ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging

C. MOHAN
IBM Almaden Research Center
and
DON HADERLE
IBM Santa Teresa Laboratory
and
BRUCE LINDSAY, HAMID PIRAHESH and PETER SCHWARZ
IBM Almaden Research Center

In this paper we present a simple and efficient method, called ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*), which supports partial rollbacks of transactions, fine-granularity (e.g., record) locking and recovery using write-ahead logging (WAL). We introduce the paradigm of *repeating history* to redo all missing updates *before* performing the rollbacks of the loser transactions during restart after a system failure. ARIES uses a log sequence number in each page to correlate the state of a page with respect to logged updates of that page. All updates of a transaction are logged, including those performed during rollbacks. By appropriate chaining of the log records written during rollbacks to those written during forward progress, a bounded amount of logging is ensured during rollbacks even in the face of repeated failures during restart or of nested rollbacks We deal with a variety of features that are very important in building and operating an *industrial-strength* transaction processing system ARIES supports fuzzy checkpoints, selective and deferred restart, fuzzy image copies, media recovery, and high concurrency lock modes (e.g., increment/decrement) which exploit the semantics of the operations and require the ability to perform operation logging. ARIES is flexible with respect to the kinds of buffer management policies that can be implemented. It supports objects of varying length efficiently. By enabling parallelism during restart, page-oriented redo, and logical undo, it enhances concurrency and performance. We show why some of the System R paradigms for logging and recovery, which were based on the shadow page technique, need to be changed in the context of WAL. We compare ARIES to the WAL-based recovery methods of

Authors' addresses: C Mohan, Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120; D. Haderle, Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, CA 95150; B. Lindsay, H. Pirahesh, and P. Schwarz, IBM Almaden Research Center, San Jose, CA 95120.

ACM Transactions on Database Systems, Vol 17, No. 1, March 1992, Pages 94–162

DB2™, IMS, and Tandem™ systems. ARIES is applicable not only to database management systems but also to persistent object-oriented languages, recoverable file systems and transaction-based operating systems. ARIES has been implemented, to varying degrees, in IBM's OS/2™ Extended Edition Database Manager, DB2, Workstation Data Save Facility/VM, Starburst and QuickSilver, and in the University of Wisconsin's EXODUS and Gamma database machine.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability—*backup procedures, checkpoint/restart, fault tolerance*; E.5. [**Data**]: Files—*backup/recovery*; H.2.2 [**Database Management**]: Physical Design—*recovery and restart*; H.2.4 [**Database Management**]: Systems—*concurrency, transaction processing*; H.2.7 [**Database Management**]: Database Administration—*logging and recovery*

General Terms: Algorithms, Design, Performance, Reliability

Additional Key Words and Phrases: Buffer management, latching, locking, space management, write-ahead logging

1. INTRODUCTION

In this section, first we introduce some basic concepts relating to recovery, concurrency control, and buffer management, and then we outline the organization of the rest of the paper.

1.1 Logging, Failures, and Recovery Methods

The transaction concept, which is well understood by now, has been around for a long time. It encapsulates the *ACID* (Atomicity, Consistency, Isolation and Durability) properties [36]. The application of the transaction concept is not limited to the database area [6, 17, 22, 23, 30, 39, 40, 51, 74, 88, 90, 101]. Guaranteeing the atomicity and durability of transactions, in the face of concurrent execution of multiple transactions and various failures, is a very important problem in transaction processing. While many methods have been developed in the past to deal with this problem, the assumptions, performance characteristics, and the complexity and ad hoc nature of such methods have not always been acceptable. Solutions to this problem may be judged using several metrics: degree of concurrency supported within a page and across pages, complexity of the resulting logic, space overhead on nonvolatile storage and in memory for data and the log, overhead in terms of the number of synchronous and asynchronous I/Os required during restart recovery and normal processing, kinds of functionality supported (partial transaction rollbacks, etc.), amount of processing performed during restart recovery, degree of concurrent processing supported during restart recovery, extent of system-induced transaction rollbacks caused by deadlocks, restrictions placed

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

on stored data (e.g., requiring unique keys for all records, restricting maximum size of objects to the page size, etc.), ability to support novel lock modes which allow the concurrent execution, based on commutativity and other properties [2, 26, 38, 45, 88, 89], of operations like increment/decrement on the same data by different transactions, and so on.

In this paper we introduce a new recovery method, called *ARIES*[1] (*Algorithm for Recovery and Isolation Exploiting Semantics*), which fares very well with respect to all these metrics. It also provides a great deal of flexibility to take advantage of some special characteristics of a class of applications for better performance (e.g., the kinds of applications that IMS Fast Path [28, 42] supports efficiently).

To meet transaction and data recovery guarantees, ARIES records in a *log* the progress of a transaction, and its actions which cause changes to recoverable data objects. The log becomes the source for ensuring either that the transaction's committed actions are reflected in the database despite various types of failures, or that its uncommitted actions are undone (i.e., rolled back). When the logged actions reflect data object content, then those log records also become the source for reconstruction of damaged or lost data (i.e., media recovery). *Conceptually*, the log can be thought of as an ever growing *sequential* file. In the actual implementation, multiple physical files may be used in a serial fashion to ease the job of archiving log records [15]. Every log record is assigned a unique *log sequence number* (*LSN*) when that record is appended to the log. The LSNs are assigned in ascending sequence. Typically, they are the *logical* addresses of the corresponding log records. At times, version numbers or timestamps are also used as LSNs [67]. If more than one log is used for storing the log records relating to *different* pieces of data, then a form of two-phase commit protocol (e.g., the current industry-standard Presumed Abort protocol [63, 64]) must be used.

The nonvolatile version of the log is stored on what is generally called *stable storage*. Stable storage means nonvolatile storage which remains intact and available across system failures. Disk is an example of nonvolatile storage and its stability is generally improved by maintaining synchronously two identical copies of the log on different devices. We would expect the online log records stored on direct access storage devices to be archived to a cheaper and slower medium like tape at regular intervals. The archived log records may be discarded once the appropriate image copies (archive dumps) of the database have been produced and those log records are no longer needed for media recovery.

Whenever log records are written, they are placed first only in the *volatile* storage (i.e., virtual storage) buffers of the log file. Only at certain times (e.g., at commit time) are the log records up to a certain point (LSN) written, in log page sequence, to stable storage. This is called *forcing* the log up to that LSN. Besides forces caused by transaction and buffer manager activi-

[1] The choice of the name ARIES, besides its use as an acronym that describes certain features of our recovery method, is also supposed to convey the relationship of our work to the Starburst project at IBM, since Aries is the name of a constellation.

ties, a system process may, in the background, periodically force the log buffers as they fill up.

For ease of exposition, we assume that each log record describes the update performed to only a single page. This is not a requirement of ARIES. In fact, in the Starburst [87] implementation of ARIES, sometimes a single log record might be written to describe updates to two pages. The *undo* (respectively, *redo*) portion of a log record provides information on how to undo (respectively, redo) changes performed by the transaction. A log record which contains both the undo and the redo information is called an *undo-redo log record*. Sometimes, a log record may be written to contain only the redo information or only the undo information. Such a record is called a *redo-only log record* or an *undo-only log record*, respectively. Depending on the action that is performed, the undo-redo information may be recorded *physically* (e.g., before the update and after the update images or values of specific fields within the object) or *operationally* (e.g., add 5 to field 3 of record 15, subtract 3 from field 4 of record 10). Operation logging permits the use of high concurrency lock modes, which exploit the semantics of the operations performed on the data. For example, with certain operations, the same field of a record could have uncommitted updates of many transactions. These permit more concurrency than what is permitted by the *strict executions* property of the model of [3], which essentially says that modified objects must be locked exclusively (X mode) for commit duration.

ARIES uses the widely accepted write ahead logging (WAL) protocol. Some of the commercial and prototype systems based on WAL are IBM's AS/400™ [9, 21], CMU's Camelot [23, 90], IBM's DB2™ [1, 10, 11, 12, 13, 14, 15, 19, 35, 96], Unisys's DMS/1100 [27], Tandem's Encompass™ [4, 37], IBM's IMS [42, 43, 53, 76, 80, 94], Informix's Informix-Turbo™ [16], Honeywell's MRDS [91], Tandem's NonStop SQL™ [95], MCC's ORION [29], IBM's OS/2 Extended Edition™ Database Manager [7], IBM's QuickSilver [40], IBM's Starburst [87], SYNAPSE [78], IBM's System/38 [99], and DEC's VAX DBMS™ and VAX Rdb/VMS™ [81]. In WAL-based systems, an updated page is written back to the same nonvolatile storage location from where it was read. That is, *in-place updating* is performed on nonvolatile storage. Contrast this with what happens in the shadow page technique which is used in systems such as System R [31] and SQL/DS [5] and which is illustrated in Figure 1. There the updated version of the page is written to a different location on nonvolatile storage and the previous version of the page is used for performing database recovery if the system were to fail before the next checkpoint.

The *WAL protocol* asserts that the log records representing changes to some data must already be on stable storage before the changed data is allowed to replace the previous version of that data on nonvolatile storage. That is, the system is not allowed to write an updated page to the nonvolatile storage version of the database until at least the undo portions of the log records which describe the updates to the page have been written to stable storage. To enable the enforcement of this protocol, systems using the WAL method of recovery store in every page the LSN of the log record that describes the most recent update performed on that page. The reader is

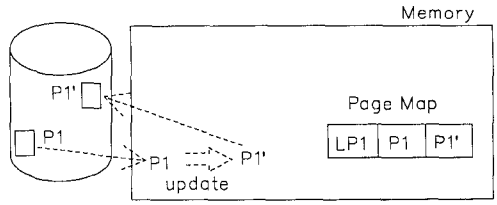


Fig. 1. Shadow page technique.

Logical page LP1 is read from physical page P1 and after modification is written to physical page P1' P1' is the *current* version and P1 is the *shadow* version During a checkpoint, the shadow version is discarded and the current version becomes the shadow version also On a failure, data base recovery is performed using the log and the shadow version of the data base

referred to [31, 97] for discussions about why the WAL technique is considered to be better than the shadow page technique. [16, 78] discuss methods in which shadowing is performed using a separate log. While these avoid some of the problems of the original shadow page approach, they still retain some of the important drawbacks and they introduce some new ones. Similar comments apply to the methods suggested in [82, 88]. Later, in Section 10, we show why some of the recovery paradigms of System R, which were based on the shadow page technique, are inappropriate in the WAL context, when we need support for high levels of concurrency and various other features that are described in Section 2.

Transaction status is also stored in the log and no transaction can be considered complete until its committed status and all its log data are safely recorded on stable storage by forcing the log up to the transaction's commit log record's LSN. This allows a restart recovery procedure to recover any transactions that completed successfully but whose updated pages were not physically written to nonvolatile storage before the failure of the system. This means that a transaction is not permitted to complete its *commit* processing (see [63, 64]) until the redo portions of all log records of that transaction have been written to stable storage.

We deal with three types of failures: transaction or process, system, and media or device. When a transaction or process failure occurs, typically the transaction would be in such a state that its updates would have to be undone. It is possible that the transaction had corrupted some pages in the buffer pool if it was in the middle of performing some updates when the process disappeared. When a system failure occurs, typically the virtual storage contents would be lost and the transaction system would have to be restarted and recovery performed using the nonvolatile storage versions of the database and the log. When a media or device failure occurs, typically the contents of that media would be lost and the lost data would have to be recovered using an image copy (archive dump) version of the lost data and the log.

Forward processing refers to the updates performed when the system is in normal (i.e., not restart recovery) processing and the transaction is updating

the database because of the data manipulation (e.g., SQL) calls issued by the user or the application program. That is, the transaction is not rolling back and using the log to generate the (undo) update calls. *Partial rollback* refers to the ability to set up *savepoints* during the execution of a transaction and later in the transaction request the rolling back of the changes performed by the transaction since the establishment of a previous savepoint [1, 31]. This is to be contrasted with *total rollback* in which all updates of the transaction are undone and the transaction is terminated. Whether or not the savepoint concept is exposed at the application level is immaterial to us since this paper deals only with database recovery. A *nested rollback* is said to have taken place if a partial rollback were to be later followed by a total rollback or another partial rollback whose point of termination is an earlier point in the transaction than the point of termination of the first rollback. *Normal undo* refers to total or partial transaction rollback when the system is in normal operation. A normal undo may be caused by a transaction request to rollback or it may be system initiated because of deadlocks or errors (e.g., integrity constraint violations). *Restart undo* refers to transaction rollback during restart recovery after a system failure. To make partial or total rollback efficient and also to make debugging easier, all the log records written by a transaction are linked via the *PrevLSN* field of the log records in reverse chronological order. That is, the most recently written log record of the transaction would point to the previous most recent log record written by that transaction, if there is such a log record.[2] In many WAL-based systems, the updates performed during a rollback are logged using what are called *compensation log records* (*CLRs*) [15]. Whether a CLR's update is undone, should that CLR be encountered during a rollback, depends on the particular system. As we will see later, in ARIES, a CLR's update is never undone and hence CLRs are viewed as redo-only log records.

Page-oriented redo is said to occur if the log record whose update is being redone describes which page of the database was originally modified during normal processing and if the same page is modified during the redo processing. No internal descriptors of tables or indexes need to be accessed to redo the update. That is, no other page of the database needs to be examined. This is to be contrasted with *logical redo* which is required in System R, SQL/DS and AS/400 for indexes [21, 62]. In those systems, since index changes are not logged separately but are redone using the log records for the data pages, performing a redo requires accessing several descriptors and pages of the database. The index tree would have to be retraversed to determine the page(s) to be modified and, sometimes, the index page(s) modified because of this redo operation may be different from the index page(s) originally modified during normal processing. Being able to perform page-oriented redo allows the system to provide *recovery independence amongst objects*. That is, the recovery of one page's contents does not require accesses to any other

[2] The AS/400, Encompass and NonStop SQL do not explicitly link all the log records written by a transaction. This makes undo inefficient since a *sequential* backward scan of the log must be performed to retrieve all the desired log records of a transaction.

(data or catalog) pages of the database. As we will describe later, this makes media recovery very simple.

In a similar fashion, we can define *page-oriented undo* and *logical undo*. Being able to perform logical undos allows the system to provide higher levels of concurrency than what would be possible if the system were to be restricted only to page-oriented undos. This is because the former, with appropriate concurrency control protocols, would permit uncommitted updates of one transaction to be moved to a different page by another transaction. If one were restricted to only page-oriented undos, then the latter transaction would have had to wait for the former to commit. Page-oriented redo and page-oriented undo permit faster recovery since pages of the database other than the pages mentioned in the log records are not accessed. In the interest of efficiency, ARIES supports page-oriented redo and its supports, in the interest of high concurrency, logical undos. In [62], we introduce the ARIES/IM method for concurrency control and recovery in B^+-tree indexes and show the advantages of being able to perform logical undos by comparing ARIES/IM with other index methods.

1.2 Latches and Locks

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

Acquiring and releasing a latch is much cheaper than acquiring and releasing a lock. In the no-conflict case, the overhead amounts to 10s of instructions for the former versus 100s of instructions for the latter. Latches are cheaper because the *latch control information* is always in virtual memory in a fixed place, and direct addressability to the latch information is possible given the latch name. As the protocols presented later in this paper and those in [57, 62] show, each transaction holds at most two or three latches simultaneously. As a result, the *latch request blocks* can be permanently allocated to each transaction and initialized with transaction ID, etc. right at the start of that transaction. On the other hand, typically, storage for individual locks has to be acquired, formatted and released dynamically, causing more instructions to be executed to acquire and release locks. This is advisable because, in most systems, the number of lockable objects is many orders of magnitude greater than the number of latchable objects. Typically, all information relating to locks currently held or requested by all the transactions is stored in a single, central hash table. Addressability to a particular lock's information is gained by first hashing the lock name to get the address of the hash anchor and then, possibly, following a chain of pointers. Usually, in the process of trying to locate the *lock control block*,

because multiple transactions may be simultaneously reading and modifying the contents of the lock table, one or more latches will be acquired and released—one latch on the hash anchor and, possibly, one on the specific lock's chain of holders and waiters.

Locks may be obtained in different *modes* such as S (Shared), X (eXclusive), IX (Intention eXclusive), IS (Intention Shared) and SIX (Shared Intention eXclusive), and at different *granularities* such as record (tuple), table (relation), and file (tablespace) [32]. The S and X locks are the most common ones. S provides the read privilege and X provides the read and write privileges. Locks on a given object can be held simultaneously by different transactions only if those locks' modes are *compatible*. The compatibility relationships amongst the above modes of locking are shown in Figure 2. A check mark ('√') indicates that the corresponding modes are compatible. With *hierarchical locking*, the intention locks (IX, IS, and SIX) are generally obtained on the higher levels of the hierarchy (e.g., table), and the S and X locks are obtained on the lower levels (e.g., record). The nonintention mode locks (S and X), when obtained on an object at a certain level of the hierarchy, *implicitly* grant locks of the corresponding mode on the lower level objects of that higher level object. The intention mode locks, on the other hand, only give the privilege of requesting the corresponding intention or nonintention mode locks on the lower level objects. For example, SIX on a table implicitly grants S on all the records of that table, and it allows X to be requested *explicitly* on the records. Additional, semantically rich lock modes have been defined in the literature [2, 38, 45, 55] and ARIES can accommodate them.

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

1.3 Fine-Granularity Locking

Fine-granularity (e.g., record) locking has been supported by nonrelational database systems (e.g., IMS [53, 76, 80]) for a long time. Surprisingly, only a few of the commercially available relational systems provide fine-granularity locking, even though IBM's System R [32], S/38 [99] and SQL/DS [5], and Tandem's Encompass [37] supported record and/or key locking from the beginning.[3] Although many interesting problems relating to providing

[3] Encompass and S/38 had only X locks for records and no locks were acquired *automatically* by these systems for reads.

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

Fig. 2. Lock mode compatability matrix

fine-granularity locking in the context of WAL remain to be solved, the research community has not been paying enough attention to this area [3, 75, 88]. Some of the System R solutions worked only because of the use of the shadow page recovery technique in combination with locking (see Section 10). Supporting fine-granularity locking and variable length records in a flexible fashion requires addressing some interesting storage management issues which have never really been discussed in the database literature. Unfortunately, some of the interesting techniques that were developed for System R and which are now part of SQL/DS did not get documented in the literature. At the expense of making this paper long, we will be discussing here some of those problems and their solutions.

As supporting high concurrency gains importance (see [79] for the description of an application requiring very high concurrency) and as object-oriented systems gain in popularity, it becomes necessary to invent concurrency control and recovery methods that take advantage of the semantics of the operations on the data [2, 26, 38, 88, 89], and that support fine-granularity locking efficiently. Object-oriented systems may tend to encourage users to define a large number of small objects and users may expect object instances to be the appropriate granularity of locking. In the object-oriented logical view of the database, the concept of a page, with its physical orientation as the container of objects, becomes unnatural to think about as the unit of locking during object accesses and modifications. Also, object-oriented system users may tend to have many terminal interactions during the course of a transaction, thereby increasing the lock hold times. If the unit of locking were to be a page, lock wait times and deadlock possibilities will be aggravated. Other discussions concerning transaction management in an object-oriented environment can be found in [22, 29].

As more and more customers adopt relational systems for production applications, it becomes ever more important to handle *hot-spots* [28, 34, 68, 77, 79, 83] and storage management without requiring too much tuning by the system users or administrators. Since relational systems have been welcomed to a great extent because of their ease of use, it is important that we pay greater attention to this area than what has been done in the context of the nonrelational systems. Apart from the need for high concurrency for user data, the ease with which online data definition operations can be performed in relational systems by even ordinary users requires the support for high concurrency of access to, at least, the catalog data. Since a leaf page in an index typically describes data in hundreds of data pages, page-level locking of index data is just not acceptable. A flexible recovery method that

allows the support of high levels of concurrency during index accesses is needed.

The above facts argue for supporting semantically rich modes of locking such as increment/decrement which allow multiple transactions to concurrently modify even the same piece of data. In funds-transfer applications, increment and decrement operations are frequently performed on the branch and teller balances by numerous transactions. If those transactions are forced to use only X locks, then they will be serialized, even though their operations commute.

1.4 Buffer Management

The buffer manager (BM) is the component of the transaction system that manages the buffer pool and does I/Os to read/write pages from/to the nonvolatile storage version of the database. The *fix* primitive of the BM may be used to request the buffer address of a logical page in the database. If the requested page is not in the buffer pool, BM allocates a buffer slot and reads the page in. There may be instances (e.g., during a B^+-tree page split, when the new page is allocated) where the current contents of a page on nonvolatile storage are not of interest. In such a case, the *fix_new* primitive may be used to make the BM allocate a *free* slot and return the address of that slot, if BM does not find the page in the buffer pool. The fix_new invoker will then format the page as desired. Once a page is fixed in the buffer pool, the corresponding buffer slot is not available for page replacement until the *unfix* primitive is issued by the data manipulative component. Actually, for each page, BM keeps a fix count which is incremented by one during every fix operation and which is decremented by one during every unfix operation. A page in the buffer pool is said to be *dirty* if the buffer version of the page has some updates which are not yet reflected in the nonvolatile storage version of the same page. The fix primitive is also used to communicate the intention to modify the page. Dirty pages can be written back to nonvolatile storage when no fix with the modification intention is held, thus allowing read accesses to the page while it is being written out. [96] discusses the role of BM in writing in the background, on a continuous basis, dirty pages to nonvolatile storage to reduce the amount of redo work that would be needed if a system failure were to occur and also to keep a certain percentage of the buffer pool pages in the nondirty state so that they may be replaced with other pages without synchronous write I/Os having to be performed at the time of replacement. While performing those writes, BM ensures that the WAL protocol is obeyed. As a consequence, BM may have to force the log up to the LSN of the dirty page before writing the page to nonvolatile storage. Given the large buffer pools that are common today, we would expect a force of this nature to be very rare and most log forces to occur because of transactions committing or entering the prepare state.

BM also implements the support for latching pages. To provide direct addressability to page latches and to reduce the storage associated with those latches, the latch on a logical page is actually the latch on the corresponding buffer slot. This means that a logical page can be latched only after it is fixed

in the buffer pool and the latch has to be released before the page is unfixed. These are highly acceptable conditions. The latch control information is stored in the buffer control block (BCB) for the corresponding buffer slot. The BCB also contains the identity of the logical page, what the fix count is, the dirty status of the page, etc.

Buffer management policies differ among the many systems in existence (see Section 11, "Other WAL-Based Methods"). If a page modified by a transaction is allowed to be written to the permanent database on nonvolatile storage before that transaction commits, then the *steal* policy is said to be followed by the buffer manager (see [36] for such terminologies). Otherwise, a *no-steal* policy is said to be in effect. Steal implies that during normal or restart rollback, some undo work might have to be performed on the nonvolatile storage version of the database. If a transaction is not allowed to commit until all pages modified by it are written to the permanent version of the database, then a *force* policy is said to be in effect. Otherwise, a *no-force* policy is said to be in effect. With a force policy, during restart recovery, no redo work will be necessary for committed transactions. *Deferred updating* is said to occur if, even in the virtual storage database buffers, the updates are not performed in-place when the transaction issues the corresponding database calls. The updates are kept in a pending list elsewhere and are performed in-place, using the pending list information, only after it is determined that the transaction is definitely committing. If the transaction needs to be rolled back, then the pending list is discarded or ignored. The deferred updating policy has implications on whether a transaction can "see" its own updates or not, and on whether partial rollbacks are possible or not.

For more discussions concerning buffer management, see [8, 15, 24, 96].

1.5 Organization

The rest of the paper is organized as follows. After stating our goals in Section 2 and giving an overview of the new recovery method ARIES in Section 3, we present, in Section 4, the important data structures used by ARIES during normal and restart recovery processing. Next, in Section 5, the protocols followed during normal processing are presented followed, in Section 6, by the description of the processing performed during restart recovery. The latter section also presents ways to exploit parallelism during recovery and methods for performing recovery selectively or postponing the recovery of some of the data. Then, in Section 7, algorithms are described for taking checkpoints during the different log passes of restart recovery to reduce the impact of failures during recovery. This is followed, in Section 8, by the description of how fuzzy image copying and media recovery are supported. Section 9 introduces the significant notion of *nested top actions* and presents a method for implementing them efficiently. Section 10 describes and critiques some of the existing recovery paradigms which originated in the context of the shadow page technique and System R. We discuss the problems caused by using those paradigms in the WAL context. Section 11 describes in detail the characteristics of many of the WAL-based recovery methods in use in different systems such as IMS, DB2, Encompass and NonStop SQL.

Section 12 outlines the many different properties of ARIES. We conclude by summarizing, in Section 13, the features of ARIES which provide flexibility and efficiency, and by describing the extensions and the current status of the implementations of ARIES.

Besides presenting a new recovery method, by way of motivation for our work, we also describe some previously unpublished aspects of recovery in System R. For comparison purposes, we also do a survey of the recovery methods used by other WAL-based systems and collect information appearing in several publications, many of which are not widely available. One of our aims in this paper is to show the intricate and unobvious interactions resulting from the different choices made for the recovery technique, the granularity of locking and the storage management scheme. One cannot make arbitrarily independent choices for these and still expect the combination to function together correctly and efficiently. This point needs to be emphasized as it is not always dealt with adequately in most papers and books on concurrency control and recovery. In this paper, we have tried to cover, as much as possible, all the interesting recovery-related problems that one encounters in building and operating an *industrial-strength* transaction processing system.

2. GOALS

This section lists the goals of our work and outlines the difficulties involved in designing a recovery method that supports the features that we aimed for. The goals relate to the metrics for comparison of recovery methods that we discussed earlier, in Section 1.1.

Simplicity. Concurrency and recovery are complex subjects to think about and program for, compared with other aspects of data management. The algorithms are bound to be error-prone, if they are complex. Hence, we strived for a simple, yet powerful and flexible, algorithm. Although this paper is long because of its comprehensive discussion of numerous problems that are mostly ignored in the literature, the main algorithm itself is quite simple. Hopefully, the overview presented in Section 3 gives the reader that feeling.

Operation logging. The recovery method had to permit operation logging (and value logging) so that semantically rich lock modes could be supported. This would let one transaction modify the same data that was modified earlier by another transaction which has not yet committed, when the two transactions' actions are semantically compatible (e.g., increment/decrement operations; see [2, 26, 45, 88]). As should be clear, recovery methods which always perform *value* or *state logging* (i.e., logging before-images and after-images of modified data), cannot support operation logging. This includes systems that do very physical—byte-oriented—logging of all changes to a page [6, 76, 81]. The difficulty in supporting operation logging is that we need to track precisely, using a concept like the LSN, the exact state of a page with respect to logged actions relating to that page. An undo or a redo of an update should not be performed without being sure that the original update

is present or is not present, respectively. This also means that, if one or more transactions that had previously modified a page start rolling back, then we need to know precisely how the page has been affected during the rollbacks and how much of each of the rollbacks had been accomplished so far. This requires that updates performed during rollbacks also be logged via the so-called *compensation log records* (*CLRs*). The LSN concept lets us avoid attempting to redo an operation when the operation's effect is already present in the page. It also lets us avoid attempting to undo an operation when the operation's effect is not present in the page. Operation logging lets us perform, if found desirable, *logical logging*, which means that not everything that was changed on a page needs to be logged explicitly, thereby saving log space. For example, changes of control information, like the amount of free space on the page, need not be logged. The redo and the undo operations can be performed logically. For a good discussion of operation and value logging, see [88].

Flexible storage management. Efficient support for the storage and manipulation of varying length data is important. In contrast to systems like IMS, the intent here is to be able to avoid the need for off-line reorganization of the data to garbage collect any space that might have been freed up because of deletions and updates that caused data shrinkage. It is desirable that the recovery method and the concurrency control method be such that the logging and locking is *logical* in nature so that movements of the data within a page for garbage collection reasons do not cause the moved data to be locked or the movements to be logged. For an index, this also means that one transaction must be able to split a leaf page even if that page currently has some uncommitted data inserted by another transaction. This may lead to problems in performing page-oriented undos using the log; *logical undos* may be necessary. Further, we would like to be able to let a transaction that has freed up some space be able to use, if necessary, that space during its later insert activity [50]. System R, for example, does not permit this in data pages.

Partial rollbacks. It was essential that the new recovery method support the concept of savepoints and rollbacks to savepoints (i.e., partial rollbacks). This is crucial for handling, in a user-friendly fashion (i.e., without requiring a total rollback of the transaction), integrity constraint violations (see [1, 31]), and problems arising from using obsolete cached information (see [49]).

Flexible buffer management. The recovery method should make the least number of restrictive assumptions about the buffer management policies (*steal*, *force*, etc.) in effect. At the same time, the method must be able to take advantage of the characteristics of any specific policy that is in effect (e.g., with a force policy there is no need to perform any redos for committed transactions.) This flexibility could result in increased concurrency, decreased I/Os and efficient usage of buffer storage. Depending on the policies, the work that needs to be performed during restart recovery after a system

failure or during media recovery may be more or less complex. Even with large main memories, it must be noted that a steal policy is still very desirable. This is because, with a no-steal policy, a page may never get written to nonvolatile storage if the page always contains *uncommitted* updates due to fine-granularity locking and overlapping transactions' updates to that page. The situation would be further aggravated if there are long-running transactions. Under those conditions, either the system would have to frequently reduce concurrency by quiescing all activities on the page (i.e., by locking all the objects on the page) and then writing the page to non-volatile storage, or by doing nothing special and then paying a huge restart redo recovery cost if the system were to fail. Also, a no-steal policy incurs additional bookkeeping overhead to track whether a page contains any uncommitted updates. We believe that, given our goal of supporting semantically rich lock modes, partial rollbacks and varying length objects efficiently, in the general case, we need to perform undo logging and in-place updating. Hence, methods like the transaction workspace model of AIM [46] are not general enough for our purposes. Other problems relating to no-steal are discussed in Section 11 with reference to IMS Fast Path.

Recovery independence. It should be possible to image copy (archive dump), and perform media recovery or restart recovery at different granularities, rather than only at the entire database level. The recovery of one object should not force the concurrent or lock-step recovery of another object. Contrast this with what happens in the shadow page technique as implemented in System R, where index and space management information are recovered *lock-step* with user and catalog table (relation) data by starting from an internally consistent state of the *whole* database and redoing changes to all the related objects of the database simultaneously, as in normal processing. Recovery independence means that, during the restart recovery of some object, catalog information in the database cannot be accessed for descriptors of that object and its related objects, since that information itself may be undergoing recovery in parallel with the object being recovered and the two may be out of synchronization [14]. During restart recovery, it should be possible to do selective recovery and defer recovery of some objects to a later point in time to speed up restart and also to accommodate some offline devices. *Page-oriented recovery* means that even if one page in the database is corrupted because of a process failure or a media problem, it should be possible to recover that page alone. To be able to do this efficiently, we need to log every page's change individually, even if the object being updated spans multiple pages and the update affects more than one page. This, in conjunction with the writing of CLRs for updates performed during rollbacks, will make media recovery very simple (see Section 8). This will also permit image copying of different objects to be performed independently and at different frequencies.

Logical undo. This relates to the ability, during undo, to affect a page that is different from the one modified during forward processing, as is

needed in the earlier-mentioned context of the split by one transaction of an index page containing uncommitted data of another transaction. Being able to perform logical undos allows higher levels of concurrency to be supported, especially in search structures [57, 59, 62]. If logging is not performed during rollback processing, logical undos would be very difficult to support, if we also desired recovery independence and page-oriented recovery. System R and SQL/DS support logical undos, but at the expense of recovery independence.

Parallelism and fast recovery. With multiprocessors becoming very common and greater data availability becoming increasingly important, the recovery method has to be able to exploit parallelism during the different stages of restart recovery and during media recovery. It is also important that the recovery method be such that recovery can be very fast, if in fact a *hot-standby* approach is going to be used (a la IBM's IMS/VS XRF [43] and Tandem's NonStop [4, 37]). This means that redo processing and, whenever possible, undo processing should be page-oriented (cf. always logical redos and undos in System R and SQL/DS for indexes and space management). It should also be possible to let the backup system start processing new transactions, even before the undo processing for the interrupted transactions completes. This is necessary because undo processing may take a long time if there were long update transactions.

Minimal overhead. Our goal is to have good performance both during normal and restart recovery processing. The overhead (log data volume, storage consumption, etc.) imposed by the recovery method in virtual and nonvolatile storages for accomplishing the above goals should be minimal. Contrast this with the space overhead caused by the shadow page technique. This goal also implied that we should minimize the number of pages that are modified (dirtied) during restart. The idea is to reduce the number of pages that have to be written back to nonvolatile storage and also to reduce CPU overhead. This rules out methods which, during restart recovery, first undo some committed changes that had already reached the nonvolatile storage before the failure and then redo them (see, e.g., [16, 21, 72, 78, 88]). It also rules out methods in which updates that are not present in a page on nonvolatile storage are undone unnecessarily (see, e.g., [41, 71, 88]). The method should not cause deadlocks involving transactions that are already rolling back. Further, the writing of CLRs should not result in an unbounded number of log records having to be written for a transaction because of the undoing of CLRs, if there were nested rollbacks or repeated system failures during rollbacks. It should also be possible to take checkpoints and image copies without quiescing significant activities in the system. The impact of these operations on other activities should be minimal. To contrast, checkpointing and image copying in System R cause major perturbations in the rest of the system [31].

As the reader will have realized by now, some of these goals are contradictory. Based on our knowledge of different developers' existing systems' features, experiences with IBM's existing transaction systems and contacts

with customers, we made the necessary tradeoffs. We were keen on learning from the past successes and mistakes involving many prototypes and products.

3. OVERVIEW OF ARIES

The aim of this section is to provide a brief overview of the new recovery method ARIES, which satisfies quite reasonably the goals that we set forth in Section 2. Issues like deferred and selective restart, parallelism during restart recovery, and so on will be discussed in the later sections of the paper.

ARIES guarantees the atomicity and durability properties of transactions in the fact of process, transaction, system and media failures. For this purpose, ARIES keeps track of the changes made to the database by using a log and it does write-ahead logging (WAL). Besides logging, on a per-affected-page basis, update activities performed during forward processing of transactions, ARIES also logs, typically using compensation log records (CLRs), updates performed during partial or total rollbacks of transactions during both normal and restart processing. Figure 3 gives an example of a partial rollback in which a transaction, after performing three updates, rolls back two of them and then starts going forward again. Because of the undo of the two updates, two CLRs are written. 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 or of nested rollbacks. This is to be contrasted with what happens in IMS, which may undo the same non-CLR multiple times, and in AS/400, DB2 and NonStop SQL, which, besides undoing the same non-CLR multiple times, may also undo CLRs one or more times (see Figure 4). These have caused severe problems in real-life customer situations.

In ARIES, as Figure 5 shows, when the undo of a log record causes a CLR to be written, the CLR, besides containing a description of the compensating action for redo purposes, is made to contain the *UndoNxtLSN* pointer which points to the *predecessor* of the just undone log record. The predecessor information is readily available since every log record, including a CLR, contains the *PrevLSN* pointer which points to the most recent preceding log record written by the same transaction. The UndoNxtLSN pointer allows us to determine precisely how much of the transaction has not been undone so far. In Figure 5, log record 3′, which is the CLR for log record 3, points to log record 2, which is the predecessor of log record 3. Thus, during rollback, the UndoNxtLSN field of the most recently written CLR keeps track of the progress of rollback. It tells the system from where to continue the rollback of the transaction, if a system failure were to interrupt the completion of the rollback or if a nested rollback were to be performed. It lets the system bypass those log records that had already been undone. Since CLRs are available to describe what actions are actually performed during the undo of an original action, the undo action need not be, in terms of which page(s) is affected, the exact inverse of the original action. That is, logical undo which allows very high concurrency to be supported is made possible. For example,

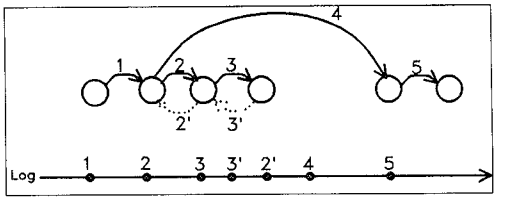
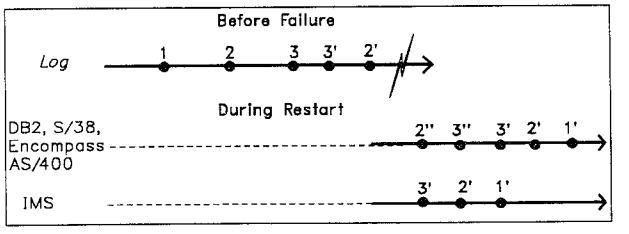


Fig. 3. Partial rollback example.

After performing 3 actions, the transaction performs a partial rollback by undoing actions 3 and 2, writing the compensation log records 3' and 2', and then starts going forward again and performs actions 4 and 5



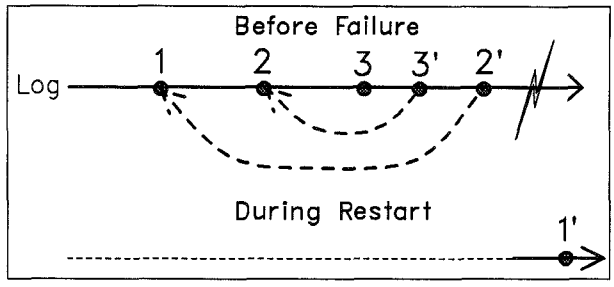
I' is the CLR for I and I'' is the CLR for I'

Fig. 4 Problem of compensating compensations or duplicate compensations, or both.

a key inserted on page 10 of a B^+-tree by one transaction may be moved to page 20 by another transaction before the key insertion is committed. Later, if the first transaction were to roll back, then the key will be located on page 20 by retraversing the tree and deleted from there. A CLR will be written to describe the key deletion on page 20. This permits page-oriented redo which is very efficient. [59, 62] describe ARIES/LHS and ARIES/IM which exploit this logical undo feature.

ARIES uses a single LSN on each page to track the page's state. Whenever a page is updated and a log record is written, the LSN of the log record is placed in the *page_LSN* field of the updated page. This tagging of the page with the LSN allows ARIES to precisely track, for restart- and media-recovery purposes, the state of the page with respect to logged updates for that page. It allows ARIES to support novel lock modes, using which, before an update performed on a record's field by one transaction is committed, another transaction may be permitted to modify the same data for specified operations.

Periodically during normal processing, ARIES takes checkpoints. The checkpoint log records identify the transactions that are active, their states, and the LSNs of their most recently written log records, and also the modified data (dirty data) that is in the buffer pool. The latter information is needed to determine from where the redo pass of restart recovery should begin its processing.



I′ is the Compensation Log Record for I
I′ points to the predecessor, if any, of I

Fig. 5. ARIES' technique for avoiding compensating compensations and duplicate compensations.

During restart recovery (see Figure 6), ARIES first scans the log, starting from the first record of the last checkpoint, up to the end of the log. During this *analysis pass*, information about dirty pages and transactions that were in progress at the time of the checkpoint is brought up to date as of the end of the log. The analysis pass uses the dirty pages information to determine the starting point (*RedoLSN*) for the log scan of the immediately following redo pass. The analysis pass also determines the list of transactions that are to be rolled back in the undo pass. For each in-progress transaction, the LSN of the most recently written log record will also be determined. Then, during the *redo pass*, ARIES *repeats history*, with respect to those updates logged on stable storage, but whose effects on the database pages did not get reflected on nonvolatile storage before the failure of the system. This is done for the updates of all transactions, including the updates of those transactions that had neither committed nor reached the in-doubt state of two-phase commit by the time of the system failure (i.e., even the missing updates of the so-called *loser* transactions are redone). This essentially reestablishes the state of the database as of the time of the system failure. A log record's update is redone if the affected page's page_LSN is less than the log record's LSN. No logging is performed when updates are redone. The redo pass obtains the locks needed to protect the uncommitted updates of those distributed transactions that will remain in the in-doubt (prepared) state [63, 64] at the end of restart recovery.

The next log pass is the *undo pass* during which all loser transactions' updates are rolled back, in reverse chronological order, in a single sweep of the log. This is done by continually taking the maximum of the LSNs of the next log record to be processed for each of the yet-to-be-completely-undone loser transactions, until no transaction remains to be undone. Unlike during the redo pass, performing undos is not a conditional operation during the undo pass (and during normal undo). That is, ARIES does not compare the page_LSN of the affected page to the LSN of the log record to decide

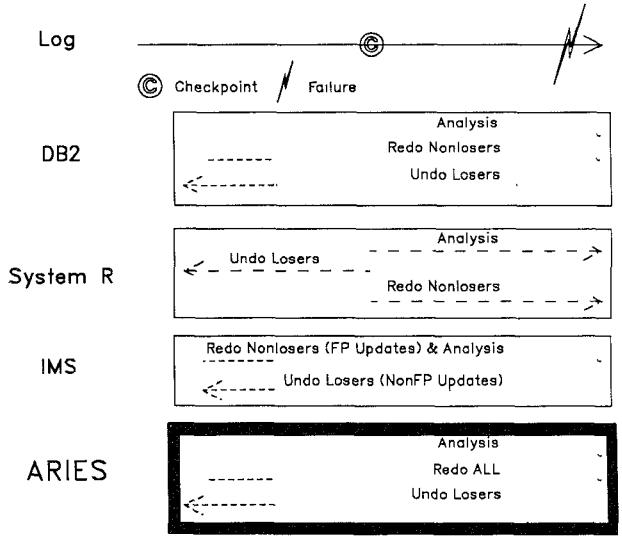


Fig. 6. Restart processing in different methods.

whether or not to undo the update. When a non-CLR is encountered for a transaction during the undo pass, if it is an undo-redo or undo-only log record, then its update is undone. In any case, the next record to process for that transaction is determined by looking at the PrevLSN of that non-CLR. Since CLRs are never undone (i.e., CLRs are not compensated—see Figure 5), when a CLR is encountered during undo, it is used just to determine the next log record to process by looking at the UndoNxtLSN field of the CLR.

For those transactions which were already rolling back at the time of the system failure, ARIES will rollback only those actions that had not already been undone. This is possible since history is repeated for such transactions and since the last CLR written for each transaction points (directly or indirectly) to the next non-CLR record that is to be undone. The net result is that, if only page-oriented undos are involved or logical undos generate only CLRs, then, for rolled back transactions, the number of CLRs written will be exactly equal to the number of undoable) log records written during forward processing of those transactions. This will be the case even if there are repeated failures during restart or if there are nested rollbacks.

4. DATA STRUCTURES

This section describes the major data structures that are used by ARIES.

4.1 Log Records

Below, we describe the important fields that may be present in different types of log records.

LSN. Address of the first byte of the log record in the ever-growing log address space. This is a monotonically increasing value. This is shown here as a field only to make it easier to describe ARIES. The LSN need not actually be stored in the record.

Type. Indicates whether this is a compensation record ('compensation'), a regular update record ('update'), a commit protocol-related record (e.g., 'prepare'), or a nontransaction-related record (e.g., 'OSfile_return').

TransID. Identifier of the transaction, if any, that wrote the log record.

PrevLSN. LSN of the preceding log record written by the same transaction. This field has a value of zero in nontransaction-related records and in the first log record of a transaction, thus avoiding the need for an explicit begin transaction log record.

PageID. Present only in records of type 'update' or 'compensation'. The identifier of the page to which the updates of this record were applied. This PageID will normally consist of two parts: an objectID (e.g., tablespaceID), and a page number within that object. ARIES can deal with a log record that contains updates for multiple pages. For ease of exposition, we assume that only one page is involved.

UndoNxtLSN. Present only in CLRs. It is the LSN of the next log record of this transaction that is to be processed during rollback. That is, UndoNxtLSN is the value of PrevLSN of the log record that the current log record is compensating. If there are no more log records to be undone, then this field contains a zero.

Data. This is the redo and/or undo data that describes the update that was performed. CLRs contain only redo information since they are never undone. Updates can be logged in a logical fashion. Changes to some fields (e.g., amount of free space) of that page need not be logged since they can be easily derived. The undo information and the redo information for the entire object need not be logged. It suffices if the changed fields alone are logged. For increment or decrement types of operations, before and after-images of the field are not needed. Information about the type of operation and the decrement or increment amount is enough. The information here would also be used to determine the appropriate action routine to be used to perform the redo and/or undo of this log record.

4.2 Page Structure

One of the fields in every page of the database is the *page_LSN* field. It contains the LSN of the log record that describes the latest update to the page. This record may be a regular update record or a CLR. ARIES expects the buffer manager to enforce the WAL protocol. Except for this, ARIES does not place any restrictions on the buffer page replacement policy. The steal buffer management policy may be used. In-place updating is performed on nonvolatile storage. Updates are applied immediately and directly to the

buffer version of the page containing the object. That is, no deferred updating as in INGRES [86] is performed. If it is found desirable, deferred updating and, consequently, deferred logging can be implemented. ARIES is flexible enough not to preclude those policies from being implemented.

4.3 Transaction Table

A table called the *transaction table* is used during restart recovery to track the state of active transactions. The table is initialized during the analysis pass from the most recent checkpoint's record(s) and is modified during the analysis of the log records written after the beginning of that checkpoint. During the undo pass, the entries of the table are also modified. If a checkpoint is taken during restart recovery, then the contents of the table will be included in the checkpoint record(s). The same table is also used during normal processing by the transaction manager. A description of the important fields of the transaction table follows:

TransID. Transaction ID.

State. Commit state of the transaction: prepared ('P'—also called in-doubt) or unprepared ('U').

LastLSN. The LSN of the latest log record written by the transaction.

UndoNxtLSN. The LSN of the next record to be processed during rollback. If the most recent log record written or seen for this transaction is an undoable non-CLR log record, then this field's value will be set to LastLSN. If that most recent log record is a CLR, then this field's value is set to the UndoNxtLSN value from that CLR.

4.4 Dirty_Pages Table

A table called the *dirty_pages table* is used to represent information about dirty buffer pages during normal processing. This table is also used during restart recovery. The actual implementation of this table may be done using hashing or via the deferred-writes queue mechanism of [96]. Each entry in the table consists of two fields: PageID and *RecLSN* (recovery LSN). During normal processing, when a nondirty page is being fixed in the buffers with the intention to modify, the buffer manager records in the *buffer pool* (BP) dirty_pages table, as RecLSN, the current end-of-log LSN, which will be the LSN of the next log record to be written. The value of RecLSN indicates from what point in the log there may be updates which are, possibly, not yet in the nonvolatile storage version of the page. Whenever pages are written back to nonvolatile storage, the corresponding entries in the BP dirty_pages table are removed. The contents of this table are included in the checkpoint record(s) that is written during normal processing. The *restart* dirty_pages table is initialized from the latest checkpoint's record(s) and is modified during the analysis of the other records during the analysis pass. The

minimum RecLSN value in the table gives the starting point for the redo pass during restart recovery.

5. NORMAL PROCESSING

This section discusses the actions that are performed as part of normal transaction processing. Section 6 discusses the actions that are performed as part of recovering from a system failure.

5.1 Updates

During normal processing, transactions may be in forward processing, partial rollback or total rollback. The rollbacks may be system- or application-initiated. The causes of rollbacks may be deadlocks, error conditions, integrity constraint violations, unexpected database state, etc.

If the granularity of locking is a record, then, when an update is to be performed on a record in a page, after the record is locked, that page is fixed in the buffer and latched in the X mode, the update is performed, a log record is appended to the log, the LSN of the log record is placed in the page_LSN field of the page and in the transaction table, and the page is unlatched and unfixed. The page latch is held during the call to the logger. This is done to ensure that the order of logging of updates of a page is the same as the order in which those updates are performed on the page. This is very important if some of the redo information is going to be logged physically (e.g., the amount of free space in the page) and repetition of history has to be guaranteed for the physical redo to work correctly. The page latch must be held during read and update operations to ensure physical consistency of the page contents. This is necessary because inserters and updaters of records might move records around within a page to do garbage collection. When such garbage collection is going on, no other transaction should be allowed to look at the page since they might get confused. Readers of pages latch in the S mode and modifiers latch in the X mode.

The data page latch is not held while any necessary index operations are performed. At most two page latches are held simultaneously (also see [57, 62]). This means that two transactions, T1 and T2, that are modifying different pieces of data may modify a particular data page in one order (T1, T2) and a particular index page in another order (T2, T1).[4] This scenario is impossible in System R and SQL/DS since in those systems, locks, instead of latches are used for providing physical consistency. Typically, all the (physical) page locks are released only at the end of the RSS (data manager) call. A single RSS call deals with modifying the data and all relevant indexes. This may involve waiting for many I/Os and locks. This means that deadlocks involving (physical) page locks alone or (physical) page locks and

[4] The situation gets very complicated if operations like increment/decrement are supported with high concurrency lock modes and indexes are allowed to be defined on fields on which such operations are supported. We are currently studying those situations.

(logical) record/key locks are possible. They have been a major problem in System R and SQL/DS.

Figure 7 depicts a situation at the time of a system failure which followed the commit of two transactions. The dotted lines show how up to date the states of pages P1 and P2 are on nonvolatile storage with respect to logged updates of those pages. During restart recovery, it must be realized that the most recent log record written for P1, which was written by a transaction which later committed, needs to be redone, and that there is nothing to be redone for P2. This situation points to the need for having the LSN to relate the state of a page on nonvolatile storage to a particular position in the log and the need for knowing where restart redo pass should begin by noting some information in the checkpoint record (see Section 5.4). For the example scenario, the restart redo log scan should begin at least from the log record representing the most recent update of P1 by T2, since that update needs to be redone.

It is not assumed that a single log record can always accommodate all the information needed to redo or undo the update operation. There may be instances when more than one record needs to be written for this purpose. For example, one record may be written with the undo information and another one with the redo information. In such cases, (1) the undo-only log record should be written before the redo-only log record is written, and (2) it is the LSN of the *redo-only log record* that should be placed in the page_LSN field. The first condition is enforced to make sure that we do not have a situation in which the redo-only record and not the undo-only record gets written to stable storage before a failure, and that during restart recovery, the redo of that redo-only log record is performed (because of the repeating history feature) only to realize later that there isn't an undo-only record to undo the effect of that operation. Given that the undo-only record is written before the redo-only record, the second condition ensures that we do not have a situation in which even though the page in nonvolatile storage already contains the update of the redo-only record, that same update gets redone unnecessarily during restart recovery because the page contained the LSN of the undo-only record instead of that of the redo-only record. This unnecessary redo could cause integrity problems if operation logging is being performed.

There may be some log records written during forward processing that cannot or should not be undone (prepare, free space inventory update, etc. records). These are identified as *redo-only* log records. See Section 10.3 for a discussion of this kind of situation for free space inventory updates.

Sometimes, the identity of the (data) record to be modified or read may not be known before a (data) page is examined. For example, during an insert, the record ID is not determined until the page is examined to find an empty slot. In such cases, the record lock must be obtained after the page is latched. To avoid waiting for a lock while holding a latch, which could lead to an undetected deadlock, the lock is requested *conditionally*, and if it is not granted, then the latch is released and the lock is requested *unconditionally*. Once the unconditionally requested lock is granted, the page is latched again, and any previously verified conditions are rechecked. This rechecking is

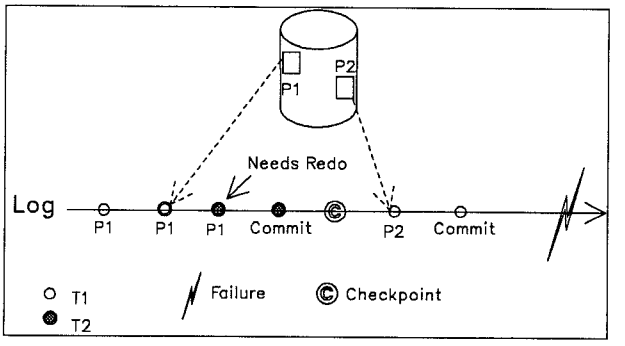


Fig. 7. Database state as a failure.

required because, after the page was unlatched, the conditions could have changed. The page_LSN value at the time of unlatching could be remembered to detect quickly, on relatching, if any changes could have possibly occurred. If the conditions are still found to be satisfied for performing the update, it is performed as described above. Otherwise, corrective actions are taken. If the conditionally requested lock is granted immediately, then the update can proceed as before.

If the granularity of locking is a page or something coarser than a page, then there is no need to latch the page since the lock on the page will be sufficient to isolate the executing transaction. Except for this change, the actions taken are the same as in the record-locking case. But, if the system is to support unlocked or *dirty* reads, then, even with page locking, a transaction that is updating a page should be made to hold the X latch on the page so that readers who are not acquiring locks are assured physical consistency if they hold an S latch while reading the page. Unlocked reads may also be performed by the image copy utility in the interest of causing the least amount of interference to normal transaction processing.

Applicability of ARIES is not restricted to only those systems in which locking is used as the concurrency control mechanism. Even other concurrency control schemes that are similar to locking, like the ones in [2], could be used with ARIES.

5.2 Total or Partial Rollbacks

To provide flexibility in limiting the extent of transaction rollbacks, the notion of a *savepoint* is supported [1, 31]. At any point during the execution of a transaction, a savepoint can be established. Any number of savepoints could be outstanding at a point in time. Typically, in a system like DB2, a savepoint is established before every SQL data manipulation command that might perform updates to the data. This is needed to support SQL statement-level atomicity. After executing for a while, the transaction or the system can request the undoing of all the updates performed after the establishment of a still outstanding savepoint. After such a partial rollback, the transaction can

continue execution and start going forward again (see Figure 3). A particular savepoint is no longer outstanding if a rollback has been performed to that savepoint or to a preceding one. When a savepoint is established, the LSN of the latest log record written by the transaction, called *SaveLSN*, is remembered in virtual storage. If the savepoint is being established at the beginning of the transaction (i.e., when it has not yet written a log record) SaveLSN is set to zero. When the transaction desires to roll back to a savepoint, it supplies the remembered SaveLSN. If the savepoint concept were to be exposed at the user level, then we would expect the system not to expose the SaveLSNs to the user but use some symbolic values or sequence numbers and do the mapping to LSNs internally, as is done in IMS [42] and INGRES [18].

Figure 8 describes the routine *ROLLBACK* which is used for rolling back to a savepoint. The input to the routine is the SaveLSN and the TransID. No locks are acquired during rollback, even though a latch is acquired during undo activity on a page. Since we have always ensured that latches do not get involved in deadlocks, a rolling back transaction cannot get involved in a deadlock, as in System R and R* [31, 64] and in the algorithms of [100]. During the rollback, the log records are undone in reverse chronological order and, for each log record that is undone, a CLR is written. For ease of exposition, assume that all the information about the undo action will fit in a single CLR. It is easy to extend ARIES to the case where multiple CLRs need to be written. It is possible that, when a logical undo is performed, some non-CLRs are sometimes written, as described in [59, 62]. As mentioned before, when a CLR is written, its UndoNxtLSN field is made to contain the PrevLSN value in the log record whose undo caused this CLR to be written. Since CLRs will never be undone, they don't have to contain undo information (e.g., before-images). Redo-only log records are ignored during rollback. When a non-CLR is encountered, after it is processed, the next record to process is determined by looking up its PrevLSN field. When a CLR is encountered during rollback, the UndoNxtLSN field of that record is looked up to determine the next log record to be processed. Thus, the UndoNxtLSN pointer helps us skip over already undone log records. This means that if a nested rollback were to occur, then, because of the UndoNxtLSN in CLRs, during the second rollback none of the log records that were undone during the first rollback would be processed again. Even though Figures 4, 5, and 13 describe partial rollback scenarios in conjunction with restart undos in the various recovery methods, it should be easy to see how nested rollbacks are handled efficiently by ARIES.

Being able to describe, via CLRs, the actions performed during undo gives us the flexibility of not having to force the undo actions to be the exact inverses of the original actions. In particular, the undo action could affect a page which was not involved in the original action. Such logical undo situations are possible in, for example, index management [62] and space management (see Section 10.3).

ARIES' guarantee of a bounded amount of logging during undo allows us to deal safely with small computer systems situations in which a circular online

```
ROLLBACK(SaveLSN,TransID);
UndoNxt := Trans_Table[TransID].UndoNxtLSN;          /* addr of 1st record to undo   */
WHILE SaveLSN < UndoNxt DO;                          /* loop thru all relevant records */
  LogRec := Log_Read(UndoNxt);                       /* read record to be processed  */
  SELECT (LogRec.Type)
    WHEN('update') DO;
      IF LogRec is undoable THEN DO;
        Page := fix&latch(LogRec.PageID,'X');
        Undo_Update(Page,LogRec);
        Log_Write('compensation',LogRec.TransID,Trans_Table[TransID].LastLSN,
                  LogRec.PageID,LogRec.PrevLSN, ...,LgLSN,Data);
                                                     /* write CLR */
        Page.LSN := LgLSN;
        Trans_Table[TransID].LastLSN := LgLSN;
        unfix&unlatch(Page);
      END;
      UndoNxt := LogRec.PrevLSN;
    END; /* WHEN('update') */
    WHEN('compensation')  UndoNxt := LogRec.UndoNxtLSN;     /* a CLR - nothing to undo */
    OTHERWISE  UndoNxt := LogRec.PrevLSN                    /* skip record and go to previous one */
  END; /* SELECT */
  Trans_Table[TransID].UndoNxtLSN := UndoNxt;
END;   /* WHILE  */
RETURN;
```

Fig. 8. Pseudocode for rollback.

log might be used and log space is at a premium. Knowing the bound, we can keep in reserve enough log space to be able to roll back all currently running transactions under critical conditions (e.g., log space shortage). The implementation of ARIES in the OS/2 Extended Edition Database Manager takes advantage of this.

When a transaction rolls back, the locks obtained after the establishment of the savepoint which is the target of the rollback may be released after the partial or total rollback is completed. In fact, systems like DB2 do not and cannot release any of the locks after a partial rollback because, after such a lock release, a later rollback may still cause the same updates to be undone again, thereby causing data inconsistencies. System R does release locks after a partial rollback completes. But, because ARIES never undoes CLRs nor ever undoes a particular non-CLR more than once, because of the chaining of the CLRs using the UndoNxtLSN field, during a (partial) rollback, when the transaction's very first update to a particular object is undone and a CLR is written for it, the system can release the lock on that object. This makes it possible to consider resolving deadlocks using partial rollbacks rather than always resorting to total rollbacks.

5.3 Transaction Termination

Assume that some form of two-phase commit protocol (e.g., Presumed Abort or Presumed Commit (see [63, 64])) is used to terminate transactions and that the *prepare* record which is *synchronously* written to the log as part of the protocol includes the list of update-type locks (IX, X, SIX, etc.) held by the transaction. The logging of the locks is done to ensure that if a system failure were to occur after a transaction enters the in-doubt state, then those locks could be reacquired, during restart recovery, to protect the uncommitted updates of the in-doubt transaction.[5] When the prepare record is written, the read locks (e.g., S and IS) could be released, if no new locks would be acquired later as part of getting into the prepare state in some other part of the distributed transaction (at the same site or a different site). To deal with actions (such as the dropping of objects) which may cause files to be erased, for the sake of avoiding the logging of such objects' complete contents, we postpone performing actions like erasing files until we are sure that the transaction is definitely committing [19]. We need to log these *pending actions* in the prepare record.

Once a transaction enters the in-doubt state, it is committed by writing an *end* record and releasing its locks. Once the end record is written, if there are any pending actions, they they must be performed. For each pending action which involves erasing or returning a file to the operating system, we write an *OSfile_return* redo-only log record. For ease of exposition, we assume that this log record is not associated with any particular transaction and that this action does not take place when a checkpoint is in progress.

[5] Another possibility is not to log the locks, but to regenerate the lock names during restart recovery by examining all the log records written by the in-doubt transaction—see Sections 6.1 and 6 4, and item 18 (Section 12) for further ramifications of this approach

A transaction in the *in-doubt* state is rolled back by writing a *rollback* record, rolling back the transaction to its beginning, discarding the pending actions list, releasing its locks, and then writing the end record. Whether or not the rollback and end records are *synchronously* written to stable storage will depend on the type of two-phase commit protocol used. Also, the writing of the prepare record may be avoided if the transaction is not a distributed one or is read-only.

5.4 Checkpoints

Periodically, checkpoints are taken to reduce the amount of work that needs to be performed during restart recovery. The work may relate to the extent of the log that needs to be examined, the number of data pages that have to be read from nonvolatile storage, etc. Checkpoints can be taken asynchronously (i.e., while transaction processing, including updates, is going on). Such a *fuzzy checkpoint* is initiated by writing a *begin_chkpt* record. Then the *end_chkpt* record is constructed by including in it the contents of the normal transaction table, the BP dirty_pages table, and any file mapping information for the objects (like tablespace, indexspace, etc.) that are "open" (i.e., for which BP dirty_pages table has entries). Only for simplicity of exposition, we assume that all the information can be accommodated in a single end_chkpt record. It is easy to deal with the case where multiple records are needed to log this information. Once the end_chkpt record is constructed, it is written to the log. Once that record reaches stable storage, the LSN of the begin_chkpt record is stored in the *master record* which is in a well-known place on stable storage. If a failure were to occur before the end_chkpt record migrates to stable storage, but after the begin_chkpt record migrates to stable storage, then that checkpoint is considered an *incomplete checkpoint*. Between the *begin_chkpt* and *end_chkpt* log records, transactions might have written other log records. If one or more transactions are likely to remain in the *in-doubt* state for a long time because of prolonged loss of contact with the commit coordinator, then it is a good idea to include in the end_chkpt record information about the update-type locks (e.g., X, IX and SIX) held by those transactions. This way, if a failure were to occur, then, during restart recovery, those locks could be reacquired without having to access the prepare records of those transactions.

Since latches may need to be acquired to read the dirty_pages table correctly while gathering the needed information, it is a good idea to gather the information a little at a time to reduce contention on the tables. For example, if the dirty_pages table has 1000 rows, during each latch acquisition 100 entries can be examined. If the already examined entries change before the end of the checkpoint, the recovery algorithms remain correct (see Figure 10). This is because, in computing the restart redo point, besides taking into account the minimum of the RecLSNs of the dirty pages included in the end_chkpt record, ARIES also takes into account the log records that were written by transactions since the beginning of the checkpoint. This is important because the effect of some of the updates that were performed since

the initiation of the checkpoint might not be reflected in the dirty page list that is recorded as part of the checkpoint.

ARIES does not require that any dirty pages be forced to nonvolatile storage during a checkpoint. The assumption is that the buffer manager is, on a continuous basis, writing out dirty pages in the background using system processes. The buffer manager can batch the writes and write multiple pages in one I/O operation. [96] gives details about how DB2 manages its buffer pools in this fashion. Even if there are some hot-spot pages which are frequently modified, the buffer manager has to ensure that those pages are written to nonvolatile storage reasonably often to reduce restart redo work, just in case a system failure were to occur. To avoid the prevention of updates to such hot-spot pages during an I/O operation, the buffer manager could make a copy of each of those pages and perform the I/O from the copy. This minimizes the data unavailability time for writes.

6. RESTART PROCESSING

When the transaction system restarts after a failure, recovery needs to be performed to bring the data to a consistent state and ensure the atomicity and durability properties of transactions. Figure 9 describes the *RESTART* routine that gets invoked at the beginning of the restart of a failed system. The input to this routine is the LSN of the master record which contains the pointer to the begin_chkpt record of the last complete checkpoint taken before site failure or shutdown. This routine invokes the routines for the analysis pass, the redo pass and the undo pass, in that order. The buffer pool dirty_pages table is updated appropriately. At the end of restart recovery, a checkpoint is taken.

For high availability, the duration of restart processing must be as short as possible. One way of accomplishing this is by exploiting parallelism during the redo and undo passes. Only if parallelism is going to be employed is it necessary to latch pages before they are modified during restart recovery. Ideas for improving data availability by allowing new transaction processing during recovery are explored in [60].

6.1 Analysis Pass

The first pass of the log that is made during restart recovery is the *analysis pass*. Figure 10 describes the *RESTART_ANALYSIS* routine that implements the analysis pass actions. The input to this routine is the LSN of the *master* record. The outputs of this routine are the transaction table, which contains the list of transactions which were in the *in-doubt* or unprepared state at the time of system failure or shutdown; the dirty_pages table, which contains the list of pages that were potentially dirty in the buffers when the system failed or was shut down; and the *RedoLSN*, which is the location on the log from which the redo pass must start processing the log. The only log records that may be written by this routine are end records for transactions that had totally rolled back before system failure, but for whom end records are missing.

```
RESTART(Master_Addr);
Restart_Analysis(Master_Addr,Trans_Table, Dirty_Pages, RedoLSN);
Restart_Redo(RedoLSN, Trans_Table, Dirty_Pages);
buffer pool Dirty_Pages table := Dirty_Pages;
remove entries for non-buffer-resident pages from the buffer pool Dirty_Pages table;
Restart_Undo(Trans_Table);
reacquire locks for prepared transactions;
checkpoint();
RETURN;
```

Fig. 9. Pseudocode for restart.

During this pass, if a log record is encountered for a page whose identity does not already appear in the dirty_pages table, then an entry is made in the table with the current log record's LSN as the page's RecLSN. The transaction table is modified to track the state changes of transactions and also to note the LSN of the most recent log record that would need to be undone if it were determined ultimately that the transaction had to be rolled back. If an OSfile_return log record is encountered, then any pages belonging to that file which are in the dirty_pages table are removed from the latter in order to make sure that no page belonging to that version of that file is accessed during the redo pass. The same file may be recreated and updated later, once the original operation causing the file erasure is committed. In that case, some pages of the recreated file will reappear in the dirty_pages table later with RecLSN values greater than the end_of_log LSN when the file was erased. The RedoLSN is the minimum RecLSN from the dirty_pages table at the end of the analysis pass. The redo pass can be skipped if there are no pages in the dirty_pages table.

It is not necessary that there be a separate analysis pass and, in fact, in the ARIES implementation in the OS/2 Extended Edition Database Manager there is no analysis pass. This is especially because, as we mentioned before (see also Section 6.2), in the redo pass, ARIES unconditionally redoes all missing updates. That is, it redoes them irrespective of whether they were logged by loser or nonloser transactions, unlike System R, SQL/DS and DB2. Hence, redo does not need to know the loser or nonloser status of a transaction. That information is, strictly speaking, needed only for the undo pass. This would not be true for a system (like DB2) in which for in-doubt transactions their update locks are reacquired by inferring the lock names from the log records of the in-doubt transactions, as they are encountered during the redo pass. This technique for reacquiring locks forces the RedoLSN computation to consider the Begin_LSNs of in-doubt transactions which in turn requires that we know, before the start of the redo pass, the identities of the in-doubt transactions.

Without the analysis pass, the transaction table could be constructed from the checkpoint record and the log records encountered during the redo pass. The RedoLSN would have to be the minimum(minimum(RecLSN from the dirty_pages table in the end_chkpt record), LSN(begin_chkpt record)). Suppression of the analysis pass would also require that other methods be used to

```
RESTART_ANALYSIS(Master_Addr, Trans_Table, Dirty_Pages, RedoLSN);
initialize the tables Trans_Table and Dirty_Pages to empty;
Master_Rec := Read_Disk(Master_Addr);
Open_Log_Scan(Master_Rec.ChkptLSN);                    /* open log scan at Begin_Chkpt record */
LogRec := Next_Log();                                      /* read in the Begin_Chkpt record */
LogRec := Next_Log();                                 /* read log record following Begin_Chkpt */
WHILE NOT(End_of_Log) DO;
  IF trans related record & LogRec.TransID NOT in Trans_Table THEN /* not chkpt/OSfile_return*/
    insert (LogRec.TransID,'U',LogRec.LSN,LogRec.PrevLSN) into Trans_Table;    /* log record */
  SELECT(LogRec.Type)
    WHEN('update' | 'compensation') DO;
      Trans_Table[LogRec.TransID].LastLSN := LogRec.LSN;
      IF LogRec.Type = 'update' THEN
        IF LogRec is undoable THEN Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.LSN;
      ELSE Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.UndoNxtLSN;
                                  /* next record to undo is the one pointed to by this CLR */
      IF LogRec is redoable & LogRec.PageID NOT IN Dirty_Pages THEN
        insert (LogRec.PageID, LogRec.LSN) into Dirty_Pages;
    END; /* WHEN('update' | 'compensation') */
    WHEN('Begin_Chkpt') ;  /* found an incomplete checkpoint's Begin_Chkpt record. ignore it */
    WHEN('End_Chkpt') DO;
      FOR each entry in LogRec.Tran_Table DO;
        IF TransID NOT IN Trans_Table THEN DO;
          insert entry(TransID,State,LastLSN,UndoNxtLSN) in Trans_Table;
        END;
      END; /* FOR */
      FOR each entry in LogRec.Dirty_PagLst DO;
        IF PageID NOT IN Dirty_Pages THEN insert entry(PageID,RecLSN) in Dirty_Pages;
        ELSE set RecLSN of Dirty_Pages entry to RecLSN in Dirty_PagLst;
      END; /* FOR */
    END; /* WHEN('End_Chkpt') */
    WHEN('prepare' | 'rollback') DO;
      IF LogRec.Type = 'prepare' THEN Trans_Table[LogRec.TransID].State := 'P';
      ELSE Trans_Table[LogRec.TransID].State := 'U';
      Trans_Table[LogRec.TransID].LastLSN := LogRec.LSN;
    END; /* WHEN('prepare' | 'rollback') */
    WHEN('end') delete Trans_Table entry for which TransID = LogRec.TransID;
    WHEN('OSfile_return') delete from Dirty_Pages all pages of returned file;
  END; /* SELECT */
  LogRec := Next_Log();
END; /* WHILE */
FOR EACH Trans_Table entry with (State = 'U') & (UndoNxtLSN = 0) DO; /* rolled back trans    */
  write end record and remove entry from Trans_Table;            /* with missing end record */
END; /* FOR */
RedoLSN := minimum(Dirty_Pages.RecLSN);                    /* return start position for redo */
RETURN;
```

Fig. 10. Pseudocode for restart analysis.

avoid processing updates to files which have been returned to the operating system. Another consequence is that the dirty_pages table used during the redo pass cannot be used to filter update log records which occur after the begin_chkpt record.

6.2 Redo Pass

The second pass of the log that is made during restart recovery is the *redo pass*. Figure 11 describes the *RESTART_REDO* routine that implements

```
RESTART_REDO(RedoLSN, Dirty_Pages);
Open_Log_Scan(RedoLSN);                          /* open log scan and position at restart pt */
LogRec := Next_Log();                             /* read log record at restart redo point   */
WHILE NOT(End_of_Log) DO;                         /* look at all records till end of log */
  IF LogRec.Type = ('update'|'compensation') & LogRec is redoable &
     LogRec.PageID IN Dirty_Pages & LogRec.LSN >= Dirty_Pages[LogRec.PageID].RecLSN
    THEN DO;              /* a redoable page update. updated page might not have made it to */
                          /* disk before sys failure. need to access page and check its LSN */
      Page := fix&latch(LogRec.PageID,'X');
      IF Page.LSN < LogRec.LSN THEN DO              /* update not on page. need to redo it */
        Redo_Update(Page,LogRec);                                        /* redo update */
        Page.LSN := LogRec.LSN;
      END;                                                             /* redid update */
      ELSE Dirty_Pages[LogRec.PageID].RecLSN := Page.LSN+1;      /* update already on page */
                    /* update dirty page list with correct info. this will happen if this */
                    /* page was written to disk after the checkpt but before sys failure  */
      unfix&unlatch(Page);
    END;                                                  /* LSN on page has to be checked */
  LogRec := Next_Log();                                          /* read next log record */
END;                                                            /* reading till end of log */
RETURN;
```

Fig. 11. Pseudocode for restart redo.

the redo pass actions. The inputs to this routine are the RedoLSN and the dirty_pages table supplied by the restart_analysis routine. No log records are written by this routine. The redo pass starts scanning the log records from the RedoLSN point. When a redoable log record is encountered, a check is made to see if the referenced page appears in the dirty_pages table. If it does and if the log record's LSN is greater than or equal to the RecLSN for the page in the table, then it is suspected that the page state might be such that the log record's update might have to be redone. To resolve this suspicion, the page is accessed. If the page's LSN is found to be less than the log record's LSN, then the update is redone. Thus, the RecLSN information serves to limit the number of pages which have to be examined. This routine reestablishes the database state as of the time of system failure. Even updates performed by loser transactions are redone. The rationale behind this repeating of history is explained in Section 10.1. It turns out that some of that redo of loser transactions' log records may be unnecessary. In [69] we have explored further the idea of restricting the repeating of history to possibly reduce the number of pages which get dirtied during this pass.

Since redo is page-oriented, only the pages with entries in the dirty_pages table may get modified during the redo pass. Only the pages listed in the dirty_pages table will be read and examined during this pass. Not all the pages that are read may require redo. This is because some of the pages that were dirty at the time of the last checkpoint or which became dirty later might have been written to nonvolatile storage before the system failure. Because of reasons like reducing log volume and saving some CPU overhead, we do not expect systems to write log records that identify the dirty pages that were written to nonvolatile storage, although that option is available and such log records can be used to eliminate the corresponding pages from

the dirty_pages table when those log records are encountered during the analysis pass. Even if such records were always to be written after I/Os complete, a system failure in a narrow window could prevent them from being written. The corresponding pages will not get modified during this pass.

For brevity, we do not discuss here as to how, if a failure were to occur after the logging of the end record of a transaction, but before the execution of all the pending actions of that transaction, the remaining pending actions are redone during the redo pass.

For exploiting parallelism, the availability of the information in the dirty_pages table gives us the possibility of initiating asynchronous I/Os in parallel to read all these pages so that they may be available in the buffers possibly before the corresponding log records are encountered in the redo pass. Since updates performed during the redo pass are not logged, we can also perform sophisticated things like building in-memory queues of log records which potentially need to be reapplied (as dictated by the information in the dirty_pages table) on a per page or group of pages basis and, as the asynchronously initiated I/Os complete and pages come into the buffer pool, processing the corresponding log record queues using multiple processes. This requires that each queue be dealt with by only one process. Updates to *different* pages may get applied in different orders from the order represented in the log. This does not violate any correctness properties since for a given page all its missing updates are reapplied in the same order as before. These parallelism ideas are also applicable to the context of supporting disaster recovery via remote backups [73].

6.3 Undo Pass

The third pass of the log that is made during restart recovery is the *undo pass*. Figure 12 describes the *RESTART_UNDO* routine that implements the undo pass actions. The input to this routine is the restart transaction table. The dirty_pages table is not consulted during this undo pass. Also, since history is repeated before the undo pass is initiated, the LSN on the page is not consulted to determine whether an undo operation should be performed or not. Contrast this with what we describe in Section 10.1 for systems like DB2 that do not repeat history but perform selective redo.

The restart_undo routine rolls back losers transactions, in reverse chronological order, in a single sweep of the log. This is done by continually taking the maximum of the LSNs of the next log record to be processed for each of the yet-to-be-completely-undone loser transactions, until no loser transaction remains to be undone. The next record to process for each transaction to be rolled back is determined by an entry in the transaction table for each of those transactions. The processing of the encountered log records is exactly as we described before in Section 5.2. In the process of rolling back the transactions, this routine writes CLRs. The buffer manager follows the usual WAL protocol while writing dirty pages to nonvolatile storage during the undo pass.

```
RESTART_UNDO(Trans_Table);
WHILE EXISTS(Trans with State = 'U' in Trans_Table) DO;
  UndoLSN := maximum(UndoNxtLSN) from Trans_Table entries with State = 'U';
                          /* pick up UndoNxtLSN of unprepared trans with maximum UndoNxtLSN */
  LogRec := Log_Read(UndoLSN);                      /* read log record to be undone or a CLR */
  SELECT(LogRec.Type)
    WHEN('update') DO;
      IF LogRec is undoable THEN DO;          /* record needs undoing (not redo-only record) */
        Page := fix&latch(LogRec.PageID,'X');
        Undo_Update(Page,LogRec);
        Log_Write('compensation',LogRec.TransID,Trans_Table[LogRec.TransID].LastLSN,
              LogRec.PageID,LogRec.PrevLSN, ...,LgLSN,Data);                  /* write CLR */
        Page.LSN := LgLSN;                                        /* store LSN of CLR in page */
        Trans_Table[LogRec.TransID].LastLSN := LgLSN;           /* store LSN of CLR in table */
        unfix&unlatch(Page);
      END;                                                          /* undoable record case */
      ELSE;                                              /* record cannot be undone - ignore it */
      Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.PrevLSN; /* next record to process is */
                                  /* the one preceding this record in its backward chain */
      IF LogRec.PrevLSN = 0 THEN DO;                    /* have undone completely - write end  */
        Log_Write('end',LogRec.TransID,Trans_Table[LogRec.TransID].LastLSN,...);
        delete Trans_Table entry where TransID = LogRec.TransID;  /* delete trans from table */
      END;                                                           /* trans fully undone */
    END; /* WHEN('update') */
    WHEN('compensation') Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.UndoNxtLSN;
                                                 /* pick up addr of next record to examine */
    WHEN('rollback'|'prepare') Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.PrevLSN;
                                                 /* pick up addr of next record to examine */
  END; /* SELECT */
END;   /* WHILE */
RETURN;
```

Fig. 12. Pseudocode for estart undo.

To exploit parallelism, the undo pass can also be performed using multiple processes. It is important that each transaction be dealt with completely by a single process because of the UndoNxtLSN chaining in the CLRs. This still leaves open the possibility of writing the CLRs first, without applying the undos to the pages (see Section 6.4 for problems in accomplishing this for objects that may require logical undos), and then redoing the CLRs in parallel, as explained in Section 6.2. In this fashion, the undo work of actually applying the changes to the pages can be performed in parallel, even for a single transaction.

Figure 13 depicts an example restart recovery scenario using ARIES. Here, all the log records describe updates to the same page. Before the failure, the page was written to disk after the second update. After that disk write, a partial rollback was performed (undo of log records 4 and 3) and then the transaction went forward (updates 5 and 6). During restart recovery, the missing updates (3, 4, 4′, 3′, 5 and 6) are first redone and then the undos (of 6, 5, 2 and 1) are performed. Each update log record will be matched with at most one CLR, regardless of how many times restart recovery is performed.

With ARIES, we have the option of allowing the continuation of loser transactions after restart recovery is completed. Since ARIES repeats history and supports the savepoint concept, we could, in the undo pass, roll back each

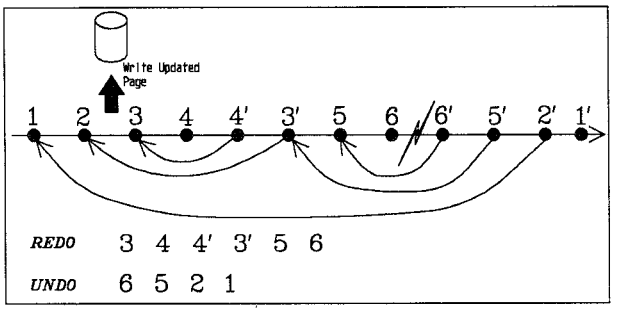


Fig. 13. Restart recovery example with ARIES.

loser only to its latest savepoint, instead of totally rolling back the loser transactions. Later, we could resume the transaction by invoking its application at a special entry point and passing enough information about the savepoint from which execution is to be resumed. Doing this correctly would require (1) the ability to generate lock names from the transaction's log records for its uncommitted, not undone updates, (2) reacquiring those locks before completing restart recovery, and (3) logging enough information whenever savepoints are established so that the system can restore cursor positions, application program state, and so on.

6.4 Selective or Deferred Restart

Sometimes, after a system failure, we may wish to restart the processing of new transactions as soon as possible. Hence, we may wish to defer doing some recovery work to a later point in time. This is usually done to reduce the amount of time during which some critical data is unavailable. It is accomplished by recovering such data first and then opening the system for the processing of new transactions. In DB2, for example, it is possible to perform restart recovery even when some of the objects for which redo and/or undo work needs to be performed are offline when the system is brought up. If some undo work needs to be performed for some loser transactions on those offline objects, then DB2 is able to write the CLRs alone and finish handling the transactions. This is possible because the CLRs can be generated based solely on the information in the non-CLR records written during the forward processing of the transactions [15]. Because page (or minipage, for indexes) is the smallest granularity of locking, the undo actions will be exact inverses of the original actions. That is, there are no logical undos in DB2. DB2 remembers, in an exceptions table (called the database allocation (DBA) table) that is maintained in the log and in virtual storage, the fact that those offline objects need to be recovered when they are brought online, before they are made accessible to other transactions [14]. The LSN ranges of log records to be applied are also remembered. Unless there are some in-doubt transactions with uncommitted updates to those objects, no locks need to be acquired to protect those objects since accesses to those objects will not be permitted until recovery is completed. When those objects are brought online, then

recovery is performed efficiently by rolling forward using the log records in the remembered ranges. Even during normal rollbacks, CLRs may be written for offline objects.

In ARIES also, we can take similar actions, provided none of the loser transactions has modified one or more of the offline objects that may require logical undos. This is because logical undos are based on the current state of the object. Redos are not at all a problem, since they are always page-oriented. For logical undos involving space management (see Section 10.3), generally we can take a conservative approach and generate the appropriate CLRs. For example, during the undo of an insert record operation, we can write a CLR for the space-related update stating that the page is 0% full. But for the high concurrency, index management methods of [62] this is not possible, since the effect of the logical undo (e.g., retraversing the index tree to do a key deletion), in terms of which page may be affected, is unpredictable; in fact, we cannot even predict when page-oriented undo will not work and hence logical undo is necessary.

It is not possible to handle the undos of some of the records of a transaction during restart recovery and handle the undos (possibly, logical) of the rest of the records at a later point in time, if the two sets of records are interspersed. Remember that in all the recovery methods, undo of a transaction is done in reverse chronological order. Hence, it is enough to remember, for each transaction, the next record to be processed during the undo; from that record, the PrevLSN and/or the UndoNxtLSN chain leads us to all the other records to be processed.

Even under the circumstances where one or more of the loser transactions have to perform, potentially logical, undos on some offline objects, if deferred restart needs to be supported, then we suggest the following algorithm:

1. Perform the repeating of history for the *online* objects, as usual; postpone it for the *offline* objects and remember the log ranges.
2. Proceed with the undo pass as usual, but stop undoing a loser transaction when one of its log records is encountered for which a CLR cannot be generated for the above reasons. Call such a transaction a *stopped transaction*. But continue undoing the other, unstopped transactions.
3. For the stopped transactions, acquire locks to protect their updates which have not yet been undone. This could be done as part of the undo pass by continuing to follow the pointers, as usual, even for the stopped transactions and acquiring locks based on the encountered non-CLRs that were written by the stopped transactions.
4. When restart recovery is completed and later the previously offline objects are made online, first repeat history based on the remembered log ranges and then continue with the undoing of the stopped transactions. After each of the stopped transactions is totally rolled back, release its still held locks.
5. Whenever an offline object becomes online, when the repeating of history is completed for that object, new transactions can be allowed to access that object in parallel with the further undoing of all of the stopped transactions that can make progress.

The above requires the ability to generate lock names based on the information in the update (non-CLR) log records. DB2 is doing that already for in-doubt transactions.

Even if none of the objects to be recovered is offline, but it is desired that the processing of new transactions start before the rollbacks of the loser transactions are completed, then we can accommodate it by doing the following: (1) first repeat history and reacquire, based on their log records, the locks for the uncommitted updates of the loser and in-doubt transactions, and (2) then start processing new transactions even as the rollbacks of the loser transactions are performed in parallel. The locks acquired in step (1) are released as each loser transaction's rollback completes. Performing step (1) requires that the restart RedoLSN be adjusted appropriately to ensure that all the log records of the loser transactions are encountered during the redo pass. If a loser transaction was already rolling back at the time of the system failure, then, with the information obtained during the analysis pass for such a transaction, it will be known as to which log records remain to be undone. These are the log records whose LSNs are less than or equal to the UndoNxtLSN of the transaction's last CLR. Locks need to be obtained during the redo pass only for those updates that have not yet been undone.

If a long transaction is being rolled back and we would like to release some of its locks as soon as possible, then we can mark specially those log records which represent the first update by that transaction on the corresponding object (e.g., record, if record locking is in effect) and then release that object's lock as soon as the corresponding log record is undone. This works only because we do not undo CLRs and because we do not undo the same non-CLR more than once; hence, it will not work in systems that undo CLRs (e.g., Encompass, AS/400, DB2) or that undo a non-CLR more than once (e.g., IMS). This early release of locks can be performed in ARIES during normal transaction undo to possibly permit resolution of deadlocks using partial rollbacks.

7. CHECKPOINTS DURING RESTART

In this section, we describe how the impact of failures on CPU processing and I/O can be reduced by, optionally, taking checkpoints during different stages of restart recovery processing.

Analysis pass. By taking a checkpoint at the end of the analysis pass, we can save some work if a failure were to occur during recovery. The entries of the transaction table of this checkpoint will be the same as the entries of the transaction table at the end of the analysis pass. The entries of the dirty_pages list of this checkpoint will be the same as the entries that the *restart* dirty_pages table contains at the end of the analysis pass. This is different from what happens during a normal checkpoint. For the latter, the dirty_pages list is obtained from the buffer pool (BP) dirty_pages table.

Redo pass. At the beginning of the redo pass, the buffer manager (BM) is notified so that, whenever it writes out a modified page to nonvolatile storage during the redo pass, it will change the restart dirty_pages table entry for that page by making the RecLSN be equal to the LSN of that log record such

that all log records up to that log record had been processed. It is enough if BM manipulates the restart dirty_pages table in this fashion. BM does not have to maintain its own dirty_pages table as it does during normal processing. Of course, it should still be keeping track of what pages are currently in the buffers. The above allow checkpoints to be taken any time during the redo pass to reduce the amount of the log that would need to be redone if a failure were to occur before the end of the redo pass. The entries of the dirty_pages list of this checkpoint will be the same as the entries of the *restart* dirty_pages table at the time of the checkpoint. The entries of the transaction table of this checkpoint will be the same as the entries of the transaction table at the end of the analysis pass. This checkpointing is not affected by whether or not parallelism is employed in the redo pass.

Undo pass. At the beginning of the undo pass, the restart dirty_pages table becomes the BP dirty_pages table. At this point, the table is cleaned up by removing those entries for which the corresponding pages are no longer in the buffers. From then onward, the BP manager manipulates this table as it does during normal processing—removing entries when pages are written to nonvolatile storage, adding entries when pages are about to become dirty, etc. During the undo pass, the entries of the transaction table are modified as during normal undo. If a checkpoint is taken any time during the undo pass, then the entries of the dirty_pages list of that checkpoint are the same as the entries of the BP dirty_pages table at the time of the checkpoint. The entries of the transaction table of this checkpoint will be the same as the entries of the transaction table at that time.

In System R, during restart recovery, sometimes it may be required that a checkpoint be taken to free up some physical pages (the shadow pages) for more undo or redo work to be performed. This is another consequence of the fact that history cannot be repeated in System R. This complicates the restart logic since the view depicted in Figure 17 would no longer be true after a restart checkpoint completes. The restart checkpoint logic and its effect on a restart following a system failure during an earlier restart were considered too complex to be describable in [31]. ARIES is able to easily accommodate checkpoints during restart. While these checkpoints are optional in our case, they may be forced to take place in System R.

8. MEDIA RECOVERY

We will assume that media recovery will be required at the level of a file or some such (like DBspace, tablespace, etc.) entity. A *fuzzy image copy* (also called a *fuzzy archive dump*) operation involving such an entity can be performed concurrently with modifications to the entity by other transactions. With such a high concurrency image copy method, the image copy might contain some uncommitted updates, in contrast to the method of [52]. Of course, if desired, we could also easily produce an image copy with no uncommitted updates. Let us assume that the image copying is performed directly from the nonvolatile storage version of the entity. This means that

more recent versions of some of the copied pages may be present in the transaction system's buffers. Copying directly from the nonvolatile storage version of the object would usually be much more efficient since the device geometry can be exploited during such a copy operation and since the buffer manager overheads will be eliminated. Since the transaction system does not have to be up for the direct copying, it may also be more convenient than copying via the transaction system's buffers. If the latter is found desirable (e.g., to support incremental image copying, as described in [13]), then it is easy to modify the presented method to accommodate it. Of course, in that case, some minimal amount of synchronization will be needed. For example, latching at the page level, but no locking will be needed.

When the fuzzy image copy operation is initiated, the location of the begin_chkpt record of the most recent complete checkpoint is noted and remembered along with the image copy data. Let us call this checkpoint the *image copy checkpoint*. The assertion that can be made based on this checkpoint information is that all updates that had been logged in log records with LSNs less than minimum(minimum(RecLSNs of dirty pages of the image-copied entity in the image copy checkpoint's end_chkpt record), LSN(begin_chkpt record of the image copy checkpoint)) would have been externalized to nonvolatile storage by the time the fuzzy image copy operation began. Hence, the image-copied version of the entity would be at least as up to date as of that point in the log. We call that point the *media recovery redo point*. The reason for taking into account the LSN of the begin_chkpt record in computing the media recovery redo point is the same as the one given in Section 5.4 while discussing the computation of the restart redo point.

When media recovery is required, the image-copied version of the entity is reloaded and then a redo scan is initiated starting from the media recovery redo point. During the redo scan, all the log records relating to the entity being recovered are processed and the corresponding updates are applied, unless the information in the image copy checkpoint record's dirty_pages list or the LSN on the page makes it unnecessary. Unlike during *restart* redo, if a log record refers to a page that is not in the dirty_pages list and the log record's LSN is greater than the LSN of the begin_chkpt log record of the image copy checkpoint, then that page must be accessed and its LSN compared to the log record's LSN to check if the update must be redone. Once the end of the log is reached, if there are any in-progress transactions, then those transactions that had made changes to the entity are undone, as in the undo pass of restart recovery. The information about the identities, etc. of such transactions may be kept separately somewhere (e.g., in an exceptions table such as the DBA table in DB2—see Section 6.4) or may be obtained by performing an analysis pass from the last complete checkpoint in the log until the end of the log.

Page-oriented logging provides recovery independence amongst objects. Since, in ARIES, every database page's update is logged separately, even if an arbitrary database page is damaged in the nonvolatile storage and the page needs recovery, the recovery can be accomplished easily by extracting

an earlier copy of that page from an image copy and rolling forward that version of the page using the log as described above. This is to be contrasted with systems like System R in which, since for some pages' updates (e.g., index and space management pages') log records are not written, recovery from damage to such a page may require the expensive operation of reconstructing the entire object (e.g., rebuilding the complete index even when only one page of an index is damaged). Also, even for pages for which logging is performed explicitly (e.g., data pages in System R), if CLRs are not written when undo is performed, then bringing a page's state up to date by starting from the image copy state would require paying attention to the log records representing the transaction state (commit, partial or total rollback) to determine what actions, if any, should be undone. If any transactions had rolled back partially or totally, then backward scans of such transactions would be required to see if they made any changes to the page being recovered so that they are undone. These backward scans may result in useless work being performed, if it turns out that some rolled back transaction had not made any changes to the page being recovered. An alternative would be to preprocess the log and place forward pointers to skip over rolled back log records, as it is done in System R during the analysis pass of restart recovery (see Section 10.2 and Figure 18).

Individual pages of the database may be corrupted not only because of media problems but also because of an abnormal process termination while the process is actively making changes to a page in the buffer pool and before the process gets a chance to write a log record describing the changes. If the database code is executed by the application process itself, which is what performance-conscious systems like DB2 implement, such abnormal terminations may occur because of the user's interruption (e.g., by hitting the *attention* key) or due to the operating system's action on noting that the process had exhausted its CPU time limit. It is generally an expensive operation to put the process in an uninterruptable state before every page update. Given all these circumstances, an efficient way to recover the corrupted page is to read the uncorrupted version of the page from the nonvolatile storage and bring it up to date by rolling forward the page state using all relevant log records for that page. The roll-forward redo scan of the log is started from the RecLSN remembered for the buffer by the buffer manager. DB2 does this kind of internal recovery operation automatically [15]. The corruption of a page is detected by using a bit in the page header. The bit is set to '1' after the page is fixed and X-latched. Once the update operation is complete (i.e., page updated, update logged and page LSN modified), the bit is reset to '0'. Given this, whenever a page is latched, for read or write, first this bit is tested to see if its value is equal to '1', in which case automatic page recovery is initiated. From an availability viewpoint, it is unacceptable to bring down the entire transaction system to recover from such a *broken page* situation by letting restart recovery redo all those logged updates that were in the corrupted page but were missing in the uncorrupted version of the page on nonvolatile storage. A related problem is to make sure that for those pages that were left in the fixed state by the abnormally

terminating process, unfix calls are issued by the transaction system. By leaving enough *footprints* around before performing operations like fix, unfix and latch, the user process aids system processes in performing the necessary clean-ups.

For the variety of reasons mentioned in this section and elsewhere, writing CLRs is a very good idea even if the system is supporting only page locking. This is to be contrasted with the no-CLRs approach, suggested in [52], which supports only page locking.

9. NESTED TOP ACTIONS

There are times when we would like some updates of a transaction to be committed, irrespective of whether the transaction ultimately commits or not. We do need the atomicity property for these updates themselves. This is illustrated in the context of file extension. After a transaction extends a file which causes updates to some system data in the database, other transactions may be allowed to use the extended area prior to the commit of the extending transaction. If the extending transaction were to roll back, then it would not be acceptable to undo the effects of the extension. Such an undo might very well lead to a loss of updates performed by the other committed transactions. On the other hand, if the extension-related updates to the system data in the database were themselves interrupted by a failure before their completion, it is necessary to undo them. These kinds of actions have been traditionally performed by starting independent transactions, called *top actions* [51]. A transaction initiating such an independent transaction waits until that independent transaction commits before proceeding. The independent transaction mechanism is, of course, vulnerable to lock conflicts between the initiating transaction and the independent transaction, which would be unacceptable.

In ARIES, using the concept of a *nested top action*, we are able to support the above requirement very efficiently, without having to initiate independent transactions to perform the actions. A nested top action, for our purposes, is taken to mean any subsequence of actions of a transaction which should not be undone once the sequence is complete and some later action which is dependent on the nested top action is logged to stable storage, irrespective of the outcome of the enclosing transaction.

A transaction execution performing a sequence of actions which define a nested top action consists of the following steps:

(1) ascertaining the position of the current transaction's last log record;
(2) logging the redo and undo information associated with the actions of the nested top action; and
(3) on completion of the nested top action, writing a *dummy CLR* whose UndoNxtLSN points to the log record whose position was remembered in step (1).

We assume that the effects of any actions like creating a file and their associated updates to system data normally resident outside the database are externalized, before the dummy CLR is written. When we discuss redo, we are referring to only the system data that is resident in the database itself.

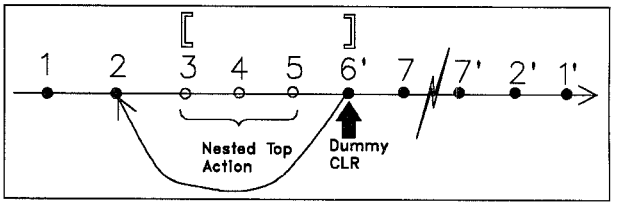


Fig. 14. Nested top action example.

Using this nested top action approach, if the enclosing transaction were to roll back after the completion of the nested top action, then the dummy CLR will ensure that the updates performed as part of the nested top action are not undone. If a system failure were to occur before the dummy CLR is written, then the incomplete nested top action will be undone since the nested top action's log records are written as undo-redo (as opposed to redo-only) log records. This provides the desired atomicity property for the nested top action. Unlike for the normal CLRs, there is nothing to redo when a dummy CLR is encountered during the redo pass. The dummy CLR in a sense can be thought of as the commit record for the nested top action. The advantage of our approach is that the enclosing transaction need not wait for this record to be forced to stable storage before proceeding with its subsequent actions.[6] Also, we do not pay the price of starting a new transaction. Nor do we run into lock conflict problems. Contrast this approach with the costly independent-transaction approach.

Figure 14 gives an example of a nested top action consisting of the actions 3, 4 and 5. Log record 6′ acts as the dummy CLR. Even though the enclosing transaction's activity is interrupted by a failure and hence it needs to be rolled back, 6′ ensures that the nested top action is not undone.

It should be emphasized that the nested top action implementation relies on repeating history. If the nested top action consists of only a single update, then we can log that update using a single *redo-only* log record and avoid writing the dummy CLR. Applications of the nested top action concept in the context of a hash-based storage method and index management can be found in [59, 62].

10. RECOVERY PARADIGMS

This section describes some of the problems associated with providing fine-granularity (e.g., record) locking and handling transaction rollbacks. Some additional discussion can be found in [97]. Our aim is to show how certain features of the existing recovery methods caused us difficulties in accomplishing our goals and to motivate the need for certain features which we had to include in ARIES. In particular, we show why some of the recovery paradigms of System R, which were developed in the context of the shadow page

[6] The dummy CLR may have to be forced if some *unlogged* updates may be performed later by other transactions which depended on the nested top action having completed.

technique, are inappropriate when WAL is to be used and there is a need for high levels of concurrency. In the past, one or more of those System R paradigms have been adopted in the context of WAL, leading to the design of algorithms with limitations and/or errors [3, 15, 16, 52, 71, 72, 78, 82, 88]. The System R paradigms that are of interest are:

—selective redo during restart recovery.
—undo work preceding redo work during restart recovery.
—no logging of updates performed during transaction rollback (i.e., no CLRs).
—no logging of index and space management information changes.
—no tracking of page state on page itself to relate it to logged updates (i.e., no LSNs on pages).

10.1 Selective Redo

The goal of this subsection is to introduce the concept of selective redo that has been implemented in many systems and to show the problems that it introduces in supporting fine-granularity locking with WAL-based recovery. The aim is to motivate why ARIES repeats history.

When transaction systems restart after failures, they generally perform database recovery updates in 2 passes of the log: a redo pass and an undo pass (see Figure 6). System R first performs the undo pass and then the redo pass. As we will show later, the System R paradigm of *undo preceding redo is incorrect with WAL and fine-granularity locking*. The WAL-based DB2, on the other hand, does just the opposite. During the redo pass, System R redoes only the actions of committed and prepared (i.e., in-doubt) transactions [31]. We call this *selective redo*. While the selective redo paradigm of System R intuitively seems to be the efficient approach to take, it has many pitfalls, as we discuss below.

Some WAL-based systems, such as DB2, support only page locking and perform selective redo [15]. This approach will lead to data inconsistencies in such systems, if record locking were to be implemented. Let us consider a WAL technique in which each page contains an LSN as described before. During the redo pass, the page LSN is compared to the LSN of a log record describing an update to the page to determine whether the log record's update needs to be reapplied to the page. If the page LSN is less than the log record's LSN, then the update is redone and the page's LSN is set to the log record's LSN (see Figure 15). During the undo pass, if the page LSN is less than the LSN of the log record to be undone, then no undo action is performed on the page. Otherwise, undo is performed on the page. Whether or not undo needs to be actually performed on the page, a CLR describing the updates that would have been performed as part of the undo operation is always written, when the transaction's actions are being rolled back. The CLR is written, even when the page does not contain the update, just to make media recovery simpler and not force it to handle rolled back updates in a special way. Writing the CLR when an undo is not actually performed on the page turns out to be necessary also when handling a failure of the system

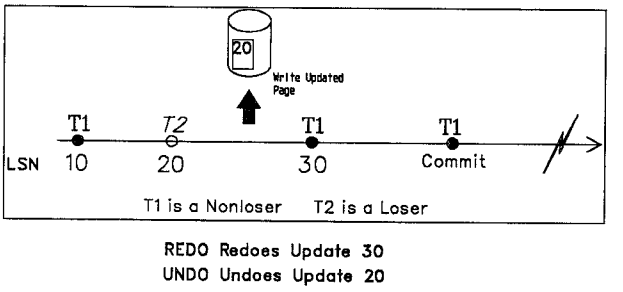


Fig. 15. Selective redo with WAL—problem-free scenario.

during restart recovery. This will happen, if there was an update U2 for page P1 which did not have to be undone, but there was an earlier update U1 for P1 which had to be undone, resulting in U1′ (CLR for U1) being written and P1's LSN being changed to the LSN of U1′ (> LSN of U2). After that, if P1 were to be written to nonvolatile storage before a system failure interrupts the completion of this restart, then, during the next restart, it would appear as if P1 contains the update U2 and an attempt would be made to undo it. On the other hand, if U2′ had been written, then there would not be any problem. It should be emphasized that this problem arises even when only page locking is used, as is the case with DB2 [15].

Given these properties of the selective redo WAL-based method under discussion, we would lose track of the state of a page with respect to a losing (in-progress or in-rollback) transaction in the situation where the page modified first by the losing transaction (say, update with LSN 20 by T2) was subsequently modified by a nonloser transaction's update (say, update with LSN 30 by T1) which had to be redone. The latter would have pushed the LSN of the page beyond the value established by the loser. So, when the time comes to undo the loser, we would not know if its update needs to be undone or not. Figures 15 and 16 illustrate this problem with selective redo and fine-granularity locking. In the latter scenario, not redoing the update with LSN 20 since it belongs to a loser transaction, but redoing the update with LSN 30 since it belongs to a nonloser transaction, causes the undo pass to perform the undo of the former update even though it is not present in the page. This is because the undo logic relies on the page_LSN value to determine whether or not an update should be undone (undo if page_LSN is greater than or equal to log record's LSN). By not repeating history, the page_LSN is no longer a true indicator of the current state of the page.

Undoing an action even when its effect is not present in a page will be harmless only under certain conditions; for example, with physical/byte-oriented locking and logging, as they are implemented in IMS [76], VAX DBMS and VAX Rdb/VMS [81], and other systems [6], there is no automatic reuse of freed space, and unique keys for all records. With operation logging, data inconsistencies will be caused by undoing an original operation whose effect is not present in the page.

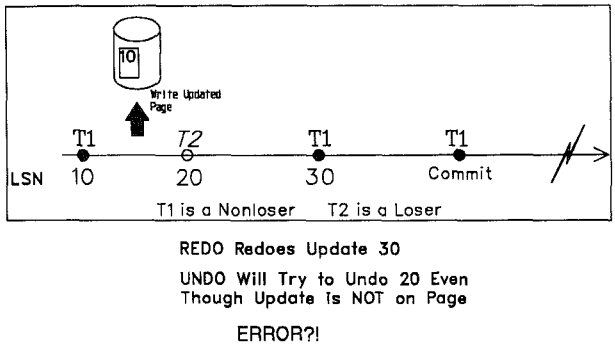


Fig. 16. Selective redo with WAL—problem scenario.

Reversing the order of the selective redo and the undo passes will not solve the problem either. This *incorrect* approach is suggested in [3]. If the undo pass were to precede the redo pass, then we might lose track of which actions need to be redone. In Figure 15, the undo of 20 would make the page LSN become greater than 30, because of the writing of a CLR and the assignment of that CLR's LSN to the page. Since, during the redo pass, a log record's update is redone only if the page_LSN is less than the log record's LSN, we would not redo 30 even though that update is not present on the page. Not redoing that update would violate the durability and atomicity properties of transactions.

The use of the shadow page technique by System R makes it unnecessary to have the concept of page_LSN in that system to determine what needs to be undone and what needs to be redone. With the shadow page technique, during a checkpoint, an action consistent version of the database, called the *shadow version*, is saved on nonvolatile storage. Updates between two checkpoints create a new version of the updated page, thus constituting the *current version* of the database (see Figure 1). During restart, recovery is performed from the *shadow* version, and shadowing is done even during restart recovery. As a result, there is no ambiguity about which updates are in the database and which are not. All updates logged after the last checkpoint are not in the database, and all updates logged before the checkpoint are in the database.[7] This is one reason the System R recovery method functions correctly even with selective redo. The other reason is that index and space management changes are not logged, but are redone or undone logically.[8]

[7] This simple view, as it is depicted in Figure 17, is not completely accurate—see Section 10.2.

[8] In fact, if index changes had been logged, then selective redo would not have worked. The problem would have come from structure modifications (like page split) which were performed after the last checkpoint by *loser* transactions which were taken advantage of later by transactions which ultimately committed. Even if logical undo were performed (if necessary), if redo was page oriented, selective redo would have caused problems. To make it work, the structure modifications could have been performed using separate transactions. Of course, this would have been very expensive. For an alternate, efficient solution, see [62].

As was described before, ARIES does not perform selective redo, but *repeats history*. Apart from allowing us to support fine-granularity locking, repeating history has another beneficial side effect. It gives us the ability to commit some actions of a transaction irrespective of whether the transaction ultimately commits or not, as was described in Section 9.

10.2 Rollback State

The goal of this subsection is to discuss the difficulties introduced by rollbacks in tracking their progress and how writing CLRs that describe updates performed during rollbacks solves some of the problems. While the concept of writing CLRs has been implemented in many systems and has been around for a long time, there has not really been, in the literature, a significant discussion of CLRs, problems relating to them and the advantages of writing them. Their utility and the fundamental role that they play in recovery have not been well recognized by the research community. In fact, whether undone actions could be undone and what additional problems these would present were left as open questions in [56]. In this section and elsewhere in this paper, in the appropriate contexts, we try to note all the known advantages of writing CLRs. We summarize these advantages in Section 13.

A transaction may totally or partially roll back its actions for any number of reasons. For example, a unique key violation will cause only the rollback of the update statement causing the violation and not of the entire transaction. Figure 3 illustrates a partial roll back. Supporting partial rollback [1, 31], at least internally, if not also at the application level, is a very important requirement for present-day transaction systems. Since a transaction may be rolling back when a failure occurs and since some of the effects of the updates performed during the rollback might have been written to nonvolatile storage, we need a way to keep track of the state of progress of transaction rollback. It is relatively easy to do this in System R. The only time we care about the transaction state in System R is at the time a checkpoint is taken. So, the checkpoint record in System R keeps track of the next record to be undone for each of the active transactions, some of which may already be rolling back. The rollback state of a transaction at the time of a system failure is unimportant since the database changes performed after the last checkpoint are not *visible* in the database during restart. That is, restart recovery starts from the state of the database as of the last checkpoint before the system failure—this is the shadow version of the database at the time of system failure. Despite this, since CLRs are never written, System R needs to do some special processing to handle those committed or in-doubt transactions which initiated and completed partial rollbacks after the last checkpoint. The special handling is to avoid the need for multiple passes over the log during the redo pass. The designers wanted to avoid redoing some actions only to have to undo them a little later with a backward scan, when the information about a partial rollback having occurred is encountered.

Figure 18 depicts an example of a restart recovery scenario for System R. All log records are written by the same transaction, say T1. In the checkpoint

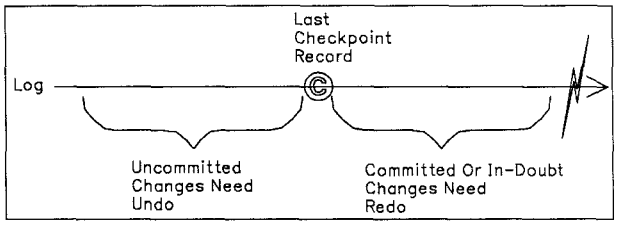


Fig. 17. Simple view of recovery processing in System R

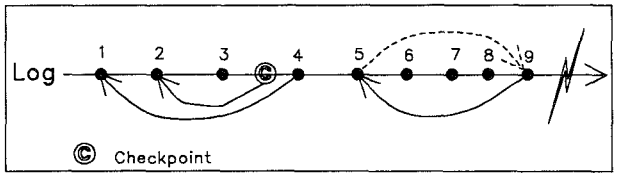


Fig. 18. Partial rollback handling in System R.

record, the information for T1 points to log record 2 since by the time the checkpoint was taken log record 3 had already been undone because of a partial rollback. System R not only does not write CLRs, but it also does not write a separate log record to say that a partial rollback took place. Such information must be inferred from the breakage in the chaining of the log records of a transaction. Ordinarily, a log record written by a transaction points to the record that was most recently written by that transaction via the PrevLSN pointer. But the first forward processing log record written after the completion of a partial rollback does not follow this protocol. When we examine, as part of the analysis pass, log record 4 and notice that its Prev_LSN pointer is pointing to 1, instead of the immediately preceding log record 3, we conclude that the partial rollback that started with the undo of 3 ended with the undo of 2. Since, during restart, the database state from which recovery needs to be performed is the state of the database as of the last checkpoint, log record 2 definitely needs to be undone. Whether 1 needs to be undone or not will depend on whether T1 is a losing transaction or not.

During the analysis pass it is determined that log record 9 points to log record 5 and hence it is concluded that a partial rollback had caused the undo of log records 6, 7, and 8. To ensure that the rolled back records are not redone during the redo pass, the log is patched by putting a forward pointer during the analysis pass in log record 5 to make it point to log record 9.

If log record 9 is a commit record then, during the undo pass, log record 2 will be undone and during the redo pass log records 4 and 5 will be redone. Here, the same transaction is involved both in the undo pass and in the redo pass. To see why the undo pass has to precede the redo pass in System R,[9]

[9] In the other systems, because of the fact that CLRs are written and that, sometimes, page LSNs are compared with log record's LSNs to determine whether redo needs to be performed or not, the redo pass *precedes* the undo pass—see the Section "10.1. Selective Redo" and Figure 6.

consider the following scenario: Since a transaction that deleted a record is allowed to reuse that record's ID for a record inserted later by the same transaction, in the above case, a record might have been deleted because of the partial rollback, which had to be dealt with in the undo pass, and that record's ID might have been reused in the portion of the transaction that is dealt with in the redo pass. To repeat history with respect to the original sequence of actions be fore the failure, the undo must be performed before the redo is performed.

If 9 is neither a commit record nor a prepare record, then the transaction will be determined to be a loser and during the undo pass log records 2 and 1 will be undone. In the redo pass, none of the records will be redone.

Since CLRs are not written in System R and hence the exact way in which one transaction's undo operations were interspersed with other transactions' forward processing or undo actions is not known, the processing, for a given page as well as across different pages during restart may be quite different from what happened during normal processing (i.e., *repeating history* is impossible to guarantee). Not logging index changes in System R also further contributes to this (see footnote 8). These could potentially cause some space management problems such as a split that did not occur during normal processing being required during the restart redo or undo processing (see also Section 5.4). Not writing CLRs also prevents logging of redo information from being done physically (i.e., the operation performed on an object has to be logged—not the after-image created by the operation). Let us consider an example: A piece of data has value 0 after the last checkpoint. Then, transaction T1 adds 1, T2 adds 2, T1 rolls back, and T2 commits. If T1 and T2 had logged the after-image for redo and the operation for undo, then there will be a data integrity problem because after recovery the data will have the value 3 instead of 2. In this case, in System R, undo for T1 is being accomplished by not redoing its update. Of course, System R did not support the fancy lock mode which would be needed to support 2 concurrent updates by different transactions to the same object. Allowing the logging of redo information physically will let redo recovery be performed very efficiently using *dumb* logic. This does not necessarily mean byte-oriented logging; that will depend on whether or not flexible storage management is used (see Section 10.3). Allowing the logging of undo information logically will permit high concurrency to be supported (see [59, 62] for examples). ARIES supports these.

WAL-based systems handle this problem by logging actions performed during rollbacks using CLRs. So, as far as recovery is concerned, the state of the data is always "marching" forward, even if some original actions are being rolled back. Contrast this with the approach, suggested in [52], in which the state of the data, as denoted by the LSN, is "pushed" back during rollbacks. That method works only with page level (or coarser granularity) locking. The immediate consequence of writing CLRs is that, if a transaction were to be rolled back, then some of its original actions are undone more than once and, worse still, the compensating actions are also undone, possibly more than once. This is illustrated in Figure 4, in which a transaction had started rolling back even before the failure of the system. Then, during

recovery, the previously written CLRs are undone and already undone non-CLRs are undone again. ARIES avoids such a situation, while still retaining the idea of writing CLRs. Not undoing CLRs has benefits relating to deadlock management and early release of locks on undone objects also (see item 22, Section 12, and Section 6.4). Additional benefits of CLRs are discussed in the next section and in [69]. Some were already discussed in the Section 8.

Unfortunately, recovery methods like the one suggested in [92] do not support partial rollbacks. We feel that this is an important drawback of such methods.

10.3 Space Management

The goal of this subsection is to point out the problems involved in space management when finer than page level granularity of locking and varying length records are to be supported efficiently.

A problem to be dealt with in doing record locking with flexible storage management is to make sure that the space released by a transaction during record deletion or update on a data page is not consumed by another transaction until the space-releasing transaction is committed. This problem is discussed briefly in [76]. We do not deal with solutions to this space reservation problem here. The interested reader is referred to [50]. For index updates, in the interest of increasing concurrency, we do not want to prevent the space released by one transaction from being consumed by another before the commit of the first transaction. The way undo is dealt with under such circumstances using a logical undo approach is described in [62].

Since flexible storage management was a goal, it was not desirable to do physical (i.e., byte-oriented) locking and logging of data within a page, as some systems do (see [6, 76, 81]). That is, we did not want to use the address of the first byte of a record as the lock name for the record. We did not want to identify the specific bytes that were changed on the page. The logging and locking have to be logical within a page. The record's lock name looks something like (page#, slot#) where the slot# identifies a location on the page which then points to the actual location of the record. The log record describes how the contents of the data record got changed. The consequence is that garbage collection that collects unused space on a page does not have to lock or log the records that are moved around within the page. This gives us the flexibility of being able to move records around within a page to store and modify variable length records efficiently. In systems like IMS, utilities have to be run quite frequently to deal with storage fragmentation. These reduce the availability of data to users.

Figure 19 shows a scenario in which not keeping track of the actual page state (by, e.g., storing the LSN in the nonvolatile storage version of the page) and attempting to perform redo from an earlier point in the log leads to problems when flexible storage management is used. Assuming that all updates in Figure 19 involve the same page and the same transaction, an insert requiring 200 bytes is attempted on a page which has only 100 bytes of free space left in it. This shows the need for exact tracking of page state

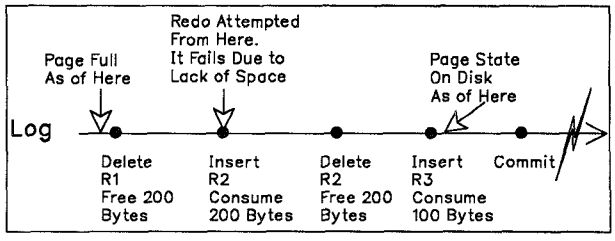


Fig. 19. Wrong redo point-causing problem with space for insert.

using an LSN to avoid attempting to redo operations which are already applied to the page.

Typically, each file containing records of one or more relations has a few pages called free space inventory pages (FSIPs). They are called space map pages (SMPs) in DB2. Each FSIP describes the space information relating to many data or index pages. During a record insert operation, possibly based on information obtained from a clustering index about the location of other records with the same key (or closely related keys) as that of the new record, one or more FSIPs are consulted to identify a data page with enough free space in it for inserting the new record. The FSIP keeps only approximate information (e.g., information such as that at least 25% of the page is full, at least 50% is full, etc.) to make sure that not every space-releasing or -consuming operation to a data page requires an update to the space information in the corresponding FSIP. To avoid special handling of the recovery of the FSIPs during redo and undo, and also to provide recovery independence, updates to the FSIPs must also be logged.

Transaction T1 might cause the space on the page to change from 23% full to 27% full, thereby requiring an update to the FSIP to change it from 0% full to 25% full. Later, T2 might cause the space to go to 35% full, which does not require an update to the FSIP. Now, if T1 were to roll back, then the space would change to 31% full and this should not cause an update to the FSIP. If T1 had written its FSIP change log record as a redo/undo record, then T1's rollback would cause the FSIP entry to say 0% full, which would be wrong, given the *current* state of the data page. This scenario points to the need for logging the changes to the FSIP as *redo-only* changes and for the need to do logical undos with respect to the free space inventory updates. That is, while undoing a data page update, the system has to determine whether that operation causes the free space information to change and if it does cause a change, then update the FSIP and write a CLR which describes the change to the FSIP. We can easily construct an example in which a transaction does not perform an update to the FSIP during forward processing, but needs to perform an update to the FSIP during rollback. We can also construct an example in which the update performed during forward processing is not the exact inverse of the update during the rollback.

10.4 Multiple LSNs

Noticing the problems caused by having one LSN per page when trying to support record locking, it may be tempting to suggest that we track each object's state precisely by assigning a separate LSN to each object. Next we explain why it is not a good idea.

DB2 already supports a granularity of locking that is less than a page. This happens in the case of indexes where the user has the option of requiring DB2 to physically divide up each *leaf* page of the index into 2 to 16 minipages and do locking at the granularity of a minipage [10, 12]. The way DB2 does recovery properly on such pages, despite not redoing actions of loser transactions during the redo pass, is as follows. DB2 tracks each minipage's state separately by associating an LSN with each minipage, besides having an LSN for the leaf page as a whole. Whenever a minipage is updated, the corresponding log record's LSN is stored in the minipage LSN field. The page LSN is set equal to the *maximum* of the minipage LSNs. During undo, it is the minipage LSN and not the page LSN that is compared to the log record's LSN to determine if that log record's update needs to be actually undone on the minipage. This technique, besides incurring too much space overhead for storing the LSNs, tends to fragment (and therefore waste) space available for storing keys. Further, it does not carry over conveniently to the case of record and key locking, especially when varying length objects have to be supported efficiently. Maintaining LSNs for deleted objects is cumbersome at best. We desired to have a *single* state variable (LSN) for each page, even when minipage locking is being done, to make recovery, especially media recovery, very efficient. The simple technique of repeating history during restart recovery before performing the rollback of loser transactions turns out to be sufficient, as we have seen in ARIES. Since DB2 *physically* divides up a page into a fixed number of minipages, no special technique is needed to handle the space reservation problem. Methods like the one proposed in [61] for fine-granularity locking do not support varying length objects (*atoms* in the terminology of that paper).

11. OTHER WAL-BASED METHODS

In the following, we summarize the properties of some other significant recovery methods which also use the WAL protocol. Recovery methods based on the shadow page technique (like that of System R) are not considered here because of their well-known disadvantages, e.g., very costly checkpoints, extra nonvolatile storage space overhead for the shadow copies of data, disturbing the physical clustering of data, and extra I/Os involving page map blocks (see the previous sections of this paper and [31] for additional discussions). First, we briefly introduce the different systems and recovery methods which we will be examining in this section. Next, we compare the different methods along various dimensions. We have been informed that the DB-cache recovery method of [25] has been implemented with significant modifications by Siemens. But, because of lack of information about the implementation, we are unable to include it here.

IBM's IMS/VS [41, 42, 43, 48, 53, 76, 80, 94], which is a hierarchical database system, consists of two parts: IMS Full Function (FF), which is relatively flexible, and IMS Fast Path [28, 42, 93], which is more efficient but has many restrictions (e.g., no support for secondary indexes). A single IMS transaction can access both FF and Fast Path (FP) data. The recovery and buffering methods used by the two parts have many differences. In FF, depending on the database types and the operations, the granularities of the locked objects vary. FP supports two kinds of databases: main storage databases (MSDBs) and data entry databases (DEDBs). MSDBs support only fixed length records, but FP provides the mechanisms (i.e., *field calls*) to make the lock hold times be the minimum possible for MSDB records. Only page locking is supported for DEDBs. But, DEDBs have many high-availability and parallelism features and large database support. IMS, with XRF, provides *hot-standby* support [43]. IMS, via global locking, also supports data sharing across two different systems, each with its own buffer pools [80, 94].

DB2 is IBM's relational database system for the MVS operating system. Limited distributed data access functions are available in DB2. The DB2 recovery algorithm has been presented in [1, 13, 14, 15, 19]. It supports different locking granularities (tablespace, table and page for data, and minipage and page for indexes) and consistency levels (*cursor stability*, *repeatable read*) [10, 11, 12]. DB2 allows logging to be turned off temporarily for tables and indexes only during utility operations like loading and reorganizing data. A single transaction can access both DB2 and IMS data with atomicity. The Encompass recovery algorithm [4, 37] with some changes has been incorporated in Tandem's NonStop SQL [95]. With NonStop, Tandem provides hot-standby support for its products. Both Encompass and NonStop SQL support distributed data access. They allow multisite updates within a single transaction using the Presumed Abort two-phase commit protocol of [63, 64]. NonStop SQL supports different locking granularities (file, key prefix and record) and consistency levels (cursor stability, repeatable read, and unlocked or dirty read). Logging can be turned off temporarily or permanently even for nonutility operations on files.

Schwarz [88] presents two different recovery methods based on value logging (a la IMS) and operation logging. The two methods have several differences, as will be outlined below. The value logging method (VLM), which is much less complex than the operation logging method (OLM), has been implemented in CMU's Camelot [23, 90].

Buffer management. Encompass, NonStop SQL, OLM, VLM and DB2 have adopted the steal and no-force policies. During normal processing, VLM and OLM write a *fetch* record whenever a page is read from nonvolatile storage and an *end-write* record every time a dirty page is successfully written back to nonvolatile storage. These are written during restart processing also in OLM alone. These records help in identifying the super set of dirty pages that might have been in the buffer pool at the time of system failure. DB2 has a sophisticated buffer manager [10, 96], and writes a log

record whenever a tablespace or an indexspace is opened, and another record whenever such a space is closed. The close operation is performed only after all the dirty pages of the space have been written back to nonvolatile storage. DB2's analysis pass uses these log records to bring the dirty objects information up to date as of the failure.

For MSDBs, IMS FP does deferred updating. This means that a transaction does not see its own MSDB updates. For DEDBs, a no-steal policy is used. FP writes, at commit time, all the log records for a given transaction in a single call to the log manager. After placing the log records in the *log buffers* (not on stable storage), the MSDB updates are applied and the MSDB record locks are released. The MSDB locks are released even before the commit log record is placed on stable storage. This is how FP minimizes the amount of time locks are held on the MSDB records. The DEDB locks are transferred to system processes. The log manager is given time to let it force the log records to stable storage ultimately (i.e., group commit logic is used—see [28]). After the logging is completed (i.e., *after* the transaction has been committed), all the pages of the DEDBs that were modified by the transaction are forced to nonvolatile storage using system processes which, on completion of the I/Os, release the DEDB locks. This does not result in any uncommitted updates being forced to nonvolatile storage since page locking with a no-steal policy is used for DEDBs. The use of separate processes for writing the DEDB pages to nonvolatile storage is intended to let the user process go ahead with the next transaction's processing as soon as possible and also to gain parallelism for the I/Os. IMS FF follows the steal and force policies. *Before* committing a transaction, IMS FF forces to nonvolatile storage all the pages that were modified by that transaction. Since finer than page locking is supported by FF, this may result in some uncommitted data being placed on nonvolatile storage. Of course, all the recovery algorithms considered in this section force the log during commit processing.

Normal checkpointing. Normal checkpoints are the ones that are taken when the system is not in the restart recovery mode. OLM and VLM quiesce all activity in the system and take, similar to System R, an operation consistent (not necessarily transaction consistent) checkpoint. The contents of the checkpoint record are similar to those of ARIES. DB2, IMS, NonStop SQL, and Encompass do take (fuzzy) checkpoints even when update and logging activities are going on concurrently. DB2's checkpoint actions are similar to what we described for ARIES. The major difference is that, instead of writing the dirty_pages table, it writes the dirty objects (tablespaces, indexspaces, etc.) list with a RecLSN for each object [96]. For MSDBs alone, IMS writes their complete contents alternately on one of two files on nonvolatile storage during a checkpoint. Since deferred updating is performed for MSDBs, no uncommitted changes will be present in their checkpointed version. Also, it is ensured that no partial committed changes of a transaction are present. Care is needed since the updates are applied after the commit record is written. For DEDBs, any *committed* updated pages which have not yet been written to nonvolatile storage are included in the check-

point records. These together avoid the need for examining, during restart recovery, any log records written before the checkpoint for FP data recovery. Encompass and NonStop SQL might force some dirty pages to nonvolatile storage during a checkpoint. They enforce the policy that requires that a page once dirtied must be written to nonvolatile storage before the completion of the second checkpoint following the dirtying of the page. Because of this policy, the completion of a checkpoint may be delayed waiting for the completion of the writing of the old dirty pages.

Partial rollbacks. Encompass, NonStop SQL, OLM and VLM do not support partial transaction rollback. From Version 2 Release 1, IMS supports partial rollbacks. In fact, the savepoint concept is exposed at the application program level. This support is available only to those applications that do not access FP data. The reason FP data is excluded is because FP does not write undo data in its log records and because deferred updating is performed for MSDBs. DB2 supports partial rollbacks for internal use by the system to provide statement-level atomicity [1].

Compensation log records. Encompass, NonStop SQL, DB2, VLM, OLM and IMS FF write CLRs during normal rollbacks. During a normal rollback, IMS FP does not write CLRs since it would not have written any log records for changes to such data until the decision to rollback is made. This is because FP is always the coordinator in two-phase commit and hence it never needs to get into the prepared state. Since deferred updating is performed for MSDBs, the updates kept in pending (to-do) lists are discarded at rollback time. Since a no-steal policy is followed and page locking is done for DEDBs, the modified pages of DEDBs are simply purged from the buffer pool at rollback time. Encompass, NonStop SQL, DB2 and IMS (FF and FP) write CLRs during restart rollbacks also. During restart recovery, IMS FP might find some log records written by (at the most) one in-progress transaction. This transaction must have been in commit processing—i.e., about to commit, with some of its log records already having been written to nonvolatile storage—when the system went down. Even though, because of the no-steal policy, none of the corresponding FP updates would have been written to nonvolatile storage and hence there would be nothing to be undone, IMS FP writes CLRs for such records to simplify media recovery [93]. Since the FP log records contain only redo information, just to write these CLRs, for which the undo information is needed, the corresponding unmodified data on nonvolatile storage is accessed during restart recovery. This should illustrate to the reader that even with a no-steal policy and without supporting partial rollbacks, there are still some problems to be dealt with at restart for FP. Too often, people assume that no-steal eliminates many problems. Actually, it has many shortcomings.

VLM does not write CLRs during restart rollbacks. As a result, a bounded amount of logging will occur for a rolled back transaction, even in the face of repeated failures during restart. In fact, CLRs are written only for normal rollbacks. Of course, this has some negative implications with respect to media recovery. OLM writes CLRs for undos *and redos* performed during

restart (called *undomodify* and *redomodify* records, respectively). This is done to deal with failures during restart. OLM might write multiple undomodify and redomodify records for a given update record if failures interrupt restart processing. No CLRs are generated for CLRs themselves. During restart recovery, Encompass and DB2 undo changes of CLRs, thus causing the writing of CLRs for CLRs and the writing of multiple, identical CLRs for a given record written during forward or restart processing. In the worst case, the number of log records written during repeated restart failures grows exponentially. Figure 5 shows how ARIES avoids this problem. IMS ignores CLRs during the undo pass and hence does not write CLRs for them. The net result is that, because of multiple failures, like the others, IMS might wind up writing multiple times the same CLR for a given record written during forward processing. In the worst case, the number of log records written by IMS and OLM grows linearly. Because of its force policy, IMS will need to redo the CLR's updates only during media recovery.

Log record contents. IMS FP writes only redo information (i.e., after-image of records) because of its no-steal policy. As mentioned before, IMS does value (or state) logging and physical (i.e., byte-range) locking (see [76]). IMS FF logs both the undo information and the redo information. Since IMS does not undo CLRs' updates, CLRs need to have only the redo information. For providing the XRF hot-standby support, IMS includes enough information in its log records for the backup system to track the lock names of updated objects. IMS FP also logs the address of the buffer occupied by a modified page. This information is used during a backup's takeover or restart recovery to reduce the amount of redo work of DEDBs' updates. Encompass and VLM also log complete undo and redo information of updated records. DB2 and NonStop SQL log only the before- and after-images of the updated fields. OLM logs the description of the update operation. The CLRs of Encompass and DB2 need to contain both the redo and the undo information since their CLRs might be undone. OLM periodically logs an operation consistent *snapshot* of each object. OLM's undomodify and redomodify records contain no redo or undo information but only the LSNs of the corresponding modify records. But OLM's modify, redomodify and undomodify records also contain a page map which specifies the set of pages where parts of the modified object reside.

Page overhead. Encompass and NonStop SQL use one LSN on each page to keep track of the state of the page. VLM uses no LSNs, but OLM uses one LSN. DB2 uses one LSN and IMS FF no LSN. Not having the LSN in IMS FF and VLM to know the exact state of a page does not cause any problems because of IMS' and VLM's value logging and physical locking attributes. It is acceptable to redo an already present update or undo an absent update. IMS FP uses a field in the pages of DEDBs as a version number to correctly handle redos after all the data sharing systems have failed [67]. When DB2 divides an index leaf page into minipages then it uses one LSN for each minipage, besides one LSN for the page as a whole.

Log passes during restart recovery. Encompass and NonStop SQL make two passes (redo and then undo), and DB2 makes three passes (analysis, redo, and then undo—see Figure 6). Encompass and NonStop SQL start their redo passes from the beginning of the penultimate successful checkpoint. This is sufficient because of the buffer management policy of writing to disk a dirty page within two checkpoints after the page became dirty. They also seem to repeat history before performing the undo pass. They do not seem to repeat history if a backup system takes over when a primary system fails [4]. In the case of a takeover by a hot-standby, locks are first reacquired for the losers' updates and then the rollbacks of the losers are performed in parallel with the processing of new transactions. Each loser transaction is rolled back using a separate process to gain parallelism. DB2 starts its redo scan from that point, which is determined using information recorded in the last successful checkpoint, as modified by the analysis pass. As mentioned before, DB2 does selective redo (see Section 10.1).

VLM makes one backward pass and OLM makes three passes (analysis, undo, and then redo). Many lists are maintained during OLM's and VLM's passes. The undomodify and redomodify log records of OLM are used only to modify these lists, unlike in the case of the CLRs written in the other systems. In VLM, the one backward pass is used to undo uncommitted changes on nonvolatile storage and also to redo missing committed changes. No log records are written during these operations. In OLM, during the undo pass, for each object to be recovered, if an operation consistent version of the object does not exist on nonvolatile storage, then it restores a snapshot of the object from the snapshot log record so that, starting from a consistent version of the object, (1) in the remainder of the undo pass any to-be-undone updates that precede the snapshot log record can be undone logically, and (2) in the redo pass any committed or in-doubt updates (modify records only) that follow the snapshot record can be redone logically. This is similar to the shadowing performed in [16, 78] using a separate log—the difference is that the database-wide checkpointing is replaced by object-level checkpointing and the use of a single log instead of two logs.

IMS first reloads MSDBs from the file that received their contents during the latest successful checkpoint before the failure. The dirty DEDB buffers that were included in the checkpoint records are also reloaded into the same buffers as before. This means that, during the restart after a failure, the number of buffers cannot be altered. Then, it makes just one forward pass over the log (see Figure 6). During that pass, it accumulates log records in memory on a per-transaction basis and redoes, if necessary, completed transactions' FP updates. Multiple processes are used in parallel to redo the DEDB updates. As far as FP is concerned, only the updates starting from the last checkpoint before the failure are of interest. At the end of that one pass, in-progress transactions' FF updates are undone (using the log records in memory), in parallel, using one process per transaction. If the space allocated in memory for a transaction's log records is not enough, then a backward scan of the log will be performed to fetch the needed records during that transaction's rollback. In the XRF context, when a hot-standby IMS

takes over, the handling of the loser transactions is similar to the way Tandem does it. That is, rollbacks are performed in parallel with new transaction processing.

Page forces during restart. OLM, VLM and DB2 force all dirty pages at the end of restart. Information on Encompass and NonStop SQL is not available.

Restart checkpoints. IMS, DB2, OLM and VLM take a checkpoint only at the end of restart recovery. Information on Encompass and NonStop SQL is not available.

Restrictions on data. Encompass and NonStop SQL require that every record have a unique key. This unique key is used to guarantee that if an attempt is made to undo a logged action which was never applied to the nonvolatile storage version of the data, then the latter is realized and the undo fails. In other words, idempotence of operations is achieved using the unique key. IMS in effect does byte-range locking and logging and hence does not allow records to be moved around freely within a page. This results in the fragmentation and the less efficient usage of free space. IMS imposes some additional constraints with respect to FP data. VLM requires that an object's representation be divided into fixed length (less than one page sized), unrelocatable quanta. The consequences of these restrictions are similar to those for IMS.

[2, 26, 56] do not discuss recovery from system failures, while the theory of [33] does not include semantically rich modes of locking (i.e., operation logging). In other sections of this paper, we have pointed out the problems with some of the other approaches that have been proposed in the literature.

12. ATTRIBUTES OF ARIES

ARIES makes few assumptions about the data or its model and has several advantages over other recovery methods. While ARIES is simple, it possesses several interesting and useful properties. Each of most of these properties has been demonstrated in one or more existing or proposed systems, as summarized in the last section. However, we know of no single system, proposed or real, which has all of these properties. Some of these properties of ARIES are:

(1) *Support for finer than page-level concurrency control and multiple granularities of locking.* ARIES supports page-level and record-level locking in a uniform fashion. Recovery is not affected by what the granularity of locking is. Depending on the expected contention for the data, the appropriate level of locking can be chosen. It also allows multiple granularities of locking (e.g., record, table, and tablespace-level) for the same object (e.g., tablespace). Concurrency control schemes other than locking (e.g., the schemes of [2]) can also be used.

(2) *Flexible buffer management during restart and normal processing.* As long as the write-ahead logging protocol is followed, the buffer manager is

free to use any page replacement policy. In particular, dirty pages of incomplete transactions can be written to nonvolatile storage before those transactions commit (*steal* policy). Also, it is not required that all pages dirtied by a transaction be written back to nonvolatile storage before the transaction is allowed to commit (i.e., *no-force* policy). These properties lead to reduced demands for buffer storage and fewer I/Os involving frequently updated (*hot-spot*) pages. ARIES does not preclude the possibilities of using deferred-updating and force-at-commit policies and benefiting from them. ARIES is quite flexible in these respects.

(3) *Minimal space overhead—only one LSN per page.* The permanent (excluding log) space overhead of this scheme is limited to the storage required on each page to store the LSN of the last logged action performed on the page. The LSN of a page is a monotonically increasing value.

(4) *No constraints on data to guarantee idempotence of redo or undo of logged actions.* There are no restrictions on the data with respect to unique keys, etc. Records can be of variable length. Data can be moved around within a page for garbage collection. Idempotence of operations is ensured since the LSN on each page is used to determine whether an operation should be redone or not.

(5) *Actions taken during the undo of an update need not necessarily be the exact inverses of the actions taken during the original update.* Since CLRs are being written during undos, any differences between the inverses of the original actions and what actually had to be done during undo can be recorded in the former. An example of when the inverse might not be correct is the one that relates to the free space information (like at least 10% free, 20% free) about data pages that are maintained in space map pages. Because of finer than page-level granularity locking, while no free space information change takes place during the initial update of a page by a transaction, a free space information change might occur during the undo (from 20% free to 10% free) of that original change because of intervening update activities of other transactions (see Section 10.3).

Other benefits of this attribute in the context of hash-based storage methods and index management can be found in [59, 62].

(6) *Support for operation logging and novel lock modes.* The changes made to a page can be logged in a logical fashion. The undo information and the redo information for the entire object need not be logged. It suffices if the changed fields alone are logged. Since history is repeated, for increment or decrement kinds of operations before- and after-images of the field are not needed. Information about the type of operation and the decrement or increment amount is enough. Garbage collection actions and changes to some fields (e.g., amount of free space) of that page need not be logged. Novel lock modes based on commutativity and other properties of operations can be supported [2, 26, 88].

(7) *Even redo-only and undo-only records are accommodated.* While it may be efficient (single call to the log component) sometimes to include the undo and redo information about an update in the same log record, at other

times it may be efficient (from the original data, the undo record can be constructed and, after the update is performed *in-place* in the data record, from the updated data, the redo record can be constructed) and/or necessary (because of log record size restrictions) to log the information in two different records. ARIES can handle both situations. Under these conditions, the undo record must be logged before the redo record.

(8) *Support for partial and total transaction rollback.* Besides allowing transactions to be rolled back totally, ARIES allows the establishment of savepoints and the partial rollback of transactions to such savepoints. Without the support for partial rollbacks, even logically recoverable errors (e.g., unique key violation, out-of-date cached catalog information in a distributed database system) will require total rollbacks and result in wasted work.

(9) *Support for objects spanning multiple pages.* Objects can span multiple pages (e.g., an IMS "record" which consists of multiple segments may be scattered over many pages). When an object is modified, if log records are written for every page affected by that update, ARIES works fine. ARIES itself does not treat multipage objects in any special way.

(10) *Allows files to be acquired or returned, any time, from or to the operating system.* ARIES provides the flexibility of being able to return files dynamically and permanently to the operating system (see [19] for the detailed description of a technique to accomplish this). Such an action is considered to be one that cannot be undone. It does not prevent the same file from being reallocated to the database system. Mappings between objects (tablespaces, etc.) and files are not required to be defined statically as in System R.

(11) *Some actions of a transaction may be committed even if the transaction as a whole is rolled back.* This refers to the technique of using the concept of a dummy CLR to implement nested top actions. File extension has been given as an example situation which could benefit from this. Other applications of this technique, in the context of hash-based storage methods and index management, can be found in [59, 62].

(12) *Efficient checkpoints (including during restart recovery).* By supporting fuzzy checkpointing, ARIES makes taking a checkpoint an efficient operation. Checkpoints can be taken even when update activities and logging are going on concurrently. Permitting checkpoints even during restart processing will help reduce the impact of failures during restart recovery. The dirty_pages information written during checkpointing helps reduce the number of pages which are read from nonvolatile storage during the redo pass.

(13) *Simultaneous processing of multiple transactions in forward processing and/or in rollback accessing same page.* Since many transactions could simultaneously be going forward or rolling back on a given page, the level of concurrent access supported could be quite high. Except for the short duration latching which has to be performed any time a page is being

physically modified or examined, be it during forward processing or during rollback, rolling back transactions do not affect one another in any unusual fashion.

(14) *No locking or deadlocks during transaction rollback.* Since no locking is required during transaction rollback, no deadlocks will involve transactions that are rolling back. Avoiding locking during rollbacks simplifies not only the rollback logic, but also the deadlock detector logic. The deadlock detector need not worry about making the mistake of choosing a rolling back transaction as a victim in the event of a deadlock (cf. System R and R* [31, 49, 64]).

(15) *Bounded logging during restart in spite of repeated failures or of nested rollbacks.* Even if repeated failures occur during restart, the number of CLRs written is unaffected. This is also true if partial rollbacks are nested. The number of log records written will be the same as that written at the time of transaction rollback during normal processing. The latter again is a fixed number and is, usually, equal to the number of undoable records written during the forward processing of the transaction. No log records are written during the redo pass of restart.

(16) *Permits exploitation of parallelism and selective/deferred processing for faster restart.* Restart can be made faster by not doing all the needed I/Os synchronously one at a time while processing the corresponding log record. ARIES permits the early identification of the pages needing recovery and the initiation of asynchronous parallel I/Os for the reading in of those pages. The pages can be processed concurrently as they are brought into memory during the redo pass. Undo parallelism requires complete handling of a given transaction by a single process. Some of the restart processing can be postponed to speed up restart or to accommodate offline devices. If desired, undo of loser transactions can be performed in parallel with new transaction processing.

(17) *Fuzzy image copying (archive dumping) for media recovery.* Media recovery and image copying of the data are supported very efficiently. To take advantage of device geometry, the actual act of copying can even be performed outside the transaction system (i.e., without going through the buffer pool). This can happen even while the latter is accessing and modifying the information being copied. During media recovery only one forward traversal of the log is made.

(18) *Continuation of loser transactions after a system restart.* Since ARIES repeats history and supports the savepoint concept, we could, in the undo pass, instead of totally rolling back the loser transactions, roll back each loser only to its latest savepoint. Locks must be acquired to protect the transaction's uncommitted, not undone updates. Later, we could resume the transaction by invoking its application at a special entry point and passing enough information about the savepoint from which execution is to be resumed.

(19) *Only one backward traversal of log during restart or media recovery.*

Both during media recovery and restart recovery one backward traversal of the log is sufficient. This is especially important if any portion of the log is likely to be stored in a slow medium like tape.

(20) *Need only redo information in compensation log records.* Since compensation records are never undone they need to contain only redo information. So, on the average, the amount of log space consumed during a transaction rollback will be half the space consumed during the forward processing of that transaction.

(21) *Support for distributed transactions.* ARIES accommodates distributed transactions. Whether a given site is a coordinator or a subordinate site does not affect ARIES.

(22) *Early release of locks during transaction rollback and deadlock resolution using partial rollbacks.* Because ARIES never undoes CLRs and because it never undoes a particular non-CLR more than once, during a (partial) rollback, when the transaction's very first update to a particular object is undone and a CLR is written for it, the system can release the lock on that object. This makes it possible to consider resolving deadlocks using partial rollbacks.

It should be noted that ARIES does not prevent the shadow page technique from being used for selected portions of the data to avoid logging of only undo information or both undo and redo information. This may be useful for dealing with long fields, as is the case in the OS/2 Extended Edition Database Manager. In such instances, for such data, the modified pages would have to be forced to nonvolatile storage before commit. Whether or not media recovery and partial rollbacks can be supported will depend on what is logged and for which updates shadowing is done.

13. SUMMARY

In this paper, we presented the ARIES recovery method and showed why some of the recovery paradigms of System R are inappropriate in the WAL context. We dealt with a variety of features that are very important in building and operating an *industrial-strength* transaction processing system. Several issues regarding operation logging, fine-granularity locking, space management, and flexible recovery were discussed. In brief, ARIES accomplishes the goals that we set out with by logging all updates on a per-page basis, using an LSN on every page for tracking page state, repeating history during restart recovery before undoing the loser transactions, and chaining the CLRs to the predecessors of the log records that they compensated. Use of ARIES is not restricted to the database area alone. It can also be used for implementing persistent object-oriented languages, recoverable file systems and transaction-based operating systems. In fact, it is being used in the QuickSilver distributed operating system [40] and in a system designed to aid the backing up of workstation data on a host [44].

In this section, we summarize as to which specific features of ARIES lead to which specific attributes that give us flexibility and efficiency.

Repeating history exactly, which in turn implies using LSNs and writing CLRs during undos, permits the following, irrespective of whether CLRs are chained using the UndoNxtLSN field or not:

(1) Record level locking to be supported and records to be moved around within a page to avoid storage fragmentation without the moved records having to be locked and without the movements having to be logged.
(2) Use only one state variable, a log sequence number, per page.
(3) Reuse of storage released by one transaction for the same transaction's later actions or for other transactions' actions once the former commits, thereby leading to the preservation of clustering of records and the efficient usage of storage.
(4) The inverse of an action origianlly performed during forward processing of a transaction to be different from the action(s) performed during the undo of that original action (e.g., class changes in the space map pages). That is, logical undo with recovery independence is made possible.
(5) Multiple transactions may undo on the same page concurrently with transactions going forward.
(6) Recovery of each page independently of other pages or of log records relating to transaction state, especially during media recovery.
(7) If necessary, the continuation of transactions which were in progress at the time of system failure.
(8) Selective or deferred restart, and undo of losers concurrently with new transaction processing to improve data availability.
(9) Partial rollback of transactions.
(10) Operation logging and logical logging of changes within a page. For example, decrement and increment operations may be logged, rather than the before- and after-images of modified data.

Chaining, using the UndoNxtLSN field, CLRs to log records written during forward processing permits the following, provided the protocol of repeating history is also followed:

(1) The avoidance of undoing CLRs' actions, thus avoiding writing CLRs for CLRs. This also makes it unnecessary to store undo information in CLRs.
(2) The avoidance of the undo of the same log record written during forward processing more than once.
(3) As a transaction is being rolled back, the ability to release the lock on an object when all the updates to that object had been undone. This may be important while rolling back a long transaction or while resolving a deadlock by partially rolling back the victim.
(4) Handling partial rollbacks without any special actions like patching the log, as in System R.
(5) Making permanent, if necessary via nested top actions, some of the

changes made by a transaction, irrespective of whether the transaction itself subsequently rolls back or commits.

Performing the analysis pass before repeating history permits the following:

(1) Checkpoints to be taken any time during the redo and undo passes of recovery.
(2) Files to be returned to the operating system dynamically, thereby allowing dynamic binding between database objects and files.
(3) Recovery of file-related information concurrently with the recovery of user data, without requiring special treatment for the former.
(4) Identifying pages possibly requiring redo, so that asynchronous parallel I/Os could be initiated for them even before the redo pass starts.
(5) Exploiting opportunities to avoid redos on some pages by eliminating those pages from the dirty_pages table on noticing, e.g., that some empty pages have been freed.
(6) Exploiting opportunities to avoid reading some pages during redo, e.g., by writing end_write records after dirty pages have been written to nonvolatile storage and by eliminating those pages from the dirty_pages table when the end_write records are encountered.
(7) Identifying the transactions in the in-doubt and in-progress states so that locks could be reacquired for them during the redo pass to support selective or deferred restart, the continuation of loser transactions after restart, and undo of loser transactions in parallel with new transaction processing.

13.1 Implementations and Extensions

ARIES forms the basis of the recovery algorithms used in the IBM Research prototype systems Starburst [87] and QuickSilver [40], in the University of Wisconsin's EXODUS and Gamma database machine [20], and in the IBM program products OS/2 Extended Edition Database Manager [7] and Workstation Data Save Facility/VM [44]. One feature of ARIES, namely *repeating history*, has been implemented in DB2 Version 2 Release 1 to use the concept of nested top action for supporting segmented tablespaces. A simulation study of the performance of ARIES is reported in [98]. The following conclusions from that study are worth noting: "Simulation results indicate the success of the ARIES recovery method in providing fast recovery from failures, caused by long intercheckpoint intervals, efficient use of page LSNs, log LSNs, and RecLSNs avoids redoing updates unnecessarily, and the actual recovery load is reduced skillfully. Besides, the overhead incurred by the concurrency control and recovery algorithms on transactions is very low, as indicated by the negligibly small difference between the mean transaction response time and the average duration of a transaction if it ran alone in a never failing system. This observation also emerges as evidence that the recovery method goes well with concurrency control through fine-granularity locking, an important virtue."

We have extended ARIES to make it work in the context of the nested transaction model (see [70, 85]). Based on ARIES, we have developed new methods, called ARIES/KVL, ARIES/IM and ARIES/LHS, to efficiently provide high concurrency and recovery for B^+-tree indexes [57, 62] and for hash-based storage structures [59]. We have also extended ARIES to restrict the amount of repeating of history that takes place for the loser transactions [69]. We have designed concurrency control and recovery algorithms, based on ARIES, for the N-way data sharing (i.e., shared disks) environment [65, 66, 67, 68]. *Commit_LSN*, a method which takes advantage of the page_LSN that exists in every page to reduce the locking, latching and predicate reevaluation overheads, and also to improve concurrency, has been presented in [54, 58, 60]. Although messages are an important part of transaction processing, we did not discuss message logging and recovery in this paper.

ACKNOWLEDGMENTS

We have benefited immensely from the work that was performed in the System R project and in the DB2 and IMS product groups. We have learned valuable lessons by looking at the experiences with those systems. Access to the source code and internal documents of those systems was very helpful. The Starburst project gave us the opportunity to begin from scratch and design some of the fundamental algorithms of a transaction system, taking into account experiences with the prior systems. We would like to acknowledge the contributions of the designers of the other systems. We would also like to thank our colleagues in the research and product groups that have adopted our research results. Our thanks also go to Klaus Kuespert, Brian Oki, Erhard Rahm, Andreas Reuter, Pat Selinger, Dennis Shasha, and Irv Traiger for their detailed comments on the paper.

REFERENCES

1. BAKER, J., CRUS, R., AND HADERLE, D. Method for assuring atomicity of multi-row update operations in a database system. U.S. Patent 4,498,145, IBM, Feb. 1985.
2. BADRINATH, B. R., AND RAMAMRITHAM, K. Semantics-based concurrency control: Beyond commutativity. In *Proceedings 3rd IEEE International Conference on Data Engineering* (Feb. 1987).
3. BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass., 1987.
4. BORR, A. Robustness to crash in a distributed database: A non-shared-memory multi-processor approach. In *Proceedings 10th International Conference on Very Large Data Bases* (Singapore, Aug. 1984).
5. CHAMBERLIN, D., GILBERT, A., AND YOST, R. A history of System R and SQL/Data System. In *Proceedings 7th International Conference on Very Large Data Bases* (Cannes, Sept. 1981).
6. CHANG, A., AND MERGEN, M. 801 storage: Architecture and programming. *ACM Trans. Comput. Syst., 6*, 1 (Feb. 1988), 28–50.
7. CHANG, P. Y., AND MYRE, W. W. OS/2 EE database manager: Overview and technical highlights. *IBM Syst. J. 27*, 2 (1988).
8. COPELAND, G., KHOSHAFIAN, S., SMITH, M., AND VALDURIEZ, P. Buffering schemes for permanent data. In *Proceedings International Conference on Data Engineering* (Los Angeles, Feb. 1986).

9. CLARK, B. E., AND CORRIGAN, M. J. Application System/400 performance characteristics. *IBM Syst. J. 28*, 3 (1989).
10. CHENG, J., LOOSELY, C., SHIBAMIYA, A., AND WORTHINGTON, P. IBM Database 2 performance: Design, implementation, and tuning. *IBM Syst. J. 23*, 2 (1984).
11. CRUS, R , HADERLE, D., AND HERRON, H. Method for managing lock escalation in a multiprocessing, multiprogramming environment. U.S. Patent 4,716,52 8, IBM, Dec. 1987.
12. CRUS, R., MALKEMUS, T., AND PUTZOLU, G. R. Index mini-pages *IBM Tech. Disclosure Bull. 26*, 4 (April 1983), 5460-5463.
13. CRUS, R., PUTZOLU, F., AND MORTENSON, J. A Incremental data base log image copy *IBM Tech. Disclosure Bull. 25*, 7B (Dec. 1982), 3730-373 2.
14. CRUS, R., AND PUTZOLU, F. Data base allocation table. *IBM Tech. Disclosure Bull. 25*, 7B (Dec. 1982), 3722-2724.
15. CRUS, R. Data recovery in IBM Database 2. *IBM Syst. J. 23*, 2 (1984).
16. CURTIS, R. Informix-Turbo, In *Proceedings IEEE Compcon Spring '88* (Feb.-March 1988).
17. DASGUPTA, P., LEBLANC, R., JR., AND APPELBE, W. The Clouds distributed operating system. In *Proceedings 8th International Conference on Distributed Computing Systems* (San Jose, Calif., June 1988).
18. DATE, C. *A Guide to INGRES*. Addison-Wesley, Reading, Mass., 1987.
19. DEY, R., SHAN, M., AND TRAIGER, I. Method for dropping data sets. *IBM Tech. Disclosure Bull. 25*, 11A (April 1983), 5453-5455.
20. DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., BRICKER, A., HSIAO, H.-I., AND RASMUSSEN, R. The Gamma database machine project. *IEEE Trans. Knowledge Data Eng. 2*, 1 (March 1990).
21. DELORME, D., HOLM, M., LEE, W., PASSE, P., RICARD, G., TIMMS, G., JR., AND YOUNGREN, L. Database index journaling for enhanced recovery. U.S. Patent 4 ,819,156, IBM, April 1989
22. DIXON, G. N., PARRINGTON, G. D., SHRIVASTAVA, S., AND WHEATER, S. M. The treatment of persistent objects in Arjuna. *Comput. J. 32*, 4 (1989).
23. DUCHAMP, D. Transaction management. Ph.D. dissertation, Tech. Rep. CMU-CS-88-192, Carnegie-Mellon Univ., Dec. 1988.
24. EFFELSBERG, W., AND HAERDER, T. Principles of database buffer management. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984).
25. ELHARDT, K , AND BAYER, R. A database cache for high performance and fast restart in database systems. *ACM Trans Database Syst. 9*, 4 (Dec. 1984).
26. FEKETE, A., LYNCH, N., MERRITT, M., AND WEIHL, W. Commutativity-based locking for nested transactions. Tech. Rep. MIT/LCS/TM-370.b, MIT, July 1989.
27. FOSSUM, B Data base integrity as provided for by a particular data base management system. In *Data Base Management*, J. W. Klimbie and K. L. Koffeman, Eds., North-Holland, Amsterdam, 1974.
28. GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS Fast Path. *IEEE Database Eng. 8*, 2 (June 1985).
29. GARZA, J., AND KIM, W. Transaction management in an object-oriented database system. In *Proceedings ACM-SIGMOD International Conference on Management of Data* (Chicago, June 1988).
30. GHEITH, A., AND SCHWAN, K. CHAOS^{art}: Support for real-time atomic transactions. In *Proceedings 19th International Symposium on Fault-Tolerant Computing* (Chicago, June 1989).
31. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the System R database manager. *ACM Comput. Surv. 13*, 2 (June 1981).
32. GRAY, J. Notes on data base operating systems. In *Operating Systems—An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, Eds., LNCS Vol. 60, Springer-Verlag, New York, 1978.
33. HADZILACOS, V. A theory of reliability in database systems. *J. ACM 35*, 1 (Jan. 1988), 121-145.
34. HAERDER, T. Handling hot spot data in DB-sharing systems. *Inf. Syst. 13*, 2 (1988), 155-166.

35. HADERLE, D., AND JACKSON, R. IBM Database 2 overview. *IBM Syst. J. 23,* 2 (1984).
36. HAERDER, T., AND REUTER, A. Principles of transaction oriented database recovery—A taxonomy. *ACM Comput. Surv. 15,* 4 (Dec. 1983).
37. HELLAND, P. The TMF application programming interface: Program to program communication, transactions, and concurrency in the Tandem NonStop system. Tandem Tech. Rep. TR89.3, Tandem Computers, Feb. 1989.
38. HERLIHY, M., AND WEIHL, W. Hybrid concurrency control for abstract data types. In *Proceedings 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Austin, Tex., March 1988).
39. HERLIHY, M., AND WING, J. M. Avalon: Language support for reliable distributed systems. In *Proceedings 17th International Symposium on Fault-Tolerant Computing* (Pittsburgh, Pa., July 1987).
40. HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. Recovery management in QuickSilver. *ACM Trans. Comput. Syst. 6,* 1 (Feb. 1988), 82–108.
41. *IMS/VS Version 1 Release 3 Recovery/Restart.* Doc. GG24-1652, IBM, April 1984.
42. *IMS/VS Version 2 Application Programming.* Doc. SC26-4178, IBM, March 1986.
43. *IMS/VS Extended Recovery Facility (XRF): Technical Reference.* Doc. GG24-3153, IBM, April 1987.
44. *IBM Workstation Data Save Facility/VM: General Information.* Doc. GH24-5232, IBM, 1990.
45. KORTH, H. Locking primitives in a database system. *JACM 30,* 1 (Jan. 1983), 55–79.
46. LUM, V., DADAM, P., ERBE, R., GUENAUER, J., PISTOR, P., WALCH, G., WERNER, H., AND WOODFILL, J. Design of an integrated DBMS to support advanced applications. In *Proceedings International Conference on Foundations of Data Organization* (Kyoto, May 1985).
47. LEVINE, F., AND MOHAN, C. Method for concurrent record access, insertion, deletion and alteration using an index tree. U.S. Patent 4,914,569, IBM, April 1990.
48. LEWIS, R. Z. *IMS Program Isolation Locking.* Doc. GG66-3193, IBM Dallas Systems Center, Dec. 1990.
49. LINDSAY, B., HAAS, L., MOHAN, C., WILMS, P., AND YOST, R. Computation and communication in R*: A distributed database manager. *ACM Trans. Comput. Syst. 2,* 1 (Feb. 1984). Also in *Proceedings 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, Oct. 1983). Also available as IBM Res. Rep. RJ3740, San Jose, Calif., Jan. 1983.
50. LINDSAY, B., MOHAN, C., AND PIRAHESH, H. Method for reserving space needed for "rollback" actions. *IBM Tech. Disclosure Bull. 29,* 6 (Nov. 1986).
51. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5,* 3 (July 1983).
52. LINDSAY, B., SELINGER, P., GALTIERI, C., GRAY, J., LORIE, R., PUTZOLU, F., TRAIGER, I., AND WADE, B. Notes on distributed databases. IBM Res. Rep. RJ2571, San Jose, Calif., July 1979.
53. MCGEE, W. C. The information management system IMS/VS—Part II: Data base facilities; Part V: Transaction processing facilities. *IBM Syst. J. 16,* 2 (1977).
54. MOHAN, C., HADERLE, D., WANG, Y., AND CHENG, J. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. In *Proceedings International Conference on Extending Data Base Technology* (Venice, March 1990). An expanded version of this paper is available as IBM Res. Rep. RJ7341, IBM Almaden Research Center, March 1990.
55. MOHAN, C., FUSSELL, D., AND SILBERSCHATZ, A. Compatibility and commutativity of lock modes. *Inf. Control 61,* 1 (April 1984). Also available as IBM Res. Rep. RJ3948, San Jose, Calif., July 1983.
56. MOSS, E., GRIFFETH, N., AND GRAHAM, M. Abstraction in recovery management. In *Proceedings ACM SIGMOD International Conference on Management of Data* (Washington, D.C., May 1986).
57. MOHAN, C. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *Proceedings 16th International Conference on Very Large Data Bases* (Brisbane, Aug. 1990). Another version of this paper is available as IBM Res. Rep. RJ7008, IBM Almaden Research Center, Sept. 1989.

58. MOHAN, C. Commit–LSN: A novel and simple method for reducing locking and latching in transaction processing systems In *Proceedings 16th International Conference on Very Large Data Bases* (Brisbane, Aug. 1990). Also available as IBM Res. Rep. RJ7344, IBM Almaden Research Center, Feb. 1990.
59 MOHAN, C. ARIES/LHS: A concurrency control and recovery method using write-ahead logging for linear hashing with separators. IBM Res. Rep., IBM Almaden Research Center, Nov. 1990.
60. MOHAN, C. A cost-effective method for providing improved data availability during DBMS restart recovery after a failure In *Proceedings of the 4th International Workshop on High Performance Transaction Systems* (Asilomar, Calif., Sept. 1991). Also available as IBM Res. Rep. RJ8114, IBM Almaden Research Center, April 1991.
61. MOSS, E., LEBAN, B., AND CHRYSANTHIS, P. Fine grained concurrency for the database cache. In *Proceedings 3rd IEEE International Conference on Data Engineering* (Los Angeles, Feb. 1987).
62. MOHAN, C., AND LEVINE, F. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. IBM Res. Rep. RJ6846, IBM Almaden Research Center, Aug. 1989.
63. MOHAN, C., AND LINDSAY, B. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). Also available as IBM Res. Rep. RJ3881, IBM San Jose Research Laboratory, June 1983.
64. MOHAN, C., LINDSAY, B., AND OBERMARCK, R. Transaction management in the R* distributed database management system. *ACM Trans. Database Syst. 11*, 4 (Dec. 1986).
65. MOHAN, C., AND NARANG, I. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proceedings 17th International Conference on Very Large Data Bases* (Barcelona, Sept. 1991). A longer version is available as IBM Res. Rep. RJ8017, IBM Almaden Research Center, March 1991.
66. MOHAN, C., AND NARANG, I. Efficient locking and caching of data in the multisystem shared disks transaction environment. In *Proceedings of the International Conference on Extending Database Technology* (Vienna, Mar. 1992). Also available as IBM Res. Rep. RJ8301, IBM Almaden Research Center, Aug. 1991.
67. MOHAN, C., NARANG, I., AND PALMER, J. A case study of problems in migrating to distributed computing: Page recovery using multiple logs in the shared disks environment. IBM Res. Rep. RJ7343, IBM Almaden Research Center, March 1990.
68. MOHAN, C., NARANG, I., SILEN, S. Solutions to hot spot problems in a shared disks transaction environment. In *Proceedings of the 4th International Workshop on High Performance Transaction Systems* (Asilomar, Calif., Sept. 1991). Also available as IBM Res Rep. 8281, IBM Almaden Research Center, Aug. 1991.
69. MOHAN, C., AND PIRAHESH, H. ARIES-RRH: Restricted repeating of history in the ARIES transaction recovery method. In *Proceedings 7th International Conference on Data Engineering* (Kobe, April 1991). Also available as IBM Res. Rep. RJ7342, IBM Almaden Research Center, Feb. 1990
70. MOHAN, C , AND ROTHERMEL, K. Recovery protocol for nested transactions using write-ahead logging. *IBM Tech. Disclosure Bull. 31*, 4 (Sept 1988).
71. MOSS, E. Checkpoint and restart in distributed transaction systems. In *Proceedings 3rd Symposium on Reliability in Distributed Software and Database Systems* (Clearwater Beach, Oct. 1983).
72. MOSS, E Log-based recovery for nested transactions. In *Proceedings 13th International Conference on Very Large Data Bases* (Brighton, Sept. 1987).
73. MOHAN, C., TRIEBER, K., AND OBERMARCK, R. Algorithms for the management of remote backup databases for disaster recovery. IBM Res. Rep. RJ7885, IBM Almaden Research Center, Nov. 1990.
74. NETT, E., KAISER, J., AND KROGER, R. Providing recoverability in a transaction oriented distributed operating system. In *Proceedings 6th International Conference on Distributed Computing Systems* (Cambridge, May 1986).

75. NOE, J., KAISER, J., KROGER, R., AND NETT, E. The commit/abort problem in type-specific locking. GMD Tech. Rep. 267, GMD mbH, Sankt Augustin, Sept. 1987.
76. OBERMARCK, R. IMS/VS program isolation feature. IBM Res. Rep. RJ2879, San Jose, Calif., July 1980.
77. O'NEILL, P. The Escrow transaction method. *ACM Trans. Database Syst. 11*, 4 (Dec. 1986).
78. ONG, K. SYNAPSE approach to database recovery. In *Proceedings 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Waterloo, April 1984).
79. PEINL, P., REUTER, A., AND SAMMER, H. High contention in a stock trading database: A case study. In *Proceedings ACM SIGMOD International Conference on Management of Data* (Chicago, June 1988).
80. PETERSON, R. J., AND STRICKLAND, J. P. Log write-ahead protocols and IMS/VS logging. In *Proceedings 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Atlanta, Ga., March 1983).
81. RENGARAJAN, T. K., SPIRO, P., AND WRIGHT, W. High availability mechanisms of VAX DBMS software. *Digital Tech. J. 8* (Feb. 1989).
82. REUTER, A. A fast transaction-oriented logging scheme for UNDO recovery. *IEEE Trans. Softw. Eng. SE-6*, 4 (July 1980).
83. REUTER, A. Concurrency on high-traffic data elements. In *Proceedings ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Los Angeles, March 1982).
84. REUTER, A. Performance analysis of recovery techniques. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984), 526–559.
85. ROTHERMEL, K., AND MOHAN, C. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In *Proceedings 15th International Conference on Very Large Data Bases* (Amsterdam, Aug. 1989). A longer version of this paper is available as IBM Res. Rep. RJ6650, IBM Almaden Research Center, Jan. 1989.
86. ROWE, L., AND STONEBRAKER, M. The commercial INGRES epilogue. Ch. 3 in *The INGRES Papers*, Stonebraker, M., Ed., Addson-Wesley, Reading, Mass., 1986.
87. SCHWARZ, P., CHANG, W., FREYTAG, J., LOHMAN, G., MCPHERSON, J., MOHAN, C., AND PIRAHESH, H. Extensibility in the Starburst database system. In *Proceedings Workshop on Object-Oriented Data Base Systems* (Asilomar, Sept. 1986). Also available as IBM Res. Rep. RJ5311, San Jose, Calif., Sept. 1986.
88. SCHWARZ, P. Transactions on typed objects. Ph.D. dissertation, Tech. Rep. CMU-CS-84-166, Carnegie Mellon Univ., Dec. 1984.
89. SHASHA, D., AND GOODMAN, N. Concurrent search structure algorithms. *ACM Trans. Database Syst. 13*, 1 (March 1988).
90. SPECTOR, A., PAUSCH, R., AND BRUELL, G. Camelot: A flexible, distributed transaction processing system. In *Proceedings IEEE Compcon Spring '88* (San Francisco, Calif., March 1988).
91. SPRATT, L. The transaction resolution journal: Extending the before journal. *ACM Oper. Syst. Rev. 19*, 3 (July 1985).
92. STONEBRAKER, M. The design of the POSTGRES storage system. In *Proceedings 13th International Conference on Very Large Data Bases* (Brighton, Sept. 1987).
93. STILLWELL, J. W., AND RADER, P. M. *IMS/VS Version 1 Release 3 Fast Path Notebook*. Doc. G320-0149-0, IBM, Sept. 1984.
94. STRICKLAND, J., UHROWCZIK, P., AND WATTS, V. IMS/VS: An evolving system. *IBM Syst. J. 21*, 4 (1982).
95. THE TANDEM DATABASE GROUP. NonStop SQL: A distributed, high-performance, high-availability implementation of SQL. In *Lecture Notes in Computer Science Vol. 359*, D. Gawlick, M. Haynie, and A. Reuter, Eds., Springer-Verlag, New York, 1989.
96. TENG, J., AND GUMAER, R. Managing IBM Database 2 buffers to maximize performance. *IBM Syst. J. 23*, 2 (1984).
97. TRAIGER, I. Virtual memory management for database systems. *ACM Oper. Syst. Rev. 16*, 4 (Oct. 1982), 26–48.
98. VURAL, S. A simulation study for the performance analysis of the ARIES transaction recovery method. M.Sc. thesis, Middle East Technical Univ., Ankara, Feb. 1990.

99 WATSON, C. T., AND ABERLE, G. F System/38 machine database support. In *IBM Syst. 38/Tech. Dev.*, Doc. G580-0237, IBM July 1980.
100. WEIKUM, G. Principles and realization strategies of multi-level transaction management. *ACM Trans. Database Syst. 16*, 1 (Mar. 1991).
101. WEINSTEIN, M., PAGE, T., JR , LIVEZEY, B., AND POPEK, G. Transactions and synchronization in a distributed operating system. In *Proceedings 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Dec. 1985).

Received January 1989; revised November 1990; accepted April 1991