ARIES/CSA: A Method for Database Recovery in 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 an algorithm, called ARIES/CSA (*Algorithm for Recovery and Isolation Exploiting Semantics for Client-Server Architectures*), for performing recovery correctly in client-server (CS) architectures. In CS, the server manages the disk version of the database. The clients, after obtaining database 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. ARIES/CSA supports write-ahead logging (WAL), fine-granularity (e.g., record) locking, partial rollbacks and flexible buffer management policies like *steal* and *no-force*. It does not require that the clocks on the clients and the server be synchronized. Checkpointing by the server and the clients allows for flexible and easier recovery.

1. Introduction

The ***client-server*** (***CS***) computing paradigm is becoming very pervasive in the computer industry. In addition to providing user-friendly interfaces, a client may perform data caching and updating for applications such as CAD, CASE, etc. Object-oriented database management systems (***OODBMSs***) support such applications in the CS architecture with such caching of data [CFLS91, Deux91, DMFV90, FrCL92, LLOW91, WaRo91, WiNe90]. Products like GemStone, ObjectStore™, VERSANT™, O_2™, and ONTOS™ support it. Unfortunately, very little has been published on the details of the recovery algorithms employed by those systems. Typically, the existing papers on CS discuss in some detail only concurrency control issues [CFLS91, DMFV90, WaRo91, WiNe90]. The exceptions are [FZTCD92, MoNa92b].

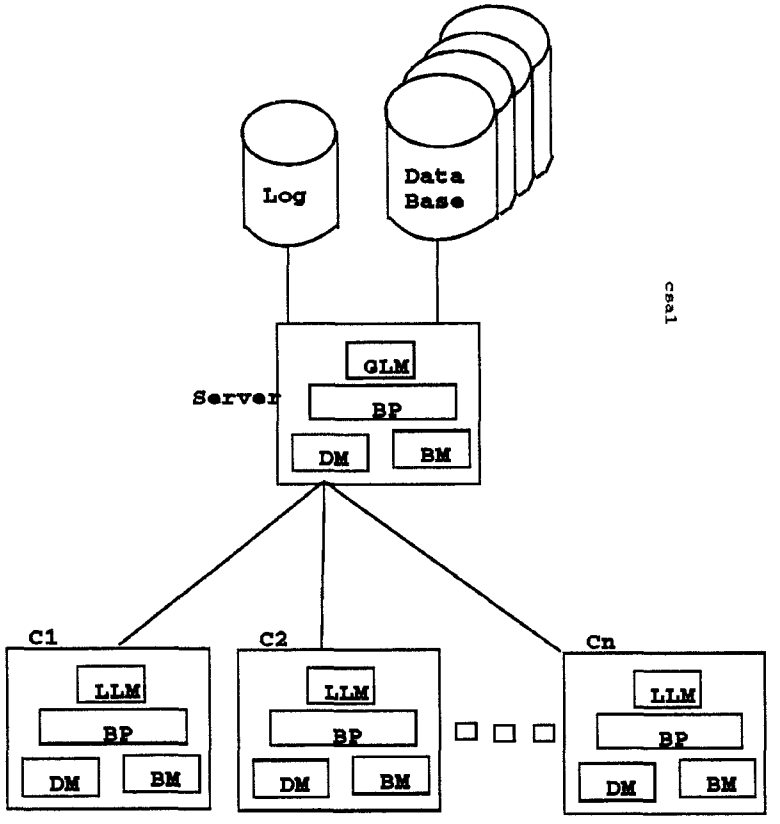
In our model of CS (see Figure 1), the server manages the disk version of the database and, possibly, takes care of global locking across the clients. The clients, after obtaining database pages from the server, cache them in their buffer pools. Clients perform their updates on the cached pages and produce log records. However, clients do not have disks for logging. They buffer log records locally in virtual storage and later send them to the single log managed by the server. The problems of keeping the caches coherent, and performing global locking and recovery are very similar to those associated with the shared disks (***SD***) architecture [DIRY89, Lome90, Rahm91]. Most of the algorithms that we have developed for SD [MoNa91, MoNa92a, MoNa92b, 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 issues and solutions relating to recovering from various types of failures when fine-granularity (e.g., record) locking is in effect.

The rest of the paper is organized as follows. In the remainder of this section, first, we describe how recovery is performed in a single-system DBMS to define the terminology used in the paper; then, we describe some of the problems which would need to be addressed in a CS architecture where a client is allowed to cache and update the pages received from the server. The rest of the paper describes how database recovery is performed correctly in the CS architecture, while at the same time not increasing overheads significantly and while retaining some optimizations currently applicable to the traditional DBMSs. In section 2, we first state our assumptions about the CS environment and then describe the ***ARIES/CSA*** (***ARIES for Client Server Architectures***) method. In section 3, we describe how log sequence numbers (LSNs) are synchronized in CSA to benefit from the Commit_LSN technique of [Moha90]. Then, in section 4, we compare our method with related work. Finally, we summarize in section 5.

1.1. Terminology for Recovery

We describe the terms which are used for recovery in the rest of the paper by describing the actions taken by a single-system DBMS during its normal processing, and its restart recovery processing after a crash.

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50



BM: Buffer Manager; BP: Buffer Pool; DM: Data Manager;
LLM: Local Lock Manager; GLM: Global Lock Manager.

Figure 1: Our Model of Client-Server Architecture

1.1.1. Normal Processing

In many DBMSs (e.g., in DB2™), every update to every page of the database gets logged. The log manager of the DBMS writes log records in time ordered sequence in *one (conceptual) log* and assigns a ***log sequence number*** (***LSN***) to every log record. An LSN is generally the *logical address* of a log record. Hence, LSNs monotonically increase in value. Every database page header has a field called ***page_LSN***. On performing an update of a page, the page's page_LSN field is set to the LSN of the log record describing that update.

The buffer manager component of the DBMS uses the page-LSN value of the page in order to make sure that 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, i.e., to implement the ***WAL (write-ahead log) protocol***. 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. This is straightforward if LSNs are log records' addresses.

Many DBMSs, such as IBM's DB2 and DB2/6000™, do not write an updated page to disk at the time of termination (i.e., after commit or rollback) 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 Rdb/VMS™ [ReSW89]. No-force improves transaction response time and concurrency, and re-

---

™DB2, DB2/2, DB2/6000, IBM and OS/2 are trademarks of International Business Machines Corp. DEC, VAXcluster and Rdb/VMS are trademarks of Digital Equipment 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. O_2 is a registered trademark of O_2 Technology.

duces I/O and CPU overheads, and the utilization of I/O devices [MHLPS92].

The LSN value is used as a way of relating the state of a page on disk with respect to updates that have been logged for that page during recovery from a DBMS crash. 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 multiple times.

During recovery from a crash, the DBMS must restore the database to a consistent state. To make this possible, some actions have to be taken during normal processing. For example, consider the ***ARIES*** (***Algorithm for Recovery and Isolation Exploiting Semantics***) 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, DB2, DB2/2™, DB2/6000, Message Queue Manager/ESA, Workstation Data Save Facility/VM and ADSM, in the IBM Research prototypes Starburst and QuickSilver, in Transarc's Encina™ product suite, and in the University of Wisconsin's Gamma and EXODUS. ARIES takes the following actions, besides other actions, during normal processing:

- It uses WAL for recovery. It does in-place updating in the buffer pool and disk versions of pages (i.e., it does not do shadowing or delayed updating).

- Periodically, it takes a checkpoint by first writing a *Begin_Checkpoint* log record, then collecting some information and finally writing an *End_Checkpoint* log record which contains that information. The collected information relates to pages that are more up to date in the buffers than on disk (***dirty*** pages) and to transactions that are in progress. For each page in the stored ***dirty pages list (DPL)***, there is an LSN called Recovery LSN (***RecLSN***, for short) which is stored in DPL and associated with the page. RecLSN is some LSN starting from which if a forward scan of the log were to be performed, then all the log records needed to bring the disk version of the page up to date with its buffer version (by possibly redoing all or some of those log records' updates) would be encountered. During normal processing, the buffer manager tracks RecLSN of a page in the buffer control block (***BCB***) for the page. The value is assigned when the page state is about to change from *clean*[1] to dirty. Typically, the current end-of-log LSN is picked conservatively as RecLSN, rather than the LSN of that log record which describes that update which makes the page become dirty.

1.1.2. Restart Recovery

During restart recovery after a crash, ARIES processes the log and takes the following actions:

- Performs the ***analysis pass*** which starts at the Begin_Checkpoint log record of the last ***completed checkpoint*** (i.e., a checkpoint for which both the Begin_Checkpoint and End_Checkpoint log records are present) and continues until the end of the log. During this pass, DPL and the transaction information found in the End_Checkpoint record are brought up to date as of the end of the log by analyzing the log records in that interval (see [MHLPS92] for a detailed description of how this is done). If a log record is encountered for a page that does not appear in DPL, then that page is added to DPL. For each such page, RecLSN is assigned to be the LSN of the first encountered log record referencing that page.

- DPL from the analysis pass determines the starting point (i.e., the ***RedoLSN*** = the *minimum* of the RecLSNs in DPL) for the log scan of the next pass (***redo pass***), and acts as a filter to determine which log records and consequently which database pages have to be examined to determine if some updates need to be redone. A page referenced in a log record is examined only if the page appears in DPL and the page's RecLSN is less than or equal to the log record's LSN. An update is redone only if the page qualifies for examination and page_LSN is less than the log record's LSN. The effect of the redo pass processing is that ARIES ***repeats history*** as recorded in the disk version of the log by redoing all those updates whose effects are missing in the disk version of the database.

- In the following ***undo pass***, using the transaction information produced by the analysis pass, ARIES rolls back those transactions that did not terminate or enter the *in-doubt* (*prepare*) state of two-phase commit. ARIES also logs, typically using compensation log records (***CLRs***), updates performed during partial or total rollbacks of transactions during both normal and restart undo. In ARIES, CLRs have the property that they are *redo-only* log records. By appropriate *chaining* of the CLRs to log records written during forward processing, a bounded amount of logging is ensured during rollbacks, even in the face of repeated failures during restart recovery or of nested rollbacks. When the undo of a log record (a nonCLR) causes a CLR to be written, the CLR is made to point, via the ***UndoNxtLSN*** field of the CLR, to the same transaction's log record that *precedes* (i.e., the one pointed to by the ***PrevLSN*** field) the

1 In the current context, a page at the *server* is clean if the buffer version at the server is the same as the disk version and a page at a *client* is clean if the client version is the same as the server version.

log record being undone. ARIES supports ***logical undo*** since the page affected during the undo of a nonCLR could be different from the page originally affected by the nonCLR's update (see [Moha93a, Moha94b, MoLe92] for examples and concurrency advantages of logical undos).

1.2. Recovery Issues in CS Architecture

In this section, we motivate the reader with some of the recovery issues which arise as a result of the CS architecture.

In CS, the clients have no local disks to store the log records for the updates. Only the server has the log disks. So, the following questions arise. The answers given here are expanded upon later in the paper.

- How does the client assign LSNs for the log records?

Clearly, in this environment, one cannot afford to wait for a log record to be sent to the server and for the server to respond back with an LSN for the newly written log record before the updated page's page_LSN field is set to the correct value. We would like to be able to assign LSNs *locally* even as log records are being created and are being buffered in the client.

This leads to the following question: if every client were to assign LSNs independently without each one somehow taking into account the values being assigned in the other systems, then how would recovery happen correctly (see [MoNa92b, MoNP90] for examples of problem scenarios)? A problem may arise because, over time, *different* clients may modify the *same* page and assign LSNs independently which may not be monotonically increasing.

- When does the client ship the log records to the server?

A client buffers log records for a period of time in virtual storage before shipping them to the server. All the log records produced by a client for its updates are sent to the server when dirty pages are sent back to the server or when a transaction commits, whichever happens earlier.

- How does the server enforce the WAL protocol when the log records and the page could be arriving independently at the server?

The server ensures that the log records corresponding to a dirty page's page_LSN are written to the stable storage before writing that page to disk. This may require keeping a mapping between log record addresses and their LSNs.

- Since dirty pages might move between the clients and the server quite unpredictably, how does the recovery method ensure that recovery is still performed correctly both for client and server failures? That is, how does the recovery logic ensure that pages which need recovery are correctly identified and also that all log records which need to be redone or undone are correctly identified and processed?

The server as well as the clients take checkpoints periodically to ensure correct operation.

2. ARIES/CSA Method

2.1. Assumptions

- A transaction executes in its entirety at a particular client or at the server. It is relatively easy to permit *reads* alone to be performed both at the server and a client by a single transaction. If a transaction needs to see its own updates, then even this requires more work since the updated pages in the client would have to be flushed to the server before a read is performed at the server. Permitting the same transaction to perform even updates at both places would make the recovery logic more complicated since any rollback of a transaction would have to be performed in reverse chronological order across the two systems.

- WAL is used for recovery. All newly produced log records currently buffered in a client are sent to the server just before any dirty page is sent back to the server (i.e., a conservative way of ensuring write-ahead logging with respect to the server) or at the time a transaction is being committed, whichever happens earlier.

- Global lock management support is provided by the server. Clients acquire global locks by communicating with the global lock manager (***GLM***) at the server. Each client may have a local lock manager (***LLM***) so that the global locks are acquired in the name of the LLMs rather than individual transactions. This would permit some optimizations which result in some message, CPU and storage savings, especially if multiple transactions were to execute *concurrently* at a given client (see [MoNa91, MoNa92a] for more discussions). The GLM function can be easily distributed, e.g., as in the DEC VAXcluster™ distributed lock manager [KrLS86].

- Record locking is in effect, which means that a page may contain the uncommitted updates of many transactions concurrently executing in different systems. However, at any given time, only one system is allowed to be actively modifying a page. This system is called the ***update privilege owning system***. This is done to avoid the need for merging different versions of a page and to avoid some problems relating to space management (such problems were not dealt with in [CFLS91] when the O2PL protocols were proposed). To guarantee this *physical* serialization of updates to a page, the ownership of update

privileges is managed using ***physical*** (***P***) locks, as in ARIES for the shared disks (SD) environment [MoNa91]. A client cannot give up the update privilege on a page before the latest updated version of the page has been received by the server. For the transfer of the update privilege, it is enough if the server has cached the latest version of the page in its buffer pool. The latest version need not have been written to disk before another client is granted the update privilege.

• Clocks across the complex of systems are not necessarily synchronized. Even if the clocks are synchronized, it is unlikely that they would be synchronized to the degree of precision that would be needed to be able to use timestamps as LSNs [MoNP90]. Furthermore, if a single-system DBMS is being extended to support CS, then it is possible that the current page_LSN field is not long enough to store timestamps and hence using timestamps would require going through the undesirable database migration work of unloading all existing data, reformatting all pages and reloading the data.

• Clients have (local) log managers which behave very much like the regular log managers, except that, instead of writing log records to a local 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* to its log file. A client does not discard a log record from its log buffer until it gets confirmation that that log record has been safely recorded on stable storage at the server. 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.

• A *no-force* policy is used between the clients and the server - that is, when a transaction terminates (by committing or rolling back), the pages modified by the transaction are not necessarily sent to the server before the transaction is allowed to terminate. Of course, a transaction is declared to have committed only after all its log records are sent to the server and the server has forced them to its stable storage. The coherency control protocol used to establish coherency amongst the multiple copies of the same page in the buffer pools of the different clients must of course be able to handle the no-force policy (see [MoNa91, Rahm91, ReSW89] for details about many protocols and their requirements).

• A *steal* policy is used between the clients and the server - that is, a page containing uncommitted updates may be sent from a client to the server. Clients do not have disks which could be used for storing updated pages. The server, which has its own buffer pool, may write to disk even pages containing uncommitted updates.

2.2. LSN Management

We first discuss how LSN values are assigned in CS to ensure that recovery is performed correctly and that certain optimizations are possible. The key point is that LSNs are assigned *locally* by the clients. In ARIES/CSA, the page_LSN field is used as an update sequence number. That is, page_LSN will no longer necessarily be a log record address. During normal processing, an update to a page involves the following:

• Look up the current value of page_LSN 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 assigns to the new log record as its LSN the *higher* of the following two values:

1 + the page_LSN passed as a parameter
1 + Local_Max_LSN

Local_Max_LSN is a value that the log manager continuously maintains. It is *typically* the LSN of the log record most recently appended by the local log manager to its log buffer. The above method of assigning the LSN 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 database *pages* (this is in contrast to the proposal in [Lome90]). The utility of this feature is covered in the sections "2.3. Page Reallocation" and "3. Commit-LSN Optimization". Also, it is possible that the value of Local_Max_LSN may be increased without writing any log records locally. We discuss this in the section "3. Commit-LSN Optimization".

The log manager places the LSN computed above into the new log record's LSN field, assigns the value to Local_Max_LSN and also returns that value to the invoker of the log manager.

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

At the *server*, for enforcing the WAL protocol, whenever a dirty page is received from a client, the server's buffer manager can *conservatively* assign as that page's ***ForceAddr*** the logical address, in the server's log file, of the most recently written log record that came from that client.[2] Alternatively, the server can associate with each received dirty page,

2 If a dirty page is sent together with a set of log records by a client, the server first appends the received log records to its log buffer and picks the ForceAddr, and then copies the page to its buffer pool and assigns it the picked ForceAddr.

the logical address of that log record whose LSN value is present in the page_LSN field of that page. This would require some extra bookkeeping by the server.

2.3. Page Reallocation

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. An example of the above kind of page is an empty index page [Moha94b, 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 DBMS environment, during page reallocation, a page-formatting log record is written and the the page_LSN is set to the address of that log record [Moha94a]. This value of page_LSN would be greater than the page_LSN value assigned when the page was deallocated. (Note that it is required that page_LSN have a monotonically increasing value for a given page.) But, in the CS architecture, the page may be deallocated in one system and then reallocated in another. How can we ensure that the second system which reallocates the page assigns a page_LSN value which is greater than what the first system assigned while deallocating the same 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 had 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 (described in the section "2.2. LSN Management"), it is 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 page-formatting log record is higher than the current LSN of the SMP page.[3]

2.4. Transaction Rollback

When a client sends a log record to the server, the server analyzes the log record to keep track of various pieces of information about the transactions that are active at the different clients. (This information is tracked by the server for computing Commit_LSN as described in the section "3. Commit_LSN Optimization".) Therefore, it is possible for a client to retrieve log records from a server for a transaction rollback if they are not available locally.

Both total and partial rollbacks of transactions can be supported. Consequently, clients can support the savepoint concept. Total or partial rollback is initiated by the client by opening a log scan starting with the latest log record written by the transaction. Due to the clients' steal policy, some of the pages on which undo has to be performed may not anymore be present in the client's buffer pool. Those pages have to be fetched from the server's buffer pool or from the disks at the server. Except for these changes, rollback is performed as in ARIES for SD [MoNa91]. For example, if a page on which undo needs to be performed is no longer in the buffer pool of the client, then, in addition to the page, the update privilege on that page would also have to be reobtained.

2.5. Process and Media Failures

A page might have to be recovered when the DBMS is in *normal* (nonrestart) operation. There are 2 cases to consider:

Case 1: The page has been in use, for example, in a client and a process failure occurs *during* an update to the page. Since, in the general case, the partially performed update's effects cannot be easily directly eliminated from the corrupted page [Crus84], the page needs to be recovered by taking an earlier uncorrupted copy of the page from the server's disk or buffer pool, and redoing any missing updates to that copy by applying the log records. The log records which need to be applied will be in the range of page_LSN of the uncorrupted copy of the page to the end-of-log when page recovery is initiated.

The issue here is how to map a page_LSN to the corresponding log record's address when the server or a client has to recover a page.

Case 2: The page cannot be read from disk at the server due to a media error.

---

3 An inefficient alternative suggested in [Lome90] requires the explicit tracking of a deallocated page's LSN in the SMP entry for that page. This has many negative consequences (see [MoNa92b]).

2.5.1. Page Corruption in Server

The page_LSN value (of the uncorrupted copy) *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 as in single-system DBMSs like DB2. To handle this situation, when the *first* update is about to be performed to a *clean* page, the buffer manager tracks, in the buffer control block (***BCB***), a *lower* bound for the log record address of that update which causes the page state to go from *clean* to *dirty*. 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 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 the server fails, this information is needed for determining the point of the log from which the redo pass of restart recovery should begin [Crus84, MHLPS92].

2.5.2. Page Corruption in Client

The buffer manager at a *client* associates with each dirty page the *LSN* of the most recently written log record at that client just before that page became *dirty at that client*. 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 (or conservatively a lower) 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 that page and that earlier dirty version has not yet been written to disk), then the server's buffer manager retains the **old** RecAddr. To be able to perform the LSN to address mapping, the server can maintain, for each client, a small set of *<LSN, address>* pairs as log records are received from the client and added to the server's log. These pairs can then be used to map a particular LSN to its exact address or conservatively to a lower valued address. With the RecAddr so obtained, the log records can be applied by the client or the server to the uncorrupted copy of the page at the server to recover the page.

2.5.3. Page Corruption on Disk

If an uncorrupted version of 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 backup copy (archive dump).
- Accessing the log at the server and performing the necessary redos by starting from the appropriate log address as recorded with the backup copy (see [MHLPS92, MoNa93] for detailed discussions of this topic).

2.6. Client Failures

We believe that client failures would be handled more effectively if the clients take checkpoints periodically, just as in a single-system DBMS. First, we describe this approach. Next, we briefly discuss the approach where the clients do not take checkpoints and the server takes on more responsibility.

2.6.1. Checkpointing by Clients

Each client periodically takes a checkpoint. The client's checkpoint records some information about the state of the client's buffer pool - e.g., the dirty pages list (DPL) which contains for each dirty page an LSN (*RecLSN*) before which there are no update log records whose updates have not yet been reflected in the server version of the page (i.e., on disk or in the server's buffer pool). It also records the states of the transactions which are active at that client. The **server** keeps track of the most recent checkpoint records of all the clients. When an End_Checkpoint log record is received from a client, the server maps, for each page in DPL, the RecLSN value to an appropriate RecAddr, as explained before, updates the End_Checkpoint log record with the RecAddr data in the place of the RecLSN data and appends the log record to its log.

If a client were to fail, then the *server*, on noticing the failure of the client, performs recovery on behalf of the failed client. It does this by processing that client's checkpoint records, and by performing the analysis, redo and undo passes as in ARIES. During these passes, only the log records written by the failed client have to be processed. The need for possibly having to perform redo would have to be checked only for those pages for which the failed client had P locks that gave the client the update privilege on those pages.[4] 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. A log record's update is redone if the page_LSN value found in the log record is *greater than* the page's current page_LSN value. During undo, any CLRs that are written will be written in the name of the failed client.

Note that a client's (say C1's) checkpoint contains enough information for a page (say P1) which it modified only to ensure the redo of C1's updates. This will be the case even if P1 had earlier been updated by C2 and the server transferred the update privilege for P1 to C1 without first writing P1 to disk.

---

4 If coarser than page locking is being done (say, table (file) locking), then only the pages belonging to the table for which the client currently holds the update privilege need to be recovered.

The reason this is sufficient is because the server treats P1 as being dirty in its buffer pool, and when C1 fails the updates of C2 will still be in the version of P1 at the server and hence only C1's updates would have to be redone. Of course, if the server itself were to fail (see the section "2.7. Server Failures") before writing P1 to disk, then C2's updates would also have to be redone. This will be guaranteed to occur since the server's RecAddr for P1 will be low enough to cover C2's updates also.

Since recovery for a failed client is performed by the server as soon as the failure of the client is noticed by the server, no work needs to be performed by the client when it comes up later. Of course, if distributed transactions are being supported, then the client would have to reacquire some locks for any in-doubt transactions (see [MHLPS92]). The server could keep the information needed to do this and hand it to the client when it reconnects.

2.6.2. No Checkpointing by Clients

The motivation for making clients take checkpoints is to be able track updates made to a table (or an object) at page level even if the table is locked at a coarse granularity, e.g., by table-level locking. If page-level locking is in effect, then it is possible to track the needed recovery information (RecAddr) at the page level in the GLM's lock table [MoNa91] at the server. The server could maintain the recovery information in the form of RecAddr as follows: assign RecAddr when P lock is granted on the page in the update-privilege mode for the first time to any client; after the client(s) has sent the updated page and the log records, the server would write the page to disk and after the I/O is complete, move RecAddr forward.[5] With the availability of such information, without making clients take checkpoints, the server can perform recovery if the client were to fail. This is because the pages for which P locks with update privilege are held by a client constitute that client's dirty pages list (DPL) and the recovery information for those pages is available from the GLM lock table. This approach is simple in concept since all recovery responsibilities are with the server which owns the database and log disks. However, this approach has the following drawbacks:

- When a client locks an entire table, it is not possible to know what the DPL for the table is since the GLM lock table has only one entry for the table.
- Even with page-level locking, RecAddr maintained by the server may get old if the client, even while holding the update privilege, does not perform an update for a while. Advancing RecAddr under these conditions is quite tricky. Taking checkpoints at the client solves that problem more easily.

Hence, we preferred clients taking checkpoints.

2.7. Server Failures

To be able to recover from its own failures, the server periodically takes its own checkpoints. The server's checkpoint records some information about the state of the server's buffer pool (DPL with RecAddrs) and the states of the transactions which are active at the server.

While recovering from its own failure, the server recovers the dirty pages that were in its buffer pool at the time of its failure and also undoes any in-flight transactions that were executing at the server. During the redo pass of this recovery, the server will be potentially redoing even updates for which log records were written by the clients. This may be necessary because one or more clients might have updated a page without the page being written to disk in between, the last client to update the page might have given up the update privilege on that page, and then the server might have failed before writing that dirty page back to disk. For such a page, it is the server's responsibility to redo the updates performed by the one or more clients.

Page recovery as described above would be performed correctly, without any special logic, provided (1) a dirty page is received at the *server* before the last checkpoint of the server was taken (i.e., the page would be included in the server's DPL), or (2) the page became dirty at the *first client* only *after* the server's last checkpoint was taken (i.e., during the server's recovery, the log scan would encounter the log record that references the page and hence would recover that page). However, a page might not be recovered at all or incorrectly recovered if it became dirty at the *first client before* the server's last checkpoint was taken (i.e., the log record for the page update may *not* be encountered during the server's recovery), and the dirty page was sent to the server and the update privilege was given up by the client only *after* the last checkpoint was taken by the server (i.e., dirty page which is cached at the server is not included in the server's DPL).

The solution to the above problem is as follows: when the server takes its checkpoint, the server includes the DPLs of the clients also in its checkpoint. This would ensure that, in the case of a dirty page for which transfer of the update privilege is in progress, the page would be encountered either in the client's memory or in the server's memory. Hence, the need for a *coordinated* checkpoint of

---

5 One needs to be careful while moving RecAddr forward since we must not include those log records whose effects are not included in the disk copy. One possibility is that RecAddr is picked corresponding to the page_LSN of the version of the page written to disk, so that the log records which are appended after that would be excluded.

DPLs between the server and the clients. The server does the following to take its checkpoint.

- Write the Begin_Checkpoint log record and then request all the operational clients to send their list of dirty pages with their RecLSNs.

- Once all the operational clients have sent in their lists, the server merges those lists with its own *current* list of dirty pages to produce the DPL for the checkpoint. If the same page appears in more than one list, then the smallest RecLSN from that page's multiple entries is chosen for inclusion in DPL.

It is important that the server wait until all the operational clients have sent in their lists before it merges its current list of dirty pages. Picking up the local list first and then requesting the lists from the clients could result in some pages not being accounted for since a client might push back a page to the server in the interval of time between those two events! Such a page will not be included in DPL and it might have log records preceding the Begin_Checkpoint log record. Those log records will not be redone during a subsequent recovery if a server failure were to occur before the page is written to disk.

- Convert RecLSNs to RecAddrs as described earlier.

- Write the End_Checkpoint record with the DPL and the server's(local) transaction list.

If any clients were to fail while the server was down, then the server would have to perform recovery on behalf of those clients before it begins normal processing. At the end of recovery, the server talks to all its *operational* clients to fetch the lock information that they have for their transactions and dirty pages. Such information is needed to reconstruct the lock table that would have disappeared when the server failed.

3. Commit-LSN Optimization

The ***Commit_LSN*** technique (see [Moha90] for details) can be used 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 isolation 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 [Moha90, Moha93b]. Commit_LSN has been implemented in DB2 V3.

The Commit_LSN method crucially depends on a log record's LSN being smaller than those of all the log records written by a given system after the writing of the former log record. In the single-system and shared nothing 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, the Commit_LSN value must be computed by taking into account all the transactions executing across the *whole* complex of systems. In CS, this is done as follows. Whenever a log record sent by a client is appended to the log by the server, the server analyzes the log record to keep track of various pieces of information about the transactions that are active at the different clients. To the latter, it also adds information about the transactions that are active at the server itself. Once this collection of information is available, the server is in a position to compute and maintain the Commit_LSN value across all the transactions in the CS complex. The server periodically and/or on demand communicates the current Commit_LSN value to the clients. The clients and the server may use that value for the Commit_LSN method as explained in [Moha90].

In the SD and CS architectures, 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 or more systems keep issuing low LSNs, the smaller values will unnecessarily prevent some transactions from benefitting from the Commit_LSN method. This is because 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 hence transactions will be forced more often to obtain locks to determine whether a piece of data is committed. Instead of computing Commit_LSN across all the files in the database, it is possible to compute it on a per-file basis and get even more benefits (see [Moha90] for details).

In CS, to assure proximity of the LSN values assigned by the different systems, periodically, the server, while interacting with every client, passes to each client the *maximum* LSN (***Max_LSN***) value that it has seen so far in the log records that it has received from *all* the clients. When Max_LSN is received by each client, if it is found to be greater than the current client's Local_Max_LSN, then Local_Max_LSN is set to Max_LSN. (As mentioned in the section "2.2. LSN Management", this is the

case when Local_Max_LSN value at the client is increased without writing any log record locally.) This essentially amounts to a Lamport logical clock scheme [Lamp78]. To make the process efficient, the transmission of Max_LSNs can be piggybacked on to the other messages being exchanged between the server and the clients.

4. Comparisons With Related Work

4.1. Client-Server EXODUS

Concurrently with our work, researchers at the University of Wisconsin proposed some modifications to ARIES for supporting CS in the context of the EXODUS Storage Manager (ESM) which also transfers pages between the server and the clients [FZTCD92]. Their method is called the ESM-CS recovery method (ESM-CS, for short, in the following). It has many features in common with ARIES/CSA (use of WAL with ARIES for recovery, steal policy for buffer management, no disks at clients, etc.). The assignment and management of LSNs is mostly similar, although our approach is more general since it handles record locking and it covers optimizations like page reallocation and Commit_LSN. The most significant ways in which ESM-CS differs from ARIES/CSA are:

• In ESM-CS, when a transaction commits, the client at which the transaction executed is required to ship all the *pages* modified by that transaction to the server (call it ***force-to-server-at-commit***) policy. Furthermore, all pages are purged from the buffer pool of the client at transaction termination time. Clearly, there would be situations where these could lead to high overheads and it would be beneficial in many ways (e.g., to reduce communication traffic and lock hold times) to avoid them, as we do in ARIES/CSA. Since their clients do not take checkpoints, avoiding *force-to-server-at-commit* would require some fundamental changes in ESM-CS.

•In ESM-CS, clients do not perform any recovery actions except for writing log records in forward processing for transaction updates. As a result, even normal transaction rollback is not performed by a client. This causes complications in ESM-CS since pages updated by a to-be-rolled-back transaction are not forced to the server before the server performs the rollback. The latter requires that conditional undo be performed in ESM-CS as designed in [MoPi91] for an optimized version of ARIES (called ARIES-RRH). This means that, even when some undo does not have to be performed (because the update is not part of the version of the page at the server), a CLR would have to be written as if the undo was actually performed. The latter is possible only if *logical undos* are not required. The methods for managing B^+-tree indexes and hash-based storage with very high concurrency [Moha93a, Moha94b, MoLe92] may require logical undos. Such methods cannot be supported in ESM-CS unless ESM-CS's approach to rollback handling is modified. Furthermore, supporting partial rollbacks would be a problem since the *client* would have to purge (at least) those pages on which undo was performed at the *server*. Unless the client examines all the relevant log records or gets the needed information from the server, it would not know which pages should be purged. Conservatively purging all the pages would be inefficient since that could potentially cause many buffer misses subsequently when the transaction resumes.

• Only page or coarser granularity locking is supported. In particular, record locking is not supported. This assumption together with the force-to-server-at-commit policy is taken advantage of in many places. It is not clear how much ESM-CS would have to be changed to accommodate fine-granularity locking.

• The client executing a transaction and the server, on receiving dirty pages, keep track of pages updated by a transaction and remember RecLSNs of those pages. Before the commit record is logged for a transaction, this Commit Dirty Page List (*CDPL*) of the transaction is logged by the server. During the analysis pass, pages in CDPL are added to DPL. This elaborate scheme is needed partly because clients do not take checkpoints. It works only because of the force-to-server-at-commit policy. Even this CDPL logging is not sufficient to deal with a situation where a transaction (which does not terminate before a server failure) modifies a clean (at server) page *prior* to and also *after* the server's last checkpoint, but the dirty page is sent to the server only *after* the server's last checkpoint. In this case, since the in-flight transaction's CDPL would not have been logged, the server's analysis pass will not encounter the log record written prior to the checkpoint and hence redo of that earlier log record's update will not be performed. To avoid this problem, ESM-CS conservatively associates the transaction's *Start_LSN* as the RecLSN of such a page! Again, it works only because of the force-to-server-at-commit policy. ARIES/CSA avoids this problem by including in the server's checkpoint log record even those pages which are not currently dirty at the server but are dirty at the client.

[FrCL92] has proposed a client-server architecture where it is possible to take advantage of client memories to cache a larger fraction of a database, thereby reducing disk I/Os. This is achieved primarily by the *forwarding* mechanism. When a client makes a page request to the server and the server does not have that page cached but another client does, it forwards the request to that remote client. The remote client ships the page directly to the requesting client. This saves disk I/Os by the server.

The paper discusses other techniques, such as hate-hints, and sending-dropped-pages. The former reduces the duplicate caching between the server and a client, and the latter could help increase the life of cached pages.

That paper's major focus is on performance but the authors have discussed recovery in the related work and future work sections of the paper. The assumptions made in the paper which are relevant for recovery are: page-level locking, one client sending a dirty page to another, and avoiding making clients send dirty pages to the server prior to committing a transaction. Below, we discuss them in the context of ARIES/CSA.

In ARIES/CSA, we assume that locking is done at a granularity finer than a page. This allows even dirty pages to be shipped from one client to another before committing a transaction in the client. Of course, the log records of the sending client must be received by the server and acknowledged, before this client can send the page to the requesting client. Based on the recovery method which is chosen for the implementation, the log records may or may not be forced to disk during the transfer of a dirty page, and the dirty page may or may not be written to disk. These options and their implications on recovery are discussed in [MoNa91] in detail, in the context of the SD architecture. The idea of forwarding has also been discussed in detail in [MoNa91], where the global lock manager assists the requestor in getting the page shipped from the system which has the dirty page cached. In the SD architecture, every system can access all the database disks. Therefore, we did not want to pay the overhead of messages and overload a system with page requests.

ARIES/CSA does not require that dirty pages be shipped to the server prior to committing a transaction.

4.2. ObjectStore

ObjectStore [LLOW91] also caches pages at the client. At commit time, modified pages are not only sent to the server but are also written to disk by the server. However, the pages accessed by the transaction continue to be cached at the client after the transaction terminates. The smallest locking granularity that is supported is a page. Other than the fact that WAL is being used, nothing more is mentioned about recovery in [LLOW91].

5. Summary

In the client-server (CS) architecture, the server owns the disks for log and data, and clients cache and update database pages. In this paper, we described the changes that had to be made to the original ARIES method to support database recovery in the CS architecture. The modified method, called ARIES/CSA, was designed to provide a significant level of flexibility. Only minimal changes were needed to the original method to accommodate CS. Some of the highlights of ARIES/CSA are:

- Server performs recovery on behalf of a failed client. Clients take checkpoints and the clients' checkpoint information is used by the server to recover them efficiently.
- Server includes the dirty page lists of the clients in its checkpoint to ensure correctness of recovery.

The key advantages of ARIES/CSA are: (1) It supports fine-granularity locking, and the steal and no-force buffer management policies. (2) It requires neither the purging of the pages at the client when a transaction commits nor the forcing of the dirty pages to the server at that time. (3) The load on the server and recovery complexity are reduced by performing normal transaction rollback (partial or total) at the client. (4) ARIES/CSA supports indexing and hashing methods like ARIES/KVL, ARIES/IM and ARIES/LHS [Moha93a, Moha94b, MoLe92] that require the ability to support logical undos (as opposed to only page-oriented undos). Nested transactions can also be supported since ARIES has been extended to handle them [RoMo89]. (5) The clocks across the complex of systems do not have to be synchronized. (6) ARIES/CSA supports the very powerful Commit_LSN optimization. (7) If a single-system DBMS is being evolved to the multisystem environment, ARIES/CSA avoids the need for migration of all existing data by avoiding using synchronized clocks' timestamps as LSNs, in case the timestamp value's length is more than the size of the currently-present page_LSN field. (8) Clients can assigns LSNs locally which improves performance.

In the future, we plan to deal with recovery issues when individual objects/records, rather than pages, are exchanged between the clients and the server.

6. References

CFLS91 Carey, M., Franklin, M., Livny, M., Shekita, E. *Data Caching Tradeoffs in Client-Server DBMS Architectures*, **Proc. ACM SIGMOD International Conference on Management of Data**, Denver, May 1991.

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

Deux91 Deux, O., et al. *The O_2 System*, **Communications of the ACM**, Vol. 34, No. 10, October 1991.

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.

FrCL92 Franklin, M., Carey, M., Livny M., *Global Memory Management in Client-Server DBMS Architecture*, **Proc. 18th International Conference on Very Large Data Bases**, Vancouver, August 1992.

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.

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

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

LLOW91 Lamb, C., Landis, G., Orenstein, J., Weinreb, D. *The ObjectStore Database System*, **Communications of the ACM**, Vol. 34, No. 10, October 1991.

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.

Moha90 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.

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.

Moha93b Mohan, C. *A Cost-Effective Method for Providing Improved Data Availability* During *DBMS Restart Recovery After a Failure*, **Proc. 19th International Conference on Very Large Data Bases**, Dublin, August 1993.

Moha94a Mohan, C. *Disk Read-Write Optimizations and Data Integrity in Transaction Systems Using Write-Ahead Logging*, **IBM Research Report RJ9741**, IBM Almaden Research Center, March 1994.

Moha94b Mohan, C. *Concurrency Control and Recovery Methods for B^+-Tree Indexes: ARIES/KVL and ARIES/IM*, To appear in **Performance of Concurrency Control Mechanisms in Centralized Database Systems**, V. Kumar (Ed.), Prentice Hall, 1994. Also available as **IBM Research Report RJ9715**, IBM Almaden Research Center, March 1994.

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.

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.

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.

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.

MoNa93 Mohan, C., Narang, I. *An Efficient and Flexible Method for Archiving a Data Base*, **Proc. ACM SIGMOD International Conference on Management of Data**, Washington, D.C., May 1993. A *corrected* version of this paper is available as **IBM Research Report RJ9733**, IBM Almaden Research Center, March 1994.

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.

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.

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

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.