Crash Recovery

CMPSCI 645

Gerome Miklau

Slide content adapted from Ramakrishnan & Gehrke
Review: the ACID Properties

- Database systems ensure the ACID properties:
  - **Atomicity**: all operations of transaction reflected properly in database, or none are.
  - **Consistency**: each transaction in isolation keeps the database in a consistent state (this is the responsibility of the user).
  - **Isolation**: should be able to understand what’s going on by considering each separate transaction independently.
  - **Durability**: updates stay in the DBMS!!!
Types of failure

- **Transaction failure**
  - partially-executed transaction cannot commit
    - changes must be removed: ROLLBACK

- **System failure**
  - volatile memory lost
    - updates of committed Xact persist
    - updates of aborted or partial Xacts removed

- **Media failure**
  - corrupted storage media
    - database brought up-to-date using backup
Motivation

- Desired Behavior after system restarts:
  - T1, T2 & T3 should be **durable**.
  - T4 & T5 should be **aborted** (effects not seen).
Undo and Redo

- **UNDO:**
  - removing effects of incomplete or aborted transaction (for atomicity)

- **REDO:**
  - re-instating effects of committed transactions (for durability)

- The work the recovery subsystem must do to support UNDO and REDO depends on **key policies** of the buffer manager.
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.
Handling the Buffer Pool

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

![Diagram showing the relationship between Force, Steal, Recovery, and Desired outcomes.](image-url)
More on Steal and Force

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

- **NO FORCE** (why enforcing Durability is hard)
  - 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.
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:
    - Before image (for UNDO), After image (for REDO)
    - and additional control info (which we’ll see soon).

<XID, pageID, offset, length, old data, new data>
Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  1. Must force the log record for an update *before* the corresponding data page is overwritten on disk.
  2. 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 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
Log Records

LogRecord fields:

- prevLSN
- XID
- type
- pageID
- 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
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.
## Log and Transaction table

<table>
<thead>
<tr>
<th>pageID</th>
<th>recLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>p500</td>
<td></td>
</tr>
<tr>
<td>p600</td>
<td></td>
</tr>
<tr>
<td>p505</td>
<td></td>
</tr>
</tbody>
</table>

### Dirty page table

<table>
<thead>
<tr>
<th>transID</th>
<th>lastLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1000</td>
<td></td>
</tr>
<tr>
<td>T2000</td>
<td></td>
</tr>
</tbody>
</table>

### Transaction table

<table>
<thead>
<tr>
<th>prevLSN</th>
<th>transID</th>
<th>type</th>
<th>pageID</th>
<th>length</th>
<th>offset</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1000</td>
<td>update</td>
<td>p500</td>
<td>3</td>
<td>21</td>
<td>ABC</td>
<td>DEF</td>
</tr>
<tr>
<td></td>
<td>T2000</td>
<td>update</td>
<td>p600</td>
<td>3</td>
<td>41</td>
<td>HIJ</td>
<td>KLM</td>
</tr>
<tr>
<td></td>
<td>T2000</td>
<td>update</td>
<td>p500</td>
<td>3</td>
<td>20</td>
<td>GDE</td>
<td>QRS</td>
</tr>
<tr>
<td></td>
<td>T1000</td>
<td>update</td>
<td>p505</td>
<td>3</td>
<td>21</td>
<td>TUV</td>
<td>WXY</td>
</tr>
</tbody>
</table>
Normal Execution of a Xact

- 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.

- 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 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

LOG

LogRecords
prevLSN
XID
type
pageID
length
offset
before-image
after-image

DB
Data pages
each
with a
pageLSN
master record

RAM

Xact Table
lastLSN
status

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.
  - Before starting UNDO, write an *Abort* log record.
    - For recovering from crash during UNDO!
Abort, cont.

- 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).
  - CLR 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 Xact’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.
Crash Recovery: Big Picture

- Start from a checkpoint (found via master record).
- Three phases. Need to:
  - **Analysis**: Figure out which Xacts committed since checkpoint, which failed.
  - **REDO** all actions.
  - **UNDO** effects of failed Xacts.
Crash during recovery

- Crashes during UNDO handled by logging CLRs
- What happens if system crashes during Analysis or Redo?
  - Analysis: all work is lost, but analysis begins again.
  - Redo: Just redo again -- redo idempotent. Some pages may have been written to disk before crash but this will be evident.
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!
Additional resources

- Very good overview paper:
  - Michael Franklin “Concurrency & Recovery”