Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems

C. MOHAN

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

Abstract This paper presents a novel and simple method, called ***Commit_LSN***, for determining if a piece of data is in the committed state in a transaction processing system. This method is a much cheaper alternative to the locking approach used by the prior art for this purpose. The method takes advantage of the concept of a log sequence number (LSN). In many systems, an LSN is recorded in each page of the data base to relate the state of the page to the log of update actions for that page. Our method uses information about the LSN of the first log record (call it *Commit_LSN*) of the *oldest* update transaction still executing in the system to infer that all the updates in pages with page_LSN *less than* Commit_LSN have been committed. This reduces locking and latching. In addition, the method may also increase the level of concurrency that could be supported. The Commit_LSN method makes it possible to use fine-granularity locking without unduly penalizing transactions which read numerous records. It also benefits update transactions by reducing the cost of fine-granularity locking when contention is not present for data on a page. We discuss in detail many applications of this method and illustrate its potential benefits for various environments. In order to apply the Commit_LSN method, extensions are also proposed for those systems in which (1) LSNs are not associated with pages (AS/400,[1] SQL/DS, System R), (2) LSNs are used only partially (IMS), and/or (3) not all objects' changes are logged (AS/400, SQL/DS, System R).

1. Introduction

In many cases of data base interactions, the sole purpose of locking is to ensure that a given piece of data that is about to be read is in the committed state. In those cases, locking is not really being done to delay *future* updates. This is the case with transactions that run with the Cursor Stability (CS) level of isolation (i.e., *consistency level 2* of System R [Gray78] - see the section "1.2. Latches and Locks") when they do not have the intention to update the data being read. Typically, ad hoc queries that examine large volumes of data and that generally do not perform any updates of the examined data are run at this level of isolation [PMCLS90]. For transactions that run with the Repeatable Read (RR) level of isolation (i.e., *consistency level 3* of System R - see the section "1.2. Latches and Locks") also, there are times when the data base management system (DBMS) does some reads using CS. This happens, for example, in DB2[1] during referential constraint checking. Avoiding the locking and unlocking interactions with the lock manager saves not only pathlength, but it may also increase concurrency, depending on the granularity of locking. In one system, when there are no conflicts, acquiring a lock and releasing it costs about 800 instructions.

Fine-granularity (e.g., record) locking is very helpful in increasing the level of concurrency that can be supported by reducing contention amongst transactions for access to data. The ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*), ARIES/NT (*Nested Transactions*), ARIES/KVL (*Key-Value Locking*), ARIES/LHS (*Linear Hashing with Separators*) and ARIES/IM (*Index Management*) methods presented in [MHLPS89, Moha90a, Moha90b, MoLe89, MoPi90, RoMo89] are examples of methods which support high concurrency. ARIES has been implemented, to varying degrees, in the IBM products OS/2 Extended Edition Database Manager[1] [ChMy88] and DB2 V2R1, in the IBM Research prototypes Starburst [SCFLM86] and QuickSilver [HMSC88], and in the University of Wisconsin's Gamma data base machine [DGSBH90]. The drawback of fine-granularity locking is that for those transactions that access large number of records, the number of locks that need to be acquired may increase dramatically compared to the situation with, for example, page locking. If, for those transactions which only need to determine that some piece of data is in the committed state, the system could somehow avoid locking, then we can have the benefits of fine-granularity locking for transactions which access few records and at the same time avoid the drawbacks of such a locking granularity for transactions which access numerous records for reading. A method for avoiding locking is expected to be useful very often since in most data bases, at any given time, most of the data is in the committed state.

1.1. Overview of the Method

In most transaction systems that use write-ahead logging (WAL) for their recovery, updates to pages get logged and every page's header has a field called **page_LSN** which contains what is called the log sequence number (**LSN**). The LSN is a monotonically increasing number which is typically the logical address of the log record

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/ or special permission from the Endowment.

Proceedings of the 16th VLDB Conference
Brisbane, Australia, August 1990

describing the *most recent* update to the page. Existing systems (e.g., IBM's DB2 [Crus84], IMS [GaKi85, PeSt83, Yama83] and the OS/2 Extended Edition Database Manager, and Tandem's NonStop SQL[1] [Tand87]) have used the LSN concept only to accomplish the recovery of the data after a system failure, to guarantee the transaction atomicity and persistence properties [HaRe83].

The crux of the Commit_LSN method is to use this LSN information and information about the currently active update transactions to come to some conclusions about whether or not all the data on a given page is in the committed state, without resorting to locking. This is done by comparing the page's LSN with the information about the *oldest* update transaction still executing in the system. The crucial fact that makes our method accomplish its objectives is that no page with an LSN value that is *less than* the LSN of the Begin_Transaction log record of the *oldest* executing update transaction could have any uncommitted data. The Commit_LSN method applies whether the lowest granularity of locking is a page or something finer than that (e.g., record).

We have extended the Commit_LSN method to use it in those transaction systems which do not necessarily log changes to all the pages of the data base (e.g., as in AS/400[1] [ClCo89, DHLPR89], Informix-Turbo[1] [Curt88], SQL/DS [ChGY81], and System R [CABGK81]) and which do not necessarily store in the page header the LSN value (e.g., AS/400, IMS, SQL/DS, System R). In such systems, we can make a change so that the systems keep track of the *youngest* transaction to have updated each page and use this information in the same fashion the LSN information is used by our method in the other systems to reduce or avoid locking.

The rest of this section introduces some concepts and terminologies relating to locking, latching, logging, and recovery. Section 2 discusses the goals that we had in mind when we designed the Commit_LSN method. The method is presented in detail in section 3, while the implementations of the method in different systems are discussed in section 4. Section 5 discusses optimizations to improve the usefulness of the Commit_LSN concept in the presence of long-running update transactions. Numerous applications of the Commit_LSN method are outlined in section 6. Section 7 discusses the impact of the *shared disks* (*data sharing*) [MoNP90, ReSW89] and the *shared nothing* (*partitioned*) [DGSBH90, Shoe86] environments on the extent of applicability of the method. Finally, section 8 presents a summary of our work.

1.2. Latches and Locks

Normally latches and locks are used to control access to shared information. Locking has been discussed to a great extent in the literature. Latches, on the other hand, have not been discussed that much. ***Latches*** are like semaphores. Usually, latches are used to guarantee physical consistency of data, while ***locks*** are used to assure logical consistency of data. Latches are usually held for a much shorter period of time than are locks. Also, the deadlock detector is not informed about latch waits. Latches are requested in such a manner so as to avoid deadlocks involving latches alone, or involving latches and locks.

Acquiring a latch is cheaper than acquiring a lock (in the no-conflict case, 10s of instructions versus 100s of instructions), because the latch control information is always in virtual memory in a fixed place, and direct addressability to the latch information is possible given the latch name. On the other hand, storage for individual *locks* may have to be acquired, formatted, and released dynamically, and more instructions need to be executed to acquire and release locks. This is because, in most systems, the number of lockable objects is many orders of magnitude greater than the number of latchable objects.

Locks may be obtained in different ***modes*** such as S (Shared), X (eXclusive), IX (Intention eXclusive), IS (Intention Shared), and SIX (Shared Intention eXclusive), and at different ***granularities*** such as record (tuple), table (relation), file (tablespace, segment, dbspace) [Gray78]. The S and X locks are the most common ones. S provides the read privilege and X provides the read and write privileges. Locks on a given object can be held simultaneously by different transactions only if those locks' modes are ***compatible***. The compatibility relationships amongst the different modes of locking are shown in Figure 1. A check mark ('√') indicates that the corresponding modes are compatible.

	S	X	IS	IX	SIX
S	√		√		
X					
IS	√		√	√	√
IX			√	√	
SIX			√		

Figure 1: Lock Mode Compatibility Matrix

With ***hierarchical locking***, the intention locks (IX, IS, and SIX) are generally obtained on the higher levels of the hierarchy (e.g., table), and the S and X locks are obtained on the lower levels (e.g., record). The nonintention mode locks (S or X), when obtained on an object at a certain level of the hierarchy, *implicitly* grant locks of the corresponding mode on the lower level objects of that higher level object. The intention mode locks, on the other hand, only give the privilege of requesting the corresponding intention or nonintention mode locks on the lower level objects (e.g., SIX on a table *implicitly* grants S on all the records of that table, and it allows X to be requested *explicitly* on the records). For more details, the reader is referred to [Gray78].

Lock requests may be made with the *conditional* or the *unconditional* option. A ***conditional*** request means that

[1] AS/400, DB2, IBM and OS/2 are trademarks of the International Business Machines Corp. NonStop SQL and Tandem are trademarks of Tandem Computers, Inc. DEC, VAX DBMS, VAX, VAXcluster and Rdb VMS are trademarks of Digital Equipment Corp. DBC/1012 is a trademark of Teradata Corp. Informix is a registered trademark of Informix Software, Inc.

the requestor is not willing to wait if the lock is not grantable immediately at the time the request is processed. An ***unconditional*** request means that the requestor is willing to wait until the lock becomes grantable. Locks may be held for different *durations*. An unconditional request for an ***instant duration*** lock means that the lock is not to be actually granted, but the lock manager has to delay returning the lock call with the *success* status until the lock becomes grantable. ***Manual duration*** locks are released some time after they are acquired and, typically, long before transaction termination. ***Commit duration*** locks are released only at the time of termination of the transaction, i.e., after commit or abort is completed. The above discussions concerning conditional requests, S and X modes, and durations, except for commit duration, apply to latches also.

Transactions may request different ***levels of isolation*** (or ***consistency***) with respect to each other. SQL/DS, the OS/2 Extended Edition Database Manager, DB2 and NonStop SQL support the isolation levels ***cursor stability*** (*consistency level 2* of System R [Gray78]) and ***repeatable read*** (*level 3* of System R). They are referred to as ***CS*** and ***RR***, respectively. Both return only committed data to the transactions, unless the accessed data is uncommitted data belonging to the accessing transaction. When the chosen level is CS, as long as an *updateable* SQL cursor is positioned on a record, a lock will continue to be held on the record and the record will be guaranteed to exist in the data base, unless the current transaction itself deletes the record after the cursor is positioned on it. As soon as the cursor is moved to a different record, the lock may be released on the previous record.

With RR, locks are held on all the accessed data until the end of the transaction. Actually, locks are somehow held even on nonexistent data, which could have satisfied the query. In [Moha90a, MoLe89], we discuss how this is done when the accesses are made via indexes. With RR, if a certain query were to be posed at a certain point in a transaction, and a little later the same query were to be posed within the same transaction, then the response to the query would be the same, even if it were a negative response like *not found*, unless the *same* transaction had changed the data base to cause a difference to be introduced in the responses. If all the transactions are run with RR, then their concurrent executions would be ***serializable*** in the sense of [EGLT76]. That is, the concurrent execution would be equivalent to some *serial* execution of those transactions. With CS, only the locks on data modified by the transaction are held for commit duration and so, repeating a query *may* give a different response due to other concurrent transactions' intervening activities. CS supports higher concurrency than RR since the S locks are held for a shorter time with CS. Typically, users posing ad hoc queries for decision support run their transactions with CS to reduce the harmful interactions with the transactions which are supporting production applications [PMCLS90]. The intention here is to read only committed data, but not to prevent future updates of the read data by other transactions before the reading transaction terminates.

In order to avoid starvation, typically lock managers process lock requests in the first-in-first-out (FIFO) discipline. As a result, sometimes a transaction may have to wait, even though its requested lock mode is compatible with the currently held mode, just because there is already a waiting request whose mode is incompatible with the held mode. While, from an overall system point of view this is a desirable wait, from the individual new request's viewpoint it is an undesirable wait. It would be desirable, at least under some circumstances, to avoid this waiting as long as it does not cause starvation.

1.3. Logging and Recovery

To meet transaction and data recovery guarantees, the transaction processing system records in a ***log*** the progress of a transaction, and its actions which cause changes to recoverable data objects. The log becomes the source for ensuring either that the transaction's committed actions are reflected in the data base in spite of various types of failures, or that its uncommitted actions are undone (i.e., rolled back). The log can be thought of logically as an ever growing sequential file.

Each log record is assigned, by the log manager, a unique ***log sequence number (LSN)*** at the time the record is written to the log. The LSNs are assigned in ascending sequence. Typically, they are the logical addresses of the corresponding log records [Crus84]. At times, version numbers or timestamps are also used as LSNs [MoNP90, ReSW89, Yama83]. On finishing the logging of an update to a page, in many systems (e.g., in DB2, Starburst, QuickSilver, the OS/2 Extended Edition Database Manager and NonStop SQL), the LSN of the log record corresponding to the *latest* update to the page is placed in a field (***page_LSN***) in the page header. Hence, knowing the LSN of a page allows the system to correlate the state of the page with respect to those logged updates relating to that page. That is, at the time of recovery, the page LSN and the log record's LSN can be compared to determine unambiguously whether or not that log record's update is already reflected in that page [MHLPS89].

The nonvolatile version of the log is stored on what is generally called ***stable storage*** (e.g., disk). This storage remains intact and available across system failures. Whenever log records are written, they are placed first only in the *volatile* storage (i.e., virtual storage) buffers of the log file. Only at certain times (e.g., at *prepare* time during the execution of the two-phase commit protocol - see [MoLi83, MoLO86]) are the log records up to a certain LSN written, in log page sequence, to stable storage. This is called ***forcing*** the log up to that LSN.

There are two general approaches to recovery: the ***write-ahead logging (WAL)*** approach [Gray78, MHLPS89] and the ***shadow-page technique*** [GMBLL81, MHLPS89]. WAL is the recovery method of choice in most systems, even though the shadow-page technique of System R or a variation of it is used in systems like SQL/DS and Informix-Turbo. In WAL systems, an updated page is written back to the same nonvolatile storage location from which it was read. The ***WAL protocol*** asserts that the log records representing changes to some data must already be on stable storage **before** the changed data is allowed to replace the previous version of that data on nonvolatile

storage. The buffer manager uses the LSN associated with the page to ensure that the log has been forced up to that LSN before it writes the modified page to non-volatile storage. In System R's implementation of the shadow-page technique, periodically an *action-consistent* checkpoint is taken by quiescing all activity in the data manager. This state of the data base (the *shadow version*) is preserved on disk until the next checkpoint by creating new copies of any pages which get modified in the interim (the *current version*). Should a system failure occur, restart recovery always happens from the most recent (internally consistent) *shadow* version of the data base. Even with this technique, logging of updates is performed.

Generally, each log record describes the update performed on only a single page. 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 ***undo-only log record***, respectively.

2. Goals

Our goals in doing this work were:

- Provide a more efficient, in terms of number of computer instructions executed, method for determining that a piece of data is in the committed state, without using locking.
- Improve the level of concurrent execution of transactions that could be supported by avoiding obtaining locks which may cause unnecessary lock waits and unnecessary interferences amongst transactions.
- Support fine-granularity locking in such a way that it does not become too expensive for transactions which access large quantities of data for reading and for transactions which update the data when there isn't contention on the affected pages.
- Make use of the fact that in some systems pages of the data base have *tags* associated with them for relating the page state to a logged history of updates in order to infer some additional facts about the data on the pages and thereby reduce locking.
- Make use of the logging that is done in transaction systems for recovery purposes to achieve the goal of reducing locking even where those systems are not presently *tagging* every page with the state information which relates the page state to logged information.

3. The Commit_LSN Method

In this section, we describe in detail the commit_LSN method and how it is used. In the next section, we describe the implementation of the method in systems which have adopted different approaches to logging and recovery.

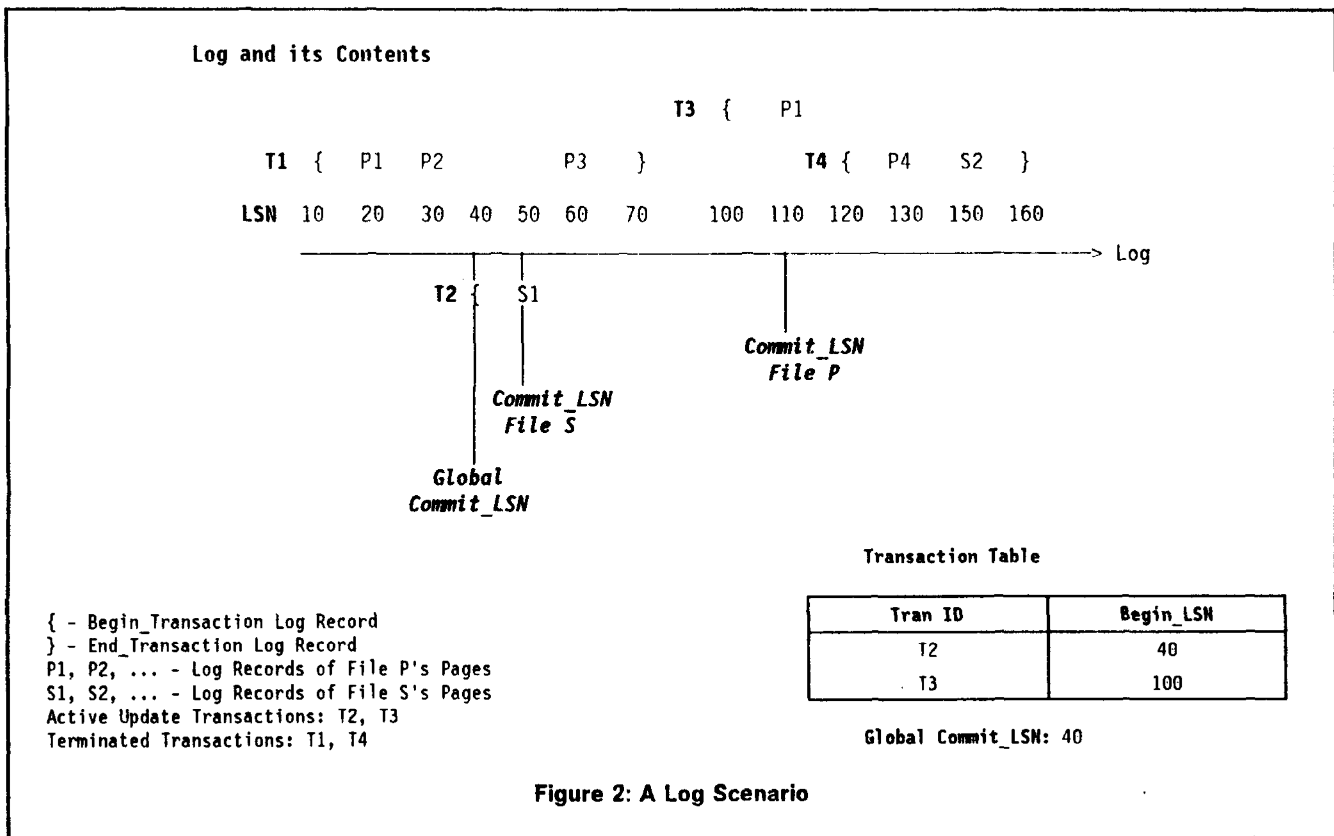
All transactions are assumed to get at least an intention lock (e.g., in the IS mode) on the higher level object (e.g., the table). The objective is only to reduce the locking at the lower levels (e.g., pages or records) of the object. Even when page is the smallest granularity of locking, an exclusive (X) latch is used by update transactions to let other nonlocking transactions (i.e., readers of the page) know that an update is in progress. This permits, when desired, uncommitted data to be read, as in systems like AS/400, IMS, and NonStop SQL, with physical consistency of the data and the validity of pointers being guaranteed by making such transactions acquire a share (S) mode latch before examining a page. With finer than page locking, all updaters and readers are in any case required to do latching in the appropriate mode before examining the page [MHLPS89].

An update transaction writes a ***Begin_Transaction*** log record just before performing its first update. Read-only transactions do not do any logging. The LSN of the Begin_Transaction log record is termed the ***Begin_LSN*** of the transaction. Typically, a component of the system called the recovery manager or the transaction manager maintains a transaction table which contains one entry for each active transaction. In the entry associated with an update transaction, the Begin_LSN value will also be stored. With the Commit_LSN method, the functionality of the recovery manager is extended so that it may be queried at any time to obtain *Commit_LSN*, which is the minimum of the Begin_LSNs of all the currently active update transactions. If there is no update transaction in the system at that point in time, then the current end-of-log LSN (***EOL_LSN***) is returned by the recovery manager as the value of Commit_LSN.

With respect to the maintenance of Commit_LSN, one possibility is to keep it in shared storage and let the recovery manager modify it using Compare & Double Swap (S/370 instruction) type logic, whenever the termination of an update transaction causes Commit_LSN to change. In this case, the transactions which need Commit_LSN can obtain it directly from this shared storage. The steps involved in maintaining this ***(Global) Commit_LSN*** are:

1. When the system restarts after a failure, initialize Commit_LSN to EOL_LSN, if no transactions are still active at the *end* of restart recovery; else, initialize it to Minimum(Begin_LSNs of Active Transactions).
2. During normal system operation, if a transaction is terminating (after *completing* commit or abort) and the terminating transaction's Begin_LSN < > Commit_LSN then leave the value as it is. Otherwise, compute the new value of Commit_LSN: If no other update transaction is active, then set the new value of Commit_LSN to EOL_LSN; else set it to Minimum(Begin_LSNs of Active Transactions).

Figure 2 shows, as an example, the state of a log as of a certain time. It shows the log records written by different transactions and the LSNs of those log records.



Tran ID	Begin_LSN
T2	40
T3	100

Figure 2: A Log Scenario

For each update log record, it also shows the file name and page number within the file whose update is recorded in that log record. For example, P2 refers to an update to page 2 of file P. The figure also shows the contents of the transaction table and the value of the (global) commit_LSN field as of the time of the log state depicted in that figure.

If Commit_LSN is not needed very often, then it may be more cost effective to compute it only when it is needed. Various possibilities exist for implementing the continuous tracking of Commit_LSN. The recovery manager could use a priority queue to keep track of the Begin_LSNs of the currently active update transactions. Alternatively, if it is expensive to maintain it continuously, then the recovery manager could compute Commit_LSN at regular intervals based on elapsed time, number of terminations of update transactions, amount of logging, etc.

Traditionally, locks are used to determine whether a piece of data that needs to be accessed by a transaction is currently in the committed state or not. The main idea behind the Commit_LSN method is the use of Commit_LSN to determine, without locking, that some data is in fact in the committed state as of that time. The interpretation of Commit_LSN is that *at least* all updates logged *prior* to that point in the log have already been committed. It is possible that some additional updates logged *after* the Commit_LSN have also been committed. It will cost the transaction more to find out which of those later updates (i.e., updates with LSN >= Commit_LSN) have been committed.

The Commit_LSN method's steps at the time of a page access are:

1. Find out Commit_LSN from the recovery manager or access it in shared storage.

   Note that it is not *necessary* for the transaction to obtain the latest value of Commit_LSN before every page access, as long as it is done at least once before the first page access. While an out of date Commit_LSN does not cause any inconsistencies, it may increase the number of times locks have to be obtained.

2. Latch the page in share (S) mode.

3. If page_LSN < Commit_LSN, then conclude that all data on the page is in the committed state; otherwise, do locking as usual and determine whether data of interest is committed or not.

4. Implementation in Different Systems

The Commit_LSN method is directly implementable in systems like DB2, Starburst, QuickSilver, the OS/2 Extended Edition Database Manager, the Gamma data base machine and NonStop SQL since those systems use the

WAL protocol and they store LSNs in the nonvolatile storage versions of pages also.

One problem with its implementation in systems like IMS is that such systems do not acquire latches on pages.[2] This can be easily taken care of by introducing latching. But the major problem with applying the Commit_LSN method to systems like IMS relates to the maintenance of LSNs on pages. Only for a dirty page in the buffer pool, IMS Full Function maintains in the page's buffer control block the LSN of the latest update log record. When the dirty page is written to nonvolatile storage, the page_LSN is not written with it, but is discarded. Changing this would be a major effort since it would require changes to page formats and the reorganization of existing data. The problem with discarding the page_LSN is that if a page with uncommitted data were to be replaced in the buffer pool (due to IMS Full Function's *steal* policy) and later on the same page were to be read in, the page_LSN information would be lost and the fact that the page contains uncommitted data could not be inferred without locking. What all these mean is that, if the buffer manager knows that a page in the buffer pool is dirty, then the page_LSN that the buffer manager is forced to track for enforcing the WAL protocol could be utilized and benefit could be derived from the Commit_LSN method. If a page is not dirty (i.e., buffer version = nonvolatile storage version), then the system cannot tell whether it contains any uncommitted updates or not. The Commit_LSN method's approach to handling such situations is to make systems like IMS Full Function associate with a page that is read from disk either (1) the current end-of-log LSN or (2) the maximum of the LSNs of all the dirty pages so far written to disk for the object (e.g., file) containing that page. This LSN can then be treated *conservatively* as the page_LSN. It is really an upper bound for the true page_LSN.[3] The page_LSN remains at this value until the page is dirtied. Having done this, the rest of the Commit_LSN method may be used as before in systems like IMS also.

Although a system like the AS/400 also uses the WAL protocol, unfortunately, in that system, even when dirty data is in virtual storage, the page_LSN is not maintained. That system does not have a buffer pool per se; because of its single-level storage concept (similar to that of the *801* [ChMe88]), the paging subsystem is the buffer manager. WAL is ensured by making an updated page ineligible for being paged out until the log record written for that update is forced to stable storage. If the AS/400's paging is modified along the lines of what was discussed earlier for IMS Full Function, then the Commit_LSN method can be applied.

The major problem in applying the Commit_LSN method to systems like SQL/DS and System R is that such systems use the shadow-paging technique for recovery [GMBLL81] and they do not maintain LSNs for pages in the buffer pool or on nonvolatile storage, although they log changes to the data and the log manager has the notion of an LSN. Additionally, no logging is done of changes to index pages and of updates to any of the pages (index or data) during transaction rollback (i.e., no *compensation log records (CLRs)* are written - see [MHLPS89]). Informix-Turbo and the AS/400 also do not log index changes.

In order to exploit the Commit_LSN method, the above systems should be modified to store in every page the Begin_LSN of the *youngest* transaction to update the page, *independent of whether logging is done for that update or not*. That is, whenever a transaction makes a change to a page, the transaction compares its own Begin_LSN with the LSN value currently stored on the page. The *higher* of the two values is now stored on the page. In the case of indexes for which no logging is done, the above applies for key inserts and key deletes. For structure modification operations (SMOs - e.g., page splits and page deletes), which cause keys or key-range responsibility to be moved from one leaf to another, the page into which keys or additional key-range responsibility is shifted is assigned a new LSN which is Maximum(Begin_LSN of current transaction, Current Page_LSN of page from which keys or key-range responsibility is shifted, Current_Page LSN of page being modified). This change in the procedure for determining the new page_LSN is to account for the fact that the SMO might cause some uncommitted data of one transaction to be moved from one page to another by a different transaction.

5. Object-Specific Commit_LSNs

Instead of having one **global** Commit_LSN that covers all objects, transactions can benefit further by computing an ***object-specific Commit_LSN*** that is specific to the object to be accessed. In this way, a long-running update transaction that accesses some other objects and keeps the global Commit_LSN quite a bit in the past will not unduly restrict the applicability of the Commit_LSN method to the object of interest. The steps involved in computing this object-specific (e.g., file specific) Commit_LSN are:

1. After the transaction locks (e.g., in IS mode) the object (e.g., the file) to be accessed, and just before it accesses the object, it notes the current EOL_LSN. Call it ***Old EOL_LSN***.

2. It queries the lock manager to ascertain the identities of the other transactions that have update-type locks (e.g., IX and SIX mode locks) on that object. Call this set of transactions the ***Updater_List***.

   Note that we are not concerned about the transactions that have read-type locks (e.g., IS and S) on that object even though some of those may be up-

2 For this reason, when unlocked reads (*consistency level 1* of System R) are performed in IMS, physical consistency of the data is not guaranteed and that may lead to abnormal termination of transactions.

3 In systems (e.g., IMS Fast Path [GaKi85]) which do not allow uncommitted data to be written to disk (*no-steal* policy [HaRe83]), the page_LSN can be set to zero, instead of the end-of-log LSN, since the page cannot contain any uncommitted data.

date transactions because of their activities against *other* objects.

3. The Updater_List is passed on to the recovery manager to find out the minimum of Begin_LSNs of those transactions.

   Note that by the time the list is passed on to the recovery manager some of those transactions may have already terminated and some other transactions might have acquired update-type locks on this object. It is to deal with the latter that we noted the EOL_LSN before requesting the Updater_List from the lock manager. We can assert that any other transactions beginning to modify the object must be writing log records (for that object) with LSNs greater than or equal to the noted EOL_LSN.

4. The recovery manager returns ***Minimum_Begin_LSN*** which is the minimum of the Begin_LSNs of all the transactions in the list that are still in existence. If none of those transactions is in existence, then it returns the value *infinity*.

5. Having obtained the Minimum_Begin_LSN from the recovery manager, the object-specific Commit_LSN is computed for the object of interest as the *minimum* of {Old EOL_LSN, Minimum_Begin_LSN}.

Another possibility for computing the object-specific Commit_LSN is to track, for each transaction and for each object that it updates, a lower bound on the LSN of the log record of the *first* update that the transaction makes to that object. This information is kept in the virtual storage descriptor associated with the object. A transaction needing the object-specific Commit_LSN can then use the *minimum* of the tracked values and the EOL_LSN as the Commit_LSN. The value computed in this fashion will be *greater than or equal to* the value derived from the previously given method which involved interacting with the lock manager and the recovery manager.

The scenario in Figure 2 illustrates the advantage of an object-specific Commit_LSN over the global Commit_LSN. When using only the global Commit_LSN value, if page P3 is accessed, then locking would be necessary since P3's LSN (60) is greater than the former (40). On the other hand, if the object-specific Commit_LSN for file P had been used, then the locking will not be necessary since the former (110) is greater than P3's LSN.

6. Applications of the Method

In this section, a number of applications of the Commit_LSN method are described. Some of them benefit all transactions, while others benefit only transactions that are performing read-only activities with CS. In this section, only the single system scenario is considered. In the next section, we consider the shared disks and the shared nothing environments.

Some of the applications of the Commit_LSN method are:

1. CS Read-Only File Scans For read-only CS *file scans*, there is no need to do locking of data while accessing data pages for which page_LSN < Commit_LSN. Latching is still needed to do this check, evaluate predicates and retrieve columns of qualifying records. To amortize the cost of latching, in one access to the page, *all* the qualifying records on that page can be retrieved. Even for *updateable* CS scans, the checking of the satisfiability of predicates can be done without locking, if the above condition is true. If a record does not qualify and the above condition is true, then no locking needs to be performed. Only after determining that a record qualifies does a manual duration lock need to be acquired in the case of updateable CS file scans.

The possibility of this optimization lets one consider keeping the granularity of locking small (e.g., at the record level) to increase concurrency for the benefit of RR transactions and CS read accesses with the update intent, while at the same time not penalizing the CS read-only accesses with the huge cost of locking and unlocking every record that is accessed. For most of the pages that the CS scan accesses, the transaction is expected to avoid locking completely, due to the Commit_LSN method.

2. CS Read-Only Index Scans For CS *index scans*, the techniques of index ANDing/ORing [MHWC90] can be combined with the Commit_LSN method to dramatically reduce the extent of locking. CS (read) index accesses for which the index contains sufficient information to respond to the query, without accessing data pages, can also behave like in the case of data page accesses mentioned above.

With CS, if an index look-up is to be followed by a data page access, and if no locking is performed during the index access, then the *same Commit_LSN value* that was used for avoiding the locking during the index access should also be used while accessing the data page for avoiding locking. If, on accessing the data page, the data page's LSN is found to be less than the Commit_LSN value passed by the index manager, then the predicates checked via the index are still guaranteed to be true and locking of the data may be avoided, if the access is a read-only one. If the former condition does not hold, then even the predicates already checked by the index manager must be rechecked. Note that in this case the record whose record ID (***RID***) was provided by the index manager may no longer exist or, if multiple table's records are allowed to be intermixed on the same page, then a record belonging to a different table might now exist with that RID. By using the *current* Commit_LSN value, it may still be possible to avoid locking.

The algorithm presented in Figure 3 may be used for CS (read-only and update) scans, if one or more indexes are to be used for identifying the qualifying records of a single table before accessing the data pages, as discussed in [MHWC90]. To simplify the pseudo-code, we have ignored the details about what happens if a lock is not grantable when it is requested. Since a page latch is held, the lock requests must be made *conditionally*. If the lock is not granted, then, to avoid deadlocks involving latches, the latch must be released and the lock requested *unconditionally*. Finally, when the lock is

```
Note Commit_LSN BEFORE starting the scan of the first index and call it Begin_Commit_LSN
Don't do any locking while accessing the indexes
Generate the list of RIDs satisfying the different predicates checked
    via the different indexes and sort the RIDs in ascending order
Start accessing the records whose RIDs are in the list
For each record in the list do the following:
  Latch record's page
  IF page_LSN < Begin_Commit_LSN THEN     /* predicates checked before are still valid and */
                                          /* all data on page is in the committed state    */
    check ANY REMAINING predicates        /* only predicates NOT checked during index accesses */
    IF predicates are satisfied THEN
      IF updateable scan THEN
        lock data for manual duration
      unlatch page and return data
    ELSE ignore record and unlatch page   /* predicates not satisfied */
  ELSE                                    /* predicates checked via index need rechecking */
    IF page_LSN < CURRENT Commit_LSN THEN /* all data in page in committed state */
      check ALL the predicates
      IF predicates are satisfied THEN
        IF updateable scan THEN
          lock data for manual duration
        unlatch page and return data
      ELSE ignore record and unlatch page /* predicates not satisfied */
    ELSE                                  /* data on page may not be in the committed state */
      check ALL the predicates
      IF predicates are satisfied THEN
        IF updateable scan THEN
          lock data for manual duration, unlatch page and return data
        ELSE lock data for instant duration, unlatch page and return data
      ELSE                                /* predicates not satisfied   */
        lock data for instant duration    /* to confirm committed state */
        ignore record and unlatch page
```

Figure 3: Combining Commit_LSN Method With Index AND/ORing Technique

granted all the predicates already evaluated must be reevaluated, if the page_LSN had changed during the time the page latch was not held.

3. Referential Constraint Enforcement One method of referential constraint enforcement during a modification to the data base does the following: (1) do the modification (delete, insert or update) first; (2) then check for referential constraint violations; (3) if constraints are violated, then undo the modification (via a partial rollback). This method has been implemented in DB2 V2R1. With this implementation, since the second step does only read operations, locks need not be obtained on the data in the pages accessed during that step if (1) the Commit_LSN method's condition (page_LSN < Commit_LSN) is satisfied and (2) the referential constraint is satisfied.[4] This is acceptable even if the transaction performing the operation has requested the RR level of isolation.[5] If the constraint is violated, RR is requested, and the repeatability of the violation must be guaranteed, then a commit duration lock must be acquired.

As an example, if the operation is an insert of a child table's record and the foreign key is not *null*, then a check is made to ensure that a record whose primary key is equal to the inserted record's foreign key exists in the parent table. When the primary key index of the parent table is accessed to perform this check, the Commit_LSN method comes into play and locking can be avoided under the above conditions. In summary, the Commit_LSN method may reduce the cost of enforcing referential integrity constraints.

4. Space Reservation When record locking is used, the space released by one transaction must be protected from being consumed by another transaction, until the

4 Today, in DB2, the second step is executed with CS as the isolation level. This causes manual duration locks to be acquired and released.

5 Not holding the locks on the data accessed in step 2 until commit does not cause problems because of the following: If any other transaction tries to modify the data accessed during the second step concurrently in a way that would cause the already checked constraint to be violated, then such a transaction would be forced to access the data modified by the first transaction in step 1 in order for it to ensure that it is not violating any constraints. If the first transaction has not terminated, then the second transaction will not be able to access the latter immediately, but would be forced to wait. It is to cause this wait to occur that the modification is performed *before* the possible violation is checked.

former commits. Otherwise, some other transactions might consume the freed space and then the space-freeing transaction might choose to rollback and find it impossible to put back the data on the original page due to lack of space. The method used in Starburst and the OS/2 Extended Edition Database Manager to do space reservation is described in [LiMP86]. The basic idea behind methods like that is to make the space *releasing* operations obtain locks (e.g., in IX mode) on the page and leave "trails" on the page to let subsequent space consuming transactions know that there is potentially some uncommitted freed space on the page. The transaction attempting to perform a space *consuming* operation, on noticing the "trail", is forced to get a lock (e.g., instant duration X mode) on the page to verify that the space releasing operation has in fact committed, before it consumes the space and, *possibly*, erases the trail. Even if the lock is granted the trail should not be erased, if the current transaction itself held the IX lock.

The application of the Commit_LSN method avoids the need for the space consuming transaction that notices the existence of the trail to have to lock the page to consume the space, if the page_LSN is less than the Commit_LSN. If the latter condition holds, then the trail can also be erased. The erasure is correct only if the current transaction itself is guaranteed not to have freed up any space on the page. Note that even transactions which access the page for update operations that neither free nor use additional space can do the erasure under the above condition. It would be useful to record the erasing of the trail in the log record of the update action so that at redo time, during system restart, the trail can be reset conveniently.

5. ARIES/IM Index Algorithm The ARIES/IM index concurrency and recovery algorithms described in [MoLe89] set the value of the Delete_Bit on a leaf page to '1' when doing the deletion of a key on that page. When a key is about to be inserted on a leaf page, if that page's Delete_Bit has the value '1', then the inserter is required to acquire the structure modification (SM) latch in S mode (this *latch becomes a global lock in the SD environment*) and then reset the Delete_Bit to '0'. What this means is that the inserter has to wait until any in-progress SM (page split or page delete) *anywhere in the index tree* is completed. This is done to make sure that the tree would be structurally consistent when the following sequence of events occurs: (1) the inserting transaction consumes all the space released by the deleting transaction and commits, (2) the deleting transaction rolls back, and (3) a traversal of the index tree from the root is necessary for performing a logical undo due to the lack of space on the original page to put back the deleted key. If this happens during normal processing, the tree may be inconsistent and the deleter can wait for it to become consistent; however, at restart time, since undo for all losing transactions is performed in a single backward scan of the log (see [MHLPS89]), waiting will not be fruitful. Hence, the burden is placed on the inserter to make sure that its action will not cause trouble for the deleter. Note that if the deleter had committed by the time the inserter attempts its operation then the acquisition of the SM latch (lock in the SD environment), which may reduce concurrency and cause delays, is unnecessary.

In the original ARIES/IM algorithms, *there is no simple way for the inserter to find out whether the deleter has committed or not committed.* The idea here is that if the Delete_Bit were to be currently set at '1' and the page_LSN were to be less than the Commit_LSN, then the inserter can reset the Delete_Bit to '0' and safely do its key insert *without* having to synchronize itself with any SM, which may be on-going, using the SM latch (lock). This improves concurrency also. If the Commit_LSN condition does not hold then the original ARIES/IM method applies. With the Commit_LSN method, the system can potentially save a latch (lock) call at the time of every key insert on a page that follows a key delete involving the same page.

6. Next Key Locking in Indexes If, for a particular index, high concurrency is extremely important and *RR is not a requirement*, then during a key *insert*, the *next key locking requirement* (i.e., the instant duration *X lock* to be acquired on the next higher key currently in the index, *before* the insertion of the current key) of the methods of ARIES/IM [MoLe89], ARIES/KVL [Moha90a], and System R [Moha90a] can be dropped, as long as the index is a *nonunique index*. Under these conditions, for key *deletes* in both unique and nonunique indexes, the next key locking (commit duration X locks) must still be performed to make sure that no uncommitted deletes are "skipped" over during a scan or a look-up operation, if not skipping over is a requirement. The intent of this lock is to let other transactions know about the uncommitted delete by blocking them.

Even if RR is not required, the next key locking requirement for *inserts* has to be enforced in *unique indexes* to make sure that the key being inserted is not the same as another one which is currently in the uncommitted deleted state.[6] If the key to be inserted is higher than the highest key on the page, then the next key locking requirement potentially causes an extra I/O to read in the next page. This next key locking during inserts causes some unnecessary synchronization and delays if it so happens that there are no uncommitted deletes in the range of interest and (1) the next key had been inserted by another transaction that has not yet committed,[7] *OR* (2) the next key is locked in the S mode by another transaction.

For a unique index also, by applying the Commit_LSN method during a key *insert*, the system can avoid the next key locking and, possibly, an I/O if (1) RR is not

6 In ARIES/KVL and System R alone, the next key locking under these conditions in unique indexes can be avoided if we changed the duration of the lock obtained on the *deleted* key from instant to commit. With this change, the lock on the key to be *inserted* will not be granted if there is an uncommitted delete of the same key value by another transaction. This of course increases the number of commit duration locks and also may cause unnecessary lock collisions due to the fact that key values are hashed to generate lock names. Of course, this would be a better method if next key locking during a delete can be avoided under the condition that not skipping over uncommitted deletes is not a requirement.

necessary, (2) the key does not already exist on the page, and (3) the page_LSN is less than the Commit_LSN. The last condition guarantees that there are no uncommitted deletes (or inserts, for that matter) on the page and hence the uniqueness constraint will not be violated, even if all the currently running update transactions were to rollback. If condition 3 is not satisfied, the system can still use other methods (like the IX locking technique of ARIES/KVL) to improve concurrency.

7. Logical Deletion of Keys One way to reduce locking in indexes and thereby improve the pathlength and the concurrency, *while still guaranteeing RR*, is to do *logical* deletion of keys instead of *physical* deletion. That is, during a key delete operation, instead of removing the key from the page, it is left there with a delete_flag set to '1', as in IMS [Ober80] and in [Mino84]. Of course, this operation must be logged. If the delete were to be rolled back, then the delete_flag is reset to '0'. This also avoids the possibility of a page split during the undo of a key delete. Performing deletes in this fashion avoids the need for next key locking during deletes which is required in ARIES/IM, ARIES/KVL and System R. This saves a commit duration X lock and improves the pathlength and concurrency, and still lets the system provide RR. If the key to be deleted is the highest key on the page, then the next key locking requirement potentially causes an extra I/O to read in the next page. Even with logical deletes, next key locking must be performed during key *inserts*, as long as RR must be guaranteed. Modifications to ARIES/IM along these and other lines to improve concurrency dramatically are explored further in [MoHP90].

The major problem with this *logical-delete* approach is that, at some point *after* the commit of the key-deleting transaction, the logically deleted key must be removed to free up the space occupied by it in order to reduce the number of page splits and to improve performance during searches. Ordinarily, such garbage collections would require getting locks on the deleted keys to make sure that they are committed. The physical deletes which are performed when such locking is successful should also be logged since the redoing of updates during restart is page oriented (i.e., same page is updated during redo as during normal processing) and the system needs to be able to repeat history. Logging is required because locks are not available at restart time to determine which keys can be physically deleted and which cannot be due to uncommitted deletes by *in-doubt* and in-flight transactions. Note that readers of keys should lock logically deleted keys. Only after the locks are granted, can they ignore them. Readers should *not* leave behind S locks on logically deleted keys to guarantee RR since those keys may get garbage collected without the knowledge of the readers. The garbage collector gets, if at all, only a conditional instant duration S lock on a logically deleted key to determine that it is committed and hence that it can be physically deleted.

The cost of garbage collection of the logically deleted keys can be reduced by using the Commit_LSN method, as explained below. Readers can also benefit from the Commit_LSN method. Locking of logically deleted keys can be avoided by readers, if the Commit_LSN method helps them to deduce immediately that all the logical deletions on the page have been committed.

During any update operation[8] on an index leaf page, if the page_LSN is less than the Commit_LSN, then ALL the logically deleted keys on that page can be physically deleted without incurring any locking cost. A field in the page header can be made to keep a count of the number of logically deleted keys on the page to trigger this action. The log records for the *physical deletes* need not include the key values themselves. In contrast, in a *logical-delete* action's log record, the complete key value is needed to be able to perform the undo if the transaction were to rollback. In the physical key delete log records, it is enough to indicate the ordinal position(s), in the key sequence on the page, of the key(s) to be deleted (e.g., the 3rd and the 17th keys on the page). This record can be written as a *redo-only* log record and the *repeating history* feature of the ARIES recovery method will ensure that the log record's effects persist even if the garbage collecting transaction were to be rolled back later on. In fact, this log record need not even be written as a separate record, but its contents could be combined with those of the record logging the main update action of this transaction on this page. The garbage collection part of such a log record is never undone.

Thus, the Commit_LSN method in combination with the logical delete approach can save a commit duration X lock during a delete operation and an instant duration S lock during the garbage collection of a logically deleted key. The avoidance of the next key lock can also increase the level of concurrency that can be supported. It can also save a reader an instant duration S lock during a look-up operation that encounters a logically deleted key.

8. Data-Only Locking When using certain index management methods like ARIES/IM, sometimes it may be found that a transaction holds an X lock on a record or a page, even though the transaction has *not* updated and does not intend updating the corresponding *data page*. This happens because, in ARIES/IM, in the interest of reducing the number of locks (see [MoLe89] for more explanations), the locks on the *index keys* were made to be the same as the locks on the underlying data from which those index entries were extracted. That is, the same lock name is used to designate objects (records and keys) which are related but which reside in different pages. Given this **data-only locking** approach, in contrast to the *index-specific locking* approaches of DB2, System R and ARIES/KVL, the next key locking done during a

7 In the ARIES/KVL method [Moha90a], which guarantees RR, such unnecessary synchronization is avoided by making the inserters lock the inserted key and the next key in the IX mode, instead of the X mode; that method, while it improves concurrency, still costs the pathlength incurred to get the lock.

8 Even readers, which are already update transactions due to their activities on other data, can play the role of Good Samaritans and do garbage collection when the appropriate conditions are true.

key delete causes an extra commit duration X lock to be acquired on the underlying data, thereby potentially causing unnecessary delays to other reader transactions, if they insist on finding out whether some data page's data is committed or not committed by acquiring locks on the data. For the same reason, a reader transaction doing an index-only access on index I1 may get blocked, if it does locking. This may happen not because any data on that *index page* is currently in the uncommitted state, but because an updater may hold an X lock on the *underlying data* (data page or some row on the data page, depending on the granularity of locking). The latter may be the case either because the latter transaction updated the underlying data, or because of its next key locking on some other index or in the same index due to an update on a different page of the current index. Under these conditions, use of the Commit_LSN method during data and index page accesses allows the transactions to save locking costs, avoid unnecessary lock waits and *also increase concurrency*. The Commit_LSN method lets the transactions have the best of both: the reduced number of locks due to the data-only locking feature of ARIES/IM and a compensation for the reduction in concurrency that data-only locking causes.

Even with index-specific locking (e.g., with key-value locking as in System R and ARIES/KVL), a single lock may cover data in multiple pages. For example, a single key value lock may lock up many index entries, even though only one of them may be in the uncommitted state.

9. FIFO Lock Request Processing Because of the FIFO discipline that is normally followed in granting lock requests to avoid starvation, some additional delays may be caused if a lock is used to determine that a piece of data is in the committed state. An example situation where, due to this observation, the Commit_LSN method may help is: T1 holds an S lock on D1; T2 has requested an X lock on D1 and is forced to wait; T3 wants to check if D1 is in the committed state, hence it requests a lock on D1, and is forced to wait.

10. Overflow Records When a record is updated and it no longer fits in the original page, in systems like DB2, the OS/2 Extended Edition Database Manager, SQL/DS, and System R, the updated record is inserted on a different page. During such an operation, the record should still retain its *original* RID since the index entries contain the original RID. To accomplish this, in the original page, the data record is replaced with a *pointer record* which contains the RID (called the *overflow RID*) of the updated data record on the overflow page. While performing the insert of the updated data record on the overflow page, the above systems have to make sure that they are not reusing the RID of an uncommitted deleted record of another transaction. This check is done by acquiring an instant lock on the overflow page or RID. It should be easy to see that the Commit_LSN method could be used to possibly avoid getting this lock. Note that since all accesses to the overflow record happen only through the original RID, the lock on the original RID is sufficient to indicate that the updated record is in the uncommitted state.

10. CS Updateable Scans As far as data qualifying for *updateable* CS scans are concerned, the Commit_LSN method may be used to delay or avoid locking the data, if the current definition of CS (see the section "1.2. Latches and Locks") is relaxed. When an attempt is made to do an update or a delete on the current scan position (using the *UPDATE* or *DELETE WHERE CURRENT OF CURSOR* SQL statement), it must be acceptable for the system to respond that the record under the cursor either no longer exists or that it no longer satisfies the predicates of the scan, if, due to the fact that no lock was acquired when the cursor was positioned during the read, some other transaction's activity had caused the original state of affairs to change. If the state is still the same as determined either by noticing that the page_LSN has not changed since the scan was positioned on the record or, if the page_LSN has changed, by reevaluating the predicates, then an attempt is made to acquire the X lock on the data and perform the update or delete. Note that even with the current definition of CS, the update or delete operation may fail due to a deadlock. So, the mere fact that an S lock is acquired currently in most systems when an updateable scan is positioned on a record and is held when the user examines the record, does not guarantee to the user that he would definitely be able to modify the record, if he desires to do so. A deadlock may cause the user's transaction to be rolled back. If the Commit_LSN method is used and a lock is not acquired at the time of positioning the scan, then it would be like an optimistic approach. This approach may or may not be acceptable and perhaps the user should be given a choice.

7. Shared Disks Versus Partitioning

In a multisystem configuration, *shared disks* (SD) and *shared nothing* (SN - i.e., partitioned approach) environments have different effects on the extent of applicability of the Commit_LSN method. In the case of SD, the disks are shared by the multiple instances of the DBMS, but each DBMS instance has its own buffer pool and global locks, and buffer coherency protocols are required to preserve consistency of the data in the face of the ability of all the systems to read and modify any of the data in the data base. SD is the approach taken in IBM's IMS Data Sharing [PeSt83, Yama83], TPF [Scru87] and the Amoeba research project [MoNP90, SNOP85], in NEC's DCS [SMMT84], and in DEC's VAX Rdb/VMS[1] and VAX DBMS[1] in the VAXcluster[1] environment [KrLS86, ReSW89]. In the case of SN, each system can read or modify directly only a portion of the data in the data base. SN is the approach taken in NonStop SQL, Teradata's DBC/1012[1] [Tera88], and the University of Wisconsin's Gamma data base machine [DGSBH90].

It should be quite clear that SN stands to benefit the most from the Commit_LSN method since in that environment there is only, at most, one copy of the data in the buffer pool. The Commit_LSN can be determined by each system independently of the other systems. With SD, conceptually, each system needs to periodically poll the other systems to determine the minimum Commit_LSN across all the systems (various alternatives are possible for implementing this efficiently). As noted before, it is acceptable to use an out-of-date Commit_LSN.

The latter can only cause unnecessary locking and not the reading of any uncommitted data.

With SD, since there can be more than one copy of each page across the different buffer pools, unless a given system is already known to have the latest version of a given page, the utility of the Commit_LSN method is reduced. It is applications 1, 2, 3, 8, and 9 of the Commit_LSN method that suffer the most in data sharing, if the CS transactions always need to see the most recent version of any data. When record locking is in effect, even if the system is forced to behave like an updater of the page, which may involve acquiring a global lock on the page to be sure that the transaction is reading the *latest* version of the page, the system can still benefit if the Commit_LSN method avoids the need for acquiring the record-level global locks. SD is penalized the most, compared to SN, when page locking is done, but it is still better compared to SD in which the Commit_LSN method is *not* adopted. The amount of *global* locking will not decrease, but some *local* locking will be avoided.

It is possible to design a method in which, if sufficiently large amount of data is going to be read in one system and the amount of concurrent update activity in the other systems is small, the updating systems can be made to let the reading system know what pages are being updated by them that belong to the object being read and for those pages alone the reading system can do locking. This penalizes the updating transactions and it is not clear that this is desirable. Another possibility is to relax the requirement that the CS reader always see the *latest* version of the data.

Also, a version number technique [MoNP90] used to avoid the need for a merged log across the systems, makes it impossible to correlate a page's version number (a counter that monotonically increases for a particular page) with the information about the Begin_LSNs of the currently executing transactions. This is because the version number assigned to a page during an update may be less than the Begin_LSN of the updating transaction even if the system equates the Begin_LSN to the timestamp assigned to that transaction's Begin_Transaction log record. This could be fixed by ensuring that the version number assigned at the time of an update to the page is always *at least* as high as the updating transaction's Begin_LSN.

Earlier, it was pointed out as to how the data-only locking performed by the ARIES/IM method and the next key locking performed during key deletes could cause some unnecessary lock waits during index and data page accesses and how the Commit_LSN method reduces the impact of those. In SD, there is another feature which could cause unnecessary waits for some types of accesses. This is due to the fact that the *instant* duration X lock acquired on the next key during key inserts in the single system environment had to be changed to a *commit* duration X lock in SD. The Commit_LSN method can reduce the impact of this change.

8. Summary

We presented a novel and simple method, called Commit_LSN, for determining if a piece of data is in the committed state in a transaction processing system. This method is a much cheaper alternative to the locking approach used by the prior art for this purpose. The method took advantage of the concept of log sequence number (LSN) which, in many systems, is recorded in each page of the data base to relate the state of a page to the log of update actions for that page. Information which is normally used only for recovery purposes is exploited by the Commit_LSN method to improve performance. The method uses information about the LSN of the first log record (called *Commit_LSN*) of the oldest update transaction still executing in the system to infer that all the updates in pages with page_LSN *less than* Commit_LSN have been committed and thereby reduces locking and latching overheads. In addition, this method was also shown to increase the level of concurrency that could be supported. We described how our method makes it possible to use fine-granularity locking without unduly affecting transactions which read numerous records. We also described its benefits to update transactions due to the reduction in the cost of fine-granularity locking when contention is not present for data on a page. We discussed in detail many applications of this method and illustrated its potential benefits. We also analyzed the impact of the shared disks and the shared nothing environments. Techniques for implementing this method in systems based on write-ahead logging or the shadow-page technique were presented. For the 5-way join query analyzed in [PMCLS90], we have estimated that the Commit_LSN method could save up to 90% of the pathlength, depending on the sizes of records, number of records per page, predicate selectivities, etc.

9. References

CABGK81 Chamberlin, D., et al., *A History and Evaluation of System R*, **Communications of the ACM**, Vol. 24, No. 10, October 1981.

ChGY81 Chamberlin, D., Gilbert, A., Yost, R. *A History of System R and SQL/Data System*, **Proc. 7th International Conference on Very Large Data Bases**, Cannes, September 1981.

ChMe88 Chang, A., Mergen, M. *801 Storage: Architecture and Programming*, **ACM Transactions on Computer Systems**, Vol. 6, No. 1, p28-50, February 1988.

ChMy88 Chang, P.Y., Myre, W.W. *OS/2 EE Database Manager Overview and Technical Highlights*, **IBM Systems Journal**, Vol. 27, No. 2, 1988.

ClCo89 Clark, B.E., Corrigan, M.J. *Application System/400 Performance Characteristics*, **IBM Systems Journal**, Vol. 28, No. 3, 1989.

Crus84 Crus, R. *Data Recovery in IBM Database 2*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.

Curt88 Curtis, R. *Informix-Turbo*, **Proc. IEEE Compcon Spring '88**, February-March 1988.

DGSBH90 DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H.-I, Rasmussen, R. *The Gamma Database Machine Project*, **IEEE Transactions on Knowledge and Data Engineering**, Vol. 2, No. 1, March 1990.

DHLPR89 DeLorme, D., Holm, M., Lee, W., Passe, P., Ricard, G., Timms, Jr., G., Youngren, L. *Database Index Journaling for Enhanced Recovery*, **U.S. Patent 4,819,156**, IBM, April 1989.

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.

GaKi85 Gawlick, D., Kinkade, D. *Varieties of Concurrency Control in IMS/VS Fast Path*, **IEEE Database Engineering**, Vol. 8, No. 2, June 1985.

GMBLL81 Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., Traiger, I. *The Recovery Manager of the System R Database Manager*, **ACM Computing Surveys**, Vol. 13, No. 2, June 1981.

Gray78 Gray, J. *Notes on Data Base Operating Systems*, In **Operating Systems - An Advanced Course**, R. Bayer, R. Graham, and G. Seegmuller (Eds.), Lecture Notes in Computer Science, Volume 60, Springer-Verlag, 1978. Also Available as IBM Research Report RJ2188, IBM San Jose Research Laboratory, February 1978.

HaRe83 Haerder, T., Reuter, A. *Principles of Transaction Oriented Database Recovery - A Taxonomy*, **Computing Surveys**, Vol. 15, No. 4, December 1983.

HMSC88 Haskin, R., Malachi, Y., Sawdon, W., Chan, G. *Recovery Management in QuickSilver*, **ACM Transactions on Computer Systems**, Vol. 6, No. 1, p82-108, 1988.

KrLS86 Kronenberg, N., Levy, H., Strecker, W. *VAXclusters: A Closely-Coupled Distributed System*, **ACM Transactions on Computer Systems**, Vol. 4, No. 2, May 1986.

LiMP86 Lindsay, B., Mohan, C., Pirahesh, H. *Method for Reserving Space Needed for "Rollback" Actions*, IBM Invention Disclosure SA885-0217, **IBM Technical Disclosure Bulletin**, Vol. 29, No. 6, November 1986.

MHLPS89 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*, To Appear in **ACM Transactions on Database Systems**. Also Available as **IBM Research Report RJ6649**, IBM Almaden Research Center, January 1989.

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 is available as **IBM Research Report RJ7341**, IBM Almaden Research Center, March 1990.

Mino84 Minoura, T. *Multi-Level Concurrency Control of a Database System*, **Proc. 4th IEEE Symposium on Reliability in Distributed Software and Database Systems**, Silver Spring, October 1984.

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

Moha90b Mohan, C. *ARIES/LHS: A Concurrency Control and Recovery Method Using Write-Ahead Logging for Linear Hashing with Separators*, **IBM Research Report**, IBM Almaden Research Center, 1990.

MoHP90 Mohan, C., Haderle, D., Peterson, A. *ARIES/IM/LDK: A Concurrency Control and Recovery Method for Index Management Using Write-Ahead Logging and Logical Deletion of Keys*, **IBM Research Report**, IBM Almaden Research Center, In Preparation, 1990.

MoLe89 Mohan, C., Levine, F. *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*, **IBM Research Report RJ6846**, IBM Almaden Research Center, August 1989.

MoLi83 Mohan, C., Lindsay, B. *Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions*, **Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing**, Montreal, Canada, August 1983. Also IBM Research Report RJ3881, IBM San Jose Research Laboratory, June 1983.

MoLO86 Mohan, C., Lindsay, B., Obermarck, R. *Transaction Management in the R* Distributed Data Base Management System*, **ACM Transactions on Database Systems**, Vol. 11, No. 4, December 1986. Also IBM Research Report RJ5037, IBM Almaden Research Center, February 1986.

MoNP90 Mohan, C., Narang, I., Palmer, J. *A Case Study of Problems in Migrating to Distributed Computing: Page Recovery Using Multiple Logs in the Shared Disks Environment*, **IBM Research Report RJ7343**, IBM Almaden Research Center, March 1990.

MoPi90 Mohan, C., Pirahesh, H. *ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method*, **IBM Research Report RJ7342**, IBM Almaden Research Center, February 1990.

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

PeSt83 Peterson, R.J., Strickland, J.P. *Log Write-Ahead Protocols and IMS/VS Logging* **Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems**, Atlanta, March 1983.

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, IEEE Computer Society Press. Also Available as **IBM Research Report**, IBM Almaden Research Center, July 1990.

ReSW89 Rengarajan, T.K., Spiro, P., Wright, W. *High Availability Mechanisms of VAX DBMS Software*, **Digital Technical Journal**, No. 8, February 1989.

RoMo89 Rothermel, K., Mohan, C. *ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions*, **Proc. 15th International Conference on Very Large Data Bases**, Amsterdam, August 1989. A longer version appears as **IBM Research Report RJ6650**, IBM Almaden Research Center, January 1989.

SCFLM86 Schwarz, P., Chang, W., Freytag, J., Lohman, G., McPherson, J., Mohan, C., Pirahesh, H. *Extensibility in the Starburst Database System*, **Proc. Workshop on Object-Oriented Data Base Systems**, Asilomar, September 1986. Also Available as IBM Research Report RJ5311, IBM Almaden Research Center, September 1986.

Scru87 Scrutchin, T. *TPF: Performance, Capacity, Availability*, **Proc. IEEE Compcon Spring '87**, San Francisco, February 1987.

Shoe86 Shoens, K. *Data Sharing vs. Partitioning for Capacity and Availability*, **Database Engineering**, Vol. 9, No. 2, March 1986.

SMMT84 Sekino, A., Moritani, K., Masai, T., Tasaki, T., Goto, K. *The DCS - A New Approach to Multisystem Data-Sharing*, **Proc. National Computer Conference**, Las Vegas, July 1984.

SNOP85 Shoens, K., Narang, I., Obermarck, R., Palmer, J., Silen, S., Traiger, I., Treiber, K. *Amoeba Project*, **Proc. IEEE Compcon Spring '85**, San Francisco, February 1985.

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. Also in **Lecture Notes in Computer Science Vol. 359**, D. Gawlick, M. Haynie, A. Reuter (Eds.), Springer-Verlag, 1989.

Tera88 Teradata *DBC/1012 Data Base Computer Concepts and Facilities - Release 3.1*, **Documber Number C02-0001-05**, Teradata Corp., May 1988.

Yama83 Yamashita, A. *Data Base Integrity at Emergency Restart in Data Sharing*, IBM Invention Disclosure SA882-0110, **IBM Technical Disclosure Bulletin**, Vol. 26, No. 2, p863, July 1983.