Logging and Recovery

If you are going to be in the logging business, one of the things that you have to do is to learn about heavy equipment.
- Robert VanNatta, Logging History of Columbia County

Motivation

- **Atomicity:**
  - Transactions may abort ("Rollback").
- **Durability:**
  - What if DBMS stops running? (Causes?)

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 DIRECT mode
  - i.e. data is overwritten on (deleted from) the disk.
- A simple scheme to guarantee Atomicity & Durability?

Handling the Buffer Pool

- **Force** write to disk at commit?
  - Poor response time (& throughput)
  - But provides durability.
- **Steal** buffer-pool frames from uncommitted Xacts?
  - If not, poor throughput.
  - Among other problems!
  - If so, how can we ensure atomicity?

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
  - A simple (physical) 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:
  - Must force the log record for an update *before* the corresponding data page gets to disk.
  - Must write all log records for a Xact *before* commit.
• #1 guarantees Atomicity.
• #2 guarantees Durability.
• Exactly how is logging (and recovery!) done?
  - We’ll study the ARIES protocol.

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: Cannot flush a data page unless pageLSN ≤ flushedLSN

Log Records

LogRecord fields:
- prevLSN
- XID
- type
- 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
  - (and some other tricks!)

Other Log-Related State

• Transaction Table:
  - One entry per active Xact.
  - Contains XID, 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 an Xact

• Series of reads & writes, followed by commit or abort.
  - We will assume that page write is atomic on disk.
• 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:
  - begin_checkpoint record: Indicates when chkpt began.
  - end_checkpoint record: Contains current Xact table and dirty page table. This is a fuzzy checkpoint:
    - Other Xacts continue to run; so these tables only known to reflect some mix of state after 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! Recall when we said that NO STEAL had other problems? This is one.)
  - Store LSN of chkpt record in a safe place (master record).
The Big Picture: What’s Stored Where

**DB**
- Data pages
  - each with a pageLSN
- Xact Table
  - lastLSN
  - status

**RAM**
- Dirty Page Table
  - recLSN
  - flushedLSN

---

Simple Transaction Abort

- 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.
  - Note: before starting UNDO, could write an Abort log record.
    - Why bother?

Abort, cont.

- To perform UNDO, must have a lock on data!
  - No problem!
- Before restoring old value of a page, write a CLR:
  - You continue logging while you UNDO!
  - CLR has one extra field: undoneXactLSN
    - Points to the next LSN to undo (i.e. the prevLSN of the record we’re currently undoing).
  - CLR contains REDO info
  - CLRs never Undone
    - Undo needn’t be idempotent (>1 UNDO won’t happen)
    - But they might be Redone when repeating history (=1 UNDO guaranteed)
- At end of all UNDOs, write an “end” log record.

Transaction Commit

- Write commit record to log.
- All log records up to Xact’s lastLSN are flushed.
  - Guarantees that flushedLSN ≥ lastLSN.
  - Note that log flushes are sequential, synchronous writes to disk.
  - Many log flushes per log page.
- Make transaction visible
  - Commit() returns, locks dropped, etc.
- Write end record to log.

Crash Recovery: Big Picture

- Start from a checkpoint (found via master record).
- Three phases. Need to:
  - Figure out which Xacts committed since checkpoint, which failed (Analysis).
  - REDO all actions.
    - Update record: If P not in Dirty Page Table, Add P to D.P.T., set its recLSN=LSN.
  - UNDO effects of failed Xacts.

Recovery: The Analysis Phase

- Reconstruct state at checkpoint.
  - via end_checkpoint record.
- Scan log forward from begin_checkpoint.
  - End record: Remove Xact from Xact table.
  - 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.
  - At end:
    - the Xact table contains all “loser” xacts
    - the DPT contains a superset of dirty data pages
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`. (requires an I/O)
- **To REDO** an action:
  - Reapply logged action.
  - Set `pageLSN` to `LSN`. No additional logging!

Recovery: The UNDO Phase

ToUndo = `{ l | 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
      - (note: we may have skipped a number of other CLRs!)
    - Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo.

Until ToUndo is empty.

Note: could do this one xact at a time; the above is a minor optimization that does them all at once.

Example of Recovery

```
LSN     LOG
00      begin_checkpoint
05      end_checkpoint
10      update: T1 writes P5
20      update T2 writes P3
30      T1 abort
40      CLR: Undo T1 LSN 10
45      T1 End
50      update: T3 writes P1
60      update: T2 writes P5
Xact Table
lastLSN
status
Dirty Page Table
reclSN
flushedLSN
ToUndo
```

Example: Crash During Restart!

```
LSN     LOG
00,05   begin_checkpoint, end_checkpoint
10      update: T1 writes P5
20      update T2 writes P3
30      T1 abort
40,45   CLR: Undo T1 LSN 10, T1 End
50      update: T3 writes P1
60      update: T2 writes P5
CRASH, RESTART
70      CLR: Undo T2 LSN 20, T2 end
80,85   CLR: Undo T3 LSN 50, T3 end
90      CLR: Undo T2 LSN 20, T2 end
```

Additional Crash Issues

- **What happens if system crashes during Analysis? During REDO?**
- **How do you limit the amount of work in REDO?**
  - Flush asynchronously in the background.
  - Watch "hot spots!"
- **How do you limit the amount of work in UNDO?**
  - Avoid long-running Xacts.

Logical vs. Physical Logging

- Roughly, ARIES does:
  - Physical REDO
  - Logical UNDO
- **Why?**
Logical vs. Physical Logging, Cont.

- **Page-oriented REDO logging**
  - Independence of REDO (e.g., indexes & tables)
  - Simplifies things enormously
  - Allows for optimizations (reordering/parallelization in Recovery)
  - Not quite physical, but close
  - Can have logical operations like increment/decrement (“escrow transactions”)
- **Logical UNDO**
  - To allow for simple management of physical structures that are invisible to users
  - To allow for logical operations

Nested Top Actions

- **Trick to support physical operations you do not want to ever be undone**
  - Example?
- **Basic idea**
  - At end of the nested actions, write a dummy CLR
    - Nothing to REDO in this CLR
    - Its UndoNextLSN points to the step before the nested action.

Summary of Logging/Recovery

- **Recovery Manager** guarantees Atomicity & Durability.
- Use 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.

Summary, Cont.

- **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!