., petabytes, 10**15 bytes), it may take several days to just scan all the pages of a table to build an index on such a table [DeGr90]! Even though a large table may be partitioned into smaller pieces with each piece having its own primary index, still building a global secondary index would require a scan of *all* the partitions [Moha92]. We are already aware of customers who would like to store more than 100 gigabytes of data in a single table! Disallowing updates while building an index may become unacceptable for several reasons. Relational DBMSs with their promise and ability to support simultaneously both *transaction* and query workloads have aggravated the situation with respect to availability by not supporting index build with concurrent updates.[1]

1.1. General Assumptions

Data Storage Model We assume that the records of a table are stored in one or more files whose pages are called ***data pages***. The indexes contain ***keys*** of the form *<key value, RID>*, where ***RID*** is the record ID of the record containing the associated key value. ***Key value*** is the concatentation of the values of the columns (fields) of the table over which the index is defined. We can handle both unique and nonunique indexes. In a ***unique index***, there can be at most only one key with a particular key value. Without loss of generality, we assume that the keys are stored in the ascending order. The section "6.2. Extensions" discusses how our algorithms can be adapted to work in the context of a storage model in which all the records of a table are stored in the primary index (*<primary key value, record data>*) and the secondary indexes contain entries of the form *<key value, primary key value>*, where the primary key value is required to be unique.

Recovery We assume that write-ahead logging (WAL) [Gray78, MHLPS92] is being used for recovery. The ***undo*** (respectively, ***redo***) portion of a log record provides information on how to undo (respectively, redo) changes performed by the transaction. A log record which contains both the undo and the redo information is called an ***undo-redo log record***. Sometimes, a log record may be written to contain only the redo information or only the undo information. Such a record is called a ***redo-only log record*** or an ***undo-only log record***, respectively.

Execution Model The term ***index-builder (IB)*** is used to refer to the process which scans the data pages, builds index keys and inserts them into the index tree. Regular user transactions can be making updates to the table while IB is performing its tasks. IB does *not lock the data* while extracting keys, but it latches[2] each page as it is accessed in the share mode. Transactions do their usual latching and locking [MHLPS92, Moha90a, MoLe92]. This execution model permits very high concurrency and decreases CPU overhead.

1.2. Problems

This section discusses the problems introduced by the execution model that we have assumed. These problems stem from the fact that IB does not lock the data.

- **Duplicate-Key-Insert Problem** An attempt may be made to insert a duplicate key (i.e., two identical <key value, RID> entries) as a result of competing actions by IB and an insert from a transaction. This is because, to avoid deadlocks involving latches, neither the transactions nor IB holds a latch on the data page while inserting keys in the index [MHLPS92, Moha90a, MoLe92]. A page's latch is held only during the time of extraction of the keys from the records in that page. Also, as we will see later, there is a long time gap between the time IB extracts a key and the time when it inserts that key into the index.
- **Delete-Key Problem** A key which was deleted by a committed transaction could be inserted later by IB because of race conditions between the two processes. The race condition can occur for the same reason as the one described above for the insert case.

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0361...$1.50

1.3. Overview

In this paper, we present two algorithms, called ***NSF*** (*No Side-File*) and ***SF*** (*Side-File*). They allow index builds concurrently with inserts and deletes of keys by transactions. NSF allows IB to *tolerate* interference by transactions. That is, while IB is inserting keys into the index, transactions could be inserting and deleting keys from the same index tree. SF does not allow transactions to interfere with IB's insertion of keys into the index. In SF, key inserts and deletes relating to the index still being constructed are maintained by the transactions in a side-file as long as IB is active. A ***side-file*** is an append-only (sequential) table in which the transactions insert tuples of the form <*operation, key*>, where *operation* is insert or delete. Transactions append entries without doing any locking of the appended entries. At the end, IB processes the side-file to bring the index up to date.

In this paper, we also describe techniques for making the index-build operation restartable so that, in case a system failure were to interrupt the completion of the index-build operation, not all the so-far-accomplished work is lost. For this purpose, we present algorithms for making a long sort operation restartable. These include algorithms for the sort and merge phases of sorting. The algorithms relating to sort have very general applicability, apart from their use in the current context of sorting keys for index creation.

The rest of the paper is organized as follows. In sections 2 and 3, we present the details of the NSF and SF algorithms, respectively. We cover in detail the mainline and recovery operations, and restarting of the index-build utility. Our emphasis has been to maximize concurrency, minimize overheads and cover all aspects of the problem. In section 4, we compare the two algorithms and discuss their performance qualitatively. Algorithms for making the sort operation restartable are presented in section 5. Finally, in section 6, we summarize our work. We also discuss extensions of our algorithms to allow multiple indexes to be built in one scan of the data and to support a different storage model.

2. Algorithm NSF: Index Build Without Side-File

In this section, we present the NSF (*No Side-File*) algorithm. First, we give a brief overview of the solutions to the problems described in the section "1.2. Problems". Then, we describe the NSF algorithm in detail. For ease of explanation, we pretend that only one index is being created at any given time for a table. Later, in the section "6.2. Extensions", we discuss how both NSF and SF can easily create multiple indexes simultaneously in one scan of the data.

2.1. Overview of NSF

Assumptions

- Both IB and the transactions write log records (e.g., as in ARIES/IM [MoLe92]) for the changes that they make to the index being built.

2.1.1. Solution to the Duplicate-Key-Insert Problem

In NSF, the IB or the transaction inserter, whichever attempts to insert the *same* key *later*, avoids inserting the duplicate key in the index when the key is already found to be present in the index. The transaction always writes a log record saying that it inserted the key even though sometimes it may not actually insert the key since IB had already inserted it. The log record is written to ensure that in case this transaction were to roll back, then the key, which was inserted earlier by IB, would be deleted by the transaction from the index. Without that log record, the transaction will not remove the key from the index and that would be wrong since it would introduce an inconsistency between the table and the index data.

2.1.2. Solution to the Delete-Key Problem

If a transaction needs to delete a key and the key is *not* found in the index, then the deleter inserts a *pseudo-deleted* key. A key present in the index is said to be ***pseudo deleted*** if the key is logically, as opposed to physically, deleted from the index (this is done, for example, in the case of IMS indexes [Ober80]). Obviously, keys deleted in such a fashion take up room in the index. A 1-bit flag is associated with every key in the index to indicate whether the key is pseudo deleted or not. There are other motivations for keeping a deleted key as a pseudo-deleted key for as long as the deleting transaction is uncommitted (see [Moha90b] for details). For example, the deleter of a key does not have to do *next key locking* (see [Moha90a, MoLe92]) which saves an exclusive (X) lock call and improves concurrency. Next key locking prevents any key inserts in the key range spanning from the currently existing key which is previous to (i.e., smaller than) the deleted key to the next key (i.e., next higher key currently in the index). The transaction, by leaving a trail in the form of a pseudo-deleted key, lets IB avoid inserting that key later on, in case IB had picked up the key before the transaction deleted or updated the corresponding data. During the insert of the pseudo-deleted key, the transaction writes a log record so that in case the transaction were to roll back, then the key will be reactivated (i.e., put in the inserted state) in the index.

2.2. Details of the NSF Algorithm

When an index is being created on a table, IB will take the following actions.

1. Create the descriptor for the index
2. Extract the keys and sort them
3. Insert the keys into the index while periodically committing the inserts
4. Make the index available for read
5. Optionally, schedule cleanup of the pseudo-deleted keys

Below, we describe most of the above actions in detail.

2.2.1. Index Descriptor Creation

After the descriptor is created, the new index is visible for key insert and delete operations by transactions. The index is still not available to the transactions to use it as an access path for retrievals. Such usage has to be delayed until

---

1 In parallel with our work, some solutions to this problem were independently proposed in [SrCa91].

2 A latch is like a semaphore and it is very cheap in terms of instructions executed [Moha90a, Moha90b]. It provides physical consistency of the data when a page is being examined. Readers of the page acquire a share (S) latch, while updaters acquire an exclusive (X) latch.

the entire index is built.[3] For key inserts and deletes into the new index, it is assumed that these operations start at transaction boundaries after the index descriptor is created. That is, there will be no uncommitted updates against the table when the descriptor is being created. This is a short term quiesce of updates against the table. This can be achieved, for example, by IB acquiring a share (S) lock on the table and holding it for the duration of the index descriptor create operation. After the descriptor is built, the update transactions are allowed to start execution. Note that this quiesce lasts for a much shorter duration than the time interval between the start and end of the complete index build operation.

The following scenario illustrates the need for quiescing the update transactions before the descriptor is built. A transaction T1 inserted a record prior to creation of the descriptor for an index I1. Therefore, T1 did not write a log record for I1. Now, IB starts and inserts the key for T1's record into I1 (note that IB does not check for uncommitted records by locking). If later T1 were to roll back, then it would not delete the key from the index, thereby leaving a key in the index which points to a deleted record. The alternative is that the IB does locking to check whether the record is uncommitted. Due to the enormous locking overhead that this might entail, we did not take that approach (of course, we would have used the Commit_LSN idea [Moha90b] to avoid the locking when the circumstances were right). Instead, we chose to quiesce the update transactions just to create the descriptor. The SF algorithm does not require this quiescing. As explained later (see the section "3.2.3. Inserts and Deletes by Transactions While IB is Active"), in NSF also we can avoid the quiescing by logging, in the data page log record, the number of visible indexes and by performing, if necessary, logical undos to indexes during transaction rollback.

2.2.2. Extraction of Keys

IB reads the data pages sequentially to extract the keys. To make the CPU processing and I/Os efficient, multiple pages may be read in one I/O by employing sequential prefetch [TeGu84]. Also, the data pages may be read in parallel using multiple processes [PMCLS90] to speed up the key create and sort operations. As IB scans all the data pages, it extracts the keys and sorts them in a pipelined fashion. It completes that processing *before* it inserts any key into the index. This approach is adopted to make the index update operation very efficient (i.e., all the keys will be handled in key sequence). Note that the final merge phase of sort can be performed as keys are being inserted into the index. Doing all of the above may involve, depending on the size of the table, a considerable amount of processing. Therefore, to guard against loss of too much work in case of a system failure, NSF would employ a restartable sort like the one described in the section "5. Restartable Sort".

When accessing a data page, IB latches the page to extract keys from the records in that page. IB does not lock records when it extracts keys from them. Therefore, it is possible that IB will insert or attempt to insert a key for a record that has been inserted or updated by an uncommitted transaction. The uncommitted transaction may have already inserted that key or it may attempt to insert that key later on. The uncommitted transaction may also try to roll back its insert and, in that process, delete that key later on. In the next section, we explain the actions that must be taken as a consequence of IB possibly competing with transactions' uncommitted operations.

2.2.3. Inserting Keys into the Index by IB

Keys are inserted into the index while holding latches on index pages, as described in [Moha90a, MoLe92]. To make IB's insert processing efficient, the index manager will accept multiple keys in a single call. For transactions, the index is traversed from the root to insert or delete a key. For IB, tree traversals are avoided most of the time by remembering the path from the root to the leaf, as in ARIES/IM [MoLe92], and by exploiting that information during a subsequent call (see [CHHIM91] for a discussion of how this is done for retrievals). The proper amount of desired free space (for future inserts during normal processing) is left in the leaf pages as multiple keys are inserted.

It is assumed that an *undo-redo* log record is written as IB's keys are inserted into a leaf page. The log record can contain multiple keys. Page splits are also logged as in ARIES/IM. Logging by IB ensures that (1) the index tree would be in a structurally consistent state after restart or process recovery, and (2) media recovery can be supported without the user being forced to take an image (dump) copy of the index immediately after the index build completes. If the strategy is to restart the index build all over in case of a failure, then log writes by IB can be avoided. This strategy is probably unacceptable for large tables.

Next, we explain the actions that must be taken as a consequence of IB possibly competing with transactions' key insert and delete operations.

IB and Insert Operations

NSF deals with the problems caused by IB competing with a transaction's insert operation by extending the index management logic to reject insertion of a duplicate key. If the *transaction* actually inserts the key, then it writes an *undo-redo* log record. If the transaction does not insert the key because it had already been inserted by IB, then it writes an *undo-only* log record. In this case, the undo-only log record is needed so that, if the transaction were to roll back later, then that key will be deleted from the index by the transaction even though the key was originally inserted by IB. If the transaction were to commit, then the undo-redo log record written by IB or the transaction would ensure that the insert operation would be reflected in the index even if that index page were not written to disk due to a system failure. However, if IB's insert is rejected because of duplication, then no log record is written by IB.

We distinguish inserts of a duplicate key *value* for a nonunique index and for a unique index. This is because, for a unique index, *unique key value violation* needs to be detected and appropriate action taken. For a nonunique index, the key must match completely (<*key value, RID*>) for rejection. For a unique index, *transactions* use the current approach to detect the duplicate key value. That is, if the key value part of the key is found to be already present, then the transaction ensures that the found key, which may be a pseudo-deleted key, belongs to a committed record (or that the key is its own uncommitted insert) before it

3 Actually, if we are ambitious, then we could make the index gradually available for a range of key values starting from the smallest possible key value in the index as the index is being continuously modified by IB to include higher and higher key values.

determines whether a unique key value violation error needs to be returned. This is normally done by locking the key. The lock may be avoidable using the Commit_LSN technique of [Moha90b].

A similar approach is used by IB except that IB has to check that (1) the record on whose behalf IB is inserting the key is committed and (2) the record whose identical key value already exists in the index is also committed. Therefore, IB would lock both records in share mode, and then access the index page and the corresponding data page(s) to verify whether the duplicate key value condition still exists. If it does, then the index-build operation is abnormally terminated since a unique index cannot be built on this table.

No *next key locking* is done during key inserts into the new index while index build is still in progress. This locking, which is done to guarantee serializability by handling the *phantom problem* [Moha90a, MoLe92], is not needed since no readers are allowed to access the index while it is still being created. For a unique index, normally, next key locking is also done to ensure that one transaction does not insert a key with a particular key value when another *still-uncommitted* transaction had earlier deleted another key with the same key value. If next key locking is not done by both transactions, then the former will be able to do its insert and commit, and later the other transaction might roll back, causing a situation from which we cannot recover correctly. Here, the pseudo deletion of keys allows the transactions to keep out of such trouble without doing next key locking (see also [Moha90b]).

IB and Delete Operations

The following extensions to the index management logic are needed to deal with the race conditions between transactions' key deletes and IB's operations. The actions performed by a transaction trying to delete a key (*deleter*) are based on whether the key exists in the index at the time the transaction looks for it in the index. The key delete may be happening as a result of a forward processing action (record delete or update) or a rollback action (undo of an earlier key insert).

If the key exists in the index, then the deleter (1) modifies the key to be a pseudo-deleted key and (2) writes the usual log record.[4] IB's attempt to insert a key which is currently present in the index in the pseudo-deleted state is rejected. Note that the deleter will not physically delete the key since it may not be aware if IB had already extracted that key from the data page for subsequent insert into the index. Even if NSF were to maintain some information about IB's data page accesses (as is done in SF), which may let the deleter determine whether IB has already extracted the key, NSF cannot physically delete the key in the case of a *unique* index. This is to avoid the problem discussed earlier with reference to next key locking and a unique key value violation scenario which involved two transactions.

If the key does *not* exist in the index, then the deleter (1) **inserts** the key with an indicator that it is pseudo deleted and (2) writes the usual log record.[4] Again, the reason for inserting the pseudo-deleted key is to correctly deal with a race condition between the deleter and IB. For example, the key might have already been extracted by IB and IB may try to insert the key after the deleter commits. By leaving a *tombstone* in the form of a pseudo-deleted key and later rejecting IB's insert, NSF correctly deals with the race condition.

Note that if a key is not inserted by IB because an uncommitted transaction had deleted the data record by the time IB's scan reaches the corresponding data page, then the key would reappear in the index if that transaction were to roll back. This is because the rollback processing of the deleter would process the undo portion of its log record for the index and that would place the key in the inserted state. This is the reason for writing an *undo-redo* log record, as opposed to a redo-only log record, when the key is not found by the deleter. Such a log record is guaranteed to exist since the deleting transaction must have begun only after the index descriptor was created. The latter will be true because of the quiescing of update transactions at the time of descriptor creation.

Next, we give examples of insert and delete operations which can happen while IB is active.

1. Transaction T1 inserts a record with RID R and key value K for a nonunique index which is being concurrently built.
2. T1 inserts the key (<K, R>) into the index being constructed.
3. IB reads the new record and tries to insert its key.
4. Since IB finds the duplicate key, it does not insert the key.
5. T1 rolls back.
6. T1 marks the key as being pseudo-deleted and deletes the record in the data page.
7. T2 inserts a record at the same location (RID R) and the same key value (K).
8. T2 inserts the key (<K, R>) which would result in resetting the pseudo-deleted flag (that is, placing the key in the inserted state).
9. T2 commits which would result in <K,R> in the index and a valid record at RID R.

If, instead, T2 had inserted the same record with RID R1, then the index would have <K, R> as a pseudo-deleted key and <K, R1> as a normal key. In this case, if the index had been a unique index, then T2 would have (1) determined that the inserter of the pseudo-deleted version of <K, R> had terminated and (2) reset the pseudo-deleted flag in the existing entry and replaced R with R1.

Periodic Checkpointing by IB

For assuring the restartability of the key insert phase of index build, IB can periodically checkpoint the *highest* key that it has so far inserted into the index. This involves IB recording on stable storage the highest key and issuing a *commit* call. For restart of IB, this key can be used to determine the keys in the sorted list which remain to be inserted into the index. Though there is no integrity problem in IB trying to insert keys which were already inserted prior to the failure (since those attempted reinsertions would be rejected as a result of the previously explained duplicate keys handling logic and hence no log records would be written), it does avoid unnecessary work after restart. Note that, since log records for the index updates are written by the transactions and IB, the index would be in a structurally

4 With ARIES/IM, for a forward processing action, it would be a redo-undo log record, and for a rollback action, it would be a compensation (redo-only) log record.

consistent state after restart recovery is completed [MoLe92].

2.2.4. Cleanup of Pseudo-Deleted Keys

After IB completes its processing, garbage collection of the pseudo-deleted keys in the index can be scheduled as a background activity. If the index is created when the table has low delete activity, then this cleanup may not be worthwhile. Otherwise, pseudo-deleted keys can cause unnecessary page splits and cause more pages to be allocated for the index than are actually required. We would expect that an index-build operation would not be scheduled during a period of time when a significant portion of the table is expected to be updated. The garbage collection of pseudo-deleted keys involves the following steps: Scan the leaf pages. For each page, latch the page and check if there are any pseudo-deleted keys. If there are, then apply the Commit_LSN check [Moha90b]. If it is successful, then garbage collect those keys; otherwise, for each pseudo-deleted key, request a conditional instant share lock on it. If the lock is granted, then delete the key; otherwise, skip it since the key's deletion is probably uncommitted.

2.3. Discussion of the NSF Algorithm

2.3.1. Performance

In NSF, IB does not have complete control over the index tree when it is inserting keys into it since transactions are allowed to concurrently insert and delete keys directly in the tree. As a result, NSF cannot build the tree in a bottom-up fashion. In a ***bottom-up index build***, the keys are sorted in key sequence and then inserted into the first index page which acts as a root as well as a leaf. When this leaf becomes full, the next two index pages are allocated with one of them becoming the new root and the other one a leaf which will be used to insert the subsequent keys in the input stream. The old root is made into a leaf. Note that this is a special form of the page split operation in which no keys are moved from the splitting page to the new page. In a normal page split, usually, half the keys in the page being split are moved to the new page [Moha90a, MoLe92].

The above process is repeated until all the keys are inserted by the index builder. Note that the new keys are always added to the rightmost leaf in the tree without a tree traversal from the root and without the cost of latching pages and comparing keys. The result of this method of inserting keys is that the tree grows in a bottom-up, left to right fashion. Needed new pages are always allocated from the end of the index file which keeps growing. The resultant tree would be such that if a range scan of all the keys in ascending sequence were to be done at the leaf level, then pages in the index file would be accessed in *ascending* order of page numbers. That is, a ***clustered index scan*** would be possible. This would enable prefetching of index pages in *physical* sequence to be quite effective [TeGu84].

In NSF, to compensate for its inability to build the index tree bottom-up and to help range scanners, we can perform prefetch of index leaf pages effectively by using an idea suggested in [CHHIM91]. The idea is to perform prefetch of leaf pages by looking up their page-IDs in their parent pages, instead of prefetching pages in physical sequence.

To avoid a tree traversal for inserting each key, NSF can (1) remember the path from the root to the leaf, as in ARIES/IM [MoLe92], and exploit that information during a subsequent call (see [CHHIM91] for a discussion of how this is done for retrievals), and (2) pass multiple keys for insertion in one call to the index manager. If splits caused by IB's key inserts were handled just like splits during normal processing, then those keys that were inserted by *transactions* before IB starts adding keys to the index may be moved through a large number of leaf pages. To avoid the unnecessary CPU and logging overhead that this would cause, IB's splits can be specialized as follows: During a split, if there are any keys on the leaf which are higher than the key that IB is attempting to insert (these keys must have been inserted earlier by transactions), then IB can move those higher keys alone to a new leaf page and try to insert the new key.[5] If there are no such keys, then IB allocates a new leaf and inserts the new key there. This approach tries to mimic what happens in a bottom-up build. As a consequence, if the concurrent update activities by transactions are not significant, then the trees generated by NSF and by bottom-up build should be close in terms of clustering and the cost of tree creation.

The following additional points are worth noting regarding the performance of IB:

- During IB's scan of the records for extraction of keys, multiple *data* pages can be read in one disk I/O because of sequential access. Data pages could also be read in parallel. We believe that I/O time to scan the data pages would be a significant portion of the total elapsed time to build the index. Therefore, parallel reads would be required.
- The last page to be processed by the data page scan can be noted before starting IB's data scan so that if there are any extensions of the file after IB starts, IB does not have to process the new pages. Transactions would insert directly into the index the keys of records belonging to those new pages.
- During extraction of keys, each data page is only latched and no locking is performed. This saves the pathlength of lock and unlock, and it supports high concurrency by reducing interferences with transactions.
- One log record for multiple keys would save the pathlength of a log call for each key and reduce the number of log records written.
- The index leaf pages are only latched and no locking is done. These have concurrency advantages [Moha90a, MoLe92].

2.3.2. Restarting or Cancelling Index Build

By using a restartable sort (see the section "5. Restartable Sort"), if a system failure were to occur when IB is still scanning the data pages, then IB can be restarted without it having to rescan the data pages from the beginning. By periodically checkpointing the highest key inserted by IB, insertion of keys needs to be resumed only from the last checkpoint onwards, rather than all the way from the beginning. The reason the index itself cannot be used to determine the highest key inserted by IB after restart is because there is interference by transactions. Hence, IB has to track its position in the list of sorted keys.

---

5 If we are very ambitious about attaining close to perfect clustering, then we could collect statistics about key value distributions during the sorting of the keys by IB and estimate what the ideal page would be, in terms of its physical location in the index file, for the higher valued keys that are moved out.

Since cancelling an in-progress index build requires that the descriptor of the index be deleted, we need to quiesce update transactions by acquiring a share lock on the table. Quiescing is required so that the transactions which roll back can process their log records against the index without running into any abnormal situations. The rest of the processing for cancelling an index build is the same as what is normally required for the dropping of an index.

3. Algorithm SF: Bottom-Up Index Build with Side-File

In this section, we present the SF (*Side-File*) algorithm. First, we give a brief overview of the solutions to the problems described in the section "1.2. Problems". Then, we describe the SF algorithm in detail.

3.1. Overview of SF

The SF algorithm has the following features:

- IB first builds the index tree bottom-up without any interference being caused by direct key inserts or deletes in the index by transactions.
- Transactions' key inserts and deletes for the index under construction are appended in a side-file while IB is active *and* the index is "visible" to the transactions (details about when the index becomes visible to a particular transaction are given later). A side-file is an append-only (sequential) table in which the transactions insert tuples of the form <*operation, key*>, where *operation* is insert or delete. Transactions append entries without doing any locking of the appended entries.
- After inserting into the tree all the keys that it extracted from the records in the data pages, IB processes the side-file. When this is happening, transactions continue to append to the side-file.
- On completing the processing of the side-file, IB signals that from then on transactions must directly insert or delete keys in the new index.

Assumptions

- IB does **not** write log records for the inserts of keys that it extracts from the records in the data pages. It does write *redo-undo* log records for the key inserts and deletes that it performs while processing the side-file.
- Transactions write *redo-only* log records for the appends that they make to the side-file.

SF and NSF are different with respect to when they make the existence of the new index *visible* to update transactions. In NSF, the index is made visible when the index descriptor is created and from then on update transactions start making key inserts and deletes directly in the new index. In SF, the index is made visible based on IB's current position in its scan of the data pages. IB maintains a ***Current-RID*** position as it scans records from page to page. The index becomes ***visible*** to an update transaction if it modifies (inserts, deletes or updates) a record with a record ID, call it the ***Target-RID***, which is *less than* Current-RID. The Current-RID and Target-RID cannot be the same because of the page latching protocol used by update transactions and IB when they access a page. As mentioned before, as long as IB is active, only when a new index is visible to a transaction does the transaction make an entry in the side-file based on its record operation in the data page.

Next, we discuss how SF avoids the Duplicate-insert-key problem and the Delete-key problem. Even though a side-file is being used, these problems still need to be taken into account.

3.1.1. Duplicate-Key-Insert Problem

SF avoids the race condition between IB and a transaction attempting to insert the same key in the index by ensuring that the transaction will generate a key insert entry in the side-file only if the index is visible to it. That is, if the record is being inserted *behind* IB's scan position (i.e., target-RID < Current-RID), then IB will not be aware of that key and hence it will not insert that key into the index. If the index is not visible to the transaction (i.e., target-RID > Current-RID), then the transaction will not make any entries in the side-file and IB will insert that key into the index. Considerations relating to the rollback of the inserter are described later.

3.1.2. Delete-Key Problem

SF ensures that if a key were to be first extracted by IB for subsequent insert into the index and later a transaction were to perform an action on the corresponding record which necessitates the deletion of that key, then that transaction will append a key delete entry to the side-file. The latter action will occur because, by the time the transaction performs its record operation, Current-RID will be greater than Target-RID. Since IB first inserts into the index tree those keys that it extracted from the data pages and only after that it processes the side-file, SF guarantees that ultimately the key of the above example will be deleted.

We now consider the impact of the rollback of a transaction with respect to the *visibility* of an index. The question is, what would happen if during the forward processing of a transaction the index was *not* visible, but by the time the transaction rolls back the index becomes visible. What this implies is that (1) for a forward processing operation necessitating a key insert (i.e, a record insert or a record update involving key columns), IB would have inserted the new key in the index, and (2) for a forward processing operation necessitating a key delete (i.e, a record delete or a record update involving key columns), IB would have missed the old key and hence would not have attempted deleting it. For both cases, we must undo those actions. That is, in the first case, the new key must be eliminated from the index and in the second case, the old key must be inserted into the index. SF's approach to dealing with this problem is to make the transaction include information, such as the count of visible indexes, in the log record for the **data page update**. From this information, it would be possible to infer that the index was not visible during forward processing, but became visible during rollback. In such a case, if the index build is not yet completed, then an entry will be appended to the side-file when the log record for the data page is undone; for a completely built index, the index would be traversed to perform the necessary undo action.

To summarize, SF requires the following changes in the transaction forward processing and undo logic:

- The record management component has to be aware whether IB is active or not and if it is, then what the current scan position of IB is. This is because, if IB is active, then an append to the side-file of the index needs to be performed only if Current-RID is greater than Target-RID.

- Maintenance of a side-file which is an append-only table to make entries for insert or delete key actions. These appends are logged. New entries may be appended during the rollback of a transaction.
- Additional information is required in the log record for a data page operation. This will be the count of the visible indexes at the time the data page update was performed.[6]
- During undo processing, the count of visible indexes recorded in the log record of the data page is compared with the current count of visible indexes. If the former is smaller, then it implies that IB's action(s) needs to be compensated as follows: (1) if the index build for the last index is not complete, by making an entry (of key delete or insert) in the side-file; (2) for the newly visible indexes for which index build has been completed, by performing a logical undo (i.e., by traversing the tree from the root).

3.2. Details of the SF Algorithm

In this section, we describe the details of SF in the same manner that we described them for NSF.

3.2.1. Index Descriptor Creation

The descriptor for the new index is created and appended to the list of descriptors for the preexisting indexes of the table without quiescing (update) transactions. IB sets a flag (*Index_Build* = '*1*') which indicates that an index-build operation is in progress. This flag is examined by a transaction as it performs a record insert, delete or update operation while holding the data page latch.

3.2.2. Extraction of Keys

Like NSF, SF also reads multiple pages with one I/O and employs parallelism for reads. Keys of the records in a data page are extracted while holding a share latch on the page. As in NSF, IB does not lock records when it extracts keys. A current scan position called Current-RID is maintained as each record is processed to extract the key. This scan position determines whether the index is visible to the transactions or not, as was described earlier (see also Figure 1 and Figure 2). When IB finishes processing the last data page, it sets Current-RID to *infinity*. This ensures that, if the file were to be subsequently extended for the addition of records, then transactions which perform those actions will make entries in the side-file. As the data pages are scanned and keys are extracted, the keys are sorted. Like NSF, SF also uses a restartable sort algorithm.

3.2.3. Inserts and Deletes by Transactions While IB is Active

Transactions take actions based on the *Index_Build* flag and the current scan position of IB. In Figure 1 and Figure 2, we give the pseudo-code for index updates during forward processing and rollback of transactions. The pseudo-code with the comments should be self-explanatory to the reader.[7] It should be observed that SF is **not** quiescing all update transactions at any time. The one point that may need some explanation is that, in the case of the pseudo-code for rollback, it is possible for the difference between the numbers of indexes visible at the time of the original data page operation and during rollback to be even greater than one. This can happen, for example, because of the following sequence of events: T1 updates data page P10; index build for I3 begins and completes; index build for I4 begins and causes IB to process P10 and move Target-RID past P10; T1 rolls back its change to P10. In this scenario, while undoing its change to P10, T1 has to make an entry in the side-file for the index undo to be performed in I4 and it should perform a logical undo (by traversing the tree) in I3.

3.2.4. Inserting Keys into the Index by IB

The keys are completely sorted before their insertion into the index. Like NSF, SF also can pipeline the output of the last merge pass into the key insert logic. When IB is active, only IB inserts keys into the index. Because of these reasons, the index is built in a bottom-up fashion which is very efficient, as explained in the section "2.3.1. Performance". IB does not traverse the index from the root to insert keys as long as it has not started processing the side-file. IB does not write log records for its index operations until it starts processing the side-file. IB can check for unique-key violation in the same way as it does in NSF.

Periodic Checkpointing by IB

Until IB starts processing the side-file, periodically, IB can checkpoint the highest key inserted into the index and the page-IDs of the rightmost branch of the index. This checkpointing to stable storage is done after all the dirty pages of the index have been written to disk. In case of a failure, the index pages can be reset in such a way that the keys higher than the checkpointed key disappear from the index.

```
Target_Page := Data page for record
     Insert/Delete/Update operation
X-latch(Target_Page)
Target_RID := RID of affected record
IF Index_Build = '1' THEN /* index being built */
| IF Target_RID < Current_RID THEN /* New Index
| |       is VISIBLE; need to make entry in SF */
| | Modify target record, log action and count of
| |          visible indexes, and Update Page_LSN
| | Unlatch(Target_Page)
| | Make entry in side-file for insert key or
| |               delete key for index being built
| | Update all other indexes directly
| ELSE     /* Target_RID >= IB's scan position */
| |   /* New index INVISIBLE; no SF entry made */
| | Modify target record, log action and count of
| |          visible indexes, and Update Page_LSN
| | Unlatch(Target_Page)
| | Update all other indexes directly, completely
| |                    ignoring index being built
ELSE          /* No index creation in progress */
| Modify target record, log action and count of
| all indexes, and Update Page_LSN
| Unlatch(Target_Page)
| Update all indexes
Return
```

Figure 1: Pseudo-Code for Index Updates by Transactions During Forward Processing in SF

6 As a result, a minor restriction is that an index cannot be dropped while update transactions are active. That is, the number of indexes can only increase while update transactions are active. Hence, a drop index operation must acquire a share lock on the table before doing the drop. NSF also has this locking requirement since it cannot make the index descriptor disappear while update transactions are active.

7 While the pseudo-code is written, for brevity, as if only one index is being created at any given time for a table, as we discuss in the section "6.2. Extensions", creation of multiple indexes simultaneously in one scan of the data can be easily accomplished.

```
Target_Page := Data page for undo of record
               Insert/Delete/Update operation
X-latch(Target_Page)
Target_RID := RID of affected record
IF Index_Build = '1' THEN /* index being built */
| IF Target_RID < Current_RID THEN  /* IB will
| |   reflect in new index old state of record */
| | Current_Count := Count of all indexes,
| |                              including new one
| ELSE /* IB will not reflect in new index old
| |                              state of record */
| | Current_Count := Count of all indexes,
| |                               excluding new one
| Modify target record, log action and
|                                  Update Page_LSN
| Unlatch(Target_Page)
| IF data page log record's count < Current_Count
|       THEN
| | undo logically index change on those indexes
| |      made visible since original data change
| | /* i.e., make entry in SF for index under
| |    construction and for others, if any,
| |    traverse the trees to reflect effect of
| |    record's undo on index key              */
ELSE         /* No index creation in progress */
| Current_Count := Count of all indexes
| Modify target record, log action and
|                                  Update Page_LSN
| Unlatch(Target_Page)
| IF data page log record's count < Current_Count
|                                 THEN
|   undo logically index change on those indexes
|     made visible since original data change by
|     retraversing their tree
Return
```

Figure 2: Pseudo-Code for Index Updates by Transactions During Rollback Processing in SF

Also, the pages which keep track of index page allocation-deallocation status will be updated to indicate that the index pages allocated after the latest index checkpoint are in the deallocated state (i.e., they are available for allocation). This is easy to do since, with a bottom-up index build, as more keys are added and new pages are needed, the pages will be allocated to the index sequentially from the beginning of the file (see the section "2.3.1. Performance").

3.2.5. Processing of the Side-File

After building the index in a bottom-up fashion, IB processes the side-file from the beginning to end. While doing so, IB traverses the index from the root and, based on the entry in the side-file, inserts or deletes the key in the index as a normal transaction would do. That is, IB writes undo-redo log records which describe its actions. In order to avoid losing too much work if a failure were to occur when the side-file is being processed, periodically IB can checkpoint its progress in processing the side-file and issue a commit call. Until IB reaches the last entry in the side-file, transactions may still be appending new entries to the side-file. After processing the last entry in the side-file, IB resets the Index_Build flag so that subsequently transactions would modify the index directly. For improved performance, IB could sort the entries of the side-file, without modifying the relative positions of the identical keys, before applying those updates to the index. The sorting and processing of the sort stream must be done carefully to make them restartable. Also, by the time the application of the sorted entries to the index is completed, some more pages might have been added to the side-file. They could be processed sequentially (i.e., without sorting).

4. Comparison of the Algorithms

The main difference between NSF and SF is the maintenance of a side-file. The other differences between SF and NSF are:

- In SF, IB is able to build the index more efficiently than in NSF for the following reasons:
  - No log records are written by IB for inserting keys until side-file processing begins. In NSF, log records are written for all key inserts by IB. NSF reduces this overhead by logging all the keys inserted on a particular index page using a single log record.
  - Tree traversal from the root page of the index tree is not required to insert keys until side-file processing begins. In NSF, most of the time, IB would avoid tree traversals by remembering the path from the root to the leaf and exploiting that information during a subsequent call.
- In SF, no quiescing of table updates by transactions is required at any time. NSF quiesces all update transactions while creating the index descriptor.
- SF does not require the support of the concept of pseudo deletion of keys. This means that no changes are required for the existing index page and key formats.

It is expected that the index built by SF would be more clustered (i.e., consecutive keys being on consecutive pages on disk) than the one built by NSF. Deviations from the perfect clustering achievable without concurrent updates would be a function of the transactions' key insert and delete activities during the time of index build. These deviations need to be quantified for both algorithms.

5. Restartable Sort

In this section, we describe algorithms for making the different phases of the sort operation restartable. The two phases to be considered are: the sort phase and the merge phase. We discuss each one in turn next. We assume that a tournament tree sort [Knut73] is used. Without loss of generality, we assume that the keys are being sorted in ascending order.

5.1. Sort Phase

We assume that the sort is being performed, using a tournament tree, in a pipelined fashion as the data is being scanned by IB and the keys are being extracted from records. Periodically, we checkpoint the sorted streams as of certain scan position up to which the IB has scanned data pages of the table. This is so that, in case of a failure, IB would not have to rescan those data pages up to which the corresponding sorted streams were checkpointed. While taking a checkpoint, we wait for the tournament tree to output all the keys that have so far been extracted. We force to disk all those keys. We checkpoint the information (file names, etc.) relating to the already output sorted streams and the position of the IB data scan up to which keys have already been extracted and sorted. For the last sorted stream that was produced, we also record the value of the highest key that was output.

When we have to restart after a failure, we take the following steps:

- Read in the information from the latest checkpoint before the failure.
- Reposition the IB scan to the position indicated in the checkpoint.
- Discard any output sorted streams that did not exist as of the last checkpoint.
- Reposition the last sorted output stream that existed during the last checkpoint to the end of file position recorded in the checkpoint.
- Restart the tournament tree by inputting from the IB scan. If the smallest key produced during this sort phase is higher than the checkpointed value (i.e., highest key output at the time of the last checkpoint before the failure), then the output keys can still be sent to the same sorted stream in which we performed repositioning in the previous step. Otherwise, a new sorted output stream must be created.

5.2. Merge Phase

At different points during the merge phase, we need to write to disk all the keys that have been output so far from the merge operation. Let's call this a checkpoint operation. When we take such a checkpoint, we need to also record enough information so that we know how to repopulate the tournament tree with keys from the different input sorted streams correctly, in case we have to later on restart from this checkpoint. We should ensure that no key is left out from the merge and that no key is output more than once. This requires that we know precisely, for each input stream, the position of the highest key which has already been output by the merge operation. This tracking can be done as follows:

- Associate with the tournament tree a vector of N counters, where each counter is associated with one input stream and N is the number of leaf nodes in the tournament tree. All the counters are initialized to 1.
- Since, in a tournament sort, during the merge phase, a particular leaf node of the tree is always fed from the same input stream and a particular input stream is associated with only one leaf node, as we produce an output from the root of the tree, we know exactly which input stream that value came from. Consequently, while outputting a value from the tree, we increment by one the counter associated with the input stream from which that value came.
- During a checkpoint operation, we record the contents of the vector of counters and the descriptions (file names, etc.) of the input streams associated with those counters. Essentially, we are checkpointing the input streams' scan positions. We also record the information relating to the output stream (the position of the end of file on the output file, etc.).

When we have to resume the merge operation after a system failure, we look at the latest checkpoint information for the merge and do the following:

- Truncate the tail of the output file so that its end of file position corresponds to the checkpointed information.
- Read in the contents of the vector of counters and use the associated input file descriptions to reposition the input files to the positions indicated by the counters' values. If the counter value for a file is k, then that file should be positioned so that the next key to be input into the merge from that file would be the key at position k.
- Restart the merge operation by initializing the counters to the checkpointed values and reading from the input files at their current positions as set up in the previous step.

6. Conclusions

As the sizes of the tables to be stored in DBMSs grow and several indexes may need to be created long after the tables were created, disallowing updates to a table while creating an index for it will not always be acceptable. Higher availability of data is becoming more and more important as many companies expand towards world-wide operations and as users' expectations about data availability increase [Moha92]. The so-called *batch window* is rapidly shrinking. As more and more companies merge, and automation of various operations become common place, the volume of data to be handled grows enormously. As disk storage prices drop and the disks' storage capacities increase, users tend to keep more and more of their data online. These trends have necessitated a new approach to the construction of indexes.

6.1. Summary

We described two efficient algorithms, called NSF (No Side-File) and SF (Side File), which allow concurrent update operations by transactions during index build. Our emphasis has been to maximize concurrency, minimize overheads and cover all aspects of the problem, including recovering from failures without complete loss of work. The efficiency of these algorithms comes from the following: (1) When data is scanned, no locks are acquired on the data pages or the records. (2) The index is built bottom up in SF and a multiple-keys interface is used in NSF. (3) Parallel reads and bulk I/Os (i.e., read of multiple pages in one I/O) are used to shorten the time to scan data. SF first builds the index bottom up and maintains a side-file for updates which occur while it is constructing the index. SF and NSF can create correctly both unique and nonunique indexes, without giving spurious unique-key-value-violation error messages in the case of unique indexes.

We also presented algorithms for making the sort operation and the tree building operation restartable. The algorithms relating to sort have very general applicability, apart from their use in the current context of sorting for index creation.

We did not consider using the log, instead of the side-file, to bring the index up to date for reasons like the following:

- The log records written for the data page updates may not contain enough information to determine how the index should be updated. For example, the new index being built may be defined on columns C1 and C2, and the log record for the data page update may contain only the before and after values of the modified column, say C2, of the updated record. Given only C2's before and after values from the log record, there is no cost-effective way to determine what key has to be deleted from the index and what key has to be inserted into the index since C1's value is not known. Extracting that information by examining the record in the data page would not be possible if the record had already advanced to a future state where the C1 value is no longer what it used to be.
- Even if the DBMS were to be inefficient enough to log even unmodified columns and hence the above is not a problem, the amount of log that would have to be scanned may be too much to make this a viable approach. Also, the system must ensure that the relevant portion of the

log is not discarded before the index build operation completes.[8]

6.2. Extensions

Since the cost of accessing all the data pages may be a significant part of the overall cost of index build, it would be very beneficial to build multiple indexes in one data scan. Our algorithms are flexible enough to accommodate that. The functions of scanning data and extracting keys for all the indexes being built simultaneously must be separated from the functions of sorting the keys, inserting them into the index and processing the side-file for each of those indexes. A process can be spawned for each index to sort the keys, insert them and process the side-file.

Our algorithms can also be easily extended to the storage model in which the records are stored in the primary index and the primary key is required to be unique. We would perform a complete range scan of the primary index to construct the keys for the new index. In SF, in the place of Current-RID, we would use the current-key as the scan position in the primary index. Since the primary key has to be unique, this position also would be a unique one in the index.

We assumed that the index manager does *data-only locking*, as in ARIES/IM [MoLe92]. In data-only locking, the lock names for the locks on the keys are the same as the names for the locks on the data from which those keys are derived. For example, with record locking, the lock on a key is the same as the lock on the corresponding record and, with page locking, it is the lock on the data page containing the corresponding record. Consequently, even if IB were to insert into the index a key of an *uncommitted* record, the transaction which performed that record operation (insert or update) does not have to acquire a new lock to protect the uncommitted key in the new index. It is because of this reason that IB, once it finishes building the index, can make available the new index for reads by transactions without the danger of exposing those transactions performing *index-only* read accesses to uncommitted keys. If the index locks were different from data locks, as in ARIES/KVL [Moha90a], then IB, on finishing building the index, would have to quiesce all update transactions before allowing reads of the new index.

Additional work needs to be done to permit concurrent index build when the DBMS supports transient versioning of index data to avoid locking by read-only transactions [MoPL92].

7. References

CHHIM91 Cheng, J., Haderle, D., Hedges, R., Iyer, B., Messinger, T., Mohan, C., Wang, Y. *An Efficient Hybrid Join Algorithm: a DB2 Prototype*, **Proc. 7th International Conference on Data Engineering**, Kobe, April 1991. An expanded version of this paper is available as **IBM Research Report RJ7884**, IBM Almaden Research Center, December 1990.

DeGr90 DeWitt, D., Gray, J. *Parallel Database Systems: The Future of Database Processing or a Passing Fad?*, **ACM SIGMOD Record**, Volume 19, Number 4, Decemeber 1990.

Gray78 Gray, J. *Notes on Data Base Operating Systems*, In **Operating Systems - An Advanced Course**, R. Bayer, R. Graham, and G. Seegmuller (Eds.), LNCS Volume 60, Springer-Verlag, 1978.

Knut73 Knuth, D. **The Art of Computer Programming: Volume 3**, Addison-Wesley Publishing Co., 1973.

MHLPS92 Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, **ACM Transactions on Database Systems**, Vol. 17, No. 1, March 1992. Also available as **IBM Research Report RJ6649**, IBM Almaden Research Center, January 1989.

Moha90a Mohan, C. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990. A different version of this paper is available as **IBM Research Report RJ7008**, IBM Almaden Research Center, September 1989.

Moha90b Mohan, C. *Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990. Also available as **IBM Research Report RJ7344**, IBM Almaden Research Center, February 1990.

Moha92 Mohan, C. *Supporting Very Large Tables*, **Proc. 7th Brazilian Symposium on Database Systems**, Porto Alegre, May 1992.

MoLe92 Mohan, C., Levine, F. *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*, **Proc. ACM SIGMOD International Conference on Management of Data**, San Diego, June 1992. A longer version of this paper is available as **IBM Research Report RJ6846**, IBM Almaden Research Center, August 1989.

MoPL92 Mohan, C., Pirahesh, H., Lorie, R. *Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions*, **Proc. ACM SIGMOD International Conference on Management of Data**, San Diego, June 1992.

Ober80 Obermarck, R. *IMS/VS Program Isolation Feature*, **IBM Research Report RJ2879**, IBM San Jose Research Laboratory, July 1980.

PMCLS90 Pirahesh, H., Mohan, C., Cheng, J., Liu, T.S., Selinger, P. *Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches*, **Proc. 2nd International Symposium on Databases in Parallel and Distributed Systems**, Dublin, July 1990. An expanded version of this paper is available as **IBM Research Report RJ7724**, IBM Almaden Research Center, October 1990.

SISU91 Silberschatz, A., Stonebraker, M., Ullman, J. (Eds.) *Database Systems: Achievements and Opportunities*, **Communications of the ACM**, Vol. 34, No. 10, October 1991.

SrCa91 Srinivasan, V., Carey, M. *On-Line Index Construction Algorithms*, **Proc. 4th International Workshop on High Performance Transaction Systems**, Asilomar, September 1991.

TeGu84 Teng, J., Gumaer, R. *Managing IBM Database 2 Buffers to Maximize Performance*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.

8 Log records may be discarded if image copies of the data have been taken and the log records are not needed for restart recovery, normal undo or media recovery using such image copies.