CSE462/562: Database Systems (Spring 23)
Lecture 22: Crash Recovery
5/9/2023
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.

- Question: which ones does the **Recovery Manager** help with?

**Atomicity & Durability (and also used for Consistency-related rollbacks)**
Motivation for crash recovery

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

• Durability:
  • What if DBMS stops running? (Causes?)

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

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

• Updates are happening “in place”.
  • i.e. data are overwritten on (or deleted from) the actual pages.

• Can you think of a simple scheme (requiring no logging) to guarantee Atomicity & Durability?
  • What happens during normal execution (what is the minimum lock granularity)?
  • What happens when a transaction commits?
  • What happens when a transaction aborts?
Buffer manager plays a key role

- **Force policy** — make sure that every update is on disk before commit.
  - Provides durability without REDO logging.
  - But, can cause poor performance.

- **No Steal policy** — don’t allow buffer-pool frames with *uncommitted* updates to overwrite *committed* data on disk.
  - Useful for ensuring atomicity without UNDO logging.
  - But can cause poor performance.
Preferred buffer management policy: steal/no-force

• This combination is most complicated but allows for highest performance.
  • **NO FORCE**: do not have to flush all dirty pages of a transaction to disk before it commits
    • complicates Durability
    • What if system crashes before a modified page written by a committed transaction makes it to disk?
    • Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.
  • **STEAL**: allows buffer pool with uncommitted updates to overwrite committed data on disk
    • complicates Atomicity
    • What if the Xact that performed updates aborts?
    • What if system crashes before Xact is finished?
    • Must remember the old value of P (to support UNDOing the write to page P).
## Buffer management policies

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

- **No Force**: No Force
- **Force**: Force
- **No Steal**: No Steal
- **Steal**: Steal

### Performance Implications
- No UNDO
- No REDO

### Logging/Recovery Implications
- UNDO
- REDO
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 flush the log record for an update \textit{before} the corresponding data page gets to disk.
  ② Must flush all log records for a Xact \textit{before commit}
    • alternatively, transaction is not considered as committed until all of its log records including its “commit” record are on the stable log.

• #1 (with \textit{UNDO} info) helps provide Atomicity.
• #2 (with \textit{REDO} info) helps provide Durability.
• This allows us to employ Steal/No-Force policy

• Exactly how is logging (and recovery) done?
  • We’ll look at the ARIES algorithms.
    • Algorithms for Recovery and Isolation Exploiting Semantics

CSE462/562 (Spring 2023): Lecture 22
• Each log record has a unique Log Sequence Number (LSN).
  • LSNs are monotonically 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 page i is flushed to disk, the log must satisfy:
  \[ \text{pageLSN}_i \leq \text{flushedLSN} \]
prevLSN is the LSN of the previous log record written by this Xact (so records of an Xact form a linked list backwards in time)

Possible log record types:

- Update
- Checkpoint (for log maintenance)
- Compensation Log Records (CLRs)
  - for UNDO actions
- Commit/Abort
- End (indicates end of commit/abort)
Other logging-related state

• Two -in-memory tables

• Transaction Table
  • One entry per currently active Xact.
  • entry removed when Xact commits or aborts
  • Contains XID, status (running/committing/aborting), and lastLSN (most recent LSN written by Xact).

• Dirty Page Table:
  • One entry per dirty page currently in buffer pool.
  • Contains recLSN -- the LSN of the log record which first caused the page to be dirty.
    • If a dirty page is flushed to disk, it is removed from dirty page table
The big picture: what’s stored and where

LogRecords
- LSN
- 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

LOG

RAM
**Normal execution of an Xact**

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

• Write **commit** record to log.

• All log records up to Xact’s **commit record** are flushed to disk.
  - Guarantees that $\text{flushedLSN} \geq \text{lastLSN}$.
  - Note that log flushes are sequential, synchronous writes to disk.
  - Many log records per log page.

• Write an **end** record to log (no need to flush immediately)

• Commit() returns.

• When does a transaction becomes durable in the database?
  - When its commit log record is flushed to disk, even if there are still dirty pages in bufmgr.
Simple transaction abort

• For now, consider an explicit abort of a Xact.
  • No crash involved.

• First, set the transaction state in the transaction table to aborting.
  • Write an *Abort* log record before starting to rollback operations

• 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.
  • Write a “CLR” (compensation log record) for each undone operation.
    • more details on next slide
  • Once its finished, write a transaction end log record in the disk

• Q: do we need to wait for abort, CLR\(_s\) and end record to be flushed?
• To perform UNDO, must have a lock on data!
  • We still have the lock because of strict 2-PL.
• Before restoring old value of a page, write a CLR:
  • Must continue logging during undo in case of crash
  • CLR has one extra field: undonextLSN
    • Points to the next LSN to undo (i.e. the prevLSN of the record we’re currently undoing).
  • CLR contains REDO info
  • CLR is 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.
Checkpointing

• Conceptually, we keep log around for all time. Obviously this has performance issues...

• 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 accurate only as of the time of the begin_checkpoint record.
    • No attempt to force all dirty pages to disk; effectiveness of checkpoint limited by oldest unwritten change to a dirty page.
    • However, the more dirty page gets flushed, the shorter time will be needed in crash recovery
  • Store LSN of most recent chkpt record in a safe place (master record).
Start from a checkpoint (found via master record).

Three phases. Need to do:

- Analysis - Figure out which Xacts committed since checkpoint, which failed.
- REDO all actions. (repeat history)
- UNDO effects of failed Xacts.
Phase 1: the analysis phase

• Re-establish knowledge of state at checkpoint.
  • via transaction 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.
  • 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 says which xacts were active at time of crash.
  • DPT says which dirty pages *might not* have made it to disk
Phase 2: the redo phase

- We *Repeat History* to reconstruct state at crash:
  - Reapply *all* updates (including those of aborted Xacts), redo CLRs.

- Scan forward from log rec containing smallest recLSN in DPT. Q: why start here?

- 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 (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=\{\text{lastLSNs of all Xacts in the Trans Table}\}

i.e., last log entry of the aborted transactions

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. Undo the update, write a CLR, add prevLSN to ToUndo.

Until ToUndo is empty.
Example of recovery

<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>

CRASH, RESTART
Example: crash during recovery

<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</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>
<tr>
<td>70</td>
<td>CLR: Undo T2 LSN 60</td>
</tr>
<tr>
<td>80</td>
<td>CLR: Undo T3 LSN 50</td>
</tr>
<tr>
<td>85</td>
<td>T3 end</td>
</tr>
<tr>
<td>90,95</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

undonextLSN
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.
• What about schema changes/disk space management?
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.

- **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!