Implementing Atomicity and Durability
Chapter 22

System Malfunctions

• Transaction processing systems have to maintain correctness in spite of malfunctions
  – Crash
  – Abort
  – Media Failure

Failures: Crash

• Processor failure, software bug
  – Program behaves unpredictably, destroying contents of main (volatile) memory
  – Contents of mass store (non-volatile memory) generally unaffected
  – Active transactions interrupted, database left in inconsistent state
• Server supports atomicity by providing a recovery procedure to restore database to consistent state
  – Since rollforward is generally not feasible, recovery rolls active transactions back

Failures: Abort

• Causes:
  – User (e.g., cancel button)
  – Transaction (e.g., deferred constraint check)
  – System (e.g., deadlock, lack of resources)
• The technique used by the recovery procedure supports atomicity
  – Roll transaction back

Failures: Media

• Durability requires that database state produced by committed transactions be preserved
• Possibility of failure of mass store implies that database state must be stored redundantly (in some form) on independent non-volatile devices

Log

• Sequence of records (sequential file)
  – Modified by appending (no updating)
• Contains information from which database can be reconstructed
  – Read by routines that handle abort and crash recovery
• Log and database stored on different mass storage devices
• Often replicated to survive media failure
• Contains valuable historical data not in database
  – How did database reach current state?
Log

• Each modification of the database causes an update record to be appended to log
• Update record contains:
  – Identity of data item modified
  – Identity of transaction (tid) that did the modification
  – Before image (undo record) – copy of data item before update occurred
    • Referred to as physical logging

Transaction Abort Using Log

• Scan log backwards using tid to identify transaction’s update records
  – Reverse each update using before image
  – Reversal done in last-in-first-out order
• In a strict system new values unavailable to concurrent transactions (as a result of long term exclusive locks); hence rollback makes transaction atomic
• Problem: terminating scan (log can be long)
• Solution: append a begin record for each transaction, containing tid, prior to its first update record

Logging Savepoints

• Savepoint record inserted in log when savepoint created
  – Contains tid, savepoint identity
• Rollback Procedure:
  – Scan log backwards using tid to identify update records
  – Undo updates using before image
  – Terminate scan when appropriate savepoint record encountered

Crash Recovery Using Log

• Abort all transactions active at time of crash
• Problem: How do you identify them?
• Solution: abort record or commit record appended to log when transaction terminates
• Recovery Procedure:
  – Scan log backwards - if T’s first record is an update record, T was active at time of crash. Roll it back
  • A transaction is not committed until its commit record is in the log
Crash Recovery Using Log

<table>
<thead>
<tr>
<th>B</th>
<th>U</th>
<th>U</th>
<th>U</th>
<th>U</th>
<th>C</th>
<th>A</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>T₁</td>
<td>A</td>
<td>2.4</td>
<td>T₁</td>
<td>T₁</td>
<td>3</td>
<td>4.5</td>
</tr>
</tbody>
</table>

Key:
- B – begin record
- U – update record
- C – commit record
- A – abort record

- T₁ and T₃ were not active at time of crash

Example

<table>
<thead>
<tr>
<th>B2</th>
<th>B3</th>
<th>U2</th>
<th>U1</th>
<th>C2</th>
<th>B5</th>
<th>U3</th>
<th>U5</th>
<th>A5</th>
<th>CK</th>
<th>U4</th>
<th>U6</th>
<th>C4</th>
<th>U1</th>
</tr>
</thead>
</table>

Key:
- B – begin record
- U – update record
- C – commit record
- CK – checkpoint record

- T₃, T₁, and T₄ active at time of crash

Write-Ahead Log

- Problem: Scan must retrace entire log
- Solution: Periodically append checkpoint record to log. Contains tid’s of all active transactions at time of append
  - Backward scan goes at least as far as last checkpoint record appended
  - Transactions active at time of crash determined from log suffix that includes last checkpoint record
  - Scan continues until those transactions have been rolled back

Write-Ahead Log: Performance

- Problem: two I/O operations for each database update
- Solution: log buffer in main memory
  - Extension of log on mass store
  - Periodically flushed to mass store
  - Flush cost pro-rated over multiple log appends
  - This effectively reduces the cost to one I/O operation for each database update

Performance

- Problem: one I/O operation for each database update
- Solution: database page cache in main memory
  - Page is unit of transfer
  - Page containing requested item is brought to cache; then a copy of the item is transferred to application
  - Retain page in cache for future use
  - Check cache for requested item before doing I/O (I/O can be avoided)
Page and Log Buffering

Cache Management

- Cache pages that have been updated are marked dirty; others are clean
- Cache ultimately fills
  - Clean pages can simply be overwritten
  - Dirty pages must be written to database before page frame can be reused

Atomicity, Durability and Buffering

- **Problem**: page and log buffers are volatile
  - Their use affects the time data becomes non-volatile
  - Complicates algorithms for atomicity and durability
- **Requirements**:
  - Write-ahead feature (move update records to log on mass store before database is updated) necessary to preserve atomicity
  - New values written by a transaction must be on mass store when its commit record is written to log (move new values to mass store before commit record) to preserve durability
  - Transaction not committed until commit record in log on mass store
- **Solution**: requires new mechanisms

New Mechanism 1

- **Forced vs. Unforced Writes**: 
  - **On database page**
    - Unforced write updates cache page, marks it dirty and returns control immediately.
    - Forced write updates cache page, marks it dirty, uses it to update database page on disk, and returns control when I/O completes.
  - **On log**
    - Unforced append adds record to log buffer and returns control immediately.
    - Forced append, adds record to log buffer, writes buffer to log, and returns control when I/O completes.

New Mechanism 2

- **Log Sequence Number (LSN)**:
  - Log records are numbered sequentially
  - Each database page contains the LSN of the update record describing the most recent update of any item in the page

Preserving Atomicity: the Write-Ahead Property and Buffering

- **Problem 1**: When the cache page replacement algorithm decides to write a dirty page, \( p \), to mass store, an update record corresponding to \( p \) might still be in the log buffer.
- **Solution**: Force the log buffer if the LSN stored in \( p \) is greater than or equal to the LSN of the oldest record in the log buffer. Then write \( p \). This preserves write-ahead policy.
Preserving Durability I

- **Problem 2:** Pages updated by T might still be in cache when T’s commit record is appended to log buffer.
  - Once commit record is in log buffer, it may be flushed to log at any time, causing a violation of durability.
- **Solution:** Force the (dirty) pages in the cache that have been updated by T before appending T’s commit record to log buffer (force policy).

Force Policy for Commit Processing

1. **Force** any update records of T in log buffer then …
2. **Force** any dirty pages updated by T in cache then …
   - (1) and (2) ensure **atomicity** (write-ahead policy)
3. Append T’s commit record to log buffer then …
   - **Force** log buffer for immediate commit or …
   - **Write** log buffer when a group of transactions have committed (group commit)
   - (2) and (3) ensure **durability**

Force Policy

- **Advantage:**
  - Transaction’s updates are in database (on mass store) when it commits.
- **Disadvantages:**
  - Commit must wait until dirty cache pages are forced
  - Pages containing items that are updated by many transactions (hotspots) have to be forced with the commit of each such transaction …
    - but an LRU page replacement algorithm would not write such a page out

Preserving Durability II

- **Problem 2:** Pages updated by T might still be in cache when T’s commit record is appended to log buffer
- **Solution:** Update record contains after image (called a redo record) as well as before image
  - Write-ahead property still requires that update record be written to mass store before page
  - But it is no longer necessary to force dirty pages when commit record is written to log on mass store since all after images precede commit record in log
  - Referred to as a **no-force** policy

No-Force Commit Processing

- Append T’s commit record to log buffer
  - **Force** buffer for immediate commit
  - T’s update records precede its commit record in buffer ensuring updates are durable before (or at the same time as) it commits
- T’s dirty pages can be flushed from cache at any time after update records have been written
  - Necessary for write-ahead policy
- T’s dirty pages can be written before or after commit record
No Force Policy for Commit Processing

No-Force Policy

- **Advantages:**
  - Commit doesn’t wait until dirty pages are forced
  - Pages with hotspots don’t have to be written out
- **Disadvantage:**
  - Crash recovery complicated: some updates of committed transactions (contained in redo records) might not be in database on restart after crash
  - Update records are larger

Recovery With No-Force Policy

- **Problem:** When a crash occurs there might exist some pages in database (on mass store)
  - containing updates of uncommitted transaction: they must be rolled back
  - that do not (but should) contain the updates of committed transactions: they must be rolled forward
- **Solution:** Use a *sharp checkpoint*

Sharp Checkpoint

- **Problem:** How far back must log be scanned in order to find update records of committed transactions that must be rolled forward?
- **Solution:** Before appending a checkpoint record, CK, to log buffer, halt processing and force all dirty pages from cache
  - Recovery process can assume that all updates in records prior to CK were written to database (only updates in records after CK might not be in database)

Recovery with Sharp Checkpoint

- **Pass 1:** Log is scanned backward to most recent checkpoint record, CK, to identify transactions active at time of crash.
- **Pass 2:** Log is scanned forward from CK to most recent record. The *after images* in all update records are used to *roll the database forward.*
- **Pass 3:** Log is scanned backwards to begin record of oldest transaction active at time of crash. The *before images* in the update records of these transactions are used to *roll these transactions back.*
Recovery with Sharp Checkpoint

- **Issue 1**: Database pages containing items updated after CK was appended to log might have been flushed before crash
  - No problem – with physical logging, roll forward using after images in pass 2 is idempotent.
  - Rollforward in this case is unnecessary, but not harmful

- **Issue 2**: Some update records after CK might belong to an aborted transaction, T₁. These updates will not be rolled back in pass 3 since T₁ was not active at time of crash
  - Treat rollback operations for aborting T₁ as ordinary updates and append compensating log records to log

- **Issue 3**: What if system crashes during recovery?
  - Recovery is restarted
  - If physical logging is used, pass 2 and pass 3 operations are idempotent and hence can be redone

Fuzzy Checkpoints

- **Problem**: Cannot stop the system to take sharp checkpoint (write dirty pages).
  - Use fuzzy checkpoint: Before writing CK, record the identity of all dirty pages (do not flush them) in volatile memory
  - All recorded pages must be flushed before next checkpoint record is appended to log buffer

Archiving the Log

- **Problem**: What do you do when the log fills mass store?
  - Initial portions of log are not generally discarded since they contain important data:
    - Record of how database got to its current state
    - Information for analyzing performance
  - **Solution**: Archive the initial portion of the log on tertiary storage. Only the portion of the log containing records of active transactions needs to be maintained on secondary store
Logical Logging

- **Problem:** With physical logging, simple database updates can result in multiple update records with large before and after images
  - *Example:* “insert t in T” might cause reorganization of a data page and an index page for each index. Before and after images might be entire pages
- **Solution:** Log the operation and its inverse instead of before and after images
  - *Example:* store “insert t in T”, “delete t from T” in update record

Logical Logging

- **Problem 1:** Logical operations might not be idempotent (e.g., “UPDATE T SET x=x+5”)
  - Pass 2 roll forward does not work (it makes a difference whether the page on mass store was updated before the crash or after the crash)
- **Solution:** Do not apply operation in update record i to database item in page P during pass 2 if P.LSN ≥ i

Logical Logging

- **Problem 2:** Operations are not atomic
  - A crash during the execution of a non-atomic operation can leave the database in a physically inconsistent state
    - *Example:* “insert t in T” requires an update to both a data and an index page. A crash might occur after t has been inserted in T but before the index has been updated
  - Applying a logical redo operation in pass 2 to a physically inconsistent state is not likely to work
    - *Example:* There might be two copies of t in T after pass 2

Physiological Logging

- **Solution:** Use physical-to-a-page, logical-within-a-page logging (physiological logging)
  - A logical operation involving multiple pages is broken into multiple logical mini-operations
    - Each mini-operation is confined to a single page and hence is atomic
      - *Example:* “insert t in T” becomes “insert t in a page of T” and “insert pointer to t in a page of index”
  - Each mini-operation gets a separate log record
  - Since mini-operations are not idempotent, use LSN check before applying operation in pass 2

Deferred-Update System

- **Update** - append new value to intentions list (in volatile memory); append update record (containing only after image) to log buffer;
  - write-ahead property does not apply since there is no before image
- **Abort** - discard intentions list
- **Commit** - force commit record to log; initiate database update using intentions list
- **Completion of intentions list processing** - write completion record to log

Recovery in Deferred-Update System

- **Checkpoint record** - contains list of committed (not active) but incomplete transactions
- **Recovery** -
  - Scan back to most recent checkpoint record to determine transactions that are committed but for which updates are incomplete at time of crash
  - Scan forward to install after images for incomplete transactions
  - No third pass required since transactions active (not committed) at time of crash have not affected database
**Media Failure**

- Durability requires that the database be stored redundantly on distinct mass storage devices
  1. Redundant copy on (mirrored) disk ⇒ high availability
  2. Redundant data in log
- **Problem**: Using the log (as in 2 above) to reconstruct the database is impractical since it requires a scan starting at first record
- **Solution**: Use log together with a periodic dump

**Simple Dump**

- **Simple dump**
  - System stops accepting new transactions
  - Wait until all active transactions complete
  - Dump: copy entire database to a file on mass storage
  - Restart log and system

**Restoring Database From Simple Dump**

- Install most recent dump file
- Scan backward through log
  - Determine transactions that committed since dump was taken
    - Ignore aborted transactions and those that were active when media failed
- Scan forward through log
  - Install after images of committed transactions

**Fuzzy Dump**

- **Problem**: The system cannot be shut down to take a simple dump
- **Solution**: Use a fuzzy dump
  - Write begin dump record to log
  - Copy database records to dump file while system active
    - Even copying records of active transactions and records that are locked

**Fuzzy Dump**

- Dump file might:
  - reflect incomplete execution of an active transaction that later commits
    \[ w_f(x) \quad dump(x) \quad dump(y) \quad w_f(y) \quad commit_t \quad time \]
  - reflect updates of an active transaction that later aborts
    \[ w_f(x) \quad dump(x) \quad abort_t \quad time \]

**Naïve Restoration Using Fuzzy Dump**

- Install dump on disk
- Scan log backwards to begin dump record to produce list, L, of all transactions that committed since start of dump
- Scan log forward and install after images in update records of all transactions in L
Naïve Restoration Using Fuzzy Dump

- It does some things correctly

- Problem: Naïve algorithm does not handle two cases:
  - T commits before dump starts but its dirty pages might not have been flushed until dump completed
  - Dump does not read T’s updates and T is not in L.
  - Dump reads T’s updates but T later aborts:

Taking a Fuzzy Dump

- Solution: Use fuzzy checkpointing and compensating log records
- Dump algorithm:
  - Write checkpoint record
  - Write begin dump record (BD)
  - Dump
  - Write end dump record (ED)

Restoration Using Fuzzy Dump

- Install dump on mass storage device
- Scan backward to CK3 to produce list, L, of all transactions active at time of media failure
- Scan forward from CK1; use redo records to roll the database forward to its state at time of media failure
- Scan backwards to begin record of oldest transaction in L, roll all transactions in L back

<table>
<thead>
<tr>
<th>CK1</th>
<th>CK2</th>
<th>BD</th>
<th>ED</th>
<th>CK3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

all dirty pages in cache at time of CK1 have been written to database

media failure