ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions

K. Rothermel, IBM European Networking Center, Tiergartenstrasse 15, D-6900 Heidelberg, W. Germany
rotherme@dhdibm1.bitnet

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

Abstract A simple and efficient recovery method for nested transactions, called ARIES/NT (Algorithm for Recovery and Isolation Exploiting Semantics for Nested Transactions), that uses write-ahead logging and supports semantically-rich modes of locking and operation logging is presented. This method applies to a very general model of nested transactions, which includes partial rollbacks of subtransactions, upward and downward inheritance of locks, and concurrent execution of ancestor and descendent subtransactions. The adopted system architecture encompasses aspects of distributed data base management also. ARIES/NT is an extension of the ARIES recovery and concurrency control method developed recently for the single-level transaction model by Mohan, et al. in the IBM Research Report RJ6649.

1. Introduction

The nested transaction concept was popularized by Moss [Moss81]. It has been implemented in several systems so far, such as ARGUS [LCJS87], Camelot [SpPB88], CLOUDS [DaLA88], LOCUS [MuMP83], and Eden [JNJBP82]. Nested transactions have at least three advantages over single-level transactions: First, they provide a means for controlling concurrency within transactions. Second, nested transactions can be used to protect a part of a transaction from failures of another part of the transaction - i.e., nested transactions can act as firewalls, preventing outside influences from affecting the internals. Third, nested transactions allow an easy and secure composition of transaction programs, by means of which the modularity of systems can be increased.

The goal of our work has been to devise an efficient and simple recovery method which guarantees the usual transaction atomicity properties for a very general model of nested transactions. Even though a number of papers have discussed the shadow-page recovery technique (as used in System R [GMBLL81]), write-ahead logging (WAL) is the method of choice in most commercial systems because of its efficiency, flexibility, and power (see [MHLPS89] for detailed comparisons). In WAL systems, an updated page is written back to the same nonvolatile storage (e.g., disk) location from where it was read. That is, **in-place updating** is done on nonvolatile storage. The ***WAL protocol*** asserts that the log records representing changes to some data must be physically written to stable storage **before** the changed data is updated on nonvolatile storage.

In this paper, a recovery method that uses WAL for nested transactions, called ARIES/NT (Algorithm for Recovery and Isolation Exploiting Semantics for Nested Transactions), is presented. ARIES/NT is an extension of the ARIES recovery and concurrency control method introduced by Mohan, et al. in [MHLPS89]. It retains all the features of the latter. ARIES tracks precisely the state of a page on nonvolatile storage as well as in main storage by associating a log sequence number (LSN) with every page. The LSN of a data base page is the address of the log record which describes the update most recently performed on the page. It is a monotonically increasing value that helps us relate the state of a page to logged updates of the page.

Like ARIES, ARIES/NT also logs all database changes, including even those performed during rollbacks of transactions. Updates performed during rollbacks are logged using what are called compensation log records (CLRs). During restart recovery, ARIES/NT guarantees that, for those transactions which were already aborting at the time of the system failure, only those actions that had not been undone ever before are rolled back. This means that the changes specified in a CLR are never undone (the changes are described in CLRs only for *redo* purposes during restart after a failure or during media recovery). This also causes a CLR to be used to keep track of how much of the transaction has already been rolled back and how much more remains to be rolled back. This tracking is done by recording, in a CLR, a set of pointers, each of which points to the next record to be dealt with in

Proceedings of the Fifteenth International Conference on Very Large Data Bases

Amsterdam, 1989

the chain of log records written by the transaction or one of its subtransactions (in ARIES, instead of a set of pointers, there will be only one pointer, called UndoNxtLSN, since ARIES was designed for the single-level transaction model).

The remainder of this paper is organized as follows. Section 2 introduces the notion of a nested transaction and describes the system architecture we assume. In section 3, the new recovery method ARIES/NT is presented. Section 4 describes other work on recovery algorithms for nested transactions. Finally, section 5 summarizes our work.

2. Transaction and System Models

In this section, we discuss the nested transaction model and the system architecture that we assume for ARIES/NT.

2.1. Nested Transactions

The transaction model that ARIES/NT is designed for is a generalization of the nested transaction model of [Moss81]. In this model, a transaction may contain any number of ***subtransactions***, and each subtransaction again may comprise any number of subtransactions.

A transaction which is not enclosed in another transaction is called a ***top-level transaction (TL-transaction)***. In the following, we use the term "transaction" to denote both TL-transactions and subtransactions. Transactions having subtransactions are called ***parents***, and their subtransactions are their ***children***. The ***ancestor*** (respectively, ***descendant***) relation is the reflexive transitive closure of the parent (child) relation. We use the term ***superior*** (respectively, ***inferior***) for the nonreflexive version of ancestor (descendant). The nesting hierarchy of a TL-transaction can be represented by a so-called ***transaction tree***, where the nodes of the tree represent transactions, and the edges represent the nesting relationship amongst the transactions.

Transactions can terminate either normally by committing or abnormally by aborting. Subtransactions appear atomic to the surrounding transactions and may commit or abort independently. A transaction is not allowed to commit until all its children have terminated. A subtransaction may abort without affecting the commit or abort outcome of the surrounding transaction. However, the commit of a subtransaction is relative; even if the subtransaction commits, aborting one of the subtransaction's superiors will undo its effects. Updates become permanent only when the enclosing TL-transaction commits.

The concurrency control scheme for nested transaction presented in [Moss81] is a locking approach. In this scheme, transactions may ***hold*** and ***retain*** locks. When a subtransaction commits, its locks are ***inherited*** by its parent, which then *retains* the locks. A transaction *holding* a lock for an object is allowed to access this object. The access is not allowed if it only *retains* the lock. Consequently, a retained lock is only a place holder indicating that transactions outside the hierarchy of the retainer cannot acquire the lock, but descendants of the retainer can.

The locking scheme introduced by Moss only allows for ***upward inheritance*** of locks, i.e., a parent may inherit locks of its children, but not vice versa. In [HaRo87b], a more general locking scheme that additionally supports ***downward inheritance*** of locks is proposed. ARIES/NT is flexible enough to be used together with concurrency control schemes enabling downward as well as upward lock inheritance [RoMo89]. Moreover, it accommodates the support of even semantically-rich lock modes like increment/decrement which permit multiple transactions to update the same data concurrently. This is the kind of feature that requires a recovery method to support operation logging (before and/or after image logging will not work) and to avoid the erroneous attempts to undo or redo some actions unnecessarily by precisely tracking the state of a page using the LSN and by writing CLRs.

Even concurrency control methods which are similar to locking, like the one described in [BaRa87], can be used with ARIES and ARIES/NT (see [MHLPS89]).

2.2. System Architecture

A distributed database system (DDBS) consists of a set of sites, which are interconnected via a communication network. Each site is comprised of a set of ***subsystems***, which cooperate with each other for the purpose of processing transactions. We assume that each subsystem has its own recovery component.

On each site of the DDBS, there exists a special subsystem called the ***bookkeeper***. Bookkeepers coordinate the initiation, migration and termination of transactions. Moreover, they maintain transaction state information, which is needed for recovery purposes, as well as information needed to build up transaction trees and to keep track of which transactions have visited which sites and which subsystems. The notations ***BK-subsystem*** and ***NBK-subsystem*** are used to denote a bookkeeper subsystem and a non-bookkeeper subsystem, respectively. An NBK-subsystem may be some component of a DDBS site, such as an index manager or a record manager.

A transaction may be distributed over several NBK-subsystems, possibly residing on different sites of the DDBS. A transaction is initiated at a subsystem either when the subsystem receives the first request to be performed within this transaction, or as soon as a

child of this transaction commits at this subsystem, whichever happens first. From an NBK-subsystem's point of view, a subtransaction may be in two different *states*, namely "unknown" and "active". At a subsystem, the initial state of a subtransaction is "unknown". The subtransaction enters the "active" state as soon as it gets initiated at the subsystem. It again becomes "unknown" when it commits or when one of its ancestors aborts.

TL-transactions alone may also reside in the "prepared" state. This state can be reached during the execution of a 2-Phase-Commit (2PC) protocol (e.g., the Presumed Abort protocol of [MoLO86]), which is used to terminate distributed TL-transactions. During the first phase of 2PC, the transaction enters the "prepared" state at a subsystem if updates were performed at that subsystem and that subsystem is willing to commit the transaction. After its termination (commit or abort) in the second phase, the transaction returns to the "unknown" state. Of course, it also becomes "unknown" when it is aborted in the "active" state.

Each BK in the DDBS guarantees the following:

- For each "active" subtransaction, an NBK-subsystem will eventually receive either a **commit** (for this subtransaction) or an **abort** (for an ancestor of this subtransaction) request from the local BK. A **commit** (respectively, **abort**) request contains the identifier of the subtransaction to be committed (aborted) as well as the identifier of the parent (inferiors, which are "active" at this subsystem) of this subtransaction. A commit request will not be issued until all the children of the committing subtransaction have terminated.
- For each "active" TL-transaction, an NBK-subsystem will eventually receive an **abort** or a **prepare** request from its local BK. While a **prepare** request contains the identifier of the TL-transaction to be prepared, an **abort** request includes the identifiers of the TL-transaction's "active" descendants at this subsystem. A prepare request will not be issued until all the children of the preparing TL-transaction have terminated.
- For each "prepared" TL-transaction, an NBK-subsystem will eventually receive a **commit** or an **abort** request from its local BK.
- For each "active" transaction, an NBK-subsystem can issue a query to the local BK asking for the transaction's inferiors residing in the "active" state at this subsystem. An NBK-subsystem needs this information when it decides to unilaterally abort a transaction.

3. ARIES/NT

In this section, first we give a brief overview of ARIES/NT, and then introduce the important data structures. Subsequently, we present the algorithm assuming normal operation, and finally describe how it operates during restart processing. Note that here we are primarily concentrating only on the ARIES/NT extensions, for the nested transaction model, to ARIES. [MHLPS89] should be consulted for detailed information and the rationale for the different basic design decisions.

3.1. Overview

In this section, we will first present the basic principles of ARIES (see also the section "1. Introduction") and then briefly describe the extensions that lead to ARIES/NT.

During restart after a failure, ARIES first scans the log, starting from the first record of the last complete checkpoint, up to the end of the log. During this **analysis pass**, information is gathered about (1) pages that were potentially more up to date in the buffers than in the nonvolatile storage version of the data base and (2) transactions that were in progress at the time of the crash. Then, ARIES ***repeats history***, with respect to those updates that were logged to stable storage but whose effects on data base pages did not get written to nonvolatile storage before the crash. This is done for ALL transactions, including for those transactions that were in progress at the time of the crash. This essentially reestablishes the state of the data base as of the time of the crash. No logging is done of the updates redone during this **redo pass**.

The next pass is the **undo pass** during which all in-progress transaction's updates are rolled back in reverse chronological order, in a single sweep of the log. Note that for those transactions which were already rolling back at the time of the crash, ARIES will only rollback 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 already rolling back transaction points to the next nonCLR record, if any, that is to be undone.

Now, we will briefly describe the extensions to the above which lead to ARIES/NT. In both ARIES and ARIES/NT, all log records written by the same transaction are linked via a so-called **backward chain (BW-chain)**. In addition, in ARIES/NT, the BW-chains of *committed* subtransactions are linked to the BW-chains of their parents to reflect the transaction trees on the log. When a subtransaction T commits, a 'child committed record', which contains a pointer to the last record of T's chain, is written to the BW-chain of T's parent. Consequently, the BW-chain of an in-progress transaction together with the chains of its committed inferiors form a tree structure, which is called the transaction's **backward chain tree (BWC-tree)**. Since the parent/child relationships of committed subtransactions are stored on the log, subsystems can

forget subtransactions after commit and the analysis pass need not collect information about committed subtransactions, which simplifies recovery.

When a transaction is aborted, the actions of that transaction and its (committed or active) inferiors are rolled back in reverse chronological order. Like ARIES, ARIES/NT logs database updates performed during rollback by means of CLRs. A CLR is also used to keep track how much of a transaction and its committed inferiors has already been rolled back and how much more remains to be undone. This is achieved by recording, in a CLR, a set of pointers, each of which points to the next log record to be processed in the BW-chain of the transaction or a committed inferior during undo.

As in ARIES, in ARIES/NT also, restart processing starts with an analysis pass, continues with a redo pass and ends with an undo pass. Redo processing of ARIES/NT works in exactly the same way as in ARIES, while the algorithms of the analysis and undo pass have been modified to support tree-structured log contents.

3.2. Data Structures

In this section, we describe some of the important data structures used by ARIES/NT at an NBK-subsystem.

Each record in the log of an NBK-subsystem belongs to a so-called ***backward chain (BW-chain)***. A BW-chain is associated with a transaction and connects the log records which are relevant for undoing and redoing this transaction at a particular subsystem.

The fields of a ***log record*** which are of interest in the subsequent discussions are:

- **PrevLSN:** Address of the preceding log record in the transaction's BW-chain. This value is NIL if this is the first log record.
- **LastLSN:** Present only in log records of the type C-Committed. When a subtransaction commits, a C-Committed record is added to the BW-chain of its parent. LastLSN contains the address of the last log record in the BW-chain of the committed subtransaction.
- **ChildId:** Present only in log records of the type *C-Committed*. It is the identifier of the child whose commit caused this log record to be written.
- **UndoNextSet:** Present only in log records of the type CLR. When written by a transaction T, it contains the address of the next log record of T that is to be processed during undo. Moreover, it also includes for each committed inferior of T which is only partially undone, the address of the next log record that is to be processed during undo in the inferior's BW-chain (for details see below).
- **PageId:** Present only in records of type Update (nonCLR) or CLR. The identifier of the data base page to which the updates of this record were applied.

In each NBK-subsystem, there exists a transaction table called ***TransTab***. For each transaction known by the subsystem, this table contains an entry consisting of **TransId**, **State**, and **LastLSN** (the address of the most recently written log record in the transaction's BW-chain).

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 table ***DirtyPages*** is used to represent the information about dirty pages. During normal processing, when a nondirty page is being fixed in the buffers for an update operation, the buffer manager inserts into the DirtyPages table a new entry containing the identifier of the page (**PageId**) and the current end-of-log address (**RecLSN** - recovery LSN), which is the address of the next log record to be written. Whenever a page is written back to disk, the corresponding entry is deleted from DirtyPages.

3.3. Normal Processing

The following two subsections describe the algorithms for database updates, transaction prepare and commit. The third subsection presents the algorithm for rollback, and finally, the fourth subsection describes how checkpoints are taken during normal processing.

3.3.1. Update

When an NBK-subsystem receives the first **work** request of a transaction, the recovery manager (RM) of this subsystem checks whether the transaction is already "active". The transaction would already be active, if a child of this transaction had executed at the subsystem and had committed before. If the transaction is still "unknown", the RM inserts an entry for the transaction in the RM's TransTab.

Whenever the execution of a work request causes a transaction to perform an update to an object in a page, this page is fixed in the buffer and latched in the exclusive (X) mode, the update is applied, an Update log record is added to the transaction's BW-chain, the LSN of the log record is placed in the page's LSN field, and the page is unlatched and unfixed (see [MHLPS89] for further explanations). The buffer manager uses the page LSN to enforce the WAL protocol. The page LSN is also used during recovery to determine the exact state of a page without having to examine any user data in the page. Before a transaction may update an object in a page, it must hold a lock for the object.

To support flexible storage management and to reduce the volume of log data, the changes to a page can be

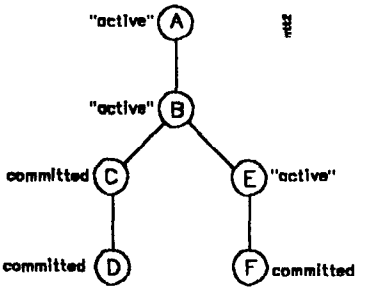

Figure 1: Transaction Tree with Transaction States

logged logically (logging operations, rather than before and after-images of modified data). This permits the semantics of the operations on the data to be exploited to permit additional concurrency. Moreover, since operations are logged, undo processing can affect a page totally different from the one affected during forward processing (see [MHLPS89, MoLe89]). Because the updates performed during rollbacks are logged, this permits us to support *page-oriented* redo processing and media recovery.

3.3.2. Prepare and Commit

Subtransactions: When an NBK-subsystem receives a **commit** request for a subtransaction T from its local BK, the RM of this subsystem checks whether there already exists an entry for T's parent in the local TransTab. If the entry does not exist, it changes the state of T's parent from "unknown" to "active" by inserting the corresponding entry in TransTab. Subsequently, it *asynchronously* writes a ***C-Committed*** (child committed) log record, which represents the first record in the BW-chain of T's parent. On the other hand, if T's parent is already "active", it only adds a C-Committed record to the BW-chain of T's parent. The C-Committed record contains, besides other information, the identifier of T (ChildId) as well as the log address of the last record in T's BW-Chain (LastLSN).

A BW-chain that represents the root of a BWC-tree is called a ***root chain.*** An ancestor which represents a root chain is denoted as a ***root ancestor.*** The BW-chains of TL-transactions as well as the BW-chains of aborted or "active" subtransactions always represent the root of a BWC-tree, whereas the BW-chains of committed subtransactions are always nonroot chains. Consequently, for each TL-transaction T, there might exist a forest of BWC-trees, which consists of T's BWC-tree and the BWC-trees of the "active" and aborted inferiors of T.

Figure 1 shows a transaction tree consisting of transactions A, B, C, D, E and F. Transactions A, B and E are in the "active" state, while the others are already committed. Figure 2 depicts a portion of the log of an NBK-subsystem S, which contains a BW chain for each transaction in this transaction tree. At time t1, there exists a forest comprised of three BWC trees, which are associated with transactions A, B and E. To the BWC-tree of B, for instance, belong the BW-chains of B, C and D, where B's BW-chain is the root chain of this BWC-tree.

TL-Transactions: When an NBK-subsystem receives a **prepare** request for a TL-transaction during the first phase of 2PC, the RM of this subsystem adds a Prepare log record to the transaction's BW-chain and *synchronously* writes the record (and all log records preceding this log record) to the log on stable storage. The log record contains, besides other information, the identifier of the TL-transaction and a list of the update type locks (X, IX, etc.) held or retained by this transaction. The nonupdate type locks (S, IS, etc.) can be released at this time.

When an NBK-subsystem receives a **commit** request for a TL-transaction, it appends an End record to the transaction's BW-chain. Whether or not this record is written synchronously to stable storage depends on the kind of 2PC protocol used (for a detailed discussion, see [MoLO86]). After writing the log record and releasing the locks held or retained by the transaction, the subsystem deletes the transaction's entry in TransTab.

3.3.3. Rollback

When an NBK-subsystem receives an **abort** request for a transaction or it decides to unilaterally abort the transaction, it has to undo the effects of the transaction's "active" or "prepared" descendants (called ***known-descendants***). We define the ***BWC-forest*** of a transaction to consist of the BWC-trees of the known-descendants of this transaction. In order to rollback a transaction, the log records belonging to the transaction's BWC-forest have to be undone and compensated for.

When rollback for a transaction starts, the rollback process only knows the root chains of the transaction's BWC-forest; however, as rollback proceeds, it learns about all the nonroot chains of this forest. In order to keep track of which BW-chains of a transaction are currently known, for each of the transaction's known-descendants, a list called ***KnownChains*** is maintained by the rollback process. The KnownChains list of a known-descendant contains an UndoNext pointer for each BW-chain (of the known-descendant's BWC-tree) which has so far been encountered by the rollback process. The UndoNext pointer of a BW-chain points to the next log record in the chain to be read and processed.

When rollback of a transaction starts, the KnownChains list of a known-descendant, say X, of this transaction contains one UndoNext pointer, which

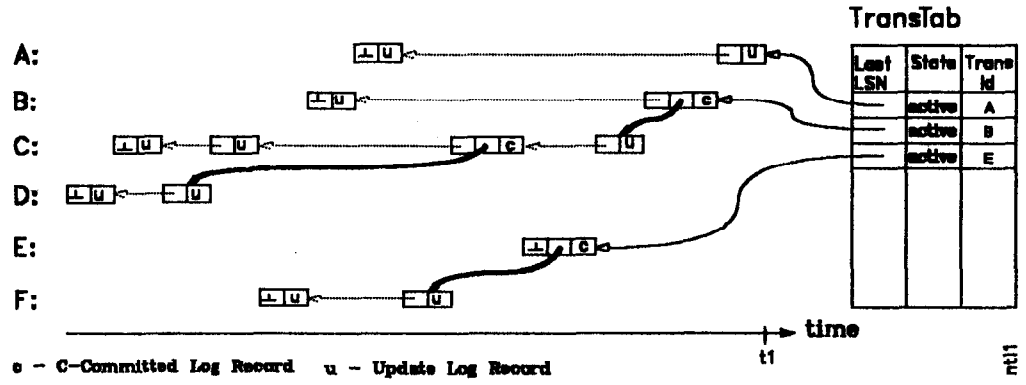

Figure 2: Log and TransTab at a Particular NBK-Subsystem

points to the last record of X's BW-chain. Whenever a C-Committed record belonging to X's BWC-tree is encountered in the log during rollback, a new UndoNext pointer is added to X's KnownChains list. Conversely, whenever the first log record of a BW-chain belonging to X's BWC-tree is read, the corresponding UndoNext pointer is removed from X's KnownChains list.

The CLRs compensating the log records belonging to a given BWC-tree are added to the root chain of this BWC-tree - i.e., CLRs can only occur in root chains (assuming there are no partial rollbacks - see [RoMo89]). If a subtransaction commits and later on it is aborted due to the rollback of one of its superiors, then that subtransaction's CLRs appear in the chain of log records of that superior.

Each CLR includes a set of log addresses: for each chain C' which is a descendant of root chain C and which is known by the rollback process, the UndoNextSet field of a CLR added to C contains the address of the next log record in C' to be processed. It is the information in the UndoNextSet that helps us avoid undoing the same nonCLR record more than once and also avoids having to undo CLRs.

Figure 3 shows the log of the NBK-subsystem S during the rollback of subtransaction B, which is a node of the transaction tree depicted in Figure 1. It is assumed that the rollback of B starts at time t1. The BWC-forest of B consists of two BWC-trees, the one of B and the one of E. In order to rollback B, the log records belonging to both these BWC-trees have to be undone and compensated for in reverse chronological order. The CLRs compensating log records belonging to the BWC-tree of B (E, respectively) are added to the BW-chain of B (E), which is the root chain of the BWC-tree of B (E). The UndoNextSet field of a CLR contains a set of log addresses. For example, CLR x6, which compensates Update log record u6, contains the LSNs of u3 and u4. Update record u3 (u4, respectively) is the next log record to be undone and compensated for in the BW-chain of D (C). At time t2, there exist two KnownChains lists, one for E and one for B. The KnownChains list for B contains two UndoNext pointers, which point to u4 and u3, whereas E's KnownChains list includes one pointer, pointing to u5. Of course, the next log record to be undone and compensated for is u5.

In order to describe how the rollback process acts on the different types of log records, we assume that it selects UndoNext pointer P residing in the KnownChains list of known-descendant X. Further, we assume that P points to log record R in BW-chain C. Depending on R's type, rollback acts as follows:

- ***Update:*** If R.PrevLSN equals NIL then R is the first log record in C - i.e., except R all undoable log records belonging to C have already been undone and compensated for. In this case, the rollback process can forget C by removing P from X's KnownChains list. If R.PrevLSN is not NIL, then P is updated to the value of R.PrevLSN. That is, after this update P points to the next log record of C to be read from the log.

  After updating X's KnownChains list, R is undone and compensated for if it is not a redo-only log record. The corresponding CLR is written to X's BW-chain, which represents C's root ancestor. The UndoNextSet field of this CLR contains the UndoNext pointers residing in X's KnownChains list. After writing the CLR, the page LSN and X's LastLSN in TransTab are updated to be the CLR's LSN.

  If X's KnownChains list is empty after removing P, then R is the last log record belonging to the BWC-tree of X that is yet to be processed. If this is the case, after processing R an End record is written to X's BW-chain, and then X is deleted from TransTab

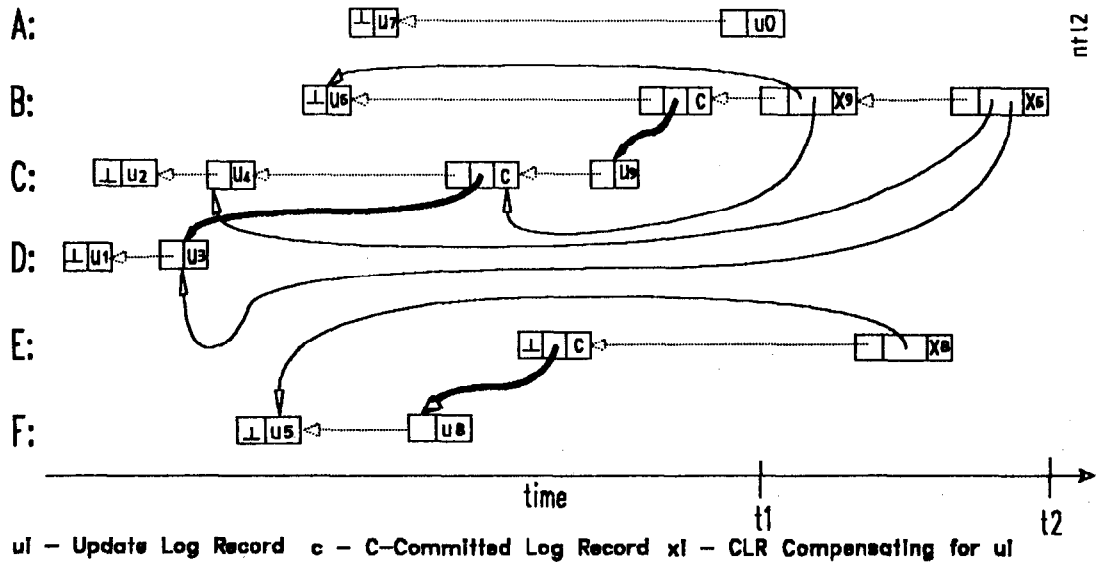

Figure 3: Logging Scenario

- *C-Committed:* The KnownChains list of X is updated by modifying P as described above. Moreover, a new UndoNext pointer P′ is added to X's KnownChains list, and is set to the value of R.LastLSN. Consequently, after this update, P′ points to the last record of a BW-chain, which is a child of C in X's BWC-tree.
- *Prepare:* P is changed to the value of R.PrevLSN - i.e., after this update P points to the next record in C to be processed.

The rollback algorithm described above only allows for *total* rollback of transactions. However, it has been extended to support *partial* rollbacks also (see [RoMo89]).

3.3.4. Checkpoints

For taking checkpoints periodically, the same *fuzzy checkpointing* mechanism as described in [MHLPS89] can be used without any changes. Taking a checkpoint is initiated by writing a ***BeginChpt*** log record. The checkpoint operation is completed by *synchronously* writing to stable storage an ***EndChpt*** record, which contains a copy of TransTab, and DirtyPages. The LSNs of the BeginChpt and EndChpt log records are stored in the ***Master*** record, which is in a well-known place on stable storage. Note that no dirty pages need to be forced to nonvolatile storage during the checkpoint operation. The assumption is that the buffer manager is periodically writing dirty pages to nonvolatile storage, as background activity, to keep the amount of redo work to be done at restart to a reasonable level.

3.4. Restart Processing

When a system failure occurs, the secondary storage version of the database may be left in an inconsistent state. It may contain uncommitted updates. Moreover, it may not contain some or all the updates of committed (or aborted) transactions. In order to reestablish a consistent database state, ***restart recovery*** has to be performed. Restart recovery is performed with three passes of the log, as outlined in the section "3.1. Overview". In the following, the three log passes are described in detail.

Figure 4 summarizes the recovery processing done by different systems. Note that, during restart, DB2, and System R do not redo any updates of the loser ("active" state) transactions. This is called ***selective redo***. In System R alone, the undo pass precedes the redo pass. There are a number of reasons for this and the correctness of this method relies on the shadow-page approach to recovery [GMBLL81]. The reader is referred to [MHLPS89] for detailed discussions concerning the problems caused by the recovery strategies used by the prior systems and how ARIES can exploit parallelism during the redo and undo passes to speed up restart. The algorithms for taking checkpoints during restart and for media recovery are the same as those described in [MHLPS89].

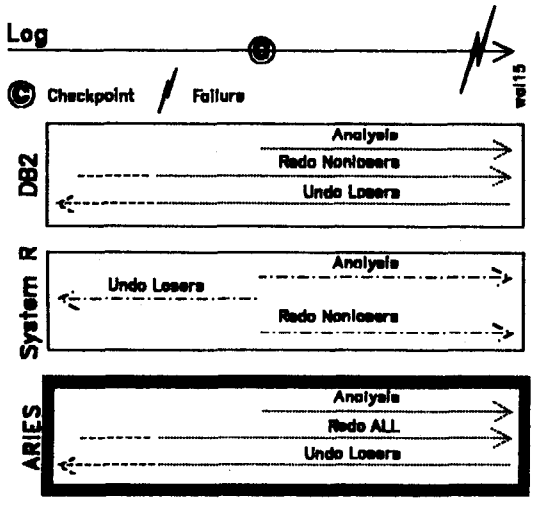

Figure 4: Restart Processing in Different Systems

3.4.1. Analysis Pass

The analysis pass begins by reading the Master record. The TransTab and DirtyPages tables are initialized from the EndChpt record. The log is scanned forward starting with the BeginChpt record until the end of the log is reached. During this scan, depending on its type, a log record R written by a transaction X is processed as follows:

- ***Update:*** If R represents the first log record in X's BW-chain, an entry is inserted into TransTab for X with State set to "active" and LastLSN set to R's LSN. Otherwise, in X's entry, LastLSN is set to R's LSN. If R.PageId is not in the DirtyPages table, a new entry is added to this table. The entry consists of R.PageId and R's LSN (as the RecLSN).
- ***CLR:*** In X's TransTab entry, LastLSN is updated. If R.PageId is not in the DirtyPages table, a new entry is added.
- ***Prepare:*** In X's TransTab entry, LastLSN is updated and State is set to "prepared".
- ***C-Committed:*** The entry in TransTab for R.ChildId is deleted.
- ***End:*** X is forgotten by removing X's entry from TransTab.

After the log is scanned until its end, **RedoLSN** is set to the minimum of the RecLSN values in DirtyPages. This is the log address from which the redo pass will start processing the log.

In summary, the output of the analysis pass is (1) the TransTab table, which contains the transactions that were in the "active" and "prepared" state at the time of the system failure, (2) the DirtyPages table, which contains the identifiers of the pages in the buffers that were *potentially* dirty when the system failure occurred, and (3) RedoLSN, which is the location on the log from which the redo pass should start processing the log.

3.4.2. Redo Pass

The redo pass reestablishes the state of the database at the time of the system failure and reacquires locks for "prepared" transactions. The redo pass algorithm described for ARIES in [MHLPS89] can be used without changes. The inputs to this algorithm are RedoLSN and DirtyPages produced during the analysis pass. Log records are examined starting from the RedoLSN and using the information in DirtyPages as a filter, the pages which may not be up to date with respect to the logged changes are examined. For each page which is examined, a log record's changes are redone if the page's LSN is less than the log record's LSN. Note that, unlike some other recovery methods (e.g., System R [GMBLL81]), ARIES redoes even the updates of transactions in the "active" state.

3.4.3. Undo Pass

During the undo pass, the effects of all transactions residing in the "active" state at the end of the analysis pass have to be undone. In the following, the set of BWC-trees of the these transactions are denoted by ***RestartUndo-Forest (RU-Forest)***. The fate of the transactions in the "prepared" state will be determined after contact is reestablished with the commit coordinator (see [MoLO86]).

Restart rollback is very similar to normal rollback. In reverse chronological order, the restart rollback process undoes and compensates for the log records which belong to the RU-Forest and which had not been undone ever before. It starts by initializing a KnownChains list for each transaction residing in the "active" state after the analysis pass. The KnownChains list of an "active" transaction contains one UndoNext pointer, which points to the last record of the transaction's BW-chain. Subsequently, it selects from these KnownChains lists the UndoNext pointer with the highest value. Then it reads the log record that the selected pointer points to and acts on it as described below. This is repeated until all the KnownChains lists are empty.

In order to describe how the restart rollback process acts on the different types of log records, we assume that it selects UndoNext pointer P residing in the KnownChains list of transaction X. Further, we assume that P points to log record R in BW-chain C. Depending on R's type the restart rollback process acts as follows:

- *Update, C-Committed:* Restart rollback acts on R in the same way as normal rollback does.
- *CLR:* P is removed from X's KnownChains list, and subsequently, all the UndoNext pointers in R.UndoNextSet are added to this list. That is, for each BW-chain which belongs to X's BWC-tree and which was known by a rollback process at the point of the system failure, an UndoNext pointer is added to X's KnownChains list. These pointers point to the log records to be undone next in the corresponding BW-chains.

4. Related Work

As far as we know, the only other recovery algorithm supporting WAL for nested transactions is the one presented in [Moss87]. Moss's WAL algorithm does not write CLRs and it does not seem to record LSNs in the nonvolatile storage versions of modified pages. That algorithm is not good enough to support fine-granularity of locking and avoid the numerous problems caused by not writing CLRs (the reader is referred to [MHLPS89] for detailed discussions concerning these topics and illustrations of the difficulties involved in supporting high concurrency efficiently using WAL). Especially with semantically-rich modes of locking (like increment/decrement) and operation logging, Moss's WAL approach of performing undo actions before redo actions during restart recovery and of not writing CLRs will not work.

Moreover, Moss does not discuss the concurrency control protocols and he assumes that the way the logging of changes is done guarantees idempotence. As discussed in [MHLPS89], the latter would make it impossible to support operation logging and efficient storage management for varying length objects. Use of LSNs on nonvolatile storage also is crucial for avoiding unnecessary and/or erroneous redo and undo work during restart recovery. Failures during restart processing will further compound the problems. ARIES and ARIES/NT support efficient restart and media recovery, and high concurrency by permitting operation logging, page-oriented redos and logical undos (see [MoLe89] for examples). For these reasons, we consider ARIES/NT to be fundamentally different from and more powerful than Moss's WAL algorithm.

Most recovery algorithms for nested transactions published so far are variations of the "version approach" described in [Moss81]. Briefly, this version algorithm goes as follows: When a transaction gets a write lock for an object, a new backup version of this object is created. This version is associated with the object and the transaction, and is used to restore the object should the transaction abort. The versions associated with an object are stored in a version stack, which is kept in volatile memory. When a subtransaction commits, its associated versions are offered to the parent. The parent accepts a version, if it does not already have an associated version for the same object. Otherwise, the offered version is discarded. If a transaction aborts, each of its associated versions is used to restore the objects directly or indirectly modified by the transaction. After that, the versions associated with the transaction can be destroyed. When a TL-transaction commits, the current state of each object directly or indirectly modified by this transaction is saved in stable storage. Then, all versions associated with this transaction are destroyed.

In most implementations of Moss's version algorithm (or variations of it), a version of an object is a complete copy of this object (e.g., see implementations in Eden [JNJBP82] and ARGUS [LCJS87]). An alternative approach is to store versions incrementally. In LOCUS [MuMP83], for example, file versions are stored incrementally - i.e., only those file pages that are new compared to the previous version need to be recorded in the new version.

Moss's version algorithm and its variations have several drawbacks:

If a transaction locks an object, a new version of the whole object has to be created. If the locked object is big, then the creation of new versions will be very expensive. On the other hand, if the objects to be locked are small, version stack management might become a substantial cost factor.

Those algorithms support only the **no-steal** buffer management policy, whereas ARIES and ARIES/NT support the no-steal as well as the **steal** policies (with *steal*, pages with uncommitted data may be written back to nonvolatile storage).

Partial rollbacks for nested transactions is not supported by them. Modelling savepoints [GMBLL81, HaRo87a, LHMW84] by means of subtransactions is too costly (inheritance of locks, generating new transaction identifier, etc.), since savepoints are used very frequently (e.g., to guarantee statement-level atomicity, which is required by ANS SQL).

5. Summary

A new recovery method for nested transactions, called ARIES/NT, has been presented. ARIES/NT is an extension of the ARIES recovery and concurrency control method that was introduced by Mohan, et al. in [MHLPS89] and that has been implemented to varying degrees in Starburst [Moha86], QuickSilver [HMSC88], the OS/2 Extended Edition[1] Database Manager [ChMy88], and DB2 V2R1. A high concurrency index management algorithm based on ARIES, and called ARIES/IM, has been implemented in the OS/2 EE Database Manager and is described, with extensions, in [MoLe89].

ARIES/NT is characterized by the following properties:

- It supports WAL for nested transactions, and, as far as we know, it is the only comprehensive WAL algorithm for nested transactions developed so far that permits semantically-rich modes of locking [BaRa87], operation logging, and efficient recovery.
- It allows arbitrary parallelism between related as well as unrelated transactions - i.e., a transaction may run concurrently with its superiors, inferiors, siblings and all other unrelated transactions.
- It supports concurrency control schemes allowing upward as well as downward inheritance of locks. That is, children may inherit locks to their parents, and vice versa.
- It supports savepoints at each transaction level - i.e., TL-transactions as well as subtransactions may establish savepoints.

Moreover, all the properties of ARIES described in [MHLPS89] hold for ARIES/NT also. No changes to fuzzy image copying (archive dumping), media recovery, buffer management, and deferred or selective restart algorithms of ARIES are necessitated by the introduction of the support for nested transactions. Some more details on ARIES/NT can be found in [RoMo89]. Additional discussions concerning BK-BK communications, BK recovery, BK data structures, and deadlocks arising from allowing concurrent execution of subtransactions and their ancestors will be presented in an expanded version of that paper [MoRo89].

6. References

BaRa87 Badrinath, B.R., Ramamritham, K. *Semantics-Based Concurrency Control: Beyond Commutativity*, **Proc. 3rd IEEE International Conference on Data Engineering**, February 1987.

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

DaLA88 Dasgupta, P., LeBlanc Jr., R., Appelbe, W. *The Clouds Distributed Operating System*, **Proc. 8th International Conference on Distributed Computing Systems**, San Jose, June 1988.

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

HaRo87a Haerder, T., Rothermel, K. *Concepts for Transaction Recovery in Nested Transactions*, **Proc. ACM-SIGMOD International Conference on Management of Data**, San Francisco, May 1987.

HaRo87b Haerder, T., Rothermel, K. *Concurrency Control Issues in Nested Transactions*, **IBM Research Report RJ5803**, Almaden Research Center, August 1987.

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

JNJBP82 Jessop, W.H., Noe, J., Jackobson, D.M., Baer, J.-L., Pu, C. *An Introduction to the Eden Transactional File System*, **Proc. 2nd IEEE Symp. on Reliability in Distributed Software and Database Systems**, Pittsburgh, July 1982.

LCJS87 Liskov, B., Curtis, D., Johnson, P., Scheifler, R. *Implementation of Argus*, **Proc. 11th ACM Symposium on Operating Systems Principles**, Austin, November 1987.

LHMW84 Lindsay, B., Haas, L., Mohan, C., Wilms, P., Yost, R. *Computation and Communication in R*: A Distributed Database Manager*, **ACM Transactions on Computer Systems**, Vol. 2, No. 1, February 1984.

MHLPS89 Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, **IBM Research Report RJ6649**, IBM Almaden Research Center, January 1989.

Moha86 Mohan, C. *An Overview of Starburst: An Extensible Relational DBMS*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Washington, May 1986.

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

MoLO86 Mohan, C., Lindsay, B., Obermarck, R. *Transaction Management in the R* Distributed Data Base Management System*, **ACM Transactions on Database Systems**, Vol. 11, No. 4, December 1986.

MoRo89 Mohan, C., Rothermel, K. *ARIES/NT: A Recovery Method Based on Write-Ahead Logging for a Very General Model of Nested Transactions*, **IBM Research Report**, IBM Almaden Research Center, Forthcoming.

Moss81 Moss, J.E.B. *Nested Transactions: An Approach to Reliable Distributed Computing*, **PhD Thesis**, Tech Rep MIT/LCS/TR-260, MIT, April 1981. Also as a modified version from MIT Press, 1985.

Moss87 Moss, E. *Log-Based Recovery for Nested Transactions*, **Proc. 13th International Conference on Very Large Data Bases**, Brighton, September 1987.

MuMP83 Mueller, E.T., Moore, J.D., Popek, G.J. *A Nested Transaction Mechanism for LOCUS*, **Proc. 9th ACM Symposium on Operating Systems Principles**, Bretton Woods, October 1983.

RoMo89 Rothermel, K., Mohan, C. *ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions*, **IBM Research Report RJ 6650**, IBM Almaden Research Center, January 1989.

SpPB88 Spector, A., Pausch, R., Bruell, G. *Camelot: A Flexible, Distributed Transaction Processing System*, **Proc. IEEE Compcon Spring '88**, San Francisco, March 1988.

1 OS/2 is a trademark of the International Business Machines Corp.