D-ARIES: A Distributed Version of the ARIES Recovery Algorithm

Jayson Speer and Markus Kirchberg

Information Science Research Centre, Department of Information Systems,
Massey University, Private Bag 11 222, Palmerston North 5301, New Zealand
JaysonSpeer@gmail.com, M.Kirchberg@massey.ac.nz

Abstract. This paper presents an adaptation of the ARIES recovery algorithm that solves the problem of recovery in Shared Disk (SD) database systems, whilst preserving all the desirable properties of the original algorithm. Previous such adaptations of the ARIES algorithm have failed to solve some of the problems associated with SD systems, resulting in a number of undesirable compromises being made. The most significant problem is how to assign log sequence numbers (LSNs) in such a way that the system is recoverable after a crash. Existing adaptations have required, among other things, a perfectly synchronised global clock and a central merge of log data either during normal processing or crash recovery, which clearly imposes a significant overhead on the database system. This adaptation of ARIES removes this requirement entirely, meaning that log merges and synchronised clocks become entirely unnecessary. Further enhancements that allow the Redo and Undo phases of recovery to be performed on a page-by-page basis have significantly reduced the recovery time. Additionally, it is possible for the database to return to normal processing at the end of the Analysis phase, rather than waiting for the recovery process to complete.

1 Introduction

Introduced by Mohan et al. [1], the ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) algorithm has had a significant impact on current thinking on database transaction logging and recovery. It has been incorporated into Lotus Notes as well as a number of major database management systems (DBMSs) [2].

ARIES, like many other algorithms, is based on the WAL (Write Ahead Logging) protocol that ensures recoverability of a database in the presence of a crash. However, ARIES' Repeating History paradigm sets it apart from all other WAL based protocols. The repeating history paradigm requires, during recovery, that the database be returned to the same state as it was before the crash occurred. This aspect of ARIES allows it to support fine granularity locking, operation locking (i.e. increment and decrement) and partial rollbacks. ARIES further supports the use of buffer managers that use the Steal and No-Force approach to page replacement [3].

1.1 Recovery in Shared Disk Architectures

Often the performance of a single DBMS is unacceptable, resulting in the use of multiple systems where the database is split across multiple systems. The two major architectures are Shared Disk (SD) or Shared Nothing (SN) architecture.

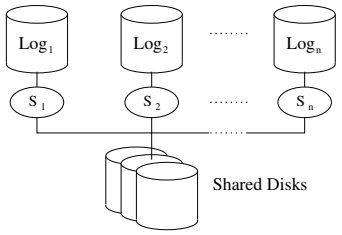

Fig. 1. Simplified Shared Disk (SD) Architecture.

In the SD architecture, the disks containing the database are shared amongst the different systems. Each system has an instance of the DBMS executing on it and may read or update any portion of the database held on the shared disks. In the simplified architecture shown in Figure 1, the disks containing the database are shared amongst the systems $\{S_1, ..., S_n\}$, with each system having an instance of the DBMS. Each system may perform updates on any portion of the database and will log each such update in its local log $\{Log_1, ..., Log_n\}$.

In the event of one or more of the systems failing (i.e. power failure, etc.), the algorithm must be able to return the system to a consistent state afterwards. Here, such nodes are referred to as crash nodes and are distinguished from non-crash nodes, since they play slightly different roles in the recovery process.

1.2 Problems

The introduction of the SD architecture adds an extra dimension of complexity to the problem of recovery, which can be almost entirely attributed to assigning log sequence numbers (LSNs) and how to interpret them during crash recovery.

Mohan and Narang ([4], p. 312) detailed a number of these problems, which include:

1. During recovery, how do we determine whether or not there is a need to redo the updates of a log record if the addresses of log records (when used as the LSN values) are not monotonically increasing across all systems?
2. If the log records representing updates to a particular page are scattered across the different systems local log files, how do we start to recover the page and how do we merge the log records from the different local log files?

ARIES has been adapted for use in SD architectures [5], with several different schemes being proposed. However, each scheme made a number of significant

compromises in order to address the aforementioned problems. The major assumptions made for all these schemes are:

1. 'For the purposes of handling data recovery, one system in the SD complex which has connectivity to all the local logs' disks produces a merged version of those logs.' ([5], p. 194)
2. '... the LSN is a timestamp and that the clocks across the SD complex are perfectly synchronised.' ([5], p. 195)

The first assumption clearly introduces an extra layer of overhead to the logging process that is not present when ARIES is applied to a single system database, due to the extra communication and processing that each system must now perform. Such problems will be exacerbated in the case where the system performing the log merge becomes a bottleneck in the process.

The second assumption not only adds an extra layer of overhead due to the requirement that all clocks in the system are perfectly synchronised, but also adds an extra layer of complexity to the handling of LSNs. Whilst logging, each record is assigned a unique LSN when that record is appended to the log. In single system databases, LSNs are typically 'the logical addresses of the corresponding log records' ([1], p. 96). This allows a correspondence to be established between LSNs and the physical location of the corresponding log record. Under the proposed assumption, this correspondence can no longer be established. This means extra data structures must be implemented to relate LSNs to the corresponding logical address, which causes a drop in database performance.

Additionally, in the case where ownership of a page is transferred to another system without first forcing that page, the scheme proposed by Mohan & Narang [5] requires a log merge in order to enable recovery of the page. Rahm [6] avoids this problem by prohibiting the change of page ownership altogether.

Lomet [7] offers the method, of all those investigated, that most resembles the one presented here, where state identifiers are used to track updates to data objects. Whilst this method does not require globally synchronised clocks, it does require the state identifiers to be monotonically increasing for any given data object, which introduces an undesirable layer of complexity and breaks the direct correspondence between the logical and physical addresses of log records. Other restrictive assumptions are also made, such as requiring that a page be forced to disk before transferring ownership to another node. Bozas & Kober [8] offer a similar method to [7] that requires state identifiers to be monotonically increasing for a page, but additionally does not support logical operation logging or Steal buffer management policies.

1.3 Solutions

The algorithm proposed in this paper addresses the problem of recovery within a SD system whilst addressing the aforementioned problems. In particular, the need to have LSNs monotonically increasing across all systems has been removed. This removes all reliance on clocks and allows the correspondence between LSNs

and the physical location of the corresponding log records to be preserved. The algorithm also removes any need for the merging of log data (during normal processing or recovery), but requires that all nodes participate in crash recovery.

Additional optimisations were made to the Redo and Undo phases of the recovery process whereby these phases are now performed on a page-by-page basis. Performing the Redo and Undo phases on a page-by-page basis allows a much higher degree of concurrency during the recovery process. This technique also provides the basis for a proposed method to improve concurrency during the rollback of transactions during normal processing.

During recovery, the time that the database is quiesced is reduced since normal processing can commence as soon as the Analysis phase of recovery is complete. Before the end of the Analysis phase, an exclusive lock is acquired (on behalf of the recovery algorithm) on those pages that must have changes reapplied (Redo phase) or removed (Undo phase). As a result, all other pages remain available for normal processing. Since the Redo and Undo phases are performed on a page-by-page basis, the exclusive lock on each page is released as soon as possible, reducing the unavailability of the database to a minimum.

It was also important not to impose any unnecessary overhead on the algorithm in terms of both communication costs and logging. The extra logging required is very small with a single extra field being added to some log record types and fields removed from others. Communication is typically the bottleneck in any distributed environment. Therefore, reducing the amount of communication required or, more significantly, increasing the concurrency of communication, will improve the overall system performance. Concurrency of communication during the Redo and Undo phases of recovery have been significantly increased by performing both the Redo and Undo phases of recovery independently on each page, hence decreasing the overall delay due to communication.

While addressing the problems experienced in previous adaptations of ARIES to the SD environment, it was important not to lose those aspects of ARIES that make it both unique and widely accepted. For this reason, the algorithm preserves the repeating history paradigm, fine granularity locking and partial rollbacks. It also places no restriction on the nature of buffer management used, thereby supporting the No-Force and Steal approaches previously supported.

1.4 Overview

This paper is organised into the following sections. Section 2 introduces the original ARIES recovery algorithm briefly. Sections 3 to 5 present our adaptation of the ARIES algorithm targeted at SD database systems. Section 6 draws conclusions and identifies the areas in which further work might be undertaken.

2 ARIES

This section provides a brief overview of the original ARIES algorithm. The reader is referred to [1] for the details about ARIES and to [9] for an introduction in the ARIES family of algorithms.

ARIES, like many other algorithms, is based on the WAL protocol that ensures recoverability of a database in the presence of a crash. All updates to all pages are logged (e.g. in logical fashion). ARIES uses an LSN stored on each page to correlate the state of the page with logged updates of that page. By examining the LSN of a page (called the PageLSN) it can be easily determined which logged updates are reflected in the page. Being able to determine the state of a page w.r.t. logged updates is critical whilst repeating history, since it is essential that any update be applied to a page once and only once. Failure to respect this requirement will in most cases result in a violation of data consistency.

Updates performed during forward processing of transactions are described by Update Log Records (ULRs). However, logging is not restricted to forward processing. ARIES also logs, using Compensation Log Records (CLRs), updates (i.e. compensations of updates of aborted / incomplete transactions) performed during partial or total rollbacks of transactions. By appropriate chaining of CLR records to log records written during forward processing, a bounded amount of logging is ensured during rollbacks, even in the face of repeated failures during crash recovery. This chaining is achieved by 1) assigning LSNs in ascending sequence; and 2) adding a pointer (called the PrevLSN) to the most recent preceding log record written by the same transaction to each log record.

When the undo of a log record causes a CLR record to be written, a pointer (called the UndoNextLSN) to the predecessor of the log record being undone is added to the CLR record. The UndoNextLSN keeps track of the progress of a 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.

Periodically during normal processing, ARIES takes fuzzy checkpoints in order to avoid quiescing the database while checkpoint data is written to disk. Checkpoints are taken to make crash recovery more efficient.

When performing crash recovery, ARIES makes three passes (i.e. Analysis, Redo and Undo) over the log. During Analysis, ARIES scans the log from the most recent checkpoint to the end of the log. It determines 1) the starting point of the Redo phase by keeping track of dirty pages; and 2) the list of transactions to be rolled back in the Undo phase by monitoring the state of transactions. During Redo, ARIES repeats history. It is ensured that updates of all transactions have been executed once and only once. Thus, the database is returned to the state it was in immediately before the crash. Finally, Undo rolls back all updates of transactions that have been identified as active at the time the crash occurred.

3 D-ARIES – Preliminaries

In the next three sections, we introduce a distributed version of the ARIES recovery algorithm (referred to as D-ARIES).

3.1 Overview

D-ARIES preserves the desirable properties of the original ARIES algorithm. Crash recovery still makes three passes over the log. However, supporting own-

ership changes of pages (without the need to flush respective pages first or merge logs) requires all nodes to be involved in crash recovery. Thus, it becomes even more important that the time the database is quiesced is reduced. D-ARIES achieves this since normal processing can resume as soon as the Analysis phase of recovery is complete. Furthermore, since the Redo and Undo phases are performed on a page-by-page basis, pages required during the Redo and Undo phases are made available to normal transaction processing as soon as possible, reducing the unavailability of the database to a minimum.

Section 4 discusses crash recovery processing in more detail. In the event of one or more of the systems failing, the crash recovery algorithm must be able to return the system(s) to the most recent consistent state.

Besides crash recovery, it is also necessary to discuss transaction rollback during normal processing. Section 5 considers two main classes of schedules that must be addressed when defining a rollback algorithm; these are schedules with cascading aborts and schedules without. D-ARIES takes advantage of multithreading and rolls back transactions on a page-by-page basis.

3.2 Logging

Previous adaptations of the ARIES algorithm require that the LSNs be globally monotonically increasing across all nodes in the system. This requirement introduces the problem of how to ensure that the LSNs are monotonically increasing on a global basis and means that the physical address of a log record can no longer be inferred from its LSN.

The adaptation of the ARIES algorithm presented here no longer requires LSNs to be globally monotonically increasing, however it is required that LSNs be monotonically increasing within each node. This requirement for monotonically increasing LSNs within a node is not a burden, but rather a benefit, since it allows a direct correspondence between a log record's physical address and logical address to be maintained.

In order to remove the need for globally monotonically increasing log numbers, a number of modifications must be made to the way the ARIES algorithm performs logging, these are:

1. Definition of a 'Distributed LSN'.
2. Modification of the CLR record.
3. Definition of a 'Special Compensation Log Record (SCR)'.
4. Addition of PageLastLSN pointers (to specific types of log records).

Distributed LSN. In order to identify which node a log record belongs to, a node identifier must be included in each LSN. Although the exact method for doing this is left open, for the purposes of this paper, the format of distributed LSNs is as follows:

```
RecNum.NodeId
```

Where `NodeId` is a globally unique identifier for the node that wrote the log record and the `RecNum` is monotonically increasing and unique within that node. For the purposes of this paper, the node identifier and record numbers for each `LSN` can be referred to individually as `LSN[NodeId]` and `LSN[RecNum]`.

Modification of the CLR Record. In this distributed incarnation of the ARIES algorithm, extensive changes are made to the CLR record, both in terms of the information it contains and the way in which it is used. Changes are:

1. The UndoneLSN field replaces the UndoNextLSN field. Whereas the UndoNextLSN records the LSN of the next operation to be undone, the UndoneLSN records the LSN of the operation that was undone.
2. The PrevLSN field is no longer required for the CLR record.
3. CLR records are now used to record undo operations during normal processing only, the newly defined SCR records (refer below) is used to record undo operations during crash recovery.

The rationale behind these modifications can be understood by observing the differences in how D-ARIES and the original ARIES algorithm perform undo operations and how this affects the information required by D-ARIES.

In the original ARIES algorithm, operations are undone one at a time in the reverse order to which they were performed by transactions. However, in order to increase concurrency, this algorithm can perform multiple undo operations concurrently, where updates to individual pages are undone independently of each other. The result of this is that although operations are undone in reverse order on a per page basis, it is now possible for operations belonging to a single transaction to be undone in an order that does correspond to the order in which they were done.

Definition of the SCR Record. The Special Compensation Log Record (SCR) is identical in almost every respect to the modified version of the CLR record, the only differences being:

- The record type field (SCR rather than CLR),
- SCRs are written during crash recovery rather than normal processing.

During normal rollback processing, operations are undone in the reverse order to which they were performed by individual transactions. However, during crash recovery rollback, operations are undone in reverse order that they were performed on individual pages. Having separate log records for compensation during recovery and normal rollback allows us to exploit this fact.

PageLastLSN Pointers. The PageLastLSN pointer[1] is added to all ULR, SCR and CLR records. It records the LSN of the record that last modified an object

[1] The PageLastLSN pointer does not only support page-by-page processing. It is also the crucial data structure that helps to avoid globally monotonically increasing LSNs.

on this page. Recording these PageLastLSN pointers provides an easy method of tracing all modifications made to a particular set of objects (stored on a single page). The PageLastLSN for each record must be a distributed LSN and include both the LSN and the node on which the record resides. This is due to the fact that the last modification to the page may have occurred on a different node.

3.3 Fuzzy Checkpoints

As with the original ARIES algorithm, a fuzzy checkpoint is performed in order to avoid quiescing the database while checkpoint data is written to disk. The following information is stored during the checkpoint: Active transaction table; and DirtyLSN value.

For each active transaction, the transaction table stores the following data:

TransId	Identifier of the active transaction.
FirstLSN	LSN of the first log record written on this node for the transaction.
Status	Either Active or Commit.

Thus, each node must maintain the FirstLSN of each active transaction. The value of FirstLSN can be easily set when the corresponding transaction table entry is created.

Given the set of pages that were dirty at the time of the checkpoint, the DirtyLSN value points to the record that represents the oldest update to any such page that has not yet been forced to disk.

4 D-ARIES – Crash Recovery

As with the original ARIES algorithm, recovery remains split into three phases, which are Analysis, Redo and Undo. However, unlike the original ARIES algorithm, recovery takes place on a page-by-page basis, where updates are reapplied (Redo phase) and removed from (Undo phase) pages independently from one another. The Redo phase reapplies changes to each page in the exact order that they were logged and the Undo phase undoes changes to each page in the exact reverse order that they were performed. Since the state of each page is accurately recorded (by use of the PageLSN), the consistency of the database will be maintained during such a process.

4.1 Data Structures.

The data collected by the algorithm during the Analysis phase is stored in the following data structures:

Transaction Status Table. The purpose of the transaction status (TransStatus) table is to determine the final status of all transactions that were active at some time after the last checkpoint. This information is used to determine whether changes made to the database should be kept or discarded.

The TransStatus table has the following fields:

TransId	Identifier of the transaction.
Status	Status of the transaction, which determines whether or not it must be rolled back. Possible states are Active, End and Commit.

Once the Analysis algorithm has completed scanning all required records, it will send the TransStatus table to all 'crash nodes' in the system. Upon receiving a TransStatus table from another node (N_i), the crash node will update its TransStatus table using the following rule:

> *The crash node will delete an entry for transaction T_j from the TransStatus table if: 1) The TransStatus table received from node N_i contains an entry for T_j; and 2) The status of that entry is either Commit or End.*

After having received the TransStatus table from all other nodes and updating the local table, any transaction with the status 'Active' is declared a 'loser transaction', whilst all other transactions are declared 'winner transactions'.

Local Page Link List. The purpose of the Local Page Link (LLink) list is to provide a linked list of records for each page modified by a node. This linked list is used in the Redo phase to navigate forwards through the log reapplying updates made to data objects on that page.

For each page that has CLR, SCR or ULR records, each node will create an LLink list, which is an ordered list of all LSNs that record changes to that page.

The data contained within the LLink list is sufficient to allow forward navigation through a set of records provided all records for the page are stored on the same node. However, in order to allow forward navigation when changes to a page are logged on multiple nodes, further data must be captured. This data is stored in the Distributed Page Link List (see below).

Distributed Page Link List. The purpose of the Distributed Page Link (DLink) list is to augment the LLink list in such a way that a linked list of records for each page is possible even when a page has log records stored on multiple nodes.

For each CLR, SCR or ULR record encountered that has a PageLastLSN pointer that points to another node, a DLink list entry will be created. Each DLink list entry has the following fields:

PageId	Identifier of the page.
LSN	LSN of the record.
PageLastLSN	PageLastLSN pointer of the record.
NodeId	Identifier of the node that the PageLastLSN pointer points to.

At the end of the Analysis phase, each node will send the relevant portion[2] of the DLink list to the other nodes. The data contained in the DLink list will

[2] Each node will be sent all entries where their node identifier is equal to NodeId.

be used to augment the LLink list for all pages. The rules for inserting data from a DLink list into an LLink list are:

1. *Locate the point in the LLink list where LSN = DLink.PageLastLSN.*
2. *Insert the corresponding LSN from the DLink list directly after this point.*

Example 1. DLink List (From Node 2)

LSN	PageLastLSN	NodeId
10.2	12.1	1

LLink List (Node 1)

5	7	12	16

First, locate the entry where LSN = 12.1 (PageLastLSN from DLink list), then insert 10.2 (LSN from DLink list) directly after this point.

Augmented LLink List (Node 1):

5	7	12	10.2	16

Page Start List. The purpose of the Page Start List is to determine, for each page, from which node to initiate the Redo phase of the recovery process. The Page Start List has the following fields:

PageId	Identifier of the page.

During the forward scan of the log, the first time the algorithm encounters a log record for a page P_j, it takes the following actions:

1. *Create a Page Start List entry for P_j.*

At the end of the Analysis phase, after having augmented the LLink list, revisit each entry of the local Page Start List and apply the following rule:

2. *Remove the entry if the first entry for the corresponding page in the augmented LLink list does not refer to the local node.*

After step 1. has been completed on all nodes, we are guaranteed to have captured all pages that have to be visited during the Redo and Undo phases. Thus, we are able to lock those pages exclusively. All other pages can then be made available for normal processing.[3] Multiple lock requests (all on behalf of the recovery algorithm) for the same page are possible, but do not cause any problems. Second, third, ... requests can simply be ignored.

Step 2. then removes all entries except those that refer to pages that have been updated first by the local node. This ensures that there is exactly one Page Start List entry per page (that is of interest to the crash recovery algorithm) across all nodes. This entry refers to the node which will be in charge of initiating the Redo pass for that particular page.

[3] Note: by this time, no communication between nodes nor access to persistent data other then the local log were required.

Page End List. The purpose of the Page End List is to determine, for each page, where the Undo phase of the recovery algorithm should stop processing. However, this is only an optional feature, which will be omitted due to space constraints. In the absence of the Page End List, the Undo phase terminates as soon as the ScanLSN record (see below) has been reached on any node.

Undone List. The purpose of the Undone List is to store a list of all operations that have been previously undone. The Undone List has the following fields:

PageId	Identifier of the page.
UndoneLSN	LSN of the record that has been undone.

During the scan of the log, whenever the algorithm encounters a CLR record, it adds an entry to the Undone List.

4.2 Analysis Phase.

During the Analysis phase, the algorithm collects all data that is required to restore the database to the most recent consistent state. This involves performing a forward scan through the log, collecting the data required for the Redo and Undo phases of recovery.

The Analysis phase of the recovery process is comprised of three steps, being: Initialisation, Data Collection and Completion.

Step 1: Initialisation. The initialisation of the Analysis phase involves reading the most recent checkpoint in order to construct an initial TransStatus table and determine the start point (ScanLSN) for the forward scan of the log.

TransStatus Table. For each transaction stored in the Active Transaction table, a corresponding entry is created in the TransStatus table.

Start Point. The start point of the scan (ScanLSN) is the point in the log where the node will start scanning and is computed as follows:

> The lowest LSN of either: 1) DirtyLSN; or 2) The lowest FirstLSN of any transaction in the TransStatus table whose status is 'Active'.

Step 2: Data Collection. During the forward scan of its log, each node will collect data to be stored in the data structures discussed in Section 4.1.

The type of record encountered during the log scan determines the data that is collected and into which data structure it is stored. The records from which the Analysis phase collects data are Commit Log Record, End Log Record, ULR, CLR, and SCR.

Commit Log Record. Each time a Commit Log Record is encountered, the recovery manager inserts an entry into the TransStatus table for the transaction with status set to Commit. Any existing entries for this transaction are replaced.

End Log Record. Each time an End Log Record is encountered, the recovery manager inserts an entry into the TransStatus table for the transaction with status set to End. Any existing entries for this transaction are replaced.

Update Log Record (ULR). Each time an ULR record is encountered, the following data is captured:

- If no entry exists in the TransStatus table for this transaction, then an entry is created for this transaction with status set to Active.
- Add an entry to the LLink list.
- If the PageLastLSN pointer points to a different node, then add an entry to the DLink list.
- Create a Page Start List entry as required.

Compensation Log Record (CLR). Each time a CLR record is encountered, in addition to capturing all data described for an ULR record, an entry will be added to the Undone List.

Special Compensation Log Record (SCR). The same data is captured for an SCR record as that captured for an ULR record.

Step 3: Completion. Once the node has completed the forward scan of the log, it performs the following actions:

1. *Acquire an exclusive lock, on behalf of the recovery algorithm, on all pages identified in the Page Start List.*
2. *Send the following data: TransStatus table (to all crash nodes) and DLink list (to all other nodes).*
3. *Upon receiving this data from other nodes, the algorithm performs the following tasks: 1) Update the TransStatus table; 2) Augment the LLink lists.*
4. *Once all LLink lists have then augmented, update the Page Start List.*
5. *Once a crash node has received the TransStatus table from all other nodes, it can send a list of 'loser transactions' to all other nodes.*

Notes:

- Once all nodes in the system have acquired the required locks (refer Step 1), the database can commence normal processing. Only those pages that are locked for recovery will remain unavailable.
- Once a node has received the DLink lists from all other nodes (and augmented its LLink lists), it can enter the Redo phase.
- Once a node has received a list of loser transactions from all crash nodes, it can potentially enter the Undo phase.

4.3 Redo Phase.

The Redo phase is responsible for returning each page in the database to the state it was in immediately before the crash. In order for a node to enter the Redo phase, it must have augment its LLink lists and the Page Start List.

For each page that the node has in its Page Start List, the Redo algorithm will spawn a thread that 'repeats history' for that page. Given a page P_j, history is repeated by performing the following tasks:

1. Start by considering the oldest log record for page P_j that was written after PageLSN. This requires reading the page into main memory.
2. Using the augmented LLink lists for page P_j, move forward through the log until no more records for this page exist.
3. Each time a re-doable record is encountered, reapply the described changes. Re-doable records are: SCR, CLR and ULR records.

Once the thread has processed the last record for this page, the recovery algorithm may enter the Undo phase for this page. The recovery algorithm may enter the Undo phase for different pages at different times, for example page P_1 might enter the Undo phase while page P_2 is still in the Redo phase.

Once the recovery algorithm has completed the Redo phase for all pages, an End Log Record can be written for all transactions whose status is Commit in the TransStatus table. Since a transaction can have a Commit entry on only a single node, it is guaranteed that exactly one End Log Record will be written for such transactions. For expediency, this can be deferred until after the Undo phase is complete if so desired.

4.4 Undo Phase.

The Undo phase is responsible for undoing the effects of all updates that were performed by so-called 'loser transactions'. In order for a node to enter the Undo phase, it must have received a list of loser transactions from all crash nodes.

The thread that was spawned for the Redo phase for page P_j will now begin working backwards through the log undoing all updates to the page that were made by loser transactions by performing the following tasks:

1. Work backwards through the log using the PageLastLSN pointers processing each log record until all updates by loser transactions have been undone.
2. Each time an SCR or ULR record is encountered, take the following actions:
   - *Special Compensation Log Record (SCR).*
     (a) Jump to the record immediately preceding the record pointed to by the UndoneLSN field.

     The UndoneLSN field indicates that during a previous invocation of the recovery algorithm, the updates recorded by the record at UndoneLSN have already been undone.

– *Update Log Record (ULR).*
If the update was not written by a loser transaction or has previously been undone (the Undone List is used to determine this), then no action is taken. Otherwise, the following actions are taken to undo the update:
(a) Write an SCR record that describes the undo action to be performed with the UndoneLSN field set equal to the LSN of the ULR record whose updates have been undone.
(b) Execute the undo action described in the SCR record written.

Once the thread has completed processing all records back to ScanLSN, the page can be unlocked, on behalf of the recovery algorithm, and made available for normal processing again. The advantage of allowing each page to be unlocked individually is that the database can return to normal processing as quickly as possible.

Once the recovery algorithm has completed the Undo phase for all pages, an End Log Record can be written for all transactions whose status is Active in the TransStatus table (these are the loser transactions). The node that is responsible for the transaction will write this record.

4.5 Crashes During Crash Recovery.

By preserving ARIES' paradigm of repeating history, it can be guaranteed that multiple crashes during crash recovery will not affect the outcome of the recovery process. The Redo phase ensures that each update lost during the crash is applied exactly once by using the PageLSN value to determine which logged updates have already been applied to the page. Since all compensation operations are logged during the Undo phase, the Redo phase and the nature of the Undo phase ensure that compensation operations are also performed exactly once. The UndoneLSN plays a similar role in D-ARIES as the UndoNextLSN does in ARIES.

4.6 Enhancements

It is possible to further enhance the fault tolerance and speed of recovery of this algorithm in the face of a node that has crashed and is slow to become available. This is achieved by making both the database partition and recovery log of each node in the system accessible to each other node in the system.

During recovery, it is then possible for a non-crash node to act as a 'proxy' for the crash node. That is, the non-crash node can perform recovery on behalf of the crashed node. Whilst acting as a proxy, a non-crash node would read the log record of the crash node and perform updates on the crash node's database partition, just as the crash node would do if it were available.

Clearly such an enhancement offers real benefits in terms of recovery speed, particularly when a crash node becomes unavailable for an extended period of time. By the time the crash node becomes available, its database partition and log will be in a consistent state and the node will be able to commence processing immediately.

5 D-ARIES – Rollback During Normal Processing

Having defined the algorithm for rollback of transactions during crash recovery, it is now necessary to do the same for normal processing. There are two main classes of schedules that must be considered when defining a rollback algorithm; these are schedules with cascading aborts and schedules without.

The case where cascading aborts do not is exist is trivial, where rolling back a transaction simply involves following the PrevLSN pointers for the transaction backwards undoing each operation as it is encountered. Since cascading aborts do not exist in these schedules, no consideration need be given to conflicts between the aborting transaction and any other transactions.

The case where cascading aborts do exist is a great deal more complex, since rolling back a transaction may necessitate the rollback of one or more other transactions. Each time an operation is undone, it is necessary to consider which transactions, if any, must be rolled back in order to avoid database inconsistencies.

In the original ARIES algorithm, rollback of transaction T_i involves undoing each operation in reverse order by following the PrevLSN pointers from one ULR record to the next. Whenever an undo operation for transaction T_i conflicts with an operation in some other transaction T_j, a cascading abort of transaction T_j must be initiated. Transaction T_i must then suspend rollback and wait for transaction T_j to rollback beyond the conflicting operation before it can recommence rollback.

Clearly this is not the most efficient method, since the rollback of the entire transaction is suspended due to a single operation being in conflict. A more desirable method is to suspend rollback only of those operations that are in conflict and to continue rollback of all other operations. It is also desirable to trigger the cascading abort of all transactions in conflict as early as possible. By taking advantage of multi-threading, it is possible to roll back a transaction on a page-by-page basis. This allows a transaction in rollback to simultaneously:

- Trigger multiple cascading aborts,
- Suspend rollback of updates to pages whilst waiting for other transactions to roll back, and
- Continue rolling back updates that do not have any conflicts.

Partial rollback of transactions is achieved by establishing save points [10] during processing, then at some later point requesting the rollback of the transaction to the most recent save point. This can be contrasted with total rollback that removes all updates performed by the transaction.

5.1 Algorithms.

Rollback of a transaction T_i is achieved by the use of a single 'Master Thread' that is responsible for coordinating the rollback process and multiple 'Slave Threads' that are responsible for the rollback of updates made to individual pages.

Master Thread. The master thread is responsible for coordinating the rollback of a transaction by performing the following actions:

- Triggering the cascading abort of transactions as required.
- Undoing all update operations that are not in conflict with update operations from other transactions.
- Spawning a new slave thread whenever a conflict detected requires the undo of updates to a page be delayed while other transaction(s) roll back.

Algorithm 1:

Parameters.	
T_i	Identifier of transaction to roll back.
SaveLSN	The save point for partial rollback of the transaction. If SaveLSN is Null, then total rollback will occur.

Variables.	
CurrLSN	The LSN of the record currently being processed.
P_S	Set of page identifiers for which slave threads are spawned.

Procedure.	
1.	CurrLSN = LSN of last update performed by T_i.
2.	WHILE (Record.LSN $\neq$ SaveLSN) DO
3.	Move to Record at CurrLSN
4.	IF (Current operation is not in conflict with any other) THEN
5.	IF (Record.PageId $\notin P_S$) THEN
6.	Undo current operation
7.	END IF
8.	ELSE
9.	IF (Record.PageId $\notin P_S$) THEN
10.	Add Record.PageId
11.	Spawn a new slave thread for page Record.PageId Parameters: (Record.PageId, SaveLSN, NodeId[4])
12.	END IF
13.	Trigger the cascading abort of any transaction(s) that have conflicting operations (and are not already aborting).
14.	END IF
15.	CurrLSN = Record.PrevLSN
16.	END DO

The algorithm terminates once the master thread has reached the save point and has received a *Done.* message from all slave threads spawned.

Slave Thread. The slave thread is responsible for undoing all updates made by the transaction to a single page. The slave thread must not undo any operation until any conflicting operation(s) have been rolled back. Once the slave thread has completed rolling back all changes to the page, it sends a *Done.* message back to the master thread.

[4] If the SaveLSN = Null, then NodeId is the node on which the master thread was spawned, otherwise NodeId = SaveLSN[NodeId]. This is to ensure that all slave threads send their *Done.* messages to the same node.

Algorithm 2:

Parameters.	
P_i	Identifier of page to roll back.
SaveLSN	The save point for partial rollback of the transaction. If SaveLSN is Null, then total rollback will occur.
NodeId	Identifier of the node that should receive the *Done.* message.

Variables.	
CurrLSN	The LSN of the record currently being processed.

Procedure.	
1.	WHILE (Record.LSN $\neq$ SaveLSN) DO
2.	Move to record at CurrLSN
3.	IF (Current operation is in conflict with any other) THEN
4.	Wait until conflicting operations(s) have been rolled back.
5.	END IF
6.	Undo current operation
7.	CurrLSN = Record.PrevLSN
8.	END DO
9.	Send *Done.* Message to the master thread.

Optimisation. In rollback, it is possible for both the master thread and the slave threads to reduce the frequency with which they check for conflicts between the current operation and operations belonging to other transactions. Given a ULR record written for page P_j by transaction T_i, it is only necessary to check for conflicts if the last ULR record written for page P_j was not written by transaction T_i. This can be achieved by checking if the PageLastLSN of the previous log record written by transaction T_i points to the current log record written by transaction T_i. If this is the case, no conflict checking need be performed.

6 Conclusions

The algorithm presented in this paper offers an adaptation of ARIES that solves the problem of recovery in SD database systems. The desirable properties of ARIES, such as fine granularity locking[5], operation locking and partial rollbacks, have been preserved without imposing significant overhead on the system.

The original ARIES algorithm relies on the LSNs being globally monotonically increasing in order to ensure recoverability of the database after a crash. As such, previous adaptations have sought to ensure that LSNs are monotonically increasing throughout the SD system rather than resolve this reliance. This has resulted in undesirable compromises being made including the requirement for a perfectly synchronised global clock and the global merge of the logs. This algorithm removes ARIES' dependency on globally monotonically increasing LSNs, which in turn renders a global log merge or clock of any kind redundant.

[5] Same as with ARIES, D-ARIES supports fine granularity locking assuming that recoverability is ensured by the transaction manager.

Further enhancements were made to decrease the time taken for the database to recover from a crash and reduce the time that the database remains unavailable for normal processing. Such enhancements include allowing normal processing to commence after completion of the Analysis phase and the use of multi-threading to increase the concurrency of recovery operations. The concept of multi-threaded recovery was further adapted to provide a mechanism of increased concurrency of transaction rollback during normal processing.

Also, it should not be forgotten to mention that the proposed algorithm offers benefits even to centralised database systems. Multi-threaded recovery allows for a high degree of parallelism. Also, support for recovery from isolated hardware failures (e.g. a single-page restore after a torn write) is provided. Moreover, the proposed algorithm readily permits to exploit a common and very effective optimisation, namely logging of disk writes.

Future work identified in this area includes the adaptation of this algorithm to the multi-level transaction model and the use of multi-threading to further improve concurrency of both recovery and normal processing.

References

1. Mohan, C., Haderle, D.J., Lindsay, B.G., Pirahesh, H., Schwarz, P.M.: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems (TODS) 17 (1992) 94–162
2. Mohan, C.: ARIES family of locking and recovery algorithms (2004) http://www.almaden.ibm.com/u/mohan/ARIES_Impact.html
3. Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Computing Surveys (CSUR) 15 (1983) 287–317
4. Mohan, C., Narang, I.: Data base recovery in shared disks and client-server architectures. In: Proceedings of the 12th International Conference on Distributed Computing Systems, IEEE Computer Society Press (1992) 310–317
5. Mohan, C., Narang, I.: Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In: Proceedings of the 17th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc. (1991) 193–207
6. Rahm, E.: Recovery concepts for data sharing systems. In: Proceedings of 21st International Symposium on Fault-Tolerant Computing. (1991) 368–377
7. Lomet, D.B.: Recovery for shared disk systems using multiple redo logs. Technical Report CLR 90/4, Digital Equipment Corp., Cambridge Research Lab, Cambridge, MA (1990)
8. Bozas, G., Kober, S.: Logging and crash recovery in shared-disk parallel database systems. Technical Report TUM-I9812, SFB-Bericht Nr. 342/06/98 A, Dept of Computer Science, Munich University of Technology, Germany (1998)
9. Mohan, C.: Repeating history beyond ARIES. In Atkinson, M.P., Orlowska, M.E., Valduriez, P., Zdonik, S.B., Brodie, M.L., eds.: VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK, Morgan Kaufmann (1999) 1–17
10. 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 (CSUR) 13 (1981) 223–242