CS 133: Databases

Fall 2016
Lec 19 – 11/14
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

Lab 4: Lock-based Concurrency Control

• Goal of Lab 4: add page-level locking to SimpleDB
  – Strict 2PL
  – Shared and Exclusive locks

  – Dealing with deadlock
  – Dealing with BufferPool eviction

Concurrency: How does it Happen?

• **Process**: executing instance of an application

• **Thread**: a path of execution within a process
  – Can be many threads within a process!
  – Threads have shared access to data structures within the process

Such as, say, a data structure managing Lock requests

Java: Thread Synchronization

- Thread synchronization in Java
  - Keyword `synchronized`

- Synchronize specific block of code:
  ```java
  synchronized(this) { // some code }
  ```

- Synchronize entire method:
  ```java
  private synchronized void flushPage(PageId pid) {
    // some code
  }
  ```

Skeleton code for Lock Manager and Buffer Pool already has these in place

Lab 4: Skeleton Code

- Modify BufferPool.java
  - `Lock Manager` as private inner class
  - Example: BufferPool.getPage() will require the transaction acquires a lock first!

- Your lock table data structure should be able to answer:
  - Given transactionId, which pages does it have locked?
  - Given a page Id, which xacts hold a lock on the page?
  - Given a page, which Permissions is it locked with?

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 UNDO and 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)
  - *Writing a page to disk is atomic*

**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**

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

**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

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

- What if system crashes before a modified page written by a committed transaction makes it to disk?
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?

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

**LogRecord fields:**
- LSN
- XactID
- prevLSN
- type
- pageID
- length
- offset
- before-image
- after-image

- prevLSN is the LSN of the previous log record written by *this xact*
- The records for a xact form a linked list backwards in time

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></td>
<td>Running, Committing, or Aborting</td>
<td>Most recent LSN written by this xact</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></td>
<td>LSN of log record that FIRST dirtied this page</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.

The Big Picture: What is Stored Where

- LogRecords
  - prevLSN
  - XID
  - type
  - pageID
  - length
  - offset
  - before-image
  - after-image
  - Store LSN of most recent checkpoint record in a safe place *master record*.

- Data pages
  - each with a pageLSN

- Xact Table
  - lastLSN
  - status

- Dirty Page Table
  - recLSN
  - LSN of most recent checkpoint

- RAM
  - flushedLSN
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.

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

Example Log: Normal Execution

<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>

<table>
<thead>
<tr>
<th>PagId</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>

Example Log: Normal Execution (cntd)

<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>
<tr>
<td>70</td>
<td>T2 abort</td>
<td>50</td>
</tr>
<tr>
<td>80</td>
<td>CLR: Undo 50, UndoNext = 30</td>
<td>70</td>
</tr>
<tr>
<td>90</td>
<td>CLR: Undo 30, UndoNext = null</td>
<td>80</td>
</tr>
<tr>
<td>100</td>
<td>T2 end (xact entry removed, not shown)</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PagId</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>

Must UNDO changes from T2!
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 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. Need to:
  1. Analysis - update structures:
     - Trans Table: 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 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 update log record 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.

Example of Recovery (up to crash)

<table>
<thead>
<tr>
<th>Xact Table</th>
<th>LSN</th>
<th>LOG RECORD</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xact Table</td>
<td>00</td>
<td>begin_checkpoint</td>
</tr>
<tr>
<td></td>
<td>05</td>
<td>end_checkpoint</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>update T2 writes P3</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>T1 abort</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>CLR: Undo T1 LSN 10, UndoNxt=Null</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>T1 End</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dirty Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageId</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>P5</td>
</tr>
<tr>
<td>P3</td>
</tr>
<tr>
<td>P1</td>
</tr>
</tbody>
</table>

Redo starts at LSN 10; in this case, reads P5, P3, and P1 from disk, redoes ops if `pageLSN < LSN`
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!

00  begin_checkpoint,
05  end_checkpoint
10  update: T1 writes P5; Prvl=null
20  update T2 writes P3; Prvl = null
30  T1 abort
40  CLR: Undo T1 LSN 10
45  T1 End
50  update: T3 writes P1; Prvl=null
60  update: T2 writes P5; Prvl=20

CRASH, RESTART

70  CLR: Undo T2 LSN 60; UndoNxtLSN=20
80  CLR: Undo T3 LSN 50; UndoNxtLSN=null
85  T3 end

CRASH, RESTART

90  CLR: Undo T2 LSN 20; UndoNxtLSN=null
100 T2 end

Aker Analysis/Redo:
ToUndo: 50 & 60
ToUndo: 50 & 20
ToUndo: 20

ToUndo: 70
ToUndo: 20
ToUndo: Finished!