CS 133: Databases

Spring 2017
Lec 19 – 3/30
Recovery
Prof. Beth Trushkowsky

Goals for Today

• Learn about transactions and locking in SimpleDB, and the role of the Buffer Manager

• 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!
Recovery Manager: Motivation

• Atomicity:
  – Transactions may abort (“Rollback”).

• Durability:
  – What if DBMS stops running?

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

Idea: Use a log of actions to help \textbf{UNDO} and \textbf{REDO} changes to the data on disk

Assumptions

• Concurrency control is in effect
  Strict 2PL, in particular

• Updates are happening “in place”
  – i.e., data is overwritten on (deleted from) the actual page copies (not private copies)
  – \textit{Writing a page to disk is atomic}

Can you think of a simple scheme (requiring no logging) to guarantee Atomicity & Durability?

Handling the Buffer Pool

• Force every write to disk at xact commit time?
  – Poor response time
  – But provides \textbf{durability}

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

Buffer Management Summary

Performance Implications

Logging/Recovery Implications
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 before a modified page written by a committed transaction makes it to disk?

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

Preferred Policy: Steal/No-Force

- **STEAL** (complicates enforcing Atomicity)
  - Dirty pages could be written to disk before xact commits or aborts
- **UNDO**: just in case, remember the old value of a page to undo the changes
  - What if the Xact that performed updates aborts?
  - What if system crashes before Xact is finished?

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 (CLRs)
  – for UNDO actions

Other Log-Related State (in RAM)

• Transaction Table
  – One entry per currently active transaction
  – Entry removed when Xact commits or aborts

<table>
<thead>
<tr>
<th>XactID</th>
<th>Status</th>
<th>lastLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running, Committing, or Aborting</td>
<td>Most recent LSN written by this xact</td>
<td></td>
</tr>
</tbody>
</table>

• Dirty Page Table
  – One entry per dirty page currently in buffer pool
  – Entry removed when page flushed to disk

<table>
<thead>
<tr>
<th>PageID</th>
<th>recLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSN of log record that FIRST dirtied this page</td>
<td></td>
</tr>
</tbody>
</table>

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.

• Note: this is a ‘fuzzy checkpoint’:
  – Xacts continue to run; tables accurate only as of the time of the begin_checkpoint record.
Example: Normal Execution

- Transaction is a series of reads and writes, followed by commit or abort

- Recall, assuming
  - Strict 2PL
  - Steal, No-Force buffer management
  - WAL

- **Commit** occurs: flush the logs to disk!

- **Abort** occurs: we need to undo all the xact’s changes.

Example Log: Normal Execution

<table>
<thead>
<tr>
<th>Trans</th>
<th>lastLSN</th>
<th>Stat</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>10</td>
<td>C</td>
</tr>
<tr>
<td>T2</td>
<td>30</td>
<td>R</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>LSN</th>
<th>Log</th>
<th>prevLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Update: T1 write P2</td>
<td>null</td>
</tr>
<tr>
<td>20</td>
<td>Update: T1 write P4</td>
<td>10</td>
</tr>
<tr>
<td>30</td>
<td>Update: T2 write P3</td>
<td>null</td>
</tr>
<tr>
<td>40</td>
<td>T1 commit</td>
<td>20</td>
</tr>
<tr>
<td>50</td>
<td>Update: T2 write P4</td>
<td>30</td>
</tr>
<tr>
<td>60</td>
<td>T1 end (xact entry removed, not shown)</td>
<td>40</td>
</tr>
</tbody>
</table>

Transaction Commit: Review

- Write **commit** record into log.
- Flush all log records up to and including the Xact’s **commit record** to log disk

**WAL Rule #2**: Ensure `flushedLSN >= lastLSN`

(Force log out up to lastLSN if necessary)

- Commit() returns
- Write **end** record to log
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

Simple Transaction Abort: Review

- Consider an explicit abort of a Xact (No crash)

- We want to “play back” the log in reverse order, UNDOing updates
  - Write an *Abort* log record before starting to rollback operations
  - Get *lastLSN* of Xact from Transaction table.
  - Follow chain of log records backward via *prevLSN*

  - For each update encountered:
    - Write a “CLR” (compensation log record) for each undone operation.
    - Undo the operation (using before image from log record)

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 CLR:
  - You continue logging while you UNDO
  - 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
Crash Recovery: Big Picture

- Start from a **checkpoint** (found via **master record**).
- **Three phases:**
  1. **Analysis** - update structures:
     - **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.
     (repeat history)
  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 \( \text{lastLSN}=\text{LSN} \), change Xact status on commit/abort.
  - Also, for Update records: If page \( P \) not in Dirty Page Table, Add \( P \) to DPT, set its \( \text{recLSN}=\text{LSN} \)

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

Phase 3: The UNDO Phase

\[ \text{ToUndo}=\{\text{lastLSNs of all Xacts in the Trans Table}\} \]

**Repeat:**

- Choose (and remove) largest LSN among ToUndo.
  - If this LSN is a **CLR** and \( \text{undonextLSN}==\text{NULL} \)
    - Write an **End** record for this Xact.
  - If this LSN is a **CLR**, and \( \text{undonextLSN} != \text{NULL} \)
    - Add \( \text{undonextLSN} \) to ToUndo
  - Else this LSN is an **update**. Write a CLR, undo the update,, add \( \text{prevLSN} \) to ToUndo
**Until ToUndo is empty.**
Example of Recovery (up to crash)

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG RECORD</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, UndoNxt=Null</td>
</tr>
<tr>
<td>50</td>
<td>T1 End</td>
</tr>
<tr>
<td>60</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td></td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td></td>
<td>CRASH, RESTART</td>
</tr>
</tbody>
</table>

Ex (cont.): Undo & Crash During Restart

After Analysis/Redo:
ToUndo: 50 & 60
ToUndo: 50 & 20
ToUndo: 20
After Analysis/Redo:
ToUndo: 70
ToUndo: 20
ToUndo: Finished!

Example (cont.): Analysis & Redo

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG RECORD</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>CLR: T1 LSN 10</td>
</tr>
<tr>
<td>40</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>50</td>
<td>CLR: T2 LSN 20</td>
</tr>
<tr>
<td>60</td>
<td>CLR: T3 LSN 50</td>
</tr>
<tr>
<td>70</td>
<td>CLR: Undo T2 LSN 20</td>
</tr>
<tr>
<td>80</td>
<td>CLR: Undo T3 LSN 50</td>
</tr>
<tr>
<td>90</td>
<td>CLR: Undo T2 LSN 20</td>
</tr>
<tr>
<td>100</td>
<td>CLR: T2 end</td>
</tr>
</tbody>
</table>

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)