An Efficient Method for Performing Record Deletions and Updates Using Index Scans

C. Mohan

IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA
mohan@almaden.ibm.com
www.almaden.ibm.com/u/mohan/

Abstract

We present a method for efficiently performing deletions and updates of records when the records to be deleted or updated are chosen by a range scan on an index. The traditional method involves numerous unnecessary lock calls and traversals of the index from root to leaves, especially when the qualifying records' keys span more than one leaf page of the index. Customers have suffered performance losses from these inefficiencies and have complained about them. Our goal was to minimize the number of interactions with the lock manager, and the number of page fixes, comparison operations and, possibly, I/Os. Some of our improvements come from increased synergy between the query planning and data manager components of a DBMS. Our patented method has been implemented in DB2 V7 to address specific customer requirements. It has also been done to improve performance on the TPC-H benchmark.

1. Introduction

Relational database management systems (***RDBMSs***) utilize B$^+$-tree indexes [BaMc72] very often to efficiently identify a set of records with certain characteristics. The CPU cost of performing a root-to-leaf traversal in an index tends to be very high since a binary search is done at every level of the tree. We would like to avoid traversals as much as possible by performing range scans at the leaf level. Here, we illustrate how traditionally RDBMSs have not been that efficient during certain types of index accesses. We present a method to improve that situation.

The query processing (optimization and execution) research community is yet to recognize the need for the query processing component (let us call it ***RDS*** (relational data system), following the System R terminology) to be conscious of concurrency control implications of its actions. In [Moha92], we argued in favor of increased synergy between the RDS and data manager (***DM***) components by giving specific examples where both correctness of query executions and/or performance were impacted. This paper, in the process of describing an efficient method to perform record deletions and updates via index scans, provides another illustration of the benefits of such increased synergy.

The rest of the paper is organized as follows. Before discussing, in section 2, the problem that we are solving, in the rest of this section, we give a brief introduction to the relevant aspects of query processing and index locking. In section 3, we describe our method for efficiently performing record deletions via an index scan. In section 4, we discuss extensions of our method to handling record updates. We conclude with section 5.

1.1 Query Processing

RDBMSs implement the concept of a cursor. A ***cursor*** is a construct (an iterator) used to scan and process a set of data (records/keys/tuples satisfying certain conditions), one at a time. RDBMSs implement two types of cursors: user cursors and system cursors. A ***user cursor*** directly corresponds to a cursor defined in a user application using an SQL DCL CURSOR statement. ***System cursors*** are the ones that the RDBMS defines and uses internally to access the tables whose data is needed to satisfy users' queries. One or more system cursors might be used to produce the output corresponding to a single user cursor. An example of a situation when a single user cursor might be implemented using multiple system cursors is one where the query requires accessing multiple tables (e.g., a join operation). In such a situation, each of the tables' data will be accessed using a separate system cursor. It is also possible that there are some system cursors in use that do not relate to any user cursors. This will be the case when the user issues ***set-oriented delete/update statements*** (i.e., SQL statements of the form DELETE/UPDATE ... FROM ...

WHERE ...). Such statements are also called ***searched deletes/updates***.

For exposition purposes, the data storage model that we assume is that of System R, where the data (i.e., the records of the table) is stored in a set of ***data pages*** that are separate from the B^+-tree indexes. All the indexes on the table contain only the key values and record identifiers (***RIDs***) of records containing those key values. Each ***key*** consists of a <key value,RID> pair. A RID consists of a data page ID concatenated with a sequence number unique within that page. The latter is the RID map slot number. The ***RID map*** on each data page is an array of pointers and it provides a level of indirection that permits the record to be moved around within the page or even be overflowed to a different page, without affecting the index entries. Traditionally, B^+-trees have been used to support range scans. Compared to hash-based storage structures, the CPU cost of traversing a B^+-tree from root to leaf is significantly higher since a binary search is performed at every level of the tree. As sizes of tables grow, the number of levels also increase. Further, the cost of fixing and unfixing a page in the buffer pool as we search down the tree also turns out to be significant. Hence, performance-conscious DBMS designers try hard to minimize the cost of tree traversals and scans via various optimizations [Anto93, Anto96, CHHIM91, GLSW93, MHWC90].

An ***access path*** is used to implement a system cursor. The most common access paths are sequential scan of a table's data pages (a ***table scan cursor*** (***TSC***)) and range scan on a B^+-tree index (an ***index scan cursor*** (***ISC***)). Both system cursors and user cursors might operate over permanent data as well as temporary data (e.g., intermediate or final results of a query stored in temporary tables or workfiles). Even if a user cursor requires accessing only a single table's data, multiple system cursors might have to be used due to the exploitation of query processing techniques like index ANDing/ORing [Anto93, MHWC90].

Information relating to a cursor is represented using a ***cursor control block*** (***CCB***). The most important attributes of a cursor are:

- Is the cursor *open* or not?

Only when a cursor is open, can a next (or fetch) call be issued to get the next piece of data in the set of data over which the cursor is defined.

- Does it have a valid position?

A user cursor has a ***valid position*** if it is positioned on a piece of data that still exists. This means that anytime a DBMS deletes a record, it has to make sure that all those user cursors of that transaction which are positioned on that record are *invalidated*. Invalidation will prevent a ***cursor-based update/delete*** (i.e., SQL statements of the form UPDATE/DELETE WHERE CURRENT OF CURSOR - also called ***positioned updates/deletes***) from being processed, until the cursor position becomes valid again by the issuance of a next call. The reason invalidation is important is because of the fact that another record might be inserted which is assigned the same RID as that of the deleted record and then if the cursor-based update or delete were to be processed after such a reuse of the RID, the *wrong* record would get updated or deleted. Of course, the situation gets much more complicated with the use of optimizations like blocking of the transfer of the records satisfying the user's query [Moha92] and the support of features like scrollable cursors. More sophisticated techniques are needed to avoid such errors when those features are supported. Even if only one system cursor is being used to support a given user cursor, the positions of the user and system cursors might be different at any given point in time due to the support of features like blocking and scrollable cursors.

Since cursor-based deletes and updates are specified by *users* only with reference to user cursors, there is no need to perform invalidations of *system* cursors. System R used to unnecessarily perform invalidations even for system cursors.

In the case of set-oriented delete and update statements, where the DBMS creates system cursors without any related user cursors, RDS will issue cursor-based deletes and updates with reference to those system cursors.

- If a cursor has a valid position, what is it?

In the case of a TSC, the cursor position is denoted by the RID of the record on which the cursor is positioned. In the case of an ISC, it is denoted by the key on which the cursor is positioned.

- Is a cursor updateable?

If a cursor is updateable, then the UPDATE/DELETE WHERE CURRENT OF CURSOR SQL statements can be used with such a cursor. When such a statement is issued against an updateable user cursor that has a valid position, the record on which the cursor is currently positioned is updated or deleted. In the case of an update, the cursor continues to be positioned on the same record. In the case of a delete, the cursor no longer has a valid position. Typically, a user cursor is updateable if it does not involve any joins and it does not contain an ORDER BY or a GROUP BY clause.

- What are the predicates of a cursor?

Many times, a cursor is defined with a set of predicates associated with it. These predicates are typically derived from the information in the WHERE clause of the SQL statement issued by the user. The set of predicates of a cursor can be divided into two classes: sargable and residual. ***Sargable*** predicates are the ones that are evaluated by DM. ***Residual*** or non-sargable predicates are the ones that are evaluated by RDS. In the case of an index scan, the sargable predicates can be further subdivided into two classes: ipreds (index predicates) and dpreds (data predicates). ***Ipreds*** are the predicates that involve only the columns in the index and hence can be evaluated by the index manager (***IM***). ***Dpreds*** involve one or more columns that are not present in the index. Hence, the data record needs to be accessed by the record manager (***RM***) for dpreds to be evaluated by RM.

1.2 Index Locking

Sophisticated concurrency control is employed while accessing indexes to assure that several properties are satisfied. One property is serializability (or repeatable read) [EGLT76]. In the case of unique indexes, another property to be guaranteed is assuring that no two keys with the same key value are present at any time in the index. Various sophisticated index locking protocols are described in [Lome93, Moha90a, Moha95, MoLe92, WeVo01]. Two types of locking are done in those protocols: *data-only locking* and *index-specific locking*. ARIES/IM is the index locking and recovery method implemented in DB2 for Unix and Windows. An extended version of it has been implemented in DB2 for the mainframe [Moha99a]. Variants of ARIES/IM can support both index-specific and data-only locking.

With ***data-only locking***, a lock on a key is equated to a lock on the corresponding piece of data that contains the key. For example, with data-only locking and *record locking granularity*, to lock a key, ARIES/IM locks the record whose *RID* is present in the key.

Data-only lock name := <TableID, RID(Key)>

With data-only locking and *page locking granularity*, to lock a key, ARIES/IM locks the data page whose *PageID* is present in the RID in the key.

Data-only lock name := <TableID, PageID(Key)>

With ***index-specific locking***, a lock on a key is made to be different from the lock on the corresponding piece of data that contains the key. For example, in ARIES/IM, with index-specific locking and *record locking granularity*,

Index-specific lock name := <IndexID, RID(Key)>

With data-only locking and *page locking granularity*,

Index-specific lock name := <IndexID, PageID(Key)>

Tradeoffs are involved in choosing between index-specific locking and data-only locking [Moha95]. Index-specific locking requires more locks to be acquired for most operations compared to data-only locking. But under some conditions, index-specific locking can support higher levels of concurrency.

In order to guarantee serializability, in ARIES/IM, whenever a key is deleted (due to a record deletion or a record update which causes the key value to change), an exclusive (X) lock for commit duration is obtained on the next higher key that currently exists in the index. It is this *next key lock* which blocks subsequent readers who look for the deleted key until the deleting transaction terminates. In a similar fashion, a next key lock is acquired *momentarily* on the next key during the insert of a key to make sure that the insert is not going to interfere with a reader who has already searched for (and not found) the key being inserted. If such a read transaction is still executing, the inserter's next key locking delays the insert since a reader obtains a share (S) lock for commit duration on the next key if the reader does not find the key that it is looking for (S is compatible with S but is incompatible with X). The mode (X, IX, etc.) in which the next key lock is acquired during inserts is unimportant for the discussions of this paper. For details, the reader should refer to the papers on ARIES/KVL and ARIES/IM [Moha90a, Moha95, MoLe92].

Typically, when DM is called by RDS for a record to be fetched via a cursor, RDS will know more about why the record is being retrieved:

- only for reading - a non-updateable cursor
- definitely for being updated/deleted - a set-oriented delete/update statement with no residual predicates
- possibly for being updated/deleted - a set-oriented delete/update statement with residual predicates or an updateable cursor (i.e., the user might issue an update/delete where current of cursor)

Consequently, RDS indicates to DM what mode of lock should be acquired on the record *when the record is fetched via a cursor*. Typically, RDS asks DM to get an S lock. In some systems, when RDS suspects or knows for sure that the fetched record would be updated/deleted subsequently by the current transaction, then, to reduce the likelihood of deadlocks, RDS asks for an update (U) lock to be acquired. Since U is incompatible with U and X, but is compatible with S, this ensures that no other transaction would be able to obtain a U lock on the same record until the current transaction terminates. For this case, if only an S lock were to be obtained, then two different transactions could both first get S locks. Later, both may try to update/delete the record at which point both will try to get X locks and thereby create a deadlock situation. Some systems (e.g., DB2) avoid such a possibility by using U locks.

Typically, acquiring a lock in the absence of contention requires 100s of instructions. DBMS and application designers normally spend a significant amount of effort in minimizing the cost of locking by reducing the number of locks, number of lock calls, etc., while at the same time permitting a high degree of concurrency. We have discussed elsewhere [Moha90b] the significance of reducing locking overhead and proposed a simple, yet powerful, technique called Commit_LSN that has been implemented in DB2 to great advantage.

2. Problem Description

Consider the following set-oriented SQL delete statement:

DELETE

FROM T1

WHERE C1 > 10 AND C1 < 20

Assume that an ascending B^+-tree index on C1 exists (call it I1) and that the optimizer chooses to use I1 and perform a range scan to determine the set of records to be deleted. In this case, there are only ipreds and no dpreds. RDBMS users very commonly issue these sorts of deletes. Deletes like these are also generated internally by RDBMSs to implement referential integrity when *cascade-on-delete* is the rule and there is an index on the foreign key of the child table, where that key is the primary key of the parent table, and a parent record deletion's effect needs to be propagated to the child table [CEHH90, HaWa90].

For the above DELETE statement, a typical execution in existing RDBMSs would involve the following steps (in addition to others which are not of interest here):

- RDS would call DM to open a range scan on the C1 index and fetch the RID of the first qualifying record. Let the ISC so created be called ISC1. RDS would request the qualifying data to be locked in S or U mode.

Key	PageID	Key Position	Page LSN	Return_Current	Lock Mode

Table 1 Some of the Fields in an Index Scan Cursor Control Block

- DM would invoke IM which would perform a root-to-leaf traversal on I1 to locate the first key whose key value is greater than 10. Let that key be k1 (<kv1, rid1>). As a result of this action, k1 would be locked in S or U mode. If index-specific locking is being done, as in System R, ARIES/KVL and a variant of ARIES/IM [Moha90a, MoLe92], then the lock would be acquired on the key itself and this lock would be different from a lock on the data from which the key was derived. If data-only locking is being done, as in the implemented (in DB2) version of ARIES/IM, then the lock would be actually on the underlying data (e.g., on rid1 if the locking granularity is a record). ISC1 would be positioned on k1, and the ID of the leaf page (say page Pi), the logical position of k1 on the page (say jth key) and the log sequence number (***LSN***) [MHLPS92] of the page (say LSNk) would be recorded in the ISCCB for use during a next call [Moha90a, MoLe92] - see Table 1.
- DM would return rid1 to RDS which would immediately turn around and ask DM to delete the record on which ISC1 is positioned (the one with RID rid1). Unlike in this example, if residual predicates had existed, then RDS would have to evaluate those predicates before it is determined whether the record should be deleted. If this were a cursor-based delete, then the application would have to decide whether the record should be deleted.
- RM would then acquire an X lock on rid1 and delete the record. If IM was doing data-only locking, then this locking would be an upgrading of the previously acquired S or U lock to an X lock; otherwise, it would be a new lock on rid1. Next, RM would examine the descriptor for I1, create the key (k1) for that index and ask IM to delete k1. Note that even though the key for I1 is already known, the existing DBMSs waste their time looking up the index descriptor for I1 and reconstructing k1!

  Of course, if there are other indexes on T1, then RM would have to make sure that their keys are also deleted.
- IM would traverse I1 from root to leaf to locate k1. Assuming that no changes had taken place on Pi between the time of key lookup and the time of key deletion, the leaf page arrived at would still be Pi. If index-specific locking is being done, then IM would lock the key in X mode. In all cases, in order to guarantee serializability, the next key would be locked in X mode for commit duration before the deletion of k1 is performed. As a result of the logging of this key deletion and the consequent updating of page_LSN [MoLe92], the LSN on Pi would become greater than LSNk.

  It is totally unnecessary to retraverse the tree to locate the key since IM already knows from ISC1's CCB as to where the key was located when it was looked up last! The traditional execution strategy in all does N root-to-leaf traversals during key deletions, where N is the number of records to be deleted. Our method avoids the traversals completely as long as the affected leaves are not modified in the interim by other transactions.
- DM would then return to RDS which would immediately issue a next call on ISC1.
- IM would examine ISC1's CCB and noting that previously the scan was positioned on a key on Pi, reaccess Pi and check if its LSN is still LSNk (which it

finds in the CCB). Since Pi's LSN is no longer LSNk, IM can no longer use the logical key position information in the ISCCB to determine cheaply what the next key is. IM can still search the same page and locate the next key safely only if it finds that the key k1 is ***bound*** on the page (i.e., there is a smaller key and a larger key compared to k1 on the same page). This boundedness condition would not be satisfied if, for example, k1 had been the very first key on the page. In the latter case, IM would retraverse the tree from the root to locate the next key. Assuming that no other transaction had altered Pi between the time the scan was positioned on k1 and now, we know that the retraversal would result in Pi again being determined to be the correct leaf page to look into to locate the next key!

Note that as long as the qualifying keys for the record deletions span more than one leaf page, we are guaranteed to encounter this deletion-of-first-key-on-page condition at least from the second of those leaf pages which contain the qualifying keys. From then on, until the rest of the keys are examined and deleted, for all those next calls the unboundedness condition would hold, thereby forcing IM into doing a root-to-leaf traversal on every next call! The traditional execution strategy does N root-to-leaf traversals during the open and next calls, where N is the number of records to be deleted, if the first qualifying key to be located happens to be the smallest key on the leaf page in which it exists. If the latter condition is not true, then there would be M traversals, where M-1 of the qualifying keys do not exist on the first leaf page in which a qualifying key is found. These traversals of course would be very expensive, especially in terms of CPU costs. In fact, these traversals may also cause unnecessary I/Os since they ensure that all the ancestors of all the accessed leaves would also be accessed. Ideally, we should be able to avoid those I/Os. Our method would guarantee that ideal performance as long as no other transactions make any changes to the leaf pages of interest during the course of the processing of the set-oriented delete statement.

- Once the next key is located, IM would behave as described before for the first key. In particular, it would unnecessarily lock in S or U mode the found key, without realizing that that key had already been locked in X mode as part of next key locking during the deletion of the previous key!
- When RDS returns with a delete call, RM, in the case of data-only locking, without realizing that the record is already locked in X mode, would request an X lock on the record!
- When RM calls IM to delete the just deleted record's key, IM will again retraverse the tree to locate that key and, in the case of index-specific locking, lock that key again in X mode, even though it had locked that key in X mode as a result of next key locking during the earlier key deletion!

To summarize, the following are the costs involved in performing a set-oriented deletion using the traditional method, where N is the number of records qualifying for deletion and M-1 is the number of qualifying keys (out of N) which are not on the first leaf that is accessed:

- **Data-only locking**: 3N lock calls involving N+1 locks

  N S/U-lock calls on records during open and next calls,

  N upgrade-to-X-lock calls on records during record deletions,

  N X-lock calls on next keys' records during key deletions.
- **Index-specific locking**: 4N lock calls involving 2N+1 locks

  N S/U-lock calls on keys during open and next calls,

  N X-lock calls on records during record deletions,

  N X-lock calls on keys during key deletions,

  N X-lock calls on next keys during key deletions.
- 2N or N+M root-to-leaf traversals

  N root-to-tree traversals during key deletions

  First qualifying key is smallest key on first leaf accessed: N root-to-tree traversals during open and next calls

  First qualifying key is not smallest key on first leaf accessed: M root-to-tree traversals during the last M next calls

Note that we do not count the unlock calls and the tree traversals that might be caused by page deletions since these would be the same with the traditional method as well as our method.

Our customers have noticed the above inefficiencies and have complained about them. That is what motivated us to tackle these inefficiencies. We are not aware of any database research literature that discusses the kind of method that we present in this paper!

It is interesting to note that current optimizers do not estimate correctly the CPU cost of using an index for the example delete statement. They assume that a range scan will be done. Hence, they account for only one traversal of the index. They assume that after the initial traversal for the first RID to be fetched, subsequent RIDs' retrieval will cost very few instructions during the next calls! As a consequence of RDS designers not paying enough attention to the actual processing that occurs in IM, in comparing the costs of a table scan versus an index scan for the above scenario, the wrong choice may be made due to the cost model not reflecting reality!

3. Our Method

In our method, during key deletions, IM avoids performing root-to-leaf traversals, and RM avoids looking up the range-scan index's descriptor and reconstructing the key by exploiting the information in the ISCCB. That is, RM, *for the range-scan index only*, instead of performing the normal descriptor lookup and key construction, and invoking the normal key delete IM routine, invokes a new *delete where current of cursor* command on the ISCCB. In processing this command, IM would look at the information about the scan position (page number, logical key position within page and LSN on page) and access immediately the corresponding leaf page to do the key deletion. If the page had not changed since the scan was positioned on it, IM would know precisely where the key is on the page and hence it would delete it right away. Even if the page had changed (leaf's current page_LSN > LSN in ISCCB), IM can check to see if the key is still on the same page. Only if it weren't on that page anymore, would it have to traverse the tree from the root to locate it. The key may not be on the same leaf anymore since another transaction could have done a split of that leaf and moved the key of interest to a different page. In many DBMSs (e.g., NonStop SQL, Informix, DB2) and index concurrency control methods like ARIES/KVL and ARIES/IM, in the interest of supporting very high concurrency, one transaction is allowed to move to a different page a key which is currently locked by another transaction.

With the above approach, when no concurrent changes by other transactions are happening to the leaf pages of interest, we eliminate completely N root-to-leaf traversals during key deletions. We also avoid the key reconstruction overheads. The other goal of our method is to avoid the root-to-leaf traversals during the next calls. These traversals traditionally happen because the LSN on the leaf page changes between the time of retrieval of a key and the time of the retrieval of the following key during a next call due to the intervening key delete call that causes the currently smallest key on the leaf to be deleted. Since the previously deleted key is no longer bound on the page, IM is unable to safely do a local search on the leaf to locate the next key that is now actually the smallest key on the page! For us to safely still return the currently smallest key on the page as the next key, what we need is a way to know that even though the LSN currently on the leaf is different from what it used to be (as recorded in the ISCCB) during the earlier next call, the only change that had been made to the page between the time of the previous key lookup and the current one is the deletion of the previously returned key. Our method does this by adding a ***return_current*** flag to the ISCCB (see Table 1).

This flag is reset to '0' when ISCCB is created and when a key is returned as part of an open or next call.

Our method takes the following *additional* steps during a delete where current of cursor operation involving the ISCCB:

- If IM had to search to locate the key to be deleted (since page_LSN of leaf whose ID was remembered in ISCCB was > LSN remembered in ISCCB), then IM updates the ISCCB to contain the page number and logical position where the key to be deleted was found. Note that the value in the key field in ISCCB is not modified since that value will be needed during the next call if a tree traversal or binary search within the leaf page were to be required at that time to get the next higher key.
- After deleting the key (which is performed after doing the X locking of the next key), logging the deletion and setting the page_LSN to the delete-key log record's LSN, set the LSN in ISCCB to the leaf's new page_LSN.
- Set to '1' the return_current flag.

When a next call is issued, our method does the following:

- Access the leaf page remembered in ISCCB and check that page's LSN. If the page_LSN is equal to the LSN in ISCCB and return_current is set to '1', then return the key that is currently in the same logical position remembered in ISCCB. Note that there is no need to lock this key before returning it since the locking would have been done during the preceding key's deletion. This is how we avoid the unnecessary relocking, and traversal or at least a local binary search that happens (in the key is bounded case) in the traditional method.

On the other hand, if the page_LSN is equal to the LSN in ISCCB but return_current is not '1', then return that key which is in the position that is next to the logical position remembered in ISCCB, *after locking that key*. If the LSNs don't match, then IM behaves as in the traditional method (i.e., try to do a local search and if it is not possible, then retraverse). Return_current will be '0' if the previously returned key was not deleted. This handles the case where dpreds and/or residual predicates exist and some of those predicates were not satisfied for the previously returned key's RID. Of course, in the example delete statement of the last section, there were only ipreds and so all returned keys' records would be deleted.

- Before returning to the caller, modify ISCCB as in the traditional method (copying the returned key, recording page number, logical position and LSN) and in addition reset return_current to '0'.

Data-only Locking

Predicates Present	Mode of Lock Acquired
Only Ipreds	X
Dpreds and/or Residuals	S/U

Index-specific Locking

Predicates Present	Mode of Locks Acquired
Only Ipreds	KLOCK=X; RLOCK=X
Dpreds, but no Residuals	KLOCK=S/U; RLOCK=X
Residuals	KLOCK=S/U; RLOCK=S/U

For determining lock modes, treat cursors used for cursor-based updates/deletes as if they have residual predicates even if they don't have them.

Table 2 Lock Modes Chosen by RDS in our Method for Non-Read-Only Cursors

One more goal of our method was to avoid, in the case of *data-only locking*, the two-step process which consists of the initial S/U locking of the record by IM and the subsequent upgrading of that lock to an X lock. In the above example scenario, since we know that all qualifying key's records are going to be deleted, we would rather have one interaction with the lock manager and acquire the X lock up front. This is done by making RDS request that an X lock be acquired (rather than an S/U lock) on qualifying data, when RDS knows that all the qualifying data will be deleted (i.e., when RDS knows that there are no *residual* predicates - cursor-based updates/deletes can be treated uniformly by pretending that they have residual predicates even when they don't have them). With data-only locking, RDS must also be sure that all the predicates involve only the key columns (i.e., there are no dpreds). This is so that we do not unnecessarily make IM get an X lock on a key's record only to discover later that RM finds that the record does not satisfy some dpreds. This would reduce concurrency and we would like to avoid it since in typical DBMSs once an X lock is obtained that lock is not released or downgraded until the transaction terminates.

With *index-specific locking*, RDS should specify one mode for the key lock (KLOCK) and, if necessary, a separate mode for the record lock (RLOCK). The latter would be necessary if the record needs to be accessed by RM after the index lookup call, but before returning to RDS, to evaluate some dpreds or to fetch some columns. If there are no residual predicates, but there are some dpreds, and all retrieved records would be updated/deleted, then KLOCK should be asked for in S/U mode and RLOCK in X mode. If there are no dpreds, then KLOCK should also be asked for in X. If there are residual predicates, then both locks should be asked for in S mode for a read-only (non-updateable) cursor. For an updateable cursor, both locks should be asked for in U. Table 2 summarizes the locks used by our method.

To make the previously given steps work correctly when dpreds are present, we add to the ISCCB a mode field (*lock mode*) which keeps track of the mode in which the returned key has been locked. In the case of index-specific locking, that information will be examined by IM during key deletes to make sure that the right lock is held on the to-be-deleted key or is acquired if necessary before performing the key deletion. Then IM will update the mode information in ISCCB to reflect the X lock that it acquires on the next key.

Traditional DBMSs that do index-specific locking, after accessing an index and retrieving a RID, lock the corresponding record *before* accessing the record and applying any dpreds. In our method, if any dpreds exist, then RM does any necessary locking of the record *after* evaluating the predicate under a latch on the data page. This is done to minimize the duration and/or exclusivity (i.e., S lock rather than X) of locking in the case where the record does *not* qualify.

This is a better strategy since most of the time most of the data is in the committed state. Hence, it is very likely that when a lock is requested after predicate evaluation it will be granted. With our strategy, when RLOCK is X, if a record does not satisfy the dpreds, then, to verify that the record is in the committed state and that it would not change later on, acquiring an S lock would be sufficient, rather than the more restrictive X lock that would have been acquired if the record had qualified.

If, instead of serializability, the isolation level of cursor stability (degree 2 of System R) had been chosen, then, in the non-qualifying case, we may be able to completely avoid any locking by using the Commit_LSN technique of [Moha90b]. Even if Commit_LSN does not work, in the non-qualifying case, an instant duration lock, which involves only a single interaction with the lock manager, would most likely be sufficient than a medium duration lock which involves two interactions with the lock manager. When locking is needed while holding the page latch, we of course need to be prepared for the rare cases where after predicate evaluation we discover that the lock request is encountering a conflict and hence cannot be granted immediately. In such cases, to avoid deadlocks involving latches, we need to release the latch, wait for the lock and, after obtaining the bck, relatch the page and check if the record's state has changed. To try to avoid the reevaluation of the dpreds, we can cache the page_LSN value of the data page in the CCB before unlatching the page. When the page is relatched after obtaining the lock, dpreds would need to be reevaluated only if the current page_LSN is greater than the cached page_LSN. This works since the page_LSN would have been increased in value if any change had been made to the page between the times of unlatching and relatching.

With *data-only locking*, RM would also examine the lock mode field in ISCCB to ensure that the X lock is already held or is acquired before a record is deleted.

To summarize, using our method, the following will be the costs involved in performing a set-oriented deletion (like the one given in the example before):

- **Data-only locking**: N+1 lock calls involving N+1 locks

  1 X-lock call on a record during the open call,

  N X-lock calls on next keys' records during key deletions.
- **Index-specific locking**: 2N+1 lock calls involving 2N+1 locks

  1 X-lock call on the first key during the open call,

  N X-lock calls on records during record deletions,

  N X-lock calls on next keys during key deletions.
- 1 root-to-leaf traversal during the open call

The reader may wish to compare the above numbers with those given in section 2 for the traditional method's way of doing such a set-oriented deletion.

4. Extensions

Even though we started out with set-oriented deletes, our method also applies, without changes, when the user specifies cursor-based deletes. As far as the data manager is concerned, a cursor-based delete is no different from a set-oriented delete with some residual predicates.

The techniques presented here are also directly applicable to set-oriented updates where the qualifying records are being determined using an index scan and that index's key is being updated. Even though for a long time RDBMSs were avoiding using such an index to prevent the so-called *Halloween Problem*, Tandem's NonStop SQL started using such an index for such an update operation as long as it was guaranteed that after the update of the key value via this statement is performed the new value will no longer be within the set of key values qualifying for update (see [Tand87] for more details). When a record is being updated, only the old key can be deleted directly on the leaf page based on information in the ISCCB. The insert of the new key would most likely require traversing the tree since it is unlikely that it will reside on the same page from which the old key was deleted. The idea of RDS specification of the X locking requirement when there are no residual predicates is also applicable to other kinds of set-oriented updates since it reduces the number of interactions with the lock manager.

Although we talked about LSNs which might have been taken to imply that our ideas work only with write-ahead logging [MHLPS92], they are also applicable with other recovery methods like shadow-paging, as long as a version number field exists in every database page and its value is incremented on every update to the page. The latter is exactly what AS/400, Informix, SQL/DS and System R do for index pages [Moha90a].

5. Conclusions

In this paper, we addressed a real-life, customer-encountered problem which arose in the context of set-oriented deletes based on an index scan with only index predicates (ipreds) and an index manager performing data-only locking. First, we proposed a method that minimizes the number of lock calls and the number of root-to-leaf tree traversals for situations like that customer scenario. Then, we generalized our method to deal with cursor-based and set-oriented deletes/updates where data predicates (dpreds) and residual (RDS-applied) predicates are also present. The extensions also dealt with index bcking algorithms that do index-specific locking. We showed how the query processing component (RDS) by becoming more aware of locking implications could improve performance. Increased awareness by RDS designers of the processing that goes on in the data manager is also important for the optimizer's cost model to reflect reality correctly. Otherwise, the optimizer will wind up making non-optimal decisions. More accurate modeling of the actual data manager processing is even more important now since, compared to System R, today's DBMSs' optimizers model different costs in much finer detail (e.g., see [CHHIM91] for samples of some of DB2 optimizer's cost equations).

Another aspect of modeling query execution costs is that even today, starting from System R days, only the costs associated with retrievals have been modeled. That is, even for update and delete statements, only the costs of identifying the records to be updated or deleted have been modeled to choose between different available access paths for doing the selection. The consequence is that the optimizer will not know the difference in delete costs between using the index as we have done before for the example scenario (i.e., immediate access of record after an index lookup) and another approach where we delay accessing the data pages until all the qualifying RIDs have been determined via a range scan. The latter is similar to what is done with index ANDing/ORing. The idea there is to sort the RIDs before data page accesses to convert an unclustered index scan into a clustered scan (see [MHWC90]). In this case, as each record is deleted, unless something special is done, every key delete will cause a root-to-leaf traversal even on the index used to choose the records for deletion. With our method, the first approach was able to avoid such traversals. If the optimizer does not model the costs of doing the deletions in the different approaches, then it might make the wrong choice during access path selection.

Our patented method [Moha99b] has been implemented in DB2 V7 to address specific customer requirements. It has also been exploited to improve performance on the TPC-H benchmark [PoFl00, TPC99]. We would like to acknowledge the work of Quanhua Hong who implemented our method in DB2.

6. References

[Anto93] Antoshenkov, G. *Dynamic Query Optimization in Rdb/VMS*, **Proc. International Conference on Data Engineering**, Vienna, April 1993.

[Anto96] Antoshenkov, G. *Dynamic Optimization of Index Scans Restricted by Booleans*, **Proc. International Conference on Data Engineering**, New Orleans, February 1996.

[BaMc72] Bayer, R., McCreighton, E. *Organization and Maintenance of Large Ordered Indices*, **Acta Informatica**, Volume 1, Number 3, 1972.

[CEHH90] Crus, R., Engles, R., Haderle, D., Herron, H. *Method for Referential Constraint Enforcement in a Database Management System*, **U.S. Patent 4,947,320**, IBM, August 1990.

[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.

[EGLT76] Eswaran, K.P., Gray, J., Lorie, R., Traiger, I. *The Notion of Consistency and Predicate Locks in a Database System*, **Communications of the ACM**, Vol. 19, No. 11, November 1976.

[GLSW93] Gassner, P., Lohman, G., Schiefer, B., Wang, Y. *Query Optimization in the IBM DB2 Family*, **Data Engineering**, Volume 16, Number 4, December 1993.

[HaWa90] Haderle, D., Watts, J. *Method for Enforcing Referential Constraints in a Database Management System*, **U.S. Patent 4,933,848**, IBM, June 1990.

[Lome93] Lomet, D. *Key Range Locking Strategies for Improved Concurrency*, **Proc. 19th International Conference on Very Large Data Bases**, Dublin, August 1993.

[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.

[MHWC90] Mohan, C., Haderle, D., Wang, Y., Cheng, J. *Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques*, **Proc. International Conference on Extending Data Base Technology**, Venice, March 1990. An expanded version of this paper is available as **IBM Research Report RJ7341**, IBM Almaden Research Center, March 1990; Revised May 1990.

[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.

[Mohan90] 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.

[Moha92] Mohan, C. *Interactions Between Query Optimization and Concurrency Control*, **Proc. 2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing**, Tempe, February 1992. Also available as IBM Research Report RJ8681, IBM Almaden Research Center, March 1992.

[Moha95] Mohan, C. *Concurrency Control and Recovery Methods for B^+-Tree Indexes: ARIES/KVL and ARIES/IM*, In **Performance of Concurrency Control Mechanisms in Centralized Database Systems**, V. Kumar (Ed.), Prentice Hall, 1995.

[Moha99a] Mohan, C. *Repeating History Beyond ARIES*, **Proc. 25th International Conference on Very Large Data Bases**, Edinburgh, September 1999.

[Moha99b] Mohan, C. *System and Method for Performing Record Deletions Using Index Scans*, **United States Patent 6,009,425**, IBM, December 1999.

[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.

[PoFl00] Poess, M., Floyd, C. *New TPC Benchmarks for Decision Support and Web Commerce*, **ACM SIGMOD Record**, Volume 29, Number 4, December 2000.

[Tand87] The Tandem Database Group *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL*, **Proc. 2nd International Workshop on High Performance Transaction Systems**, Asilomar, September 1987.

[TPC99] **TPC Benchmark H (Decision Support) Standard Specification**, Revision 1.3.0, 1999.

[WeVo01] Weikum, G., Vossen, G. **Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery,** Morgan Kaufmann, 2001.