A Cost-Effective Method for Providing Improved Data Availability During DBMS Restart Recovery After a Failure

C. MOHAN

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

Abstract We present a cost-effective method for improving data availability during restart recovery of a data base management system (DBMS) after a failure. The method achieves its objective by enabling the processing of new transactions to begin even before restart recovery is completed by exploiting the *Commit_LSN* concept. It supports fine-granularity (e.g., record) locking with semantically-rich lock modes and operation logging, partial rollbacks, write-ahead logging, and the *steal* and *no-force* buffer management policies. The overhead imposed by this method during normal transaction processing is insignificant. We require very few changes to an existing DBMS in order to support our method. Our method can be implemented with different degrees of sophistication depending on the existing features of a DBMS.

1. Introduction

Increased demands are being placed on data base management systems (DBMSs) to provide improved data availability to user transactions [Moha93c]. Our motivation for designing the algorithms presented in this paper stems from our knowledge of some customers who very regularly produce many tapes worth of log records as a result of the execution of a *single transaction*! As can be easily imagined by the reader, if such a transaction's execution were to be interrupted by a DBMS failure, then even after the DBMS is restarted it would be a very long time before the processing of *new* transactions will begin, assuming that the DBMS does not start handling new transactions until *all* of restart recovery (i.e., both the redo and the undo passes) is completed. Today, in almost all DBMSs, all restart recovery work is done by a single process and, at least, all read I/Os are performed *synchronously, one page at a time*!

Over the years, various solutions to provide improved data availability have been proposed with different continual system operational costs and DBMS development costs. One of the solutions is the concept of *hot standby* as implemented in Tandem's NonStop™ architecture [Tand87], and IBM's XRF (eXtended Recovery Facility) for IMS [IBM87] and CICS [IBM89, ScRi88]. More recently, proposals have been made which exploit nonvolatile memory to improve data availability [CKKS89, LeCa87, Levy91]. Compared to them, our solution is much cheaper to implement for the DBMS implementers. It is also much more cost effective to operate and support for the users of the DBMS. We desired a solution which supports the ARIES [MHLPS92] recovery method's features like fine-granularity (e.g., record) locking with semantically-rich lock modes and operation logging, partial rollbacks, write-ahead logging, and the *steal* and *no-force* buffer management policies. The ***no-force*** buffer management policy states that it is *not* required that *before* a transaction is allowed to commit, all *pages* modified by that transaction must be forced to disk. The ***steal*** policy states that a page with *uncommitted* updates may be written to disk. We did not want a solution that required that all data be versioned and that the force policy be followed, as [Ston87] does for POSTGRES. The numerous advantages of no-force and steal are discussed in detail in [MHLPS92].

The rest of this paper is organized as follows. In the remainder of this section, we discuss the problems that need to be dealt with in permitting new

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 19th VLDB Conference
Dublin, Ireland, August 1993

™ NonStop, NonStop SQL and Tandem are trademarks of Tandem Computers, Inc. DB2, DB2/2, DB2/6000 and IBM are trademarks of the International Business Machines Corp. DEC and Rdb/VMS are trademarks of Digital Equipment Corp. Informix is a registered trademark of Informix Software, Inc. Oracle is a trademark of Oracle Corp.

transaction processing to begin before the completion of restart recovery and the assumptions that we make in proposing our method. Section 2 presents our method in two parts. The first subsection presents the method for permitting new transaction processing to be initiated only after the undo pass of restart recovery begins. The second subsection extends the method to permit the processing of new transactions to begin even earlier - from the start of the redo pass itself. The third subsection discusses the applicability of our method to the shared disks (*data sharing*) environment. Section 3 discusses related work as implemented in systems like IMS/XRF, CICS/XRF and Tandem's NonStop. We conclude with section 4.

1.1. Improving Data Availability

One way to improve data availability would be to permit new transaction processing to start as soon as the DBMS is brought up after a failure, instead of waiting for all of the DBMS recovery to be completed. The latter is the case in almost all DBMSs (e.g., DB2™, Informix™, Oracle™, ...). The difficulty in permitting new transaction activity to occur even before restart recovery is completed comes from the fact that some of the pages that the new transactions want to access might be in such states that permitting those accesses may lead to data inconsistencies. The undesirable states are:

Undesirable State 1 A page on disk at restart may *not* contain some updates for which log records exist. These updates might be the ones that were performed by uncommitted and/or committed transactions. Permitting accesses to such a page might lead to a *new* transaction reading an older version of a piece of data (e.g., a record) to which one or more log records written by one or more *committed* transactions remain to be applied. Assuming that record locking is being used with flexible storage management,[1] even if the new transaction were to access the page for updating or inserting some *record* for which no unapplied log records exist, permitting that operation to proceed before completion of recovery might result in some space on the page being consumed. The latter might result in a state in which it is impossible to redo some of the unapplied log records' changes relating to *other records* on the *same page*. With ARIES, redo is not performed logically across pages [MHLPS92], but is page oriented (i.e., the same page as the originally updated page gets updated during redo also). Whether a particular log record's update needs to be redone is determined by comparing the LSNs of the log record and the data base page referred to in the log record. Given these properties, it is an unacceptable situation to allow a new transaction to read or modify a page to which some log records remain to be applied.

Undesirable State 2 A page on disk may contain some uncommitted updates. Such a page may not require any redo since it *may* contain all the updates logged for that page. Even if redo is not required, inconsistencies may be caused if the DBMS were to allow access by a new transaction to such a page. This is because the page may contain updates of (1) some transactions which are to be rolled back as part of restart (i.e., the so called ***in-flight transactions***) or (2) those transactions which will remain in the *in-doubt* state (of two-phase commit [MoLO86]) at the end of restart recovery and for which locks will be reacquired during the course of the redo pass to protect their uncommitted updates (i.e., the so called ***in-doubt transactions***). Allowing an access under these conditions might result in a new transaction reading some uncommitted data even though the *new* transaction will be acquiring locks. Of course, the second point would not be a concern if the new transaction's access is happening during the *undo* pass since by then the *in-doubt* transactions would have reacquired locks to protect their uncommitted updates. Permitting updates by new transactions might make it impossible to undo updates of some in-flight transactions due to lack of space [MoHa93].

At the time of initialization of restart recovery, a given page on disk may be in both of the above undesirable states. Our method allows us to cost effectively determine when the above conditions could possibly exist for a given page in order to disallow access to such a page by a *new* transaction before one or more passes of restart recovery are completed. We require very few changes to an existing DBMS in order to support our method. Our method can be implemented with different degrees of sophistication depending on the existing features of a DBMS.

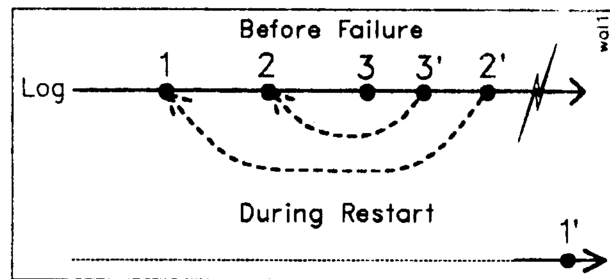
---

1 *Flexible storage management* means that a given record may exist anywhere on the page. At different times, it may exist in different locations on the page. If the record is moved around within the page, then there is no need to lock it or log its movements. This allows efficient support of garbage collection which brings together to a contiguous area all the free space on the page. As a consequence, varying length records can be managed efficiently. It also means that logging is logical within a page [MHLPS92]. This is the approach taken in System R, DB2, DB2/2, DB2/6000 and SQL/DS [Moha93b].

1.2. Assumptions

We assume that the DBMS supports distributed transactions and implements the write-ahead logging (WAL) based ARIES recovery method described in [MHLPS92, MoLe92, MoNa93, MoPi91, RoMo89] or a similar method. ARIES has become very popular. It has been implemented in IBM's DB2 family of products (DB2, DB2/2™, DB2/6000™) [Moha93b], Starburst, QuickSilver, WDSF/VM and Message Queue Manager, Transarc's Encina, and University of Wisconsin's Gamma and EXODUS. ARIES has been extended by others also [FZTCD92, Lome92]. When ARIES is used, every data base page has a ***page_LSN*** field which contains the ***log sequence number (LSN)*** of the log record that describes the most recent update to the page. Since LSNs monotonically increase over time, by comparing at recovery time a page_LSN with the LSN of a log record for that page, we can unambiguously determine whether that version of the page contains that log record's update. That is, if the page_LSN is *less than* the log record's LSN, then the effect of the latter is *not* present in the page. ARIES supports fine-granularity (e.g., record) locking with semantically-rich lock modes (e.g., increment/ decrement-type locks), partial rollbacks, nested transactions, write-ahead logging, and the steal and no-force buffer management policies.

In ARIES, restart recovery consists of three passes of the log: *analysis*, *redo* and *undo*. While recovering from a system failure, ARIES first scans the log, starting from the first log record of the last complete checkpoint and continuing up to the end of the log. During this ***analysis pass***, the information included in the checkpoint record about pages that were more up to date in the buffers than on disk (the so-called ***dirty*** pages) and about transactions that were in progress is brought up to date as of the end of the log by analyzing the log records in that interval. For each page in the ***dirty pages list (DPL)***, the LSN of the log record (call it the ***dirty LSN***) from which redo might have to be performed is also determined based on information in the checkpoint log record and the subsequent log records. DPL from the analysis pass determines the starting point (i.e., the ***RedoLSN*** = the minimum of the LSNs in DPL) for the log scan of the next pass, and acts as a filter to determine which log records and consequently which data base pages have to be examined to see if some updates need to be redone. The analysis pass also provides the list of in-flight and in-doubt transactions, and the LSN of the latest log record written by each such transaction.



I' is the CLR for I. Only UndoNxtLSN chain is shown (1' has a NULL pointer). PrevLSN chain should be obvious.

Figure 1: ARIES Recovery Scenario - Log Records of a Single Transaction

In the ***redo pass***, ARIES ***repeats history*** with respect to those updates logged on stable storage but whose effects on the data base pages did not get reflected on disk before the crash. This is done for the updates of *all* transactions, *including the updates of in-flight transactions*. This essentially reestablishes the state of the data base as of the time of the crash, as far as the actions represented in the log on stable storage as of the crash time are concerned. The redo pass also reacquires the locks needed to protect the uncommitted updates of the *in-doubt* transactions.

The next pass is the ***undo pass*** during which all in-flight transactions' updates are rolled back, in reverse chronological order, in a single sweep of the log. This is done by continually taking the maximum of the LSNs of the next log record to be processed for each of the yet-to-be-completely-undone transactions, until no transaction remains to be undone. ARIES also logs, typically using compensation log records (***CLRs***), updates performed during partial or total rollbacks of transactions during both normal and restart processing. In ARIES, CLRs have the property that they are redo-only log records. By appropriate *chaining* of the CLRs written by a rolling-back transaction to log records written by that transaction during forward processing, a bounded amount of logging is ensured during rollbacks. The latter will be the case even in the face of repeated failures during restart recovery or of nested rollbacks. When the undo of a log record (***nonCLR***) causes a CLR to be written, the CLR is made to point, via the ***UndoNxtLSN*** field of the CLR, to the *predecessor* (i.e., setting it equal to the ***PrevLSN*** value) of the log record being undone (see Figure 1).

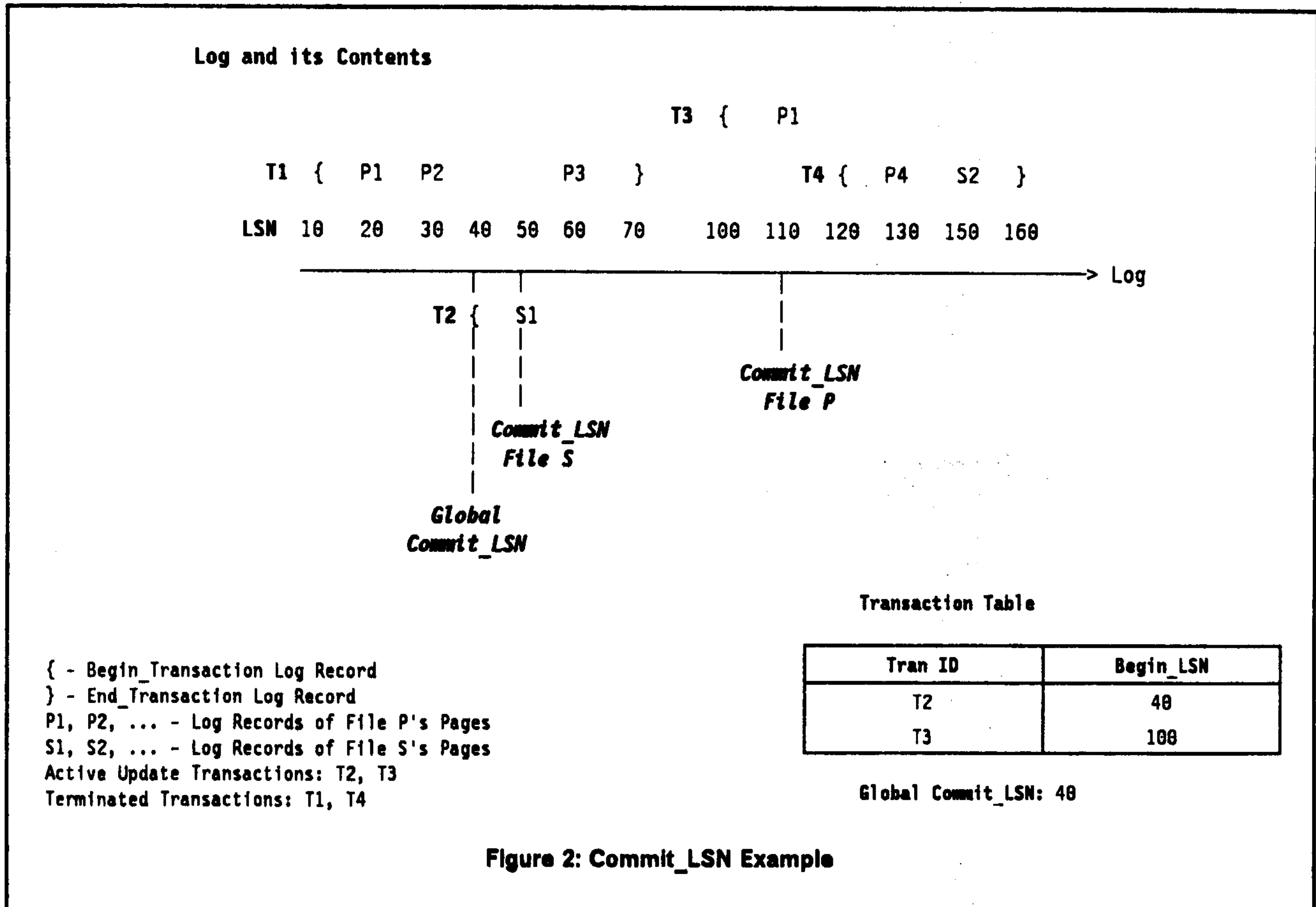


Figure 2: Commit_LSN Example

We assume that when a system failure occurs, all the locks held by the in-flight and in-doubt transactions are lost. That is, we do not assume that nonvolatile main memory is available to preserve the lock and buffer pool states across the DBMS failure. We are *not* assuming that during the redo pass locks are obtained to protect the uncommitted updates of *in-flight* (*not* in-doubt) transactions also. This means that the redo pass need not examine *all* the log records of the in-flight transactions for locking purposes. We assume that latching[2] of pages is performed as part of page accesses during the redo and undo passes, just as it is done during normal (forward and undo) processing in ARIES.

2. A Cost-Effective Solution

There are two parts to our cost-effective solution for the improved data availability problem. We discuss them in the next two subsections. Depending on the existing features of a DBMS, our method can be implemented to different degrees of sophistication with minimal changes to the DBMS.

We use a global flag, called ***Restart***, in the DBMS which can be tested to see if restart recovery is in progress. Restart is equal to 'Y' if restart recovery is still in progress; otherwise, Restart is equal to 'N'. When Restart is equal to 'Y', the system holds the ***Restart latch*** in the X mode and another field called ***Pass*** can be checked to see which pass of restart recovery is in progress (*Analysis*, *Redo* or *Undo*). When Pass is equal to 'Redo', the system holds the ***Redo latch*** in the X mode. The latter is

2 A latch is like a semaphore and it is a cheaper implementation of a short-duration lock. It is typically used for ensuring the physical consistency of some object (typically, a page) that is about to be read or modified. Latch waits are not communicated to the deadlock detector and hence latch usage must be such that deadlocks are avoided. For more details on the differences between locks and latches, see [MHLPS92].

released once the system completes the redo pass. The Restart latch is released only after restart recovery is completed. We assume that at the end of the analysis pass, the current end-of-log LSN, call it ***fail_LSN*** (***FLSN***), is determined and the global variable FLSN is set to that value.

2.1. Executing New Transactions During the Undo Pass

First, we deal with the case where we admit new transactions into the system only after the DBMS finishes the redo pass. The first undesirable state described in the section "1.1. Improving Data Availability" will no longer be a problem since the redo pass would have been completed. When new transactions' read and write accesses to pages are processed, we use the Commit_LSN idea from [Moha90a] in a novel way to determine efficiently when they are encountering pages in the second undesirable state. ***Commit_LSN*** is the *minimum* of the LSNs of the Begin_Transaction log records of all the *in-flight* transactions (see Figure 2). The interpretation of Commit_LSN is that no page with an LSN *less than* Commit_LSN can contain any uncommitted updates belonging to in-flight transactions. Originally, Commit_LSN was proposed to reduce or eliminate locking under certain conditions. It has been implemented in DB2 for those reasons [Moha93b]. Here, we use it to know when it is safe to let a new transaction read or modify a page before recovery is completed.

For the purposes of this paper, the Commit_LSN value is computed at the end of the *analysis pass*. We need not worry about the *in-doubt* transactions' uncommitted updates since those updates will be protected by locks. The needed locks would have been reacquired on behalf of those transactions by the time the undo pass starts. If new transaction activity is going to be permitted even during the redo pass (see the next subsection), then, *during the redo pass alone*, the Commit_LSN that is used should be computed by taking into account the *in-doubt* transactions also. The latter is necessary because, *during* the course of the redo pass, locks might not yet have been reacquired to protect the uncommitted updates of the in-doubt transactions.

Basically, any time a page access is attempted by a new transaction in *forward processing* (i.e., not rolling back), *if Restart = 'Y'*, then the transaction is allowed to access the page (for read or write) only if the following condition, called the ***Undo_Pass Condition***, holds:

page_LSN < Commit_LSN

If the above condition does not hold, then the transaction requests the Restart latch in the S mode, thereby waiting for restart recovery to finish.[3] The reason for waiting in this case is that it is possible that the page has some (uncommitted) changes which are not yet undone. Note that the action that is taken in this case is a conservative one. Just because the page_LSN is not smaller than Commit_LSN it does not mean that the page definitely has some uncommitted changes.

We can do better than the above conservative approach if we log at checkpoint time, as DB2 does for example, for each active transaction, the list of objects (e.g., at the gross granularity of file or table) that have its uncommitted updates. We call the union of these lists the ***Uncommitted Objects List (UOL)***. During the analysis pass, this list could be brought up to date as of the end of the log. In fact, although *during* the redo pass UOL has to contain even objects for which there are uncommitted updates by **only** *in-doubt* transactions (i.e., objects for which there are *no* uncommitted updates by *in-flight* transactions), at the *end* of the redo pass such objects may be safely removed from UOL since by then the in-doubt transactions would have reacquired their locks on such objects. Furthermore, as the undo pass progresses, UOL could be kept up to date. That is, once *all* the in-flight transactions which had, as of the failure of the system, uncommitted updates in a particular object are completely undone, that object can be removed from UOL.

Assuming UOL is available, *if Restart = 'Y'*, then a new transaction in forward processing is allowed to access a page (for read or write) only if the following (*modified*) ***Undo_Pass Condition*** holds:

(page belongs to object not in UOL) OR

((page belongs to object in UOL) AND (page_LSN < Commit_LSN))

Instead of using the above (global) Commit_LSN, an even better method would be to compute, for each object in UOL, the ***object-specific Commit_LSN***

3 In order to avoid deadlocks involving the restart latch, any other latches that are held (e.g., on the page that has been accessed and, in the case of an index access, possibly the latch on the parent of the current page [Moha90b, MoLe92]) must be released *before* waiting on the Restart latch. Once the Restart latch is obtained, the previously released page latches must be reacquired and the previously inferred information must be revalidated. Such revalidations are discussed in [Moha90b, MoLe92].

[Moha90a]. That is, for each object, consider only those transactions that have uncommitted updates on it and compute the minimum of the Begin_Transaction LSNs of only those transactions. Then, in the Undo_Pass Condition, use the object-specific Commit_LSN for that object, instead of the global Commit_LSN. The advantage of an object-specific Commit_LSN over the global Commit_LSN is that in general the former will be greater than the latter (in the worst case they will be equal), thereby allowing more page accesses by new transactions. The former reduces the negative impact of some long running transactions accessing private or semi-private data [Moha90a]. It has been implemented in DB2. Object-specific Commit_LSNs can also be computed at the end of the *analysis pass*. Similar to what was said earlier with reference to UOL, note that for computing the object-specific Commit_LSN *for this purpose*, during the redo pass, we must take into account, apart from the in-flight transactions, even those transactions that were in the in-doubt state at the time of the failure. On the contrary, during the undo pass, we need to consider only the in-flight transactions that are being undone as part of that pass.

The unfortunate aspect of the above mentioned test involving the Commit_LSN is that once a page of an object in UOL is found to have its page_LSN < Commit_LSN and the *first update* to that page is performed by a new transaction, no subsequent update or read by new transactions in forward processing will be possible on that page until restart recovery is finished or the object is removed from UOL, whichever happens first. This is because that first update would make the page_LSN be *greater than* Commit_LSN (global or object-specific), thereby violating the Undo_Pass Condition that must be satisfied for permitting access by a new transaction in forward processing. Unfortunately, as a result of that first update, we lose track of the fact that the page does not contain any uncommitted updates of the transactions being rolled back in the undo pass. Preserving the latter information requires having a bit called the ***unlocked_uncommitted_data_bit (UUD_Bit)*** on every page of the data base. If the bit is '1' then that means that the page *may* contain some uncommitted updates which are *not* protected by locks. If the bit is '0' then the page definitely does *not* contain any uncommitted updates which are not protected by locks. Note that if the bit is '0' the page may still contain some *uncommitted* updates. The crucial distinction is that in the latter case the uncommitted updates will definitely be protected by locks.

The following are the rules for manipulating the UUD_Bit:

•*Normal* transaction updates (forward processing or normal undo (i.e., not undo during restart recovery))

```
Set to '0'
```

It is not incorrect to do this since we would allow an update to this page by a *new* transaction only if the page was definitely known not to have any unlocked uncommitted updates. If a system failure were to occur, thereby causing the loss of the lock information, the setting of the bit by the buffer manager during reads from disk (see below) will ensure the existence of the desired state for the bit at the appropriate time during restart.

•Restart redo of an *in-flight* transaction's log record (assuming that *in-doubt* transactions reacquire their locks before the start of the undo pass)

```
Set to '1'
```

This is necessary since an update which is not being protected by a lock is being redone. As of this time, the UUD_Bit on the page may have the value '0' and in that case this update will be the first unprotected one for the page.

•Restart redo of structure modification (page split and page delete) related log records for leaf pages in an index or record relocations in a hash-based storage method for **all** transactions

```
IF log record's LSN >= Commit_LSN THEN
  Set to '1'
```

The above is needed because, with the high concurrency supported by index protocols like ARIES/IM [MoLe92] and ARIES/KVL [Moha90b], and hash-based storage's recovery methods like ARIES/LHS [Moha93a], one transaction's *uncommitted* updates on a certain page may be moved to a totally different page by *another* transaction. The second transaction may terminate or get into the in-doubt state even as the first transaction remains in the *in-flight* state. Under these conditions, after a failure, the only way to ensure that the first transaction's uncommitted updates remain protected is to ensure that the *mover* of the uncommitted data to a different page causes the UUD_Bit to get set to '1' on the second page. Again, Commit_LSN is taken advantage of to determine whether such a situation is a possibility.

•Restart redo for a *non-in-flight* transaction *and* the log record does *not* relate to structure modification as described in the last item

```
IF existing page_LSN (i.e., page_LSN before redo
   is performed) < Commit_LSN THEN Set to '0'
```

Here, we are trying to take advantage of the facts that the page is known to contain only committed updates before this update is redone (since page_LSN is less than Commit_LSN) and that this update itself is either a committed update or an uncommitted update, by an in-doubt transaction, which will be protected by locks by the time the undo pass starts. Note that after the current log record's update is redone, the page's LSN may become greater than Commit_LSN since page_LSN will be set to the log record's LSN.

•Buffer manager when reading into the buffer pool a page from disk

```
IF Restart = 'Y' THEN
  IF (page_LSN >= Commit_LSN) AND
     (page_LSN < FLSN) THEN Set to '1'
```

Setting the UUD_Bit to '1' under the above conditions is a conservative action since the fact that the page is in that range does not necessarily mean that there is some uncommitted data on the page that is not protected by locks. Since a system failure could happen anytime, the burden is placed on the buffer manager to ensure that the correct UUD_bit setting is present on a page when the page is read from disk and restart recovery is still in progress. This is important since, during recovery after a failure, it is the disk version of the data base that recovery processing and new transactions will be dealing with. The buffer manager can use in this check the global Commit_LSN or, better still, the object-specific Commit_LSN. As mentioned before, an object-specific Commit_LSN is always better than or at least as good as the global Commit_LSN [Moha90a]. In any case, the important point to note is that the computation of those Commit_LSN values must take into account only those transactions that were active at the time of the last failure.

The setting of the UUD_Bit, if required, by the buffer manager does **not** cause the page to become dirty. The buffer manager does not disturb the existing UUD_Bit setting on the page if the condition in the above test is not satisfied. If the page_LSN is greater than FLSN then it is *incorrect* to always assume that there is no *uncommitted* data on the page that is not protected by a lock and set the UUD_Bit to '0'. This is because this may be the second time that the page is being read from disk during this restart recovery and when it was read from disk (and before it was subsequently written to disk) the first time, some *in-flight* transaction's undo might have been performed on the page which caused the page_LSN to become greater than FLSN due to the writing of a CLR and the assignment of the CLR's LSN to the page_LSN. Under these conditions, it is essential that the UUD_Bit remains at the value of '1' since the page may still contain some uncommitted updates which are not protected by locks. If, on the other hand, the page_LSN is greater than FLSN because, during the first time it was read in, a *new* transaction had modified it, then we would like to retain the UUD_Bit value of '0' that would exist as a result.

With the introduction of the UUD_Bit, the ***Undo_Pass Condition*** becomes:

```
(page belongs to object not in UOL) OR
((page belongs to object in UOL) AND
                    (UUD_Bit = '0')) OR
((page belongs to object in UOL) AND
        (page_LSN < Commit_LSN))
```

With the introduction of the UUD_Bit, in the unfortunate scenario discussed earlier, the first update by the new transaction will cause the UUD_Bit to be set to '0' and so the condition for allowing accesses to new transactions will still be true.

2.2. Executing New Transactions During the Redo Pass

Allowing new transaction activity concurrently with redo processing also requires that we deal with both undesirable states described in the section "1.1. Improving Data Availability". We do that by taking advantage of some information that typically gets logged at the time of a checkpoint. As in ARIES, first we assume that at the time of a checkpoint the dirty pages list (***DPL***) is logged and that during the analysis pass this DPL is brought up to date as of the end of the log. Many advantages of logging DPL are discussed in [MHLPS92]. DPL includes only those pages that might *potentially* be involved in *redo* operations. In particular, DPL *may not* include any (or some) pages on which *undo* may have to be performed (i.e., pages of objects in UOL).

Assuming DPL is available, *if Restart = 'Y' and Pass = 'Redo'*, then a new transaction is allowed to access a page (for read or write) only if the following ***Redo_Pass Condition*** holds:

```
(Undo_Pass Condition) AND (page not in DPL)
```

Basically, these checks ensure that the page does not contain any uncommitted data and that it will not be modified in the redo pass, respectively. If

the Redo_Pass Condition does not hold, then the transaction requests the *Redo* latch in the S mode for instant duration, thereby waiting for the redo pass to finish. Once the Redo latch is granted, the new transaction has to check the Undo_Pass Condition.

If, unlike in ARIES but like in DB2 [TeGu84], the buffer manager logs at checkpoint time the dirty object information only at the granularity of a file which is then brought up to date during the analysis pass, thereby providing the system with the ***dirty objects list (DOL)***, then the ***Redo_Pass Condition*** becomes

```
(Undo_Pass Condition) AND
  (page belongs to object not in DOL)
```

Even in those systems which do not log DPL during checkpoint time, if we would like to get the benefit of what something like DPL could provide, then we could implement a more expensive solution which involves scanning the log from the Redo_LSN to the end of the log. This extra pass can be performed by a separate process. It is initiated when the redo pass is initiated and it uses DOL to determine what *pages* of the objects in DOL might need some log records' updates to be redone on them. This is done just by noting the page numbers in the log records relating to the objects in DOL. Since no data page accesses are made, this extra pass would be completed long before the redo pass completes. As soon as this extra pass is finished, new transactions can be let into the system. DPL generated in this fashion will be equal to or be a superset of DPL that we would have obtained if the system were to log it at checkpoint time and bring it up to date during the analysis pass, as in ARIES.

Since processing new transactions will result in the log growing continuously, in order to make sure that the redo pass terminates, we should terminate the redo pass when the FLSN point is reached.

2.3. Shared Disks Environment

What we have discussed so far is usable in the single-system and partitioned (also called *shared nothing*) DBMS environments. In addition, our method is also usable in the shared disks (***SD*** - also called the data sharing) environment [Haer88, Lome90, MoNa91, MoNa92a, MoNa92b, Rahm91, Reut86]. With SD, all the disks containing the data base are shared amongst the different systems. Every system that has an instance of the DBMS executing on it may access and modify any portion of the data base on the shared disks. Since each DBMS instance has its own buffer pool and because conflicting accesses to the same data may be made from different systems, the interactions amongst the systems must be controlled via various synchronization protocols. This necessitates global locking and protocols for the maintenance of buffer coherency. SD is the approach used in IBM's IMS/VS Data Sharing product and TPF, and in DEC's Rdb/VMS™ [ReSW89]. Fujitsu, Hitachi, INGRES and Oracle™ Parallel Server have also adopted this approach.

In the SD context, when a system fails, the locks needed to protect the failed systems' uncommitted updates may be retained in one or more other systems [MoNa91, MoNa92a]. In such an event, the still-operational systems will continue to be able to access the *rest of the data*. By using our method, even as the failed system is recovering, we would be able to allow new transaction processing to begin on the recovering system. Depending on the level of sharing that was in effect at the time of the system failure, the granularity at which the other systems retain the failed system's locks may vary all the way from table level to the record level, even if the locking being done by the transactions at the failed system was at the record level. The coarser the granularity at which the locks are retained the more beneficial our method would be.

It should be noted that typically the locks are retained by the other systems with the failed *system* being identified as the owner of those locks rather than by using the identifiers of the individual *transactions* which caused those locks to be acquired [MoNa91, MoNa92a]. As a result, the retained *logical* locks cannot be released until the failed system finishes its undo pass and the retained *physical* locks cannot be released until the failed system finishes its redo pass [MoNa91]. ***Physical*** locks are acquired to ensure that at any given time only one system is modifying a given page. That is, the physical locks are used to assure coherency of the data in the different systems' buffer pools. ***Logical*** locks are used to perform the more traditional concurrency control amongst the different transactions.

If the physical locks are retained at the page level, then the list of such locks can be used to generate DPL and our method can be applied to allow new transaction processing during the redo pass. DPL will consist of exactly those pages for which physical locks have been retained. Under these conditions, if the logical locks had been retained at the same granularity at which the transactions were acquiring those locks, then we can use our method by pre-

tending that all the retained logical locks were acquired by a *system* transaction at the failed system and by treating the Undo_Pass Condition to be TRUE. What this means is that a new transaction will be able to access a page and make progress during the redo pass if the page is not in DPL and if the lock needed by the new transaction is not one of the retained locks. If either condition is not true, then the new transaction would wait for the retained locks to be released by the *system* transaction.

On the other hand, if the logical locks had been retained at a coarser granularity than the granularity at which the transactions were acquiring those locks, then we can use our method by using the techniques (Commit_LSN, UOL, DOL, generating DPL via log analysis, ...) described earlier for the single system (nonSD) case.

2.4. Discussion

The overheads of our method during *normal* processing are:

- the extra check on every page access to see if restart is in progress
- the manipulations of the UUD_Bit

We consider these overheads to be insignificant since only simple comparisons of values are involved.

The extent of the benefit of our method for executing new transactions during the undo pass would depend on the particular mix of transactions that is run by the users of a given DBMS installation. This is something that the DBMS would have no control over. The higher the number and the longer the duration of update transactions, the more would be the benefit to be derived. Under these circumstances, new transaction processing will start much earlier with our method than otherwise. As we mentioned in the introduction, our motivation for doing the work reported here stems from our knowledge of some customers who very regularly produce many tapes worth of log records as a result of the execution of a *single transaction*! For such users, our method would be of immense value.

The extent of the benefit of our method for executing new transactions during the redo pass would depend on the speed at which the buffer manager writes dirty pages to disk. Of course, there is a trade-off here between reducing restart redo work versus impacting in a negative manner normal transaction processing work. The more quickly the buffer manager writes the pages, the lesser would be the amount of redo work to be performed in case of a system failure. But then, if there is any locality of reference amongst a set of pages across different transactions, then the quick writing of dirty pages would not allow us to amortize the cost of disk write of a page across multiple updates to the same page by different transactions or by a single long transaction. Frequent writing of hot spot pages would also cause concurrency problems if a page is not going to be allowed to be modified when it is being written to disk. Frequent writes may also impact negatively on being responsive to read I/O operations. Systems like DB2 delay doing writes also in order to accumulate multiple dirty pages for a single file so that the capability of the operating system to write multiple pages using a single *start I/O* command could be exploited to reduce the CPU and I/O overheads [TeGu84]. For reasons like these and also to reduce the lock holds times, all the IBM RDBMSs and many others follow the no-force buffer management policy.

3. Related Work

In [MHLPS92], we discussed some techniques to reduce the time spent in restart recovery processing.[4] Basically, they involved exploiting parallelism during the redo pass and subsequently during the undo pass. Essentially, these permitted I/O parallelism during the numerous I/Os that have to be performed during the redo and undo passes, and CPU parallelism during the processing of log records for different pages (during redo) and for different transactions (during undo). Unfortunately, during the undo pass, all the log records of a single transaction have to be processed by a single process in order to chain the CLRs properly. Similarly, during the redo pass, all the log records relating to a particular page must be processed by a single process to ensure that redo actions are performed in chronological sequence. To lessen the impact of these unfortunate situations and to improve data availability even further, in this paper, we introduced ways to permit parallelism between recovery processing and new transaction processing.

4 Those techniques were extended to the remote backup context in [MoTO93].

In IBM's IMS/XRF [IBM87] and Tandem's NonStop architecture [Tand87], on the failure of a primary system, a *hot standby* (the backup system) takes over and it first reacquires, as part of the redo pass, the necessary locks to protect *all* the uncommitted updates. This is possible since the backup system on a continuous basis keeps monitoring the primary, keeps analyzing the log records written by the primary and accumulates in virtual storage information about the transactions that are active in the primary and their update activities. Of course, the cost of doing this is very high in terms of extra processing capacity needed in the backup and in the primary. In NonStop, the primary sends the log records directly to the backup, thereby imposing extra overhead in the primary system. In IMS, the backup reads directly from disk the log records written by the primary. This increases the contention on the log disk, thereby impacting the processing on the primary.

When the XRF support was introduced in IMS, existing log records' contents had to be enhanced to identify what lock had to be reacquired in order to protect a given log record's update. The previously existing contents were not sufficient to infer the lock name. In DB2, on the other hand, from the beginning there was enough information available in the log records to determine the lock names. In fact, for in-doubt transactions, DB2 reacquires locks by accessing all their log records during the redo pass [Crus84, MHLPS92]. In contrast, in SQL/DS and R* [MoLO86], locks held by in-doubt transactions are included in their *prepare* log records. The information available in the update log records is not good enough to reacquire all the locks (e.g., no log records are written for index page changes and the *next key* lock names (see [Moha90b, MoLe92]) cannot be computed using solely the information in the log records for the data page changes). Even in ARIES/IM, which is implemented in DB2/2 and DB2/6000, and which does log index changes, such information is not available in the index log records.

On a takeover, in IMS/XRF and NonStop, *once the redo pass is completed*, the backup then starts processing new transactions in parallel with the undo pass. If no backup system is defined or it is not currently operational, then new transaction activity is begun only after the failed system is completely recovered (i.e., only after the undo pass is also completed). In spite of all the extra developmental and run-time expenses incurred in supporting the concept of hot standby in IMS/XRF and NonStop, it should be noted that neither system allows the processing of new transactions during the redo pass, unlike in our much more cost-effective method. Since we are not reacquiring locks to protect the uncommitted updates, naturally we cannot support as much concurrency during the undo pass as IMS or NonStop can. Of course, if one is willing to pay the price of reacquiring locks for in-flight transactions' updates during the redo pass, which may involve starting the redo pass from an earlier point in the log and scanning many more log records, in our method also we can support the same amount of concurrency during the undo pass. Our method would still be better since it would support new transaction processing during the redo pass.

In CICS/XRF [IBM89, ScRi88], the backup system does *not* continuously monitor the log records written by the primary. It only tracks the states of the terminals that are connected to the primary so that on a takeover the terminals can be quickly switched to use the backup sessions established to the backup system. As a result, on a takeover, the backup performs data recovery *completely* before permitting new transaction activity. Since CICS follows the force policy, the recovery work involved is just rolling back the in-flight transactions (i.e., there is no redo pass). In spite of that, in some of the measurements presented in [ScRi88], it has been reported that on an IBM 4381-2 machine 11% of the takeover time involved the undo pass and on an IBM 3084-QX it was 19%. One of the primary reasons for these high percentages is that undo activities are highly I/O bound. Typically, the I/Os would be random ones necessitating significant disk arm movements. Also, opening all the files on which recovery needs to be performed takes a significant amount of time. These numbers should allow us to conclude that in a system which has to perform redo also, the percentage of time spent on data recovery would be more and that our method could be beneficial in a significant manner in the hot standby context as well as in the no-standby context. More modern systems like the DB2 family of products [Moha93b] follow the no-force policy. Hence the need for performing redo in those systems. Even in IMS, for Fast Path data, redo may be necessary since for such data a force *after* commit policy is followed [MHLPS92].

4. Conclusions

We presented a cost-effective method for improving data availability during restart recovery of a DBMS after a failure. The method achieves its objective

by enabling the processing of new transactions to begin even before restart recovery is completed. A partial implementation of the new method would enable new transaction processing to begin only at the start of the undo pass. A more complete implementation would enable it to begin at the start of the redo pass itself. The overhead imposed by our method during normal transaction processing is insignificant since very little additional processing is needed. The method does not require nonvolatile memory to accomplish its goals. Our method has applicability even if the DBMS supports the hot standby concept and/or the shared disks environment. It supports fine-granularity (e.g., record) locking with semantically-rich lock modes and operation logging, partial rollbacks, write-ahead logging, and flexible buffer management policies. The method is easy to implement and requires few changes to an existing DBMS that uses the ARIES recovery method or a similar method. It is flexible in the sense that, depending on the affordable development cost, it can be implemented to different degrees of sophistication. It takes advantage of the information which is logged by the buffer manager at the time of a checkpoint and the Commit_LSN concept which we developed originally for reducing locking overhead and for increasing concurrency during normal processing. We compared our method with the techniques employed in IBM's IMS/XRF and CICS/XRF, and Tandem's NonStop architecture.

We can extend our method's effectiveness during the redo pass even further by associating with each page in the dirty page list (DPL) the LSN of the last log record (LastLSN) for that page. This information can be brought up to date during the analysis pass. With this additional information in hand, during the redo pass, a page can be removed from DPL when the page_LSN is found to have become equal to LastLSN.

We have extended the results of this paper to the remote backup context in [MoTO93]. As a result, when the primary system fails, even as a remote backup is taking over, processing of new transactions can be initiated.

Acknowledgements I would like to convey my thanks to Hamid Pirahesh and Julie Watts for their feedback and discussions.

5. References

CKKS89 Copeland, G., Keller, T., Krishnamurthy, R., Smith, M. *The Case for Safe RAM*, **Proc. 15th International Conference on Very Large Data Bases**, Amsterdam, August 1989.

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

FZTCD92 Franklin, M., Zwilling, M., Tan, C.K., Carey, M., DeWitt, D. *Crash Recovery in Client-Server EXODUS*, **Proc. ACM SIGMOD International Conference on Management of Data**, San Diego, June 1992.

Haer88 Haerder, T. *Handling Hot Spot Data in DB-Sharing Systems*, **Information Systems**, Vol. 13, No. 2, p155-166, 1988.

IBM87 *IMS/VS Extended Recovery Facility (XRF): Technical Reference*, **Document Number GG24-3153**, IBM, April 1987.

IBM89 *CICS/MVS Version 2.1 XRF Guide*, **Document Number SC33-0522-1**, IBM, March 1989.

LeCa87 Lehman, T., Carey, M. *A Recovery Algorithm for a High-Performance Memory-Resident Database System*, **Proc. ACM-SIGMOD International Conference on Management of Data**, San Francisco, May 1987.

Levy91 Levy, E. *Incremental Restart*, **Proc. 7th International Conference on Data Engineering**, Kobe, April 1991.

Lome90 Lomet, D. *Recovery for Shared Disk Systems Using Multiple Redo Logs*, **Technical Report CRL 90/4**, DEC Cambridge Research Laboratory, October 1990.

Lome92 Lomet, D. *MLR: A Recovery Method for Multi-Level Systems*, **Proc. ACM SIGMOD International Conference on Management of Data**, San Diego, June 1992.

MHLPS92 Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, **ACM Transactions on Database Systems**, Vol. 17, No. 1, March 1992. Also available as **IBM Research Report RJ6649**, IBM Almaden Research Center, January 1989; Revised November 1990.

Moha90a Mohan, C. *Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990. Also available as **IBM Research Report RJ7344**, IBM Almaden Research Center, February 1990.

Moha90b 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.

Moha93a Mohan, C. *ARIES/LHS: A Concurrency Control and Recovery Method Using Write-Ahead*

Logging for Linear Hashing with Separators, **Proc. 9th International Conference on Data Engineering**, Vienna, April 1993. A longer version is available as **IBM Research Report RJ8682**, IBM Almaden Research Center, March 1992.

Moha93b Mohan, C. *IBM's Relational DBMS Products: Features and Technologies*, **Proc. ACM SIGMOD International Conference on Management of Data**, Washington, D.C., May 1993.

Moha93c Mohan, C. *A Survey of DBMS Research Issues to Support Large Tables*, To appear in **Proc. 4th International Conference on Foundations of Data Organization and Algorithms**, Chicago, October 1993.

MoHa93 Mohan, C., Haderle, D. *Algorithms for Space Management in Transaction Systems Supporting Fine-Granularity Locking*, **IBM Research Report**, IBM Almaden Research Center, June 1993.

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.

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 available as **IBM Research Report RJ5037**, IBM Almaden Research Center, February 1986.

MoNa91 Mohan, C., Narang, I. *Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment*, **Proc. 17th International Conference on Very Large Data Bases**, Barcelona, September 1991. A longer version is available as **IBM Research Report RJ8017**, IBM Almaden Research Center, March 1991.

MoNa92a Mohan, C., Narang, I. *Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment*, **Proc. International Conference on Extending Data Base Technology**, Vienna, March 1992. Also available as **IBM Research Report RJ8301**, IBM Almaden Research Center, August 1991.

MoNa92b Mohan, C., Narang, I. *Data Base Recovery in Shared Disks and Client-Server Architectures*, **Proc. 12th International Conference on Distributed Computing Systems**, Yokohama, June 1992. Also available as **IBM Research Report RJ8685**, IBM Almaden Research Center, March 1992.

MoNa93 Mohan, C., Narang, I. *ARIES/CSA: A Method for Data Base Recovery in Client-Server Architectures*, **IBM Research Report**, IBM Almaden Research Center, June 1993.

MoPi91 Mohan, C., Pirahesh, H. *ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method*, **Proc. 7th International Conference on Data Engineering**, Kobe, April 1991. Also available as **IBM Research Report RJ 7342**, IBM Almaden Research Center, February 1990.

MoTO93 Mohan, C., Treiber, K., Obermarck, R. *Algorithms for the Management of Remote Backup Data Bases for Disaster Recovery*, **Proc. 9th International Conference on Data Engineering**, Vienna, April 1993. A longer version is available as **IBM Research Report RJ7885**, IBM Almaden Research Center, December 1990; Revised June 1991.

Rahm91 Rahm, E. *Recovery Concepts for Data Sharing Systems*, **Proc. 21st International Symposium on Fault-Tolerant Computing**, Montreal, June 1991.

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

Reut86 Reuter, A. *Load Control and Load Balancing in a Shared Database Management System*, **Proc. International Conference on Data Engineering**, February 1986.

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 of this paper is available as **IBM Research Report RJ6650**, IBM Almaden Research Center, January 1989.

ScRi88 Scott, J.W., Richards, A.J.M. *CICS/MVS Extended Recovery Facility (XRF) Performance*, **Document Number GG66-0281**, IBM Washington Systems Center, June 1988.

Ston87 Stonebraker, M. *The Design of the POSTGRES Storage System*, **Proc. 13th International Conference on Very Large Data Bases**, Brighton, September 1987.

Tand87 The Tandem Database Group *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL*, In **Lecture Notes in Computer Science Vol. 359**, D. Gawlick, M. Haynie, A. Reuter (Eds.), Springer-Verlag, 1989.

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