Crash Recovery

Chapter 18

Final Tuesday 12/11 @ 8am, FB141
~50 questions multiple choice
Open book, notes, no computers
Review: The ACID properties

- **Atomicity:** All actions of a transaction 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.

The **Recovery Manager** guarantees Atomicity & Durability.
Motivation

- **Atomicity:**
  - Transactions may abort ("Rollback").

- **Durability:**
  - What if DBMS Crashes?
    - ("Worse case", a few unfinished Xacts are lost)

- Desired Behavior after system restarts:
  - T1, T2 & T3 should be durable.
  - T4 & T5 should be aborted (no effect).
Assumptions

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

- Updates are happening “in place”.
  - i.e. data is overwritten on (or deleted from) non-volatile disk.

- A simple scheme to guarantee Atomicity & Durability?
Handling the Buffer Pool

- **Force** every write to disk? Stall DBMS until completed
  - Poor response time.
  - But provides durability.

- **Steal** buffer-pool frames from uncommitted Xacts? (flush dirty frames, only when a new frame is needed)
  - If not, poor throughput (multiple writes to same page).
  - If so, how can we ensure atomicity?

**Matrix:**

<table>
<thead>
<tr>
<th></th>
<th>No Steal</th>
<th>Steal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force</td>
<td>Trivial</td>
<td>Desired</td>
</tr>
<tr>
<td>No Force</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More on Steal and Force

- **STEAL** (why enforcing Atomicity is hard)
  - What if a page, P, dirtied by some unfinished Xact is written to disk to free up a buffer slot, and the Xact later aborts?
    - Must remember the old value of P at steal time (to UNDO the page write).

- **NO FORCE** (why enforcing Durability is hard)
  - What if system crashes before a page dirtied by a committed Xact is flushed to disk?
    - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.
Basic Idea: Logging

- Record sufficient information to REDO and UNDO every change in a log.
  - Write and Commit sequences saved to log (on a separate disk or replicated on multiple disks).
  - 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, \text{pageID}, \text{offset}, \text{length}, \text{old data}, \text{new data}\>
  - and additional control info (which we’ll see soon).
Write-Ahead Logging (WAL)

- The **Write-Ahead Logging** Protocol:
  1. Modification of a database object must *first* be recorded in the log, and the log updated, *before* any change to the object
  2. Must *write all log records* of a Xact *before it commits*.

- #1 guarantees Atomicity.
- #2 guarantees Durability.
- Exactly how is logging (and recovery!) done?
  - We’ll study the ARIES algorithms.
Each log record has a unique Log Sequence Number (LSN).
- LSNs are always increasing.

Each *data page* contains a *pageLSN*.
- LSN of most recent page modification.

System keeps track of *flushedLSN*.
- Max LSN flushed from the page buffer so far.

**WAL: Before** a page is written,
- pageLSN ≤ flushedLSN
Log Records

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

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

- **Transaction Table:**
  - One entry per active Xact.
  - Contains XID, status (running/committed/aborted), and lastLSN due to Xact

- **Dirty Page Table:**
  - One entry per dirty page in buffer pool
  - Contains recLSN -- the LSN of the log record which *first* dirtied the page
## Log and Table Entries

### Log’s “Tail”

<table>
<thead>
<tr>
<th>prevLSN</th>
<th>XID</th>
<th>type</th>
<th>pageID</th>
<th>length</th>
<th>offset</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1000</td>
<td>update</td>
<td>500</td>
<td>1</td>
<td>2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td>T2000</td>
<td>update</td>
<td>600</td>
<td>3</td>
<td>1</td>
<td>DEF</td>
<td>GHI</td>
</tr>
<tr>
<td></td>
<td>T2000</td>
<td>update</td>
<td>500</td>
<td>2</td>
<td>1</td>
<td>AZ</td>
<td>MN</td>
</tr>
<tr>
<td></td>
<td>T1000</td>
<td>update</td>
<td>505</td>
<td>1</td>
<td>3</td>
<td>Q</td>
<td>R</td>
</tr>
</tbody>
</table>

### Dirty Page Table

<table>
<thead>
<tr>
<th>pageID</th>
<th>recLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td></td>
</tr>
<tr>
<td>600</td>
<td></td>
</tr>
<tr>
<td>505</td>
<td></td>
</tr>
</tbody>
</table>

### Transaction Table

<table>
<thead>
<tr>
<th>transID</th>
<th>status</th>
<th>lastLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1000</td>
<td>running</td>
<td></td>
</tr>
<tr>
<td>T2000</td>
<td>running</td>
<td></td>
</tr>
</tbody>
</table>
Normal Execution of an Xact

- Series of **reads & writes**, terminated by **commit** or **abort**.
  - We will assume that write is atomic on disk.
    - In practice, additional details to deal with non-atomic writes.
- **Strict 2PL**.
- **STEAL, NO-FORCE buffer management, with Write-Ahead Logging**.
Checkpointing

- Periodically, the DBMS creates a **checkpoint**, to minimize recovery time in the event of a system crash. What is written to log and disk:
  - **begin_checkpoint** record: Indicates when chkpt began.
  - **end_checkpoint** record: Contains current *Xact table* and *dirty page table*. This is a “**fuzzy checkpoint**”:
    - Xacts continue to run; so these tables are accurate only as of the time of the **begin_checkpoint** record.
    - No attempt to force dirty pages to disk; effectiveness of checkpoint limited by oldest unwritten change to a dirty page. (So it’s a good idea to periodically flush dirty pages to disk!)
  - Store LSN of chkpt record in a safe place (**master** record).
The Big Picture: What’s Stored Where

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

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

**Xact Table**
- lastLSN
- status

**Dirty Page Table**
- recLSN

**flushedLSN**

**DB**

**RAM**
For now, consider an explicit abort of a Xact.
  - No crash involved.

We want to “play back” the log in reverse order, UNDOing updates.
  - Get lastLSN of Xact from Xact table.
  - Can follow chain of log records backward via the prevLSN field.
  - Before starting UNDO, write an Abort log record.
    - For recovering from crash during UNDO!
Abort, cont.

- To perform UNDO, must have a lock on data!
- Before restoring old value of a page, write a Compensation Log Record (CLR):
  - Continue logging while you UNDO!!
  - CLR has one extra field: undonextLSN
    - Points to the next LSN to undo
  - CLRs are *never* Undone (but they might be Redone when repeating history: guarantees Atomicity!)
- At end of UNDO, write an “end” log record.
Transaction Commit

- Write commit record to log.
- All log records up to Xact’s lastLSN are flushed on a commit.
  - Guarantees that flushedLSN ≥ lastLSN.
  - Note that log flushes are sequential, synchronous writes to disk.
  - Many log records per log page.
- Commit() returns.
- Write end record to log.
Crash Recovery: Big Picture

- Start from a checkpoint (found via master record).
- ARIES 3 phases. Need to:
  - Analysis: Figure out which Xacts committed since last checkpoint, and which did not finish.
  - REDO all logged actions.
    - repeat “writing” history
  - UNDO effects of unfinished “loser” Xacts.
Recovery: The Analysis Phase

- Reconstruct state at checkpoint.
  - via the **end_checkpoint** record.

- Scan log forward from checkpoint.
  - **End** record: Remove Xact from Xact table because it safely completed.
  - **Other records**: Add Xact to Xact table, set **lastLSN=LSN**, change Xact status on **commit**.
  - **Update** record: If P not in Dirty Page Table,
    - Add P to D.P.T., 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 rec containing smallest recLSN in D.P.T. For each CLR or update log rec 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 (in DB) ≥ LSN.

- To **REDO** an action:
  - Reapply logged changes (restore to before state).
  - Set pageLSN to LSN. No additional logging!
Recovery: The UNDO Phase

ToUndo = \{ l \mid 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 of Recovery

#### Xact Table
- lastLSN
- status

#### Dirty Page Table
- recLSN
- flushedLSN

#### To Undo

<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</td>
</tr>
<tr>
<td>45</td>
<td>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>
</tbody>
</table>

---

**prevLSNs**

**CRASH, RESTART**
**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, RESTART</td>
</tr>
<tr>
<td>80,85</td>
<td>CLR: Undo T3 LSN 50, T3 end</td>
</tr>
<tr>
<td>90</td>
<td>CRASH, RESTART</td>
</tr>
<tr>
<td></td>
<td>CLR: Undo T2 LSN 20, T2 end</td>
</tr>
</tbody>
</table>

**RAM**

**Xact Table**
- lastLSN
- status

**Dirty Page Table**
- recLSN
- flushedLSN

**ToUndo**

**LSN**
- 00,05
- 10
- 20
- 30
- 40,45
- 50
- 60
- 70
- 80,85
- 90
Additional Crash Issues

- What happens if system crashes during Analysis? During REDO?
- How to limit the amount of work in REDO?
  - Flush dirty pages asynchronously in the background.
  - Watch out for “hot spots”!
- How to limit the amount of work in UNDO?
  - Avoid long-running Xacts.
Summary of Logging/Recovery

- **Recovery Manager** guarantees Atomicity & Durability.
- Uses WAL to allow STEAL/NO-FORCE w/o sacrificing correctness.
- LSNs identify log records; linked into backwards chains per transaction (via prevLSN).
- pageLSN allows comparison of data page and log records.
**Checkpointing**: A quick way to limit the amount of log to scan on recovery.

Recovery works in 3 phases:
- **Analysis**: Forward from checkpoint.
- **Redo**: Forward from oldest recLSN.
- **Undo**: Backward from end to first LSN of oldest Xact alive at crash.

Upon Undo, write CLRs.

Redo “repeats history”: Simplifies the logic!