Data Base Recovery in Shared Disks and Client-Server Architectures

C. Mohan, Inderpal Narang

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

Abstract This paper presents solutions for the problem of performing recovery correctly in shared disks (SD) and client-server (CS) architectures. In SD, all the disks containing the data bases are shared amongst multiple instances of the DBMS. Any DBMS instance in the complex may directly access and update any data. Each instance maintains its own log and buffer pool. The local logs are later merged for media recovery purposes and, possibly, for restart recovery purposes. In CS, the server manages the disk version of the data base. The clients, after obtaining data base pages from the server, cache them in their buffer pools. Clients perform their updates on the cached pages and produce log records. The log records are buffered locally in virtual storage and later sent to the single log at the server. In write-ahead logging (WAL) systems, a monotonically increasing value called the log sequence number (LSN) is associated with each log record. Every data base page contains the LSN of the log record describing the most recent update to that page. This is required for proper recovery after a system failure. We describe a technique with some valuable features (e.g., avoiding reading empty pages and supporting the Commit_LSN optimization) for generating monotonically increasing LSNs in SD and CS architectures without using synchronized clocks.

1. Introduction

One approach to improving the capacity and availability characteristics of a single-system data base management system (DBMS) is to use multiple systems. There are two major architectures in use in the multisystem environment: ***shared disks*** (***SD***) or also called ***data sharing*** [DIRY89, Rahm91, Shoe86], and ***shared nothing*** (***SN***) or also called ***partitioned*** [Ston86]. But, in the context of object-oriented DBMSs (***OODBMSs***), the ***client-server*** (***CS***) architecture has become extremely popular [CFLS91, DMFV90, WaRo91, WiNe90]. There are quite a few similarities between SD and CS. We next provide a brief introduction to SD, SN and CS.

1.1. Shared Disks (SD) Architecture

The shared disks (SD) architecture is illustrated in Figure 1. Here, 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. A single transaction executes in only one system since all the data that it needs is available *locally*. SD is the approach used in IBM's IMS/VS Data Sharing product [PeSt83], TPF product [Scru87] and the Amoeba research project [MoNa91, MoNa92, MoNP90, MoNS91], and in DEC's VAX DBMS[1] and VAX Rdb/VMS[1] [ReSW89]. For the VAXcluster environment, third-party DBMSs like ORACLE and INGRES have been modified to support data sharing. Hitachi and Fujitsu also have products which support SD. SD has also, of late, become popular in the area of distributed virtual memory [NiLo91].

1.2. Shared Nothing (SN) Architecture

In SN, each system *owns* a portion of the data base and only that portion may be directly read or modified by that system. That is, the data base is *partitioned* amongst the multiple systems. The kind of synchronization protocols mentioned before for SD are not needed for SN. Depending on where it begins and what its data requirements are, a transaction might have to execute at multiple systems since not all the data is available *locally* as in SD. A transaction accessing data in multiple systems would need a form of two-phase commit protocol (e.g., the current industry standard Presumed Abort protocol of [MoLO86]) to coordinate its activities. SN is the approach taken in Tandem's NonStop SQL[1] [Pong90], Teradata's DBC/1012[1] [Nech86], MCC's Bubba [BACCD90], and the University of Wisconsin's Gamma [DGSBH90]. There are many advantages and disadvantages with both SD and SN [PMCLS90, Shoe86, Ston86].

Our intention in this paper is not to argue the relative merits of the SD and SN approaches. Even though major products have come out which support SD or SN, the debate still goes on. Of late, algorithms developed originally for use in the SD architecture have become very relevant to the areas of distributed virtual memory, distributed shared memory and CS architectures. Here, we concentrate on the problems relating to SD and CS. Our ideas should also apply to distributed, recoverable file systems in the SD environment.

1.3. Client-Server (CS) Architecture

The CS architecture has become very popular in the OODBMS area. Products like ObjectStore[1], VERSANT[1] and ONTOS[1] support it. Unfortunately, very little has been published on the algorithms employed by those systems.

---

1 DB2, IBM and OS/2 are trademarks of 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. Transarc is a registered trademark of Transarc Corp. Encina is a trademark of Transarc Corp. VERSANT is a trademark of Versant Object Technology Corp. Object Design and ObjectStore are registered trademarks of Object Design, Inc. ONTOS is a trademark of ONTOS, Inc.

0-8186-2865-0 /92 $3.00 © 1992 IEEE

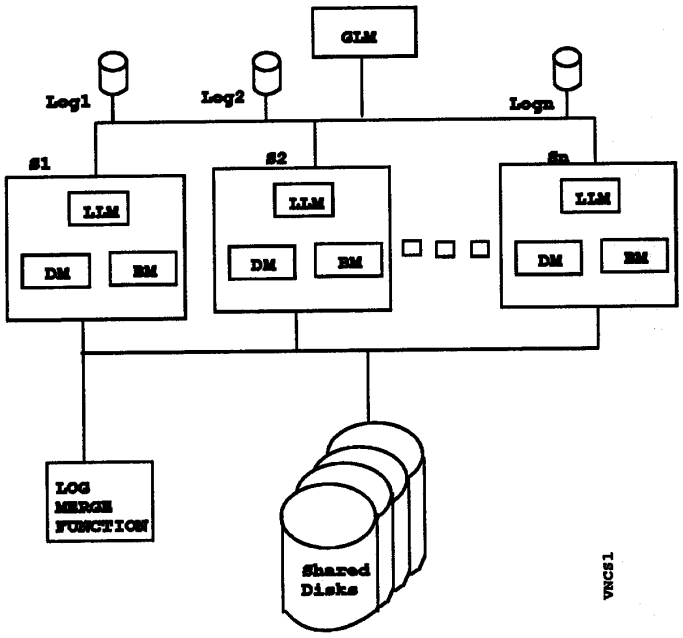

Figure 1: Shared Disks Architecture

While the existing papers on CS [CFLS91, DMFV90, WaRo91, WiNe90] discuss concurrency control, they do not deal with the recovery problem.

In CS, the server manages the disk version of the data base and takes care of global locking across the clients. The clients, after obtaining data base pages from the server, cache them in their buffer pools. Clients perform their updates on the cached pages and produce log records. The log records are buffered locally and later sent to the single log at the server. The problems of keeping the caches coherent, and performing global locking and recovery are very similar to those in SD. Most of the algorithms that we have developed for SD [MoNa91, MoNa92, MoNP90, MoNS91] are very much applicable to CS. For the purposes of this paper, the specifics of the protocols used for concurrency and coherency control are not that important. We will concentrate on some issues relating to recovery.

1.4. Recovery in Single-System DBMSs

Many DBMSs, such as IBM's DB2[1] [Crus84] and Tandem's NonStop SQL, do not write an updated page to disk at the time of termination (i.e., after commit or abort) of the transaction which performed the update. This is termed the ***no-force*** policy. This is to be contrasted with the force policy adopted by IBM's IMS, and DEC's VAX DBMS and VAX Rdb/VMS. No-force improves transaction response time and concurrency, and reduces I/O and CPU overheads, and the utilization of I/O devices. In many transaction systems that use write-ahead logging (***WAL***) for their recovery (e.g., in DB2 and the OS/2 Extended Edition Database Manager[1]), every update to every page gets logged and the page header has a field called ***page_LSN*** which contains what is called the ***log sequence number*** (***LSN***). The log records are written in time ordered sequence in *one (conceptual) log* and the log component assigns an LSN to every log record. The LSN is a monotonically increasing number and is generally the *logical* address of the log record. On performing an update of a page, the page_LSN field is set to the LSN of the log record describing that update.

The LSN value is used today for different purposes in the non-shared-disks (***nonSD***) systems, as described below:

- As a way of relating, for restart recovery purposes, the state of a page on disk with respect to updates that have been logged for that page. That is, given a log record and the page for which that log record was written, the recovery manager needs to know whether or not that page already contains that log record's update. Note that this means that only the LSN *for a given page* needs to monotonically increase over time as the page is updated.
- For use by the buffer manager to ensure that the ***WAL protocol*** is followed - a modified page is not written to disk *before* all the log records that describe changes to that page have been written to stable storage. That is, given the page_LSN of a *dirty* page, the buffer manager needs to know the *address* of the log record that describes the *most recent* change to the page.

During recovery from a system failure, transaction systems go through different stages of processing of the log. For example, consider the ARIES transaction recovery and concurrency control method which was introduced in [MHLPS92] and which has been implemented, to varying

degrees, in the IBM products OS/2 Extended Edition Database Manager, Workstation Data Save Facility/VM and DB2, in the IBM Research prototypes Starburst and QuickSilver, in Transarc's Encina[1] product suite, and in the University of Wisconsin's Gamma [DGSBH90] and EXODUS. ARIES takes the following actions, besides other actions, at the time of restart recovery:

- During the redo pass, appropriately positions for a forward scan of the log so as to encounter the log records for the *potentially* unapplied updates (to the disk version) of all objects (files, tables, ...) that were open at the time of the system failure.

- As each log record is encountered, reads the referenced data base page from disk and applies the update of the log record, if the LSN of the log record is *greater than* the page_LSN.

1.5. Recovery in SD Architecture Systems

Consider Figure 1's SD architecture where each system has its own local log, as in IBM's IMS Data Sharing product.[2] In such an architecture, whenever a log record is written because of an update in one system, the LSN of the log record would be assigned independently by that system - i.e., without taking into account the logging that is being done in the other systems. For example, let the LSN be the *logical* address of the log record in the *local* log file, as is the case in DB2 which does not currently support the SD architecture. Therefore, the LSN will not necessarily be a monotonically increasing number *across systems* (it will be monotonically increasing within a system, however). In this type of an architecture, when a page is updated in different systems over time, storing such an LSN in the page header may result in an incorrect data base state, when a system goes through restart recovery. The following example shows that.

Transaction T2 in system S2 updates page P1, writes a log record with LSN = 10, and sets P1's page_LSN to 10. Another transaction T1 in system S1 wants to update P1, but it is made to wait for T2 to commit since page locking is being assumed. When T2 commits, S2 writes P1 to disk and *simultaneously* transfers it to S1 (this is the *medium* page transfer scheme of [MoNa91]). As a consequence, page_LSN in the disk version of P1 will be 10. Next, T1 updates P1, writes a log record in its system with LSN = 9, modifies page_LSN to be 9 in its buffer version of P1, and commits. Then, S1 crashes. P1 is not written to disk before S1 crashes. During restart recovery of S1, S1 will not redo the update of the log record with LSN = 9 because the disk version of P1 has a page_LSN of 10 which is greater than the LSN of the log record. *As a result, a committed update will be missing in the page*, thereby violating the transaction durability property.

1.6. Recovery in CS Architecture Systems

In CS, since there is no local log, *all* the log records so far produced by the client are sent to the server when *dirty* data base pages are sent back to the server or when the transaction commits, whichever happens earlier. The client will normally buffer log records for a period of time in virtual storage before shipping them to the server. Clearly, in this environment, one cannot afford to wait for the log records to be sent to the server and for the server to respond back with LSNs for the newly written log records before the updated pages' page_LSN fields are set to the correct values. We would like to be able to assign the LSNs *locally* even as the log records get buffered in the client. If every client were to assign LSNs independently without each one somehow taking into account the values being assigned in the other systems, then errors like the one discussed before for the SD architecture can arise in CS also. This can happen because, over time, *different* clients may modify the *same* page.

The rest of this paper describes how the page_LSN field is assigned values to ensure that correct page recovery is possible in the SD and CS architectures, while at the same time retaining some existing optimizations and not increasing the overheads significantly. We also describe how the buffer manager is provided enough information for it to enforce the WAL protocol. The rest of the paper is organized as follows. In section 2, we discuss more the problems involved in assigning and using LSNs in the SD and CS architectures. In section 3, we describe a technique which solves those problems. Then, in section 4, we compare our approach with related work. Finally, we summarize in section 5.

2. Problems

First, we show that in the SD architecture, if each system has its own log file, and the LSN is equated to a log record's logical address and is assigned independently in each system, then the following problems related to the page_LSN exist. We list some of the points relating to LSNs and the questions that they give rise to in the SD and CS architectures.

1. *Page_LSN on disk*

The page_LSN value in the disk version of a page is used as follows:

1.a. To determine whether or not there is a need to redo the updates of a log record to recover the page to bring it up to date with respect to the log during the *redo* pass of restart recovery [Crus84, MHLPS92].

How do we handle this requirement, if the log record's address when used as the LSN value is not necessarily a monotonically increasing number across the different systems sharing the data base?

1.b. As a (very conservative) starting point in the log to recover the page to bring it up to date.

If the log records representing updates to a particular page are scattered in different systems' local log files (SD only), where do we start to recover the page and how do we merge the log records from the different local log files?

2. *Page_LSN of the buffer version of a dirty page*

In DB2 and the OS/2 Extended Edition Database Manager, for enforcing the WAL protocol, page_LSN of the buffer version of a *dirty* page is used to *force* (i.e., write to disk)

---

2 In DEC's VAXcluster environment, the VAX DBMS and VAX Rdb/VMS have only one global log on a shared disk for use by all the sharing systems (see the section "4.1. DEC VAXcluster"). In the absence of a log server, a single log that could be written into by any system directly leads to inefficient usage of resources because of the need for global synchronization. With high transaction rates, it would also become a bottleneck.

the log up to the log record of the last update to that page *before* the page is allowed to be written back to disk.

How should the WAL protocol's requirement be handled in SD and CS?

3. *Assigning the page_LSN value to a reallocated page*

In a DBMS such as DB2, when a previously deallocated page is reallocated, a slot is assigned to it in the buffer pool and the slot is formatted according to the type of the page being allocated. That is, the deallocated version of the page is *not* read from disk. This is a very useful optimization since it saves a *synchronous* I/O for reading a page whose contents are not of interest. Currently, during such formatting, the page_LSN is set to the address of the page-formatting log record. In SD, since the logical log addresses are not monotonically increasing across all the systems, how should this value be assigned?

An example of the above kind of page is an empty index page [Moha90a, MoLe92] which is reused. An index page is deallocated when there are no keys left in the page and is then reallocated during a subsequent page split operation. During reallocation, the page is not read from disk. In a single system environment, during allocation, setting the page_LSN to the current end of log address ensures that this value would be greater than the page_LSN value assigned when the page was deallocated. But, in the SD and CS architectures, the page may be deallocated in one system and then reallocated in another. If the architecture is SD with local logs, then it is possible that the second system's log is growing slower than the first system's log. In this case, we could suffer from the kind of problem illustrated in the last section. So the question is: how do we ensure, *without reading the page from disk*, that the value assigned to the page_LSN during reallocation is always greater than any page_LSN value previously assigned to that page?

4. *Use of a page_LSN to decide if all data on page is committed*

In [Moha90b], a very simple technique, called ***Commit_LSN***, was introduced to cheaply determine if all the data on a page was committed. This method uses the LSN of the first log record of the *oldest* update transaction still executing in the system (which is called Commit_LSN) to infer that all the updates in pages with page_LSN *less than* Commit_LSN have been committed. This simple observation turns out to be a very powerful one. This method reduces locking and latching, especially for transactions desiring the degree of consistency called cursor stability (degree 2 in System R terminology). In addition, the method may also increase the level of concurrency that could be supported and decrease the need for reevaluation of selection predicates. Several applications of the method, including many nontrivial ones, were presented in [Moha90b]. Recently, a totally different application of the method was presented in [Moha91] to allow access to data to new transactions even while recovery from a system failure is in progress.

The Commit_LSN method crucially depends on a log record's LSN being smaller than those of all the log records written after the writing of the former log record. In the single system and SN environments this is trivially satisfied if the LSN is a log address. In the SD and CS environments if (synchronized clocks') timestamps are not used as LSNs, then the alternative method used to determine LSNs must satisfy this property.

In the SD and CS architectures, since the Commit_LSN value must be computed by taking into account all the transactions executing across the whole complex of systems, it is also important that the LSNs issued by the different systems be as close to each other as possible, even though the log production rate at the different systems may be very different. While no inconsistency will arise if one ore more systems keep issuing low LSNs, the smaller values will unnecessarily prevent transactions from benefitting from the Commit_LSN idea since the low LSNs will keep the global Commit_LSN value too much in the past and the conservative check (*is page_LSN less than Commit_LSN*) will fail more often, and the transactions will be forced more often to obtain locks to determine whether a piece of data is committed.

3. Solutions

For the LSN assignment problem, the solution that might come to mind immediately is the one that requires that all the systems have synchronized clocks and that then uses the clock value as the LSN. We did not want to require that the clocks across the systems be synchronized. We also felt that if a solution could be found which did not require clock synchronization that would give us some flexibility (e.g., avoiding a problem like the following: if the timestamp's size is longer than the currently in-use page_LSN field's size, then requiring the unloading of the data, the reformatting of the pages and the reloading of the data).[3] As far as we know, even in the DEC VAXcluster, which has been in existence for a long time, the clocks are not synchronized close enough.

For SD, if at all possible, we also desired a solution which did not require the realtime merging of all the different systems' local logs for performing transaction rollbacks and restart recovery. Building a realtime log merge facility for a very high transaction throughput system is very expensive because of its high availability and performance requirements. In CS, the server, in addition to its other duties, performing the role of a log server also may cause it to become overloaded in due course of time. But, in the current usage of CS for OODBMSs, this is not yet a problem since it is not a high-volume, on-line transaction processing (OLTP) environment.

3.1. Assumptions

- Record locking is in effect, which means that a page may contain the uncommitted updates of many transactions concurrently executing in different systems.
- While it may sometimes be the case that clocks across an SD complex are synchronized, it is significantly less likely that that would be the case in CS, especially at the degree of precision that would be needed. Hence, we assume that clocks are not synchronized across the complex of systems both in SD and CS.

---

3 Even if the clocks were being synchronized, they would have to be synchronized within a microsecond and even then some special tricks would be necessary to make it work (see [MoNP90]).

• **SD Architecture** A page contains dirty updates of only one *system*. That is, when a modified page is transferred from one system to another, it is written to disk *before* another system is allowed to update it. This is one of the many schemes described in [MoNa91]. Note that this does *not* prevent record locking from being supported.

This ensures that only one system's log is needed for restart *redo* recovery. That is, a *realtime* merged log is not required. Note that undo may still require accessing multiple systems' local logs, but, in that case, the logs do not have to be merged for undo to be performed correctly. Note that our technique works whether or not the above assumption holds. The extra information that needs to be tracked if the above assumption is relaxed is described in [MoNa91].

• **CS Architecture** Each client periodically takes a checkpoint. The server keeps track of the most recent checkpoint records of all the clients. If a client were to fail, then the server performs recovery for it very much the same way single system failure is handled in SD (e.g., as it is done in ARIES for SD [MoNa91]). It does this by processing that client's checkpoint record, and by performing the analysis, redo and undo passes. Redo would be needed only for those pages for which the failed client had write locks. Even for some of those pages, redo would not be needed if the *server's* buffer pool already had the latest versions of those pages. If the server were to fail, then that would be handled similarly to the way an SD-complex failure is handled (see [MoNa91]).

The clients also have (local) log managers which behave very much like the regular log managers, except that, instead of writing log records to disk, they just buffer them in virtual storage and then at various points in time ship them to the server. When the server receives log records from a client, it *appends them*, as they are, to its log file. The log records written by a client contain the client's identity. This information would be used during an analysis pass for separating the log records written by a particular client from those written by the other clients.

3.2. Page_LSN Value on Disk

Now, we show how the page_LSN value is maintained and used for recovery and as to how we respond to the questions raised in the section "2. Problems".

3.2.1. Solution to Problem 1.a

Normal Processing

The page_LSN field is used as an ***update sequence number*** (***USN***). During normal processing, an update to the page involves the following:

• Look up the current value of the page_LSN field on the page being updated.

• While writing the log record describing the update, pass to the log manager the page_LSN value.

• The log manager will assign to the new log record the *higher* of the following two values:

1. 1 + the page_LSN passed as a parameter
2. 1 + Local_Max_LSN

Local_Max_LSN is the value that the log manager continuously maintains. It is typically the LSN of the most recently written log record in the log file.

The log manager will place the LSN so computed into the new log record's LSN field, assign the value to Local_Max_LSN and also return that value to the invoker of the log manager. Note that this method guarantees that for a particular system all the log records written by it will have LSNs which are monotonically increasing, even across log records for different *pages*.

• On return from the log manager, the page updater will place the returned LSN value in the page_LSN field.

Restart Processing

There is no change in the logic for redo: a log record's update is redone if the LSN value found in the log record is *greater than* the page's current page_LSN value.

3.2.2. Solution to Problem 1.b

The page_LSN value *cannot* be used as the starting point to scan the log to recover a page when the system is operational since it is no longer the address of a log record. There are 2 cases to consider:

1. The page has been in use in this system (e.g., process failure occurs during page update and page needs to be recovered from the disk version).
2. The page cannot be read from disk due to media error.

To handle the first case, the buffer manager tracks, in the buffer control block (***BCB***), the log record address of that update which causes the page state to go from *nondirty* to *dirty* (i.e., when the *first* update is performed to a *clean* page).[4] This *log address*, called ***RecAddr***, becomes the starting point for page recovery. The ending point for the log scan is the end of log address when the page recovery is initiated. The above RecAddr is also needed for recording, at the time of a checkpoint, a summary of the state of the buffer pool. In case a system failure were to occur, this information is needed for determining the point of the log from which the redo pass of restart recovery should begin [Crus84, MHLPS92].

In CS, the buffer manager at the client associates with each dirty page the LSN of the most recently written log record at that client just before that page became dirty. When this dirty page is sent to the server, this ***RecLSN*** information is also sent with it. The buffer manager at the server then maps the RecLSN to the corresponding RecAddr and stores the information in its BCB for that dirty page. If the server already had a dirty version of that page (i.e., earlier, the same or a different client had dirtied the page and that earlier dirty version has not yet been written to disk), then the buffer manager retains the **old** RecAddr.

If the page cannot be read from disk, the page is recovered by doing media recovery. Media recovery involves the following:

• Obtaining a copy of the page from the last image copy (archive dump).

4 In SD, a page is ***clean*** if the disk version is up to date with the buffer version. In CS, a page is clean if the client version is the same as the server version.

• **SD Architecture** Redoing updates of this page's log records from the logs of the different systems. This necessitates merging the log records from the different local logs by comparing the LSN fields in the log records. Two log records from two different local logs may have the same LSN. In this case, the merge logic can include the two log records into the merged stream in any order. This would not create problems since such log records could only be relating to two different pages. The latter is guaranteed since we ensure that, for a given page, all its log records are assigned monotonically increasing LSNs *across the whole complex*.

CS Architecture Media recovery requires accessing the log at the server. As explained before, since the log records produced by the clients are appended to this log as they are sent by the clients, there is no need for a special merge operation. Note that the server log's successive records may not always have increasing LSNs, unlike in the case of SD's merged log. This is not of any consequence with respect to recovery or the application of the Commit_LSN optimization since all the log records written by a single client will have increasing LSNs and since the LSNs assigned by the different clients are kept close together (see below).

3.3. Page_LSN Value in the Buffer Pool

For enforcing the WAL protocol, in the SD architecture, the *logical address* of the most recently written log record for a page is tracked in the BCB of that page. This value is needed only as long as the page is in the buffer pool.

In the case of the CS architecture, we assume that all buffered log records are sent to the server at the time any dirty page is sent back to the server or at the time a transaction is being committed, whichever happens earlier. At the server, whenever a dirty page is received from a client, the server's buffer manager can *conservatively* associate, with the received page, the logical address, in the server's log file, of the most recently written log record that came from that client. Alternatively, the server can associate with each received dirty page, the logical address of that log record whose LSN value is present in the page_LSN field of that page.

3.4. Page Reallocation

If a page is being reallocated without its old version being read from disk, then the new LSN assigned to the page_LSN field (e.g., while formatting the page and writing a log record) must be guaranteed to be *greater than* the LSN value which exists in the disk version of the page. The way we ensure this property is by making the new page_LSN be higher than the current LSN of the space map page (SMP) which describes the allocation-deallocation status of the page in question. We pass the LSN of the SMP as the LSN for the page being allocated. This works since at the time of deallocating a page, we would have to modify the SMP entry for that page to say that the page is available for future allocation. During that operation, because of the algorithm used by the log manager to assign LSNs, it will be ensured that the SMP update log record's LSN is *higher than* the latest LSN of the page being deallocated. During allocation, since we have to look at the SMP anyway (to even realize that the page is available for allocation and then to mark the page as being allocated), we can ensure that the LSN assigned for the format log record is higher than the current LSN of the SMP page.

3.5. Synchronizing LSNs Across Systems

In order make the Commit_LSN check succeed more often, it is important to make the LSNs issued by the different systems be reasonably close to each other in terms of their values, as discussed earlier. To accomplish this, we ensure that periodically all the systems are informed of the other systems' Local_Max_LSNs. As each Local_Max_LSN is received, the maximum of the received Local_Max_LSN and the current system's Local_Max_LSN is computed and the current system's log manager is informed about the new Local_Max_LSN so that log manager can update its data structure which keeps track of that value. This essentially amounts to a Lamport logical clock scheme [Lamp78]. To make the process efficient, the transmission of Local_Max_LSNs can be piggybacked onto the other messages being exchanged between the systems. With the above scheme in place and the Commit_LSN value is computed by taking into account the transactions executing across all the systems, it can be used as explained in [Moha90b, Moha91].

3.6. Advantages

The key advantages of our solutions are: (1) The clocks across the complex of systems do not have to be synchronized. (2) The solutions are applicable to the SD and CS architectures. (3) For the SD architecture, a realtime merged log is not required for restart recovery or transaction rollback. (4) They support the very powerful Commit_LSN optimization. (5) If a single system DBMS is being evolved to the multisystem environment, they avoid the need for migration of all existing data in case the timestamp value's length is more than the size of the currently present page_LSN field.

4. Comparisons With Existing Work

4.1. DEC VAXcluster

In the VAXcluster environment, DEC's VAX DBMS and VAX Rdb/VMS support SD using a global lock manager [ReSW89]. Those DBMSs also use something like our USN, even though they use only a single global log from all the systems. Their USN is not used for recovery purposes since those systems implement the force policy and migrate modified data to disk *before* commit is logged. USN is used only to implement buffer coherency.

Having a single log for direct use by all the systems becomes expensive since *every write* to the global log requires acquiring a *global lock* to serialize the space allocation in the log file. Acquiring a global lock involves sending and receiving messages. To avoid making every log record's write acquire this global lock, each transaction first places its log records in a process private log buffer and later transfers them to its system's main memory log cache which is shared across that system's transactions. A log force necessitated by the WAL protocol or transaction commit causes all the cached records to be placed into the global log which requires acquiring the global lock. This means that if two transactions in a system modified the same page in a certain order, then it is possible for

the log records representing those changes to be written into the local log cache in the opposite order. This method works only because of the force-before-commit policy and the physical nature of the logging and locking done by the DEC DBMSs. With the sophisticated features (like semantically-rich lock modes, and operation logging) permitted by recovery and/or concurrency control methods like ARIES [MHLPS92], ARIES/NT [RoMo89], ARIES-RRH [MoPi91], ARIES/KVL [Moha90a], ARIES for SD [MoNa91, MoNa92, MoNP90, MoNS91] and ARIES/IM [MoLe92], this type of reordering of log records is not acceptable. [ReSW89] does not explain how the buffer manager determines up to what point the log needs to be forced, if at all, to obey the WAL protocol, when a modified page needs to be written to disk.

4.2. Lomet's Scheme

The work that we have reported in this paper is an evolution of our initial work reported in [MoNP90]. In our prior work, we assumed the existence of tightly synchronized clocks and we considered only the SD architecture. During the time that we came up with the results presented in this paper, Lomet independently designed the scheme presented in [Lome90]. Lomet's work was based on our initial work and it also eliminated the need for synchronized clocks.

Lomet's algorithm assigns the new page_LSN for a particular page to be always one higher than the previously assigned value for that page. Furthermore, in each log record, instead of storing the LSN of that log record like we do, it stores the LSN of the corresponding data base page *before* the update described by the current log record was performed (Lomet calls them *before state identifiers* (BSIs)). In order to ensure this property for reallocation of empty pages, Lomet is forced to track in the space map entry for the deallocated page the exact page_LSN value of the deallocated page. The space overhead of this approach is enormous. In DB2, only one bit is used to track the allocated/deallocated status of index pages. Lomet's scheme would increase that overhead 47-63 times, depending on whether the LSN is a 6 byte or 8 byte quantity! The Lomet method crucially depends on the *before* LSN value being captured in the log record. Redo of a log record's update is performed only if the page_LSN is *equal to* the LSN stored in the log record.

The Lomet algorithm is even more inefficient when one considers operations like *mass delete* which DB2 supports very efficiently for segmented tablespaces in which records from different tables are not intermixed on a given data page [CrHT90]. That is, when all the records of a table are deleted using a set-oriented delete statement, DB2, instead of deleting one record at a time, just visits the space map pages and marks all the corresponding pages as being empty. None of the deallocated pages is read from disk. Log records are written only for the space map page changes. With the Lomet algorithm, this efficient implementation would not be possible since it needs to record the current LSNs of those *emptied* pages in the space map pages! It would require the expensive reads of all the pages. The alternative approach of, at the time of every update to a page, keeping the LSN information for that page in the space map page up to date would be prohibitively expensive.

The log merge logic of the Lomet method is also more complicated since it requires that both the page number field and the LSN field of the log records be compared to determine which local log's record should be sent to the merged stream next. This is necessary because successive log records, relating to different data base pages, from a single local log may have LSNs with lower as well as higher values. That is, there is no predictable pattern to the LSN values in the log records of a local log. With our method, since we ensure that all successive log records in a local log have higher and higher LSN values, the comparison for merging can be done simply, based solely on the LSN field.

Because of the above reasons, the Lomet scheme does not support the Commit_LSN optimization. On the other hand, our method does support it since it makes LSNs increase across all pages and keeps them close together across systems.

5. Summary

We presented solutions for the problem of performing recovery correctly in the shared disks (SD) and client-server (CS) architectures. We have described how it is possible to use the page_LSN field for page recovery in a multisystem shared disks architecture, where the use of the log address as the LSN will not guarantee a monotonically increasing number across systems since each system has its own log and the logs may grow at different rates. We described the changes required to (1) assign the page_LSN value when a reallocated page is formatted, (2) maintain the page_LSN value in normal processing when the page is updated, and (3) recover the page during restart and after a media failure.

The key advantages of our solutions are: (1) The clocks across the complex of systems do not have to be synchronized. (2) The solutions are applicable to the SD and CS architectures. (3) For the SD architecture, a realtime merged log is not required for restart recovery or transaction rollback. (4) They support the very powerful Commit_LSN optimization. (5) If a single system DBMS is being evolved to the multisystem environment, they avoid the need for migration of all existing data in case the timestamp value's length is more than the size of the currently present page_LSN field. Our solutions also work with the more sophisticated page transfer schemes described in [MoNa91] which rely on a realtime merged log. For brevity, we did not describe here the details of our solutions for those schemes. In SD, those schemes allow a dirty page to be transferred from one system to a second system for update without the page having to be written to disk in between. The most sophisticated scheme even allows the transfer to take place before the *log* is forced to disk in the first system.

6. References

BACCD90 Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M., Valduriez, P. *Prototyping Bubba, a Highly Parallel Database System*, **IEEE Transactions on Knowledge and Data Engineering**, Vol. 2, No. 1, March 1990.

CFLS91 Carey, M., Franklin, M., Livny, M., Shekita, E. *Data Caching Tradeoffs in Client-Server DBMS Architect*

ures, **Proc. ACM SIGMOD International Conference on Management of Data**, Denver, May 1991.

CrHT90 Crus, R., Haderle, D., Teng, J. *Method for Minimizing Locking and Reading in a Segmented Storage Space*, **U.S. Patent 4,961,134**, IBM, 1990.

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

DIRY89 Dias, D., Iyer, B., Robinson, J., Yu, P. *Integrated Concurrency-Coherency Controls for Multisystem Data Sharing*, **IEEE Transactions on Software Engineering**, Vol. 15, No. 4, April 1989.

DMFV90 DeWitt, D., Maier, D., Futtersack, P., Velez, F. *A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990.

Lamp78 Lamport, L. *Time, Clocks, and the Ordering of Events in a Distributed System*, **Communications of the ACM**, Vol. 21, No. 7, July 1978.

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

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

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

Moha90b Mohan, C. *Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990.

Moha91 Mohan, C. *A Cost-Effective Method for Providing Improved Data Availability* During *DBMS Restart Recovery After a Failure*, **Proc. 4th International Workshop on High Performance Transaction Systems**, Asilomar, September 1991. Also available as **IBM Research Report RJ8114**, IBM Almaden Research Center, May 1991.

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 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.

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.

MoNa92 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.

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.

MoNS91 Mohan, C., Narang, I., Silen, S. *Solutions to Hot Spot Problems in a Shared Disks Transaction Environment*, **Proc. 4th International Workshop on High Performance Transaction Systems**, Asilomar, September 1991. Also available as **IBM Research Report RJ8281**, IBM Almaden Research Center, August 1991.

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.

Nech86 Neches, P. *The Anatomy of a Data Base Computer - Revisited*, **Proc. IEEE Compcon Spring '86**, San Francisco, March 1986.

NILo91 Nitzberg, B., Lo, V. *Distributed Shared Memory: A Survey of Issues and Algorithms*, **Computer**, August 1991.

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. An expanded version is available as **IBM Research Report RJ7724**, IBM Almaden Research Center, October 1990.

Pong90 Pong, M. *An Overview of NonStop SQL Release 2*, **Tandem Systems Review**, Vol. 6, No. 2, October 1990.

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.

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.

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. 1, March 1986.

Ston86 Stonebraker, M. *The Case for Shared Nothing*, **IEEE Database Engineering**, Vol. 9, No. 1, 1986.

WaRo91 Wang, Y., Rowe, L. *Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Denver, May 1991.

WINe90 Wilkinson, K., Neimat, M.-A. *Maintaining Consistency of Client-Cached Data*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990.