CS 133: Databases

Warm-up Exercise

(See exercise sheet. You can start before class.)

The page is evicted from buffer pool, so is flushed first

Goals for Today

• Consider the implications of the buffer manager’s strategy for flushing pages on consistency

• Understand the role of the **recovery manager** in achieving xact Atomicity and Durability

• Reason about Write-Ahead-Logging and the ARIES recovery algorithm

Review: The ACID properties

• **Atomicity**: All actions in the Xact happen, or none happen.

• **Consistency**: If each Xact is consistent, and the DB starts consistent, it ends up consistent.

• **Isolation**: Execution of one Xact is isolated from that of other Xacts.

• **Durability**: If a Xact commits, its effects persist.

*Recovery Manager* helps with Atomicity and Durability!

Lab 2 Solution code on Sakai!
Recovery Manager: Motivation

- **Atomicity:**
  - Transactions may abort and “roll back” their changes.
- **Durability:**
  - What if DBMS stops running and data in memory is lost?

**Example:**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit</td>
<td>Abort</td>
<td>Commit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Desired state after system restarts:
- T1 & T3 should be **durable**
- T2, T4 & T5 should be **aborted** (effects not seen)

Use a log of actions to help UNDO and REDO changes to data on disk

Assumptions

- Concurrency control is in effect
  - **Strict 2PL**, in particular

- Updates are happening “in place”
  - i.e., data is overwitten on (deleted from) the actual page copies (not private copies)
  - *Writing a page to disk is atomic*

Can you think of a **simple** scheme (requiring no logging of a xact’s changes) to guarantee Atomicity & Durability?!

Handling the Buffer Pool

- **Force** every write to disk at xact commit time?
  - Poor response time
  - But provides **durability**

- **Steal** buffer-pool frames from uncommitted xacts?
  - If not, hurts concurrency
  - If so, how can we ensure **atomicity**?

Buffer Management Summary

- **No Force**
  - No Steal: Desired
  - Steal: Easy!

- **Force**
  - No Steal: Fastest
  - Steal: Slowest

Performance Implications

- No UNDO
  - No REDO
  - UNDO
  - No REDO

Logging/Recovery Implications

- No UNDO REDO
  - No REDO
  - UNDO
Preferred Policy: Steal/No-Force

- This combination is most complicated but allows for highest flexibility/performance

- **NO FORCE** (complicates enforcing Durability)
  - Dirty pages not forced to disk when xact commits
  - What if system crashes after a transaction commits but before a modified page written by that transaction makes it to disk?

  REDO info: Write just the changes in a safe place at commit time, just in case need to redo those modifications.

- **STEAL** (complicates enforcing Atomicity)
  - Dirty pages could be written to disk before transaction commits or aborts
  - What if transaction that performed updates aborts?
  - What if system crashes before transaction is finished?

  UNDO info: just in case, remember the old value of a page to undo the changes

Basic Idea: Logging

- Record REDO and UNDO information, for every update, in a log.

- **Log**: An ordered list of REDO/UNDO actions
  - Log record contains:
    
    `< xactID, pageID, offset, length, old data, new data >`

    In our examples we’ll simplify this to records like "update: T1 writes P2"

    – and additional control info (which we’ll see soon)

Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  1) **Must force** the log record for an update before the corresponding data page gets to disk.

    UNDO → Atomicity despite STEAL

  2) **Must force all log records** for a Xact before commit.

    (transaction is not committed until all of its log records including its “commit” record are on the stable log.)

    REDO → Durability despite NO FORCE

We’ll be looking at the **ARIES algorithm from IBM**
WAL & the Log

- Each log record has a unique Sequence Number (LSN)
  - LSNs always increasing
- System keeps track of flushedLSN
  - max LSN flushed to stable log so far.
- Each data page contains a pageLSN.
  - The LSN of the most recent log record for an update to that page.
- WAL (rule 1): For a page \( i \) to be written, must flush log at least to the point where:
  \[ \text{pageLSN}_i \leq \text{flushedLSN} \]

Log Records

prevLSN is the LSN of the previous log record written by this xact

Possible log record types:
- Update, Commit, Abort
- End
  - After commit or abort
  - Bookkeeping only, means clean-up is finished
- Checkpoint (for log maintenance)
- Compensation Log Records (CLR) – for UNDO actions

Other Log-Related State (in RAM)

- Transaction Table
  - One entry per currently active transaction
  - Entry removed when Xact commits or aborts
    - XactID | Status | lastLSN
      - Running, Committing, or Aborting
      - Most recent LSN written by this xact
- Dirty Page Table
  - One entry per dirty page currently in buffer pool
  - Entry removed when page flushed to disk
    - PageID | recLSN
      - LSN of log record that FIRST dirtied this page

Exercise 3

(a) No. DPT thinks first LSN that dirtied page 5 was LSN 50
(b) Yup. Page 2 is not in dirty page table. It could have been flushed to disk due to STEAL policy
Checkpointing

- Conceptually, keep log around for all time
  - this has performance/implementation issues...

- Periodically, the DBMS creates a **checkpoint**
  - Minimize time taken to recover if system crashes
  - Write to log:
    - `begin_checkpoint` record: Indicates when chkpt began.
    - `end_checkpoint` record: Contains current Xact table and dirty page table.
  - After `end_checkpoint`, log flushed

- Note: this is a ‘fuzzy checkpoint’:
  - Xacts continue to run; tables accurate only as of the time of the `begin_checkpoint` record.

Example Log: Normal Execution

<table>
<thead>
<tr>
<th>Trans</th>
<th>lastLSN</th>
<th>Stat</th>
<th>LSN</th>
<th>Log</th>
<th>prevLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1020</td>
<td>R</td>
<td>10</td>
<td>Update: T1 write P2</td>
<td>null</td>
</tr>
<tr>
<td>T2</td>
<td>3050</td>
<td>R</td>
<td>20</td>
<td>Update: T1 write P4</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>Update: T2 write P3</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pageld</th>
<th>recLSN</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

Store LSN of most recent checkpoint record in a safe place (master record).

Example Log: Normal Execution (cntd)

<table>
<thead>
<tr>
<th>Trans</th>
<th>lastLSN</th>
<th>Stat</th>
<th>LSN</th>
<th>Log</th>
<th>prevLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1020</td>
<td>R</td>
<td>10</td>
<td>Update: T1 write P2</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>Update: T1 write P4</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>Update: T2 write P3</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pageld</th>
<th>recLSN</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

More on Abort

- **To perform UNDO**, must have a lock on data!
  - No problem (we’re doing Strict 2PL)!

- **Before** restoring old value of a page, write a **compensation log record** (CLR):
  - CLR has one extra field: `undoNextLSN`
  - CLRs are *never* Undone (but they might be Redone when repeating history: guarantees **Atomicity**!)

- At end of UNDO, write an end log record

Assumptions: Strict 2PL, WAL, Steal/No-Force
Crash Recovery: Big Picture

- Start from a **checkpoint** (on disk)
- Three phases:
  1. **Analysis** – Determine dirty pages and active xacts at time of crash → **updates** tables from checkpoint:
     - **XactTable**: which Xacts were active at time of crash.
     - **Dirty Page Table**: which pages *might* have been dirty in the buffer pool at time of crash.
  2. **REDO** all actions to restore state at time of crash
  3. **UNDO** effects of failed Xacts

The Analysis Phase

- Re-establish knowledge of state at checkpoint
  - via **xact table** and **dirty page table** stored in the checkpoint
- Scan log forward from checkpoint:
  - **End** record: Remove Xact from Xact table.
  - All other records: Add Xact to Xact table, set lastLSN=LSN, change Xact status on commit/abort.
  - Also, for **Update** records: If page P not in Dirty Page Table, Add P to DPT, set its recLSN=LSN
- At end of Analysis...
  - Transaction table has which xacts were active at time of crash.
  - Dirty page table has which dirty pages *might not* be on disk

Phase 2: 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 dirty pages table
- For each redoable log record (update or CLR) with a given LSN, REDO the action **unless**:
  - Affected page is not in the Dirty Page Table, or
  - Affected page is in D.P.T., but has recLSN > LSN, or
  - pageLSN (on actual page in DB) > LSN. (this last case requires I/O)
- To REDO an action:
  - Reapply logged action.
  - Set pageLSN to LSN. No additional logging, no forcing

Phase 3: The UNDO Phase

**ToUndo** = {lastLSNs of all Xacts in the Trans Table}

**Repeat**:

- Choose (and remove) 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**. Write a CLR, undo the update,, add prevLSN to ToUndo

**Until** ToUndo is empty.
Exercise 4

(a) Xacts: T1, T3, T4, T5,
DPT: P5, P1, P3, P2

(b) Note: start REDO at LSN 40 (smallest in DPT)
so redo: 40, 50, 60, 90, 110, 130, 160, 180
(don’t need to redo 70 since Page 2’s recLSN > 70)