A Transaction Recovery Method

Outline

- What's the problem?
- Terminology
- ARIES in action
  - Normal processing
  - System crash

ACID

- **Atomicity**: Either all actions in the transaction occur, or none occur
- **Consistency**: If each transaction is consistent and the DB starts in a consistent state, then the DB ends up being consistent.
- **Isolation**: The execution of one Transaction is isolated from that of other transactions
- **Durability**: The result of a committed transaction is stored persistently.

Discussion

- How much of the success of a database management system depends on reliable and efficient transaction management?
- Given that relational database management systems have been very successful, do you believe relational model has made the design of transaction management algorithms easier and more efficient? Why or why not?

What is ARIES good for?

- Problem: How to ensure the Atomicity and Durability if a transaction gets aborted or a media or device failure occurs?
  - Unroll transactions
  - redo transactions
- ARIES supports methods to deal with the problem
- ARIES features: fine granularity locking
  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 ➔ need less tuning
  4. Metadata is accessed often; cannot all be locked at once

Goals

1. Simplicity (Concurrency & recovery are complex)
2. Operation Logging (higher concurrency level)
3. Flexible storage management (avoid offline reorganization of data ➔ garbage collect)
4. Partial rollbacks (faster than total rollback)
5. Flexible buffer management ( concurrency I/O)
6. Recovery independence (selective recovery + image copy at different granularities e.g page-oriented)
7. Logical undo (concurrency)
8. Parallelism and fast recovery (multiprocessors, normal processing while recovery)
9. Minimal overhead (min log data, min CPU usage)
Excursus: Buffer management

- **Page Requests from Higher Levels**
  - **BUFFER POOL**
  - **MAIN MEMORY**
  - **DISK**

Q: When should an updated page be written to disc? → Need for a policy

Handling the buffer pool → Policies

- **Force**: make sure that every update is on disk before commit
  - Durability without REDO logging
  - Bad performance → Transaction has to wait for the disk

- **no Steal**: don’t allow buffer-pool frames with uncommitted updates to overwrite committed data on disk.
  - Atomicity without UNDO logging
  - Bad performance

<table>
<thead>
<tr>
<th>No Force</th>
<th>Force</th>
<th>No Steal</th>
<th>Steal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastest</td>
<td>Slowest</td>
<td>No REDO</td>
<td>No UNDO</td>
</tr>
</tbody>
</table>

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 (difference) 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.
  - With UNDO info (ARIES: logical undo, concurrency)
  - #2 guarantees Durability.
  - With REDO info (ARIES: physical REDO, simplicity, independency)

  **Note**: Note: we can implement Steal/No-force

Log in WAL

- **LSN**: log sequence number for every log record
  - Always increasing
- **pageLSN**:
  - LSN of the most recent log record for an update to that page
- Part of the log is in RAM another part is already on disc

Following the WAL Protocol requires that `flushedLSN >= pageLSN`

Outline

- What’s the problem?
- **Terminology**
  - ARIES in action
  - Normal processing
  - System crash
The Big Picture: What’s Stored Where

LogRecords
  LSN
  prevLSN
  XID
  type
  pageID
  length
  offset
  before-image
  after-image

DB
  Data pages
  each
  with a
  pageLSN

Xact Table
  lastLSN

Dirty Page Table
  recLSN
  flushedLSN

Master record

Log Records

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

CLR only
  UndoNxtLSN

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

before and after image are the data before and after the update.

Dirty page & Transaction table

Outline

• What’s the problem?
• Terminology
• ARIES in action
  • Normal processing
  • System crash

Normal processing

• Updating / forward processing
  • Adding records the log file
• Checkpoints (→ next Slide)
• Total/partial rollback
  • If transaction is aborted. Rollback to the last savepoint or the whole transaction → no double UNDO

Checkpoints

• Motivation: reduce the amount of recovery work after a System crash
• Idea: make a fuzzy snapshot of the DPT and TAT
  • 1st log entry: begin_chk
  • 2nd log entry end_chk. Save DPT and TAT on stable storage
  • Write begin_chk LSN to a save place (master record)
• Fuzzy, because there might be transaction between begin_chk and end_chk
• No attempt to force dirty pages to disk
  • effectiveness of checkpoint limited by oldest unwritten change to a dirty page
Outline

- What's the problem?
- Terminology
- ARIES in action
  - Normal processing
  - System crash

Crash Recovery: Big Picture

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

Analysis Phase

- Recreate Transaction & Dirty page table using the checkpoint
- Follow the log data from the checkpoint until the last LSN (like normal processing)
- End record: Remove Xact from Xact table.
- All Other records: Add Xact to Xact table, set lastLSN=LSN, change Xact status on commit.
- Also, for Update records: If page P not in Dirty Page Table, Add P to DPT, set its recLSN=LSN.

Redo pass

- Motivation: Repeat history to reconstruct state at crash
- Reapply all updates, also updates of looser transactions
- Procedure
  - Start at the log with the smallest recLSN
  - Redo all actions of log record or CLR unless
    - Affected Pages is not in the DPT or
    - Affected page is in DPT and recLSN > LSN or
    - pageLSN >= LSN (requires I/O, therefore last check)
  - Redo = apply action + set pageLSN = LSN
  - 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.

Example: Crash

<table>
<thead>
<tr>
<th>RAM</th>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin_checkpoint</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>end_checkpoint</td>
<td>05</td>
<td></td>
</tr>
<tr>
<td>update: T1 writes P5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>update: T2 writes P3</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>T1 abort</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>CLR: UndoT1 LSN 10</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>T1 End</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>update: T3 writes P1</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>update: T2 writes P5</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>CRASH, RESTART</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ToUndo = \{ l | l a lastLSN of a "loser" Xact \}

Repeat:

- Choose largest LSN among ToUndo
- If this LSN is a CLR and undoNextLSN==NULL
  - Write an End record for this Xact
- 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, add prevLSN to ToUndo

Until ToUndo is empty
Example: Crash During Restart!

```
<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>CRASH, START</td>
</tr>
<tr>
<td>80,85</td>
<td>CLR: Undo T2 LSN 60, T3 end</td>
</tr>
</tbody>
</table>
```

Discussion

- **Goals of ARIES:** Simplicity, operation logging, flexible storage management, partial rollbacks, flexible buffer management, recovery independence, logical undo, parallelism and fast recovery, minimal overhead.
- The authors claim that the system is simple and efficient. Do you agree or disagree with each claim? Why or why not? Do you think all of these goals are among the primary requirements of every transaction management system?

Limit the recovery work

- How do you limit the amount of work in REDO?
  - Flush asynchronously in the background.
  - Watch "hot spots"!
- How do you limit the amount of work in UNDO?
  - Avoid long-running Xacts.

Sources

- Slides Crash Recovery by Robert VanNatta
- Slides ARIES: Database Logging and Recovery by Zachary G. Ives
- Slides "Buffer Management Notes" by Amol Deshpande