Topics
- Motivation
- Logging
- Recovery
- Summary

Layers in a DBMS

These layers must consider concurrency control and recovery.
Review: The ACID properties

- **Atomicity**: All actions in the txn happen, or none happen.
- **Consistency**: If each txn is consistent, and the DB starts consistent, it ends up consistent.
- **Isolation**: Execution of one txn is isolated from that of other Xacts.
- **Durability**: If a txn commits, its effects persist.

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

Motivation

- **Atomicity**: Transactions may abort (“Rollback”)
- **Durability**: What if DBMS stops running?
- Desired Behavior after system restarts:
  - T1, T2 & T3 should be durable
  - T4 & T5 should be aborted (effects not seen)

Assumptions

- Concurrency control is in effect
  - Strict 2PL, in particular
- Updates are happening “in place”
  - i.e. data is overwritten on (deleted from) the disk
- A simple scheme to guarantee Atomicity & Durability?
Handling the Buffer Pool

- Force every write to disk?
  - Poor response time
  - But provides durability
- Steal buffer-pool frames from uncommitted txns?
  - If not, poor throughput
  - If so, how can we ensure atomicity?

<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

- To steal frame F:
  - Current page in F (say P) is written to disk
  - Some txn holds lock on P
    - What if the txn with the lock on P aborts?
      - Must remember the old value of P at steal time
      - To support UNDOing the write to page P
  - Stealing is why enforcing Atomicity is hard

More on Steal and Force: No Force

- What if system crashes before a modified page is written to disk?
  - Write as little as possible
    - In a convenient place
    - At commit time
    - To support REDOing modifications
  - No force is why enforcing durability is hard
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 (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, 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 the log record for an update before the corresponding data page gets to disk
  2. Must write all log records for a transaction before commit
- #1 guarantees Atomicity
- #2 guarantees Durability
- Exactly how is logging (and recovery!) done?
  - We’ll study the ARIES algorithms
WAL & the Log

- Each log record has a unique Log Sequence Number (LSN)
  - LSNs always increasing
- Each data page contains a pageLSN
  - The LSN of the most recent log record for an update to that page
- System keeps track of flushedLSN
  - The max LSN flushed so far
- WAL: Before a page is written,
  - pageLSN ≤ flushedLSN

Types of Log Records

- Commit
  - Force-written to the log when a txn commits
- Abort
  - Written to the log when a txn aborts
- End
  - Signifies end of commit or abort
  - Written after some more actions taken
    - E.g., modifying transaction table
- LogRecord fields:
  LSN  prevLSN  transID  type

Types of Log Records: Update

- Written after modifying a page
- pageLSN of the modified page is set to the LSN of the Update log record
- Contains:
  - Before-image: for undo
  - After-image: for redo

<table>
<thead>
<tr>
<th>LSN</th>
<th>prevLSN</th>
<th>transID</th>
<th>pageID</th>
<th>length</th>
<th>offset</th>
<th>before-image</th>
<th>after-image</th>
</tr>
</thead>
<tbody>
<tr>
<td>807</td>
<td>634</td>
<td>T1000</td>
<td>P500</td>
<td>3</td>
<td>21</td>
<td>ABC</td>
<td>DEF</td>
</tr>
</tbody>
</table>
Types of Log Records: Compensation (CLR)

- Written during txn rollback just before undoing action of an Update LR
- `undoNextLSN` is LSN of next LR to undo
  - Set to `prevLSN` in the Update LR just processed
- Contains:
  - before-image: for undo
  - Not needed: after-image

<table>
<thead>
<tr>
<th>LSN</th>
<th>prevLSN</th>
<th>transID</th>
<th>type</th>
<th>undoNextLSN</th>
<th>pageID</th>
<th>length</th>
<th>offset</th>
<th>before-image</th>
</tr>
</thead>
<tbody>
<tr>
<td>934</td>
<td>734</td>
<td>T1000</td>
<td>undo</td>
<td>634</td>
<td>5000</td>
<td>3</td>
<td>21</td>
<td>ABC</td>
</tr>
</tbody>
</table>

Other Log-Related State

- Transaction Table:
  - One entry per active txn
  - Contains transID, status (running/committed/aborted), and lastLSN
- Dirty Page Table:
  - One entry per dirty page in buffer pool
  - Contains recLSN -- the LSN of the log record which first caused the page to be dirty

Normal Execution of a Transaction

- Series of reads & writes, followed 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, in order to minimize the time taken to recover in the event of a system crash.
- Write to log:
  1. `begin_checkpoint` record:
     - Indicates when chkpt began
  2. `end_checkpoint` record:
     - Contains current `txn table` and `dirty page table`
     - This is a ‘fuzzy checkpoint’:
       - Other txns continue to run, so these tables 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**

<table>
<thead>
<tr>
<th>LOG</th>
<th>DB</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogRecords</td>
<td></td>
<td></td>
</tr>
<tr>
<td>prevLSN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>page0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>before-image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>after-image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data pages</td>
<td></td>
<td></td>
</tr>
<tr>
<td>each</td>
<td></td>
<td></td>
</tr>
<tr>
<td>with a pageSN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>master record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Txn Table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lastLSN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dirty Page Table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>recLSN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>flushedLSN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Simple Transaction Abort**
- For now, consider an explicit abort of a txn
  - No crash involved
- We want to “play back” the log in reverse order, UNDOing updates
  - Get lastLSN of txn from txn 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
  - No problem since 2PL is being followed
- Before restoring old value of a page, write a CLR:
  - You continue logging while you UNDO!
  - CLR has one extra field: undonextLSN
    - Points to the next LSN to undo (i.e. the prevLSN of the record we're currently undoing)
  - CLRs 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 txn’s lastLSN are flushed
  - 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

Topics

- Motivation
- Logging
- Recovery
- Summary
Crash Recovery: Big Picture

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

The Analysis Phase

- Performs 3 tasks:
  1. Determines the point in the log at which to start Redo pass
  2. Determines (a conservative set of) the pages in the buffer pool that were dirty at the point of the crash
  3. Identifies txns that were active at the time of the crash
- How? Scan log forward from checkpoint
  - End record: Remove txn from txn table
  - Other records: Add txn to txn table, set lastLSN=LSN, change txn status on commit/abort
  - Update record: If P not in Dirty Page Table,
    - Add P to D.P.T., set its recLSN=LSN

The REDO Phase

- We repeat History to reconstruct state at crash:
  - Reapply all updates (even of aborted txns!), redo CLR(s)
  - 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 action
    - Set pageLSN to LSN. No additional logging!
The UNDO Phase

ToUndo=\{ l | l a lastLSN of a “loser” txn\}

Repeat:
  \begin{itemize}
    \item Choose largest LSN among ToUndo
    \item If this LSN is a CLR and undonextLSN==NULL
      \hspace{1cm} Write an End record for this Xact
    \item If this LSN is a CLR, and undonextLSN != NULL
      \hspace{1cm} Add undonextLSN to ToUndo
    \item Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo
  \end{itemize}

Until ToUndo is empty

Example of Recovery

Example: Crash During Restart!
**Additional Crash Issues**

- What happens if system crashes during Analysis?
  - All the work done is lost
- What if system crashes during REDO?
  - Again do Analysis phase
  - During REDO may find that some changes had been written to disk, so those update actions don’t need to be done again
- How do you limit the amount of work in REDO?
  - Flush asynchronously in the background
- How do you limit the amount of work in UNDO?
  - Avoid long-running transactions
- Checkpoints can be taken during Restart to minimize repeated work in the event of yet another crash

**Topics**

- Motivation
- Logging
- Recovery
- Summary

**Summary of Logging/Recovery**

- Recovery Manager guarantees Atomicity & Durability
- Use WAL to allow STEAL/NO-FORCE without 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
Summary – contd.

- Recovery works in 3 phases:
  - Analysis: Forward from checkpoint
  - Redo: Forward from oldest recLSN
  - Undo: Backward from end to first LSN of oldest txn alive at crash
- Upon Undo, write CLRs
- Redo “repeats history”: Simplifies the logic!
- Note: If log files are archived, these can be used to recover a database after media failure which renders the datafiles unrecoverable
  - By using the current log file, archived log files and a previous backup of the datafiles
  - The current log file is generally multiplexed to guard against media failure