ADMINISTRIVIA

Homework 4 ongoing
→ Due Friday, April 7\textsuperscript{th} at 11:59 p.m.

Project 3 ongoing
→ Due Sunday, April 9\textsuperscript{th} at 11:59 p.m.

Final exam Monday, May 1\textsuperscript{st}, 8:30 – 11:30 a.m.
LAST TIME: LOGGING

Failure Classification
Buffer Pool Policies
Shadow Paging
Write-Ahead Log
Logging Schemes
Checkpoints
CHECKPOINTS

Use the `<CHECKPOINT>` record as the starting point for analyzing the WAL. Anytxn that committed before the checkpoint is ignored ($T_1$).

$T_2 + T_3$ did not commit before the last checkpoint.

→ Need to **redo** $T_2$ because it committed after checkpoint.

→ Need to **undo** $T_3$ because it did not commit before the crash.
CHECKPOINTS – CHALLENGES

In this example, the DBMS must stall txns when it takes a checkpoint to ensure a consistent snapshot. → We will see how to get around this problem next class.

Scanning the log to find uncommitted txns can take a long time. → Unavoidable but we will add hints to the <CHECKPOINT> record to speed things up next class.

How often the DBMS should take checkpoints depends on many different factors…
CHECKPOINTS – FREQUENCY

Checkpointing too often causes the runtime performance to degrade.
→ System spends too much time flushing buffers.

But waiting a long time is just as bad:
→ The checkpoint will be large and slow.
→ Makes recovery time much longer.

Tunable option that depends on application recovery time requirements.
LOGGING CONCLUSION

Write-Ahead Logging is (almost) always the best approach to handle loss of volatile storage.

Use incremental updates (STEAL + NO-FORCE) with checkpoints.

On Recovery: undo uncommitted txns and redo committed txns.
CRASH RECOVERY

Recovery algorithms are techniques to ensure database consistency, transaction atomicity, and durability despite failures.

Recovery algorithms have two parts:
→ Actions during normal txn processing to ensure that the DBMS can recover from a failure.
→ Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability.
ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging

C. MOHAN
IBM Almaden Research Center
and
DON HADIELE
IBM Santa Teresa Laboratory
and
BRUCE LINDGAY, HAMID PIRAEI and PETER SCHWARTZ
IBM Almaden Research Center

In this paper, we present a simple and efficient method, called ARIES (Algorithms for Recovery and Isolation Exploiting Semantics), which supports partial rollbacks of transactions. Fine-granularity (e.g., row or tuple) locking and recovery using write-ahead logging (WAL). We introduce the paradigm of operating history to redo all missing updates before performing the rollbacks of the failed transactions during restart after a system failure. ARIES uses a big sequence number in each page to constrain the state of a page with respect to logged updates of that page. All updates of a transaction are logged, including those performed during rollbacks. By appropriate use of the sequence number, the amount of redo generated is bounded, and the amount of log that is stored during rollbacks is limited. In the face of repeated failures during restart or of serial rollbacks, we deal with a variety of failure scenarios very efficiently in building and operating an online transaction processing system. ARIES supports fast checkpointing, selective and deferred rollback, fast image copy, media recovery, and high concurrency (i.e., no concurrency checkpointing) which exploits the semantics of the operations and requires the ability to perform operation logging. ARIES is flexible with respect to the kinds of recovery management policies that can be implemented. It supports objects of varying length efficiently. By making parallelism during restart, propagating rollbacks, and using a pipelined (write-ahead log) recovery algorithm, the recovery performance and performance of the system is improved. Recovery algorithms and recovery techniques are implemented in a simple way based on the ARIES page structure, and is the implement of ARIES in the WAL-based recovery method of WALT.

Authors' address: C. Mohan, Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120; D. Hadiele, Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, CA 95134; B. Lindgaye, H. Piraesh, and P. Schwartz, IBM Almaden Research Center, San Jose, CA 95120.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage. ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

ACM Transactions on Database Systems, Vol. 15, No. 1, March 1990, Pages 94 - 132
ARIES – MAIN IDEAS

Normal execution: Write-Ahead Logging:
→ Any change is recorded in log on stable storage before the database change is written to disk.
→ Works with STEAL + NO-FORCE buffer pool policies.

Recovery: Three phases
→ Analysis: Use log to determine what transaction were executing and what pages were dirty before the crash.
→ Redo: Replay history to restore database to exact state before the crash.
→ Undo: Undo transactions that had not committed before the crash.
**ARIES – OVERVIEW**

**Analysis:** Figure out which txns committed or failed since checkpoint & which bufferpool pages were dirty.

**Redo:** Repeat all actions.

**Undo:** Reverse effects of failed txns.
TODAY'S AGENDA

- Log Sequence Numbers
- Normal Commit & Abort Operations
- Fuzzy Checkpointing
- Recovery Algorithm
WAL RECORDS

We need to extend our log record format from last class to include additional info.

Every log record now includes a globally unique log sequence number (LSN).
→ LSNs represent the physical order that txns make changes to the database.

Various components in the system keep track of LSNs that pertain to them…
### LOG SEQUENCE NUMBERS

<table>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>flushedLSN</td>
<td>Memory</td>
<td>Last LSN in log on disk</td>
</tr>
<tr>
<td>pageLSN</td>
<td>page&lt;sub&gt;x&lt;/sub&gt;</td>
<td>Newest update to page&lt;sub&gt;x&lt;/sub&gt;</td>
</tr>
<tr>
<td>recLSN</td>
<td>page&lt;sub&gt;x&lt;/sub&gt;</td>
<td>Oldest update to page&lt;sub&gt;x&lt;/sub&gt; since it was last flushed</td>
</tr>
<tr>
<td>lastLSN</td>
<td>T&lt;sub&gt;i&lt;/sub&gt;</td>
<td>Latest record of txn T&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>MasterRecord</td>
<td>Disk</td>
<td>LSN of latest checkpoint</td>
</tr>
</tbody>
</table>
WRITING LOG RECORDS

Each data page contains a pageLSN.
→ The LSN of the most recent update to that page.

System keeps track of flushedLSN.
→ The max LSN flushed so far.

Before the DBMS can write page x to disk, it must flush the log at least to the point where:
→ pageLSN_x ≤ flushedLSN
15-445/645 (Spring 2023)

WAL (Tail)

Buffer Pool

Log Sequence Numbers

017 <T₅ BEGIN>
018 <T₅, A, 9, 8>
019 <T₅, B, 5, 1>
020 <T₅ COMMIT>

Not safe to evict because pageLSN > flushedLSN

Log Sequence Numbers

Database

WAL

001 <T₁ BEGIN>
002 <T₁, A, 1, 2>
003 <T₁ COMMIT>
004 <T₂ BEGIN>
005 <T₂, A, 2, 3>
006 <T₂ BEGIN>
007 <CHECKPOINT>
008 <T₂ COMMIT>
009 <T₃, A, 3, 4>
010 <T₄ BEGIN>
011 <T₄, X, 5, 6>
012 <T₄, Y, 9, 7>
013 <T₄, B, 4, 2>
014 <T₄ COMMIT>
015 <T₅, B, 2, 3>
016 <T₅, C, 1, 2>

PageflushedLSN

recLSN

pageLSN

A=9 B=5 C=2

Not safe to evict because pageLSN > flushedLSN

Safe to evict because pageLSN ≤ flushedLSN
WRITING LOG RECORDS

All log records have an **LSN**.

Update the **pageLSN** every time a txn modifies a record in the page.

Update the **flushedLSN** in memory every time the DBMS writes out the WAL buffer to disk.
NORMAL EXECUTION

Each txn invokes a sequence of reads and writes, followed by commit or abort.

Assumptions in this lecture:
→ All log records fit within a single page.
→ Disk writes are atomic.
→ Single-versioned tuples with Strong Strict 2PL.
→ **STEAL + NO-FORCE** buffer management with WAL.
TRANSACTION COMMIT

When a txn commits, the DBMS writes a **COMMIT** record to log and guarantees that all log records up to txn's **COMMIT** record are flushed to disk.

→ Log flushes are sequential, synchronous writes to disk.
→ Many log records per log page.

When the commit succeeds, write a special **TXN-END** record to log.

→ Indicates that no new log record for atxn will appear in the log ever again.
→ This does not need to be flushed immediately.
We can trim the in-memory log up to flushedLSN.

Transaction Commit:

Buffer Pool

Buffer Pool:

MasterRecord

WAL

Database

flushedLSN = 015
Aborting a txn is a special case of the ARIES undo operation applied to only one txn.

We add another field to our log records:
→ **prevLSN**: The previous **LSN** for the txn.
→ This maintains a linked-list for each txn that makes it easy to walk through its records.
TRANSACTION ABORT

Important: Need to record what steps we took to undo the txn.
A **CLR** describes the actions taken to undo the actions of a previous update record.

It has all the fields of an update log record plus the **undoNext** pointer (the next-to-be-undone LSN).

**CLRs** are added to log records but the DBMS does not wait for them to be flushed before notifying the application that the txn aborted.
## TRANSACTION ABORT – CLR EXAMPLE

<table>
<thead>
<tr>
<th>LSN</th>
<th>prevLSN</th>
<th>TxnId</th>
<th>Type</th>
<th>Object</th>
<th>Before</th>
<th>After</th>
<th>UndoNext</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>nil</td>
<td>T₁</td>
<td>BEGIN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>002</td>
<td>001</td>
<td>T₁</td>
<td>UPDATE</td>
<td>A</td>
<td>30</td>
<td>40</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>002</td>
<td>T₁</td>
<td>ABORT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
### TRANSACTION ABORT – CLR EXAMPLE

<table>
<thead>
<tr>
<th>LSN</th>
<th>prevLSN</th>
<th>TxnId</th>
<th>Type</th>
<th>Object</th>
<th>Before</th>
<th>After</th>
<th>UndoNext</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>nil</td>
<td>T1</td>
<td>BEGIN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>002</td>
<td>001</td>
<td>T1</td>
<td>UPDATE</td>
<td>A</td>
<td>30</td>
<td>40</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>002</td>
<td>T1</td>
<td>ABORT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>026</td>
<td>011</td>
<td>T1</td>
<td>CLR-002</td>
<td>A</td>
<td>40</td>
<td>30</td>
<td>001</td>
</tr>
</tbody>
</table>

**TIME**

The LSN of the next log record to be undone.
**TRANSACTION ABORT - CLR EXAMPLE**

<table>
<thead>
<tr>
<th>LSN</th>
<th>prevLSN</th>
<th>TxnId</th>
<th>Type</th>
<th>Object</th>
<th>Before</th>
<th>After</th>
<th>UndoNext</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>nil</td>
<td>T₁</td>
<td>BEGIN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>002</td>
<td>001</td>
<td>T₁</td>
<td>UPDATE</td>
<td>A</td>
<td>30</td>
<td>40</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>002</td>
<td>T₁</td>
<td>ABORT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>026</td>
<td>011</td>
<td>T₁</td>
<td>CLR-002</td>
<td>A</td>
<td>40</td>
<td>30</td>
<td>001</td>
</tr>
<tr>
<td>027</td>
<td>026</td>
<td>T₁</td>
<td>TXN-END</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>nil</td>
</tr>
</tbody>
</table>
ABORT ALGORITHM

First write an **ABORT** record to log for the txn. Then undo the txn's updates in reverse order. For each update record:

→ Write a **CLR** entry to the log.
→ Restore old value.

Lastly, write a **TXN-END** record and release locks.

Notice: **CLRs** never need to be undone.
TODAY’S AGENDA

Log Sequence Numbers
Normal Commit & Abort Operations
Fuzzy Checkpointing
Recovery Algorithm
NON-FUZZY CHECKPOINTS

The DBMS halts everything when it takes a checkpoint to ensure a consistent snapshot:

→ Halt the start of any new txns.
→ Wait until all active txns finish executing.
→ Flushes dirty pages on disk.

This is bad for runtime performance but makes recovery easy.
SLIGHTLY BETTER CHECKPOINTS

Pause modifying txns while the DBMS takes the checkpoint.
→ Prevent queries from acquiring write latch on table/index pages.
→ Don't have to wait until all txns finish before taking the checkpoint.

We must record internal state as of the beginning of the checkpoint.
→ Active Transaction Table (AT'T)
→ Dirty Page Table (DPT)
ACTIVE TRANSACTION TABLE

One entry per currently active txn.

→ **txnId**: Unique txn identifier.
→ **status**: The current "mode" of the txn.
→ **lastLSN**: Most recent *LSN* created by txn.

Remove entry after the **TXN-END** record.

Txn Status Codes:

→ **R** → Running
→ **C** → Committing
→ **U** → Candidate for Undo
DIRTY PAGE TABLE

Keep track of which pages in the buffer pool contain changes that have not been flushed to disk.

One entry per dirty page in the buffer pool:

→ **recLSN**: The *LSN* of the log record that first caused the page to be dirty.
SLIGHTLY BETTER CHECKPOINT

At the first checkpoint, assuming $P_{11}$ was flushed, $T_2$ is still running and there is only one dirty page ($P_{22}$).

At the second checkpoint, assuming $P_{22}$ was flushed, $T_2$ and $T_3$ are active and the dirty pages are ($P_{11}$, $P_{33}$).

This still is not ideal because the DBMS must stall txns during checkpoint…
FUZZY CHECKPOINTS

A fuzzy checkpoint is where the DBMS allows active txns to continue the run while the system writes the log records for checkpoint. → No attempt to force dirty pages to disk.

New log records to track checkpoint boundaries:
→ CHECKPOINT-BEGIN: Indicates start of checkpoint
→ CHECKPOINT-END: Contains ATT + DPT.
Assume the DBMS flushes $P_{11}$ before the first checkpoint starts.

Any txn that begins after the checkpoint starts is excluded from the ATT in the CHECKPOINT-END record.

The LSN of the CHECKPOINT-BEGIN record is written to the MasterRecord when it completes.
ARIES – RECOVERY PHASES

Phase #1 – Analysis
→ Examine the WAL in forward direction starting at MasterRecord to identify dirty pages in the buffer pool and active txns at the time of the crash.

Phase #2 – Redo
→ Repeat all actions starting from an appropriate point in the log (even txns that will abort).

Phase #3 – Undo
→ Reverse the actions of txns that did not commit before the crash.
ARIES – OVERVIEW

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**.

**Analysis:** Figure out which txns committed or failed since checkpoint.

**Redo:** Repeat all actions.

**Undo:** Reverse effects of failed txns.
ANALYSIS PHASE

Scan log forward from last successful checkpoint. If the DBMS finds a **TXN-END** record, remove its corresponding txn from **ATT**.

All other records:
→ If txn not in **ATT**, add it with status **UNDO**.
→ On commit, change txn status to **COMMIT**.

For update log records:
→ If page **P** not in **DPT**, add **P** to **DPT**, set its **recLSN=LSN**.
ANALYSIS PHASE

At end of the Analysis Phase:
→ **ATT** identifies which txns were active at time of crash.
→ **DPT** identifies which dirty pages might not have made it to disk.
ANALYSIS PHASE EXAMPLE

Modify A in page $P_{33}$

CRASH!
The goal is to repeat history to reconstruct the database state at the moment of the crash:
→ Reapply all updates (even aborted txns!) and redo CLRs.

There are techniques that allow the DBMS to avoid unnecessary reads/writes, but we will ignore that in this lecture...
REDO PHASE

Scan forward from the log record containing smallest \text{recLSN} in \text{DPT}.

For each update log record or CLR with a given \text{LSN}, redo the action unless:
→ Affected page is not in \text{DPT}, or
→ Affected page is in \text{DPT} but that record's \text{LSN} is less than the page's \text{recLSN}.
REDO PHASE

To redo an action:
→ Reapply logged update.
→ Set $\text{pageLSN}$ to log record's $\text{LSN}$.  
→ No additional logging, no forced flushes!

At the end of Redo Phase, write $\text{TXN-END}$ log records for all txns with status $\text{C}$ and remove them from the $\text{ATT}$.  

UNDO PHASE

Undo all txns that were active at the time of crash and therefore will never commit.

→ These are all the txns with U status in the ATT after the Analysis Phase.

Process them in reverse LSN order using the lastLSN to speed up traversal.
Write a CLR for every modification.
FULL EXAMPLE

**LSN**  | **LOG**
--- | ---
00 | \(<\text{CHECKPOINT-BEGIN}>\)
05 | \(<\text{CHECKPOINT-END}>\)
10 | \(<T_1, \text{A→P}_5, 1, 2>\)
20 | \(<T_2, \text{B→P}_3, 2, 3>\)
30 | \(<T_1 \text{ ABORT}>\)
40 | \(<\text{CLR: Undo } T_1 \text{ LSN } 10>\)
45 | \(<T_1 \text{ TXN-END}>\)
50 | \(<T_3, \text{C→P}_1, 4, 5>\)
60 | \(<T_2, \text{D→P}_5, 6, 7>\)

**prevLSNs**

X \text{ CRASH!}
FULL EXAMPLE

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00,05</td>
<td>&lt;CHECKPOINT-BEGIN&gt;, &lt;CHECKPOINT-END&gt;</td>
</tr>
<tr>
<td>10</td>
<td>&lt;T₁, A→P₅, 1, 2&gt;</td>
</tr>
<tr>
<td>20</td>
<td>&lt;T₂, B→P₃, 2, 3&gt;</td>
</tr>
<tr>
<td>30</td>
<td>&lt;T₁ ABORT&gt;</td>
</tr>
<tr>
<td>40,45</td>
<td>&lt;CLR: Undo T₁ LSN 10&gt;, &lt;T₁ TXN-END&gt;</td>
</tr>
<tr>
<td>50</td>
<td>&lt;T₃, C→P₁, 4, 5&gt;</td>
</tr>
<tr>
<td>60</td>
<td>&lt;T₂, D→P₅, 6, 7&gt;</td>
</tr>
<tr>
<td></td>
<td><strong>CRASH! RESTART!</strong></td>
</tr>
<tr>
<td>70</td>
<td>&lt;CLR: Undo T₂ LSN 60, UndoNext&gt;</td>
</tr>
<tr>
<td>80,85</td>
<td>&lt;CLR: Undo T₃ LSN 50&gt;, &lt;T₃ TXN-END&gt;</td>
</tr>
<tr>
<td></td>
<td><strong>CRASH! RESTART!</strong></td>
</tr>
</tbody>
</table>

Flush dirty pages + WAL to disk!
**FULL EXAMPLE**

### ATT

<table>
<thead>
<tr>
<th>TxnId</th>
<th>Status</th>
<th>lastLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(_2)</td>
<td>U</td>
<td>70</td>
</tr>
</tbody>
</table>

### DPT

<table>
<thead>
<tr>
<th>PageId</th>
<th>recLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(_1)</td>
<td>50</td>
</tr>
<tr>
<td>P(_3)</td>
<td>08</td>
</tr>
<tr>
<td>P(_5)</td>
<td>10</td>
</tr>
</tbody>
</table>

### LSN

<table>
<thead>
<tr>
<th>Time</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00,05</td>
<td>&lt;CHECKPOINT-BEGIN&gt;, &lt;CHECKPOINT-END&gt;</td>
</tr>
<tr>
<td>10</td>
<td>&lt;T(_1), A→P(_5), 1, 2&gt;</td>
</tr>
<tr>
<td>20</td>
<td>&lt;T(_2), B→P(_3), 2, 3&gt;</td>
</tr>
<tr>
<td>30</td>
<td>&lt;T(_1) ABORT&gt;</td>
</tr>
<tr>
<td>40,45</td>
<td>&lt;CLR: Undo T(_1) LSN 10&gt;, &lt;T(_1) TXN-END&gt;</td>
</tr>
<tr>
<td>50</td>
<td>&lt;T(_3), C→P(_1), 4, 5&gt;</td>
</tr>
<tr>
<td>60</td>
<td>&lt;T(_2), D→P(_5), 6, 7&gt;</td>
</tr>
<tr>
<td>70</td>
<td>X CRASH! RESTART!</td>
</tr>
<tr>
<td>80,85</td>
<td>&lt;CLR: Undo T(_2) LSN 60, UndoNext 20&gt;</td>
</tr>
<tr>
<td>90,95</td>
<td>X CRASH! RESTART!</td>
</tr>
<tr>
<td>90,95</td>
<td>&lt;CLR: Undo T(_2) LSN 20&gt;, &lt;T(_2) TXN-END&gt;</td>
</tr>
</tbody>
</table>
What does the DBMS do if it crashes during recovery in the Analysis Phase?
→ Nothing. Just run recovery again.

What does the DBMS do if it crashes during recovery in the Redo Phase?
→ Again nothing. Redo everything again.
ADDITIONAL CRASH ISSUES (2)

How can the DBMS improve performance during recovery in the Redo Phase?
→ Assume that it is not going to crash again and flush all changes to disk asynchronously in the background.

How can the DBMS improve performance during recovery in the Undo Phase?
→ Lazily rollback changes before new txns access pages.
→ Rewrite the application to avoid long-running txns.
CONCLUSION

Mains ideas of ARIES:
→ WAL with **STEAL/NO-FORCE**
→ Fuzzy Checkpoints (snapshot of dirty page ids)
→ Redo everything since the earliest dirty page
→ Undo txns that never commit
→ Write **CLR**s when undoing, to survive failures during restarts

Log Sequence Numbers:
→ **LSN**s identify log records; linked into backwards chains per transaction via **prevLSN**.
→ **pageLSN** allows comparison of data page and log records.
NEXT CLASS

You now know how to build a single-node DBMS.

So now we can talk about distributed databases!