Aries: A Transaction Recovery Method

Slides modified by Rachel Pottinger from slides from “Database Management Systems” by Ramakrishnan and Gehrke
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.
What happens if the system fails?

- The goal of transaction recovery is to resurrect the db if this happens
- Aries is one example of such a system
- A key tenant of Aries is 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 \(\rightarrow\) more hotspots \(\rightarrow\) need less tuning
  4. Metadata is accessed often; cannot all be locked at once
The 9 Goals of Aries

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
Operation logging

“let one transaction modify the same data that was modified earlier by another transaction which has not yet committed, when the two transactions’ actions are semantically compatible”
Partial rollbacks

Support save points and rollbacks to save points in order to be user friendly
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?
Flexible buffer management

Make the least number of restrictive assumptions about buffer management policies
Recovery independence

“The recovery of one object should not force the concurrent or lock-step recovery of another object”
Group Discussion on the 9 Goals

Rank the goals from 1 to 9 where 1 is the most important and 9 is the least important
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>
- and additional control info (which we’ll see soon).
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.
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

I.e., the latest thing on disk must also be written to disk on the log

- WAL & the Log
- LSNs
- pageLSNs
- flushedLSN

Log records flushed to disk

“Log tail” in RAM

I.e., the latest thing on disk must also be written to disk on the log
Log Records

Possible log record types:
- **Update**
- **Commit**
- **Abort**
- **End** (signifies end of commit or abort)

Compensation Log Records (CLR): 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 (i.e., page updated)

**Commit:**
- When Xact commits a record is written in the log and is forcibly written to stable storage.

**Abort:**
- created when Xact is aborted

**End:**
- created when Xact has completed all work (after commit or abort)

**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
  - Has one entry per active Xact
  - 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.
The Big Picture: What’s Stored Where

**LogRecords**
- prevLSN
- transID
- type
- pageID
- length
- offset
- before-image
- after-image

**Data pages**
- each
- with a
- pageLSN

**master record**
- Last to update page

**Xact Table**
- lastLSN
- status

**Dirty Page Table**
- recLSN

First thing made it dirty

Part of DBMS, but not in db (too slow)
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.
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

Go back far because “fuzzy” checkpoint
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 bookkeeping happens
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 it’s clear that it’s already been recorded (details omitted)

- **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

Those are the trans. we must undo
Discussion Questions

If you are designing a system for transaction processing,

- would you redo “loser” transactions?
- would you use selective redo?
- would you do a checkpoint after the analysis phase?

Why or why not?
Example of Recovery

Assume flush at checkpoint

### Transaction Table
<table>
<thead>
<tr>
<th>T#</th>
<th>lastLSN</th>
</tr>
</thead>
</table>

### Dirty Page Table
<table>
<thead>
<tr>
<th>T#</th>
<th>recLSN</th>
</tr>
</thead>
</table>

### ToUndo

### FlushedLSN

<table>
<thead>
<tr>
<th>Action</th>
<th>LSN</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin_checkpoint</td>
<td>00</td>
<td>begin_checkpoint</td>
</tr>
<tr>
<td>end_checkpoint</td>
<td>05</td>
<td>end_checkpoint</td>
</tr>
<tr>
<td>update: T1 writes P5</td>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>update T2 writes P3</td>
<td>20</td>
<td>update T2 writes P3</td>
</tr>
<tr>
<td>T1 abort</td>
<td>30</td>
<td>T1 abort</td>
</tr>
<tr>
<td>CLR: Undo T1 LSN 10</td>
<td>40</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>T1 End</td>
<td>45</td>
<td>T1 End</td>
</tr>
<tr>
<td>update: T3 writes P1</td>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>update: T2 writes P5</td>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
</tbody>
</table>

CRASH, RESTART

---

**Notes:**
- **First** made dirty
- **Max log flushed**
- **Check prev lsn, undo**
- **Delete from transaction table**
- **PrevLSNs**
### Example: Crash During Restart!

Still assume flush at checkpoint

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00,05</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>40,45</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>80,85</td>
<td>CLR: Undo T3 LSN 50, T3 end</td>
</tr>
</tbody>
</table>

**Transaction Table**

<table>
<thead>
<tr>
<th>T#</th>
<th>lastLSN</th>
</tr>
</thead>
</table>

**Dirty Page Table**

<table>
<thead>
<tr>
<th>T#</th>
<th>recLSN</th>
</tr>
</thead>
</table>

**ToUndo**

**FlushedLSN**
**Example: Crash During Restart!**

Still assume flush at checkpoint

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00,05</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>40,45</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>80,85</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>

**Still maybe not on disk**

**CRASH, RESTART**

**CRASH, RESTART**

**Still assume flush at checkpoint**

**Still maybe not on disk**
Discussion

The authors claim that the system is simple and efficient. Do you agree or disagree with each claim? Why or why not?
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”

  Can lead to trouble because must log undos (for media recovery), then would attempt to redo undo