

# C-ARIES: A Multi-Threaded 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  
Contact: [Markus.Kirchberg@ieee.org](mailto:Markus.Kirchberg@ieee.org)

**Abstract.** The ARIES recovery algorithm has had a significant impact on current thinking on transaction processing, logging and recovery. In this paper, we present the C-ARIES algorithm, which extends the original algorithm with the capability to perform transaction aborts and crash recovery in a highly concurrent manner. Concurrency is achieved by performing transaction aborts and the Redo and Undo recovery phases on a page-by-page basis. An additional benefit of C-ARIES is that the database system can commence 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 IBM's DB2 Universal Database, Lotus Notes and a number of other systems [2].

ARIES, like many other recovery algorithms, is based on the WAL (Write Ahead Logging) protocol that ensures recoverability of databases in the presence of a crash. However, ARIES' Repeating History paradigm sets it apart from other WAL based protocols. The repeating history paradigm involves returning the database to the exact state it was in before the crash occurred and allows ARIES to support properties such as fine granularity locking, operation logging and partial rollbacks.

### 1.1 Contribution

We propose an adaptation of the original ARIES recovery algorithm. The proposed C-ARIES algorithm extends the original algorithm with the capability to perform transaction aborts during normal processing and crash recovery in a highly concurrent manner. Concurrency is achieved by performing transaction aborts and the Redo and Undo crash recovery phases on a page-by-page basis. An additional benefit of our approach is that the database system can be returned to normal processing at the end of the Analysis phase, rather than waiting for the recovery process to complete.

## 1.2 Outline

This paper is organised as follows: Section 2 introduces the original ARIES recovery algorithm. Sections 3 to 5 present the proposed concurrent ARIES (i.e. C-ARIES) adaptation. Finally, Section 6 concludes this paper.

## 2 ARIES

This section provides a brief overview of the original ARIES algorithm [1, 3].

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. ARIES uses a log sequence number (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 C-ARIES: The Multi-Threaded ARIES Algorithm

In the next three sections, we introduce a multi-threaded version of the ARIES recovery algorithm (referred to as *C-ARIES*).

In this section, basic concepts such as modifications to data structure, logging and checkpointing are outlined. Based on these concepts, Section 4 discusses crash recovery processing in greater detail. Subsequently, Section 5 considers necessary modifications to transaction rollback during normal processing.

C-ARIES preserves the desirable properties of the original ARIES algorithm. However, enhancements were made to the Redo and Undo phases of the crash recovery process, whereby these phases are now performed on a page-by-page basis. This results in a much higher degree of concurrency since operations that would normally be performed serially can now be performed concurrently. This page-by-page technique also provides the basis for the improved method to transaction aborts during normal processing.

It was important not to impose any unnecessary costs on the algorithm in terms of logging, since this is purely overhead on the system that offers no benefit until recovery is required [4]. The extra logging required for C-ARIES is very small with a single extra field being added to some log record types and fields removed from others.

#### 3.1 Logging

ARIES and also C-ARIES require that LSNs be monotonically increasing. This, however, is not a burden but rather a benefit. It allows a direct correspondence between a log record's physical and logical address to be maintained.

However, in order to adopt a page-by-page approach to crash recovery and transaction abort, a number of modifications must be made to the way the ARIES algorithm performs logging, these are as follows:

**Modification of the CLR.** In C-ARIES, extensive changes are made to the CLR, both in terms of the information it contains and the way in which it is used:

1. The `UndoneLSN` field replaces the `NextUndoLSN` 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 C-ARIES and the original ARIES algorithm perform undo operations and how this affects the information required by C-ARIES. In ARIES, operations are undone one at a time in the reverse order to which they were

performed by transactions. However, in order to increase concurrency, C-ARIES can perform multiple undo operations concurrently, where updates to individual pages are undone independently of each other.

**Definition of the SCR.** The Special Compensation log Record (SCR) is almost identical to the modified version of the CLR, the only differences being:

1. The record type field (SCR rather than CLR).
2. When SCR records are written.

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 is added to all ULR, SCR and CLR records. It records the LSN of the record that last modified an object 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 page).

### 3.2 Fuzzy Checkpoint

As with ARIES, 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**, i.e. an identifier of the active transaction.
- **FirstLSN**, i.e. the LSN of the first log record written for the transaction.
- **Status**, i.e. either *Active* or *Commit*.

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 C-ARIES: Crash Recovery

With C-ARIES, recovery remains split into three phases: Analysis, Redo and Undo. However, 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.

Data collected during the Analysis pass is stored in the following data structures:

**Transaction Status Table.** The *transaction status* (*TransStatus*) table determines 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 holds:

- **TransId**, i.e. the identifier of the active transaction.
- **Status**, i.e. the status of the transaction, which determines whether or not it must be rolled back. Possible states are: *Active*, *End* and *Commit*.

Any transaction with status ‘*Active*’ is declared a *Loser Transaction (LT)*, whilst all other transactions are declared *Winner Transactions (WTs)*.

**Page Link List.** The *page link (PLink)* list provides a linked list of records for each modified page. This list is used in the Redo phase to navigate forwards through the log. For each page that has CLR, SCR or ULR records, such a *PLink* list, which is an ordered list of all LSNs that modified that page, is created.

**Page Start List.** The *Page Start List* determines, for each page, from where to commence recovery. It holds the following field: **PageId**, i.e. the page identifier.

During the forward scan of the log, the first time the algorithm encounters a log record for a page  $P_j$ , it creates a *Page Start List* entry for page  $P_j$ . The *Page Start List* captures all pages that are to be visited during the Redo and Undo phases of recovery. Thus, we can lock those pages exclusively on behalf of the recovery algorithm and make available all other pages for normal processing<sup>1</sup>.

**Page End List.** The *Page End List* is an optional data structure intended to optimise the Undo phase of recovery by accurately determining where this phase should stop processing. The *Page End List* has the following fields:

- **PageId**, i.e. the identifier of the page.
- **TransId**, i.e. the identifier of the transaction that has modified the page.
- **EndLSN**, i.e. the LSN of the last record to process this page during Undo.

Given a set of records, the rule for creating and updating an entry for some page  $P_j$  in the *Page End List* proceeds as follows:

1. The first time a log record written for page  $P_j$  by transaction  $T_k$  is encountered, the following entry should be inserted into the *Page End List*: ( **PageId**, **TransId**, **EndLSN** ) = (  $P_j$ ,  $T_k$ , LSN from record ).  
Each page should have only one entry in the *Page End List* for each transaction that has written a log record for it.

<sup>1</sup> Note, at this stage there was no access to persistent data other than that of the log.

2. Once the scanning of all records back to `ScanLSN` is completed and the set of all winner and loser transactions is known, the following actions are performed:
  - (a) Delete all entries where `TransId` is in `TW`.
  - (b) For each page, retain only the entry with the lowest `EndLSN`.

Finally, there will be at most a single Page End List entry for any page. The `EndLSN` value indicates where the Undo phase will terminate for that page.

**Undone List.** The *Undone List* stores a list of all operations that have been previously undone. It has the following fields:

- `PageId`, i.e. the identifier of the page.
- `UndoneLSN`, i.e. the 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 a 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.** Initialisation 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 log scan starts from the lowest LSN of either the `DirtyLSN`, or the lowest `FirstLSN` of any active transaction in the `TransStatus` table.

**Step 2: Data Collection.** During the forward scan of the log, data for all data structures as discussed in Section 4.1 is collected. The type of record encountered during the log scan determines the data that is collected and into which data structure(s) 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, an entry is inserted 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, an entry is inserted into the TransStatus table for the transaction with status set to *End*.

*Update Log Record (ULR).* Upon encountering an ULR:

- 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 PLink list.
- Create a Page Start List entry and a Page End List entry as required.

*Compensation Log Record (CLR).* Upon encountering a CLR, the same step as for ULRs are performed and an entry is added to the Undone List.

*Special Compensation Log Record (SCR).* Same as for an ULR record.

**Step 3: Completion.** Once the forward scan of the log is completed, the recovery algorithm acquires an exclusive lock on all pages identified in the Page Start List. Subsequently, the DBS can commence normal processing. Only those pages that are locked for recovery will remain unavailable.

Now, the Redo phase may be entered. Once all loser transactions are known, a page can potentially enter the Undo phase – Page End Lists are not required but rather help the algorithm to determine where to terminate. In the absence of Page End Lists, Undo terminates as soon as the ScanLSN record is reached.

### 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. For each page in the 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 PLink lists for page  $P_j$ , move forward through the log until no more records for this page exist.
3. Each time a redoable record is encountered, reapply the described changes. Redoable 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. 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. 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 loser transactions. The thread that was spawned for the Redo phase will now begin working backwards through the log undoing all updates to the page that were made by loser transactions:

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:
  - (SCR): 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.
  - (ULR): If the update was not written by a loser transaction or has previously been undone (i.e. check against the Undone List), then no action is taken. Otherwise:
    - (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 `EndLSN`, the page can be unlocked and, thus, 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.

#### 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 C-ARIES as the `UndoNextLSN` does in ARIES.

### 5 C-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 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 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 ARIES, 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 [5] 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 Sketch of Algorithms

Rollback of a transaction 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.

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 active transactions. By consulting the lock table in the usual way, the concurrency control manager is able to detect these conflicts.
- 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.

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

The *slave thread* is responsible for undoing all updates made by the transaction to a single page. This thread must not undo any operation until any conflicting operation(s) have been undone. Once the slave thread has completed rolling back all changes to the page, it sends a 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  $T_i$ . The PageLastLSN pointers are ideally suited for determining whether or not the last ULR record written for page  $P_j$  was written by transaction  $T_i$ .

## 6 Conclusion

In this paper, we presented the C-ARIES algorithm, which extends the original ARIES recovery algorithm with the capability to perform transaction aborts and crash recovery in a highly concurrent manner. Concurrency is achieved by performing transaction aborts and the Redo and Undo crash recovery phases on a page-by-page basis. Additional enhancements are included that decrease the time taken for the DBS to recover from a crash and reduce the time that the database remains unavailable for normal processing.

It should also be mentioned that the proposed algorithm offers 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.

## 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 roll-backs using write-ahead logging. *ACM Transactions on Database Systems (TODS)* **17** (1992) 94–162
2. Mohan, C.: ARIES family of locking and recovery algorithms. On the Internet at [http://www.almaden.ibm.com/u/mohan/ARIES\\_Impact.html](http://www.almaden.ibm.com/u/mohan/ARIES_Impact.html) (2004)
3. Mohan, C.: Repeating history beyond ARIES. In Atkinson, M.P., Orlowska, M.E., Valduriez, P., Zdonik, S.B., Brodie, M.L., eds.: *Proceedings of 25th International Conference on Very Large Data Bases*, Morgan Kaufmann (1999) 1–17
4. Mohan, C., Treiber, K., Obermarck, R.: Algorithms for the management of remote backup data bases for disaster recovery. In: *Proceedings of the 9th International Conference on Data Engineering*, Washington, DC, USA, IEEE Computer Society (1993) 511–518
5. 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