**Aries: A Transaction Recovery Method**

Presenter **Dibesh Shakya**
Discussion Leader **Alex Bradley**
Oct 14, 2009

Adapted from slides from "Database Management System" by Ramakrishnan and Gehrke

---

**Transaction**

- A block of operations on database objects
- Foundation for concurrent execution and recovery from system failure

**Example**

```sql
<begin transaction>
Read(A)
A=A-50
Write(A)
Read(B)
B=B+50
Write(B)
<end transaction>
```

---

**Review: ACID Properties**

- **Atomicity**: Either all actions in the Xact occur, or none occur.
- **Consistency**: If each Xact is consistent, and the DB starts in a consistent state, then the DB ends up being consistent.
- **Isolation**: The execution of one Xact is isolated from that of other Xacts.
- **Durability**: If a Xact commits, then its effects persist.

---

**States of a Transaction**

![Diagram showing the states of a transaction: Active, Failed, Aborted, Committed.]

---

**What happens if the system fails?**

- The goal of transaction recovery is to resurrect the db if this happens
- Maintains atomicity and durability
- Job of recovery manager
- Aries is one example of such a system
- A key tenet of Aries in fine-granularity locking for 4 reasons
  1. OO systems make users think in small objects
  2. "Object-oriented system users may tend to have many terminal interactions during …"
  3. More system use → more hotspots → require need less tuning from DBA
  4. Metadata is accessed often; cannot all be locked at once

---

**The 9 Goals of Aries (Desirable features)**

1. Simplicity
2. Operation Logging
3. Flexible storage management
4. Partial rollbacks
5. Flexible buffer management
6. Recovery independence
7. Logical undo
8. Parallelism and fast recovery
9. Minimal overhead

Achieving them result in increased concurrency, reduced I/O dependence and efficient CPU and buffer usage
Group Discussion on the 9 Goals

Considering the nine goals of the system:
- Which do you consider most important?
- Are there any of these goals that you would remove? If so, why?
- Are there any that you would add?
- What, if any, contradictions do you see between the goals?

Basic Idea: Logging

- Record REDO and UNDO information, for every update, in a log.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.
- Log: An ordered list of REDO/UNDO actions
  - Log record contains:
    - <XID, pageID, offset, length, old data, new data>
  - Size of log is much smaller than the size of the pages affected by the updates being recorded by the log
  - Log is maintained as a sequential file and results in sequential write to the stable storage

Reminder: Handling the Buffer Pool

- Transactions modify pages in memory buffers
- Writing to disk is more permanent
- When should updated pages be written to disk?
  - Force every write to disk?
    - Poor response time.
    - But provides durability.
  - Steal buffer-pool frames from uncommitted Xacts? (resulting in write to disk)
    - If not, poor throughput.
    - If so, how can we ensure atomicity?

Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  1. Must force log record for an update before the corresponding data page gets to disk.
  2. Must write all log records for a Xact before commit.
- #1 guarantees Atomicity.
- #2 guarantees Durability.

WAL & the Log

- Each log record has a unique Log Sequence Number (LSN).
- LSNs always increasing.
- Each data page contains a pageLSN.
  - The LSN of the most recent log record for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN flushed so far.
  - WAL: Before a page is written, pageLSN ≤ flushedLSN

Log Records

- Possible log record types:
  - Update
  - Commit
  - Abort
  - End (signifies end of commit or abort)
  - Compensation Log Records (CLRs)
  - for UNDO actions

LogRecord fields:
- prevLSN
- transID
- type
- pageID
- length
- offset
- before-image
- after-image

before and after image are the data before and after the update.
Creating Log Entries

- **Update**: Inserted when modifying a page.
  - Contains all the fields.
  - pageLSN of that page is set to the LSN of the record.
- **Commit**: When Xact commits a record is written in the log and is forcibly written to stable storage.
- **Abort**: created when Xact is aborted
  - Contains undoNextLSN field: LSN of next log record to be undone.
- **Compensation Log Records (CLR)**: Inserted before undoing an action described by an update log record
  - It happens during aborting or recovery.
  - Contains undoNextLSN field: LSN of next log record to be undone.

Other Log-Related Structures

Transaction manager also maintains the following tables:

- **Transaction Table**: Maintained by transaction manager
  - Contains tranID, status (running/committed/aborted), and lastLSN (LSN of most recent log record for it)
  - Xact removed from table when end record is inserted in the log
- **Dirty Page Table**: Maintained by buffer manager
  - Has one entry per dirty page in buffer pool
  - Contains recLSN – LSN of action which first made the page dirty
  - Entry is removed when page is written to the disk
  - Both tables must be reconstructed during recovery.

Checkpoints

- Periodically checkpoint, to minimize recovery time in system crash. Write to log:
  - **begin_checkpoint** record: when checkpoint began
  - **end_checkpoint** record: current Xact table and dirty page table.
  - Aries uses a ‘fuzzy checkpoint’:
    - Xacts continue to run; so these tables are accurate only as of time of begin_checkpoint
    - Dirty pages are not forced to disk;
    - Store LSN of checkpoint record in a safe place (master record).
  - When system starts after a crash:
    - Locate the most recent checkpoint
    - Restore Xact table and dirty page table from there.

Discussion on Fuzzy Checkpoint

- Do you think that fuzzy checkpoints are a good idea? Would you use them? Why or why not? Does it depend on the circumstances?
  - One alternative is a “full” checkpoint: all updates block while checkpoint runs, all pages are flushed to disk

Crash Recovery: Big Picture

- Start from a checkpoint (found via master record)
- Three phases. Need to:
  - Figure out which Xacts committed since checkpoint, which failed (Analysis)
  - REDO all actions
    - (repeat history)
  - UNDO effects of failed Xacts
Recovery: The Analysis Phase

- **Goals:**
  - Determine log record that Redo has to start at
  - Determine pages that were dirty at crash
  - Identify Xact's active at crash
  - Reconstruct state at checkpoint
  - Reconstruct Xact & dirty page tables using end_checkpoint record
  - Scan log forward from checkpoint
- **End record:**
  - Remove Xact from Xact table
- **Other records:**
  - Add Xact to Xact table, if not there
  - set lastLSN=LSN, set Xact status to C if log record is commit; otherwise set it to U (to be undone)
  - Update record:
    - If P not in Dirty Page Table, add P to DPT
    - set its recLSN=LSN

Recovery: The REDO Phase

- **We repeat history to reconstruct state at crash:**
  - Reapply all updates (even of aborted Xacts), redo CLRs
  - Scan forward from log record containing smallest recLSN in DPT. For each CLR or update log record, REDO the action unless:
    - Affected page is in the Dirty Page Table.
    - Affected page is in DPT, but has recLSN > LSN, or
    - pageLSN (in DB) > LSN
  - To REDO an action:
    - Reapply logged action
    - Set pageLSN to LSN.
    - No additional logging is required!
  - At the end of REDO, and End record is inserted in the log for each transaction with status C which is removed from Xact table.

Recovery: The UNDO Phase

- **Loser Xact's** = Xact active at the crash
- Need to undo all records of loser Xact's in reverse order
- ToUndo = set of all lastLSN values of all loser Xact's

**Algorithm:**

- **Repeat:**
  - Choose largest LSN among ToUndo
  - If this LSN is a CLR and undonextLSN=NULL
    - write an End record for this Xact.
    - remove record from ToUndo set
  - If this LSN is a CLR, and undonextLSN = NULL
    - add undonextLSN to ToUndo
  - Else this LSN is an update.
    - undo the update, write a CLR,
    - remove record from toUndo
    - add prevLSN of this record to ToUndo.

Until ToUndo is empty

Example of Recovery

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>begin_checkpoint</td>
</tr>
<tr>
<td>05</td>
<td>end_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update: T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>T1 abort</td>
</tr>
<tr>
<td>40</td>
<td>CLR: Undo T1 LSN 10, T1 End</td>
</tr>
<tr>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td>70</td>
<td>CLR: Undo T2 LSN 60</td>
</tr>
<tr>
<td>80</td>
<td>CLR: Undo T3 LSN 50, T3 end</td>
</tr>
</tbody>
</table>

Example: Crash During Restart!

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>begin_checkpoint, end_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update: T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>T1 abort</td>
</tr>
<tr>
<td>40</td>
<td>CLR: Undo T1 LSN 10, T1 End</td>
</tr>
<tr>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td>70</td>
<td>CLR: Undo T2 LSN 60</td>
</tr>
<tr>
<td>80</td>
<td>CLR: Undo T3 LSN 50, T3 end</td>
</tr>
<tr>
<td>90</td>
<td>CLR: Undo T2 LSN 20, T2 end</td>
</tr>
</tbody>
</table>
Today’s Recovery Algorithms

- Most popular are like ARIES:
  - maintain a log
  - use WAL
- Some Redo phases are different:
  - they don’t repeat the whole history
  - they only redo the non-loser transactions – “selective redo”

Final Discussion Questions

- Consider the arguments for fine-grained locking (at right). Do you find these arguments persuasive? Do you see any disadvantages to fine-grained locking? Would you have chosen to include fine-grained locking in ARIES?

- WAL vs. the shadow page method: What are the advantages and disadvantages of each? Which do you prefer?

- OO systems make users think in small objects
- Object-oriented system users may tend to have many terminal interactions during a transaction
- More system use → need less tuning
- Metadata is accessed often; cannot all be locked at once