Database System Recovery

CSE593 Transaction Processing
Philip A. Bernstein
Copyright ©2001 Philip A. Bernstein

Outline

1. Introduction
2. Recovery Manager
3. Two Non-Logging Algorithms
4. Log-based Recovery
5. Media Failure

1. Introduction

• A database may become inconsistent because of a
  – transaction failure (abort)
  – database system failure (possibly caused by OS crash)
  – media crash (disk-resident data is corrupted)
• The recovery system ensures the database contains
  exactly those updates produced by committed transactions
  – I.e. atomicity and durability, despite failures

Assumptions

• Two-phase locking, holding write locks until after
  a transaction commits. This implies
  – recoverability
  – no cascading aborts
  – strictness (never overwrite uncommitted data)
• Page-level everything (for now)
  – page-granularity locks
  – database is a set of pages
  – a transaction’s read or write operation operates on an
    entire page
  – we’ll look at record granularity later

Storage Model

• Stable database - survives system failures
• Cache (volatile) - contains copies of some pages, which are lost by a system failure

Stable Storage

• Write(P) overwrites the entire contents of P on the disk
• If Write is unsuccessful, the error might be detected on the next read ...
  – e.g. page checksum error => page is corrupted
• … or maybe not
  – Write correctly wrote to the wrong location
• Write is the only operation that’s atomic with respect to failures and whose successful execution can be determined by recovery procedures.
The Cache

- Cache is divided into page-sized slots.
- Each slot’s dirty bit tells if the page was updated since it was last written to disk.
- Pin count tells number of pin ops without unpins

<table>
<thead>
<tr>
<th>Page</th>
<th>Dirty Bit</th>
<th>Cache Address</th>
<th>Pin Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>1</td>
<td>91976</td>
<td>1</td>
</tr>
<tr>
<td>P47</td>
<td>0</td>
<td>812</td>
<td>2</td>
</tr>
<tr>
<td>P21</td>
<td>1</td>
<td>10101</td>
<td>0</td>
</tr>
</tbody>
</table>

- Fetch(P) - read P into a cache slot. Return slot address.
- Flush(P) - If P’s slot is dirty and unpinned, then write it to disk (i.e. return after the disk acks)
- Pin(P) - make P’s slot unflushable. Unpin releases it.
- Deallocate - allow P’s slot to be reused (even if dirty)

The Log

- A sequential file of records describing updates:
  - address of updated page
  - id of transaction that did the update
  - before-image and after-image of the page
- Whenever you update the cache, also update the log
- Log records for Commit(T) and Abort(T)
- Some older systems separated before-images and after-images into separate log files.
- If op2 conflicts with and executes before op1, then op1’s log record must precede op2’s log record.
  “recovery will replay operations in log record order

The Log (cont’d)

- With record granularity operations, short-term locks, called latches, control concurrent record updates to the same page:
  - Fetch(P) read P into cache
  - Pin(P) ensure P isn’t flushed
  - write lock (P) for two-phase locking
  - latch P get exclusive access to P
  - update P update P in cache
  - log the update to P append it to the log
  - unlatch P release exclusive access
  - Unpin(P) allow P to be flushed
  - There’s no deadlock detection for latches.

2. Recovery Manager

- Processes Commit, Abort and Restart
- Commit(T)
  - Write T’s updated pages to stable storage atomically, even if the system crashes.
- Abort(T)
  - Undo the effects of T’s writes
- Restart = recover from system failure
  - Abort all transactions that were not committed at the time of the previous failure
  - Fix stable storage so it includes all committed writes and no uncommitted ones (so it can be read by new txns)

Recovery Manager Model
Implementing Abort(T)
- Suppose T wrote page P.
- If P was not transferred to stable storage, then deallocate its cache slot
- If it was transferred, then P’s before-image must be in stable storage (else you couldn’t undo after a system failure)
- Undo Rule - Do not flush an uncommitted update of P until P’s before-image is stable. (Ensures undo is possible.)
  - Write-Ahead Log Protocol - Do not … until P’s before-image is in the log

Avoiding Undo
- Avoid the problem implied by the Undo Rule by never flushing uncommitted updates.
  -Avoids stable logging of before-images
  -Don’t need to undo updates after a system failure
- A recovery algorithm requires undo if an update of an uncommitted transaction can be flushed.
  -Usually called a steal algorithm, because it allows a dirty cache page to be “stolen.”

Implementing Commit(T)
- Commit must be atomic. So it must be implemented by a disk write.
- Suppose T wrote P, T committed, and then the system fails. P must be in stable storage.
- Redo rule - Don’t commit a transaction until the after-images of all pages it wrote are on stable storage (in the database or log). (Ensures redo is possible.)
  - Often called the Force-At-Commit rule

Avoiding Redo
- To avoid redo, flush all of T’s updates to the stable database before it commits. (They must be in stable storage.)
  - Usually called a Force algorithm, because updates are forced to disk before commit.
  - It’s easy, because you don’t need stable bookkeeping of after-images
  - But it’s inefficient for hot pages. (Consider TPC-A/B.)
- Conversely, a recovery algorithm requires redo if a transaction may commit before all of its updates are in the stable database.

Avoiding Undo and Redo?
- To avoid both undo and redo
  - never flush uncommitted updates (to avoid undo), and
  - flush all of T’s updates to the stable database before it commits (to avoid redo).
- Thus, it requires installing all of a transaction’s updates into the stable database in one write to disk
- It can be done, but it isn’t efficient for short transactions and record-level updates.
  - We’ll show how in a moment

Implementing Restart
- To recover from a system failure
  - Abort transactions that were active at the failure
  - For every committed transaction, redo updates that are in the log but not the stable database
  - Resume normal processing of transactions
- Idempotent operation - many executions of the operation have the same effect as one execution
- Restart must be idempotent. If it’s interrupted by a failure, then it re-executes from the beginning.
- Restart contributes to unavailability. So make it fast!
3. Two Non-Logging Algorithms

- First, we’ll look at a Force (= no-redo) algorithm [Rappaport, SIGMOD ’75 Conference]
- Use multiversions, as in the locking chapter
  - with one change (or clarification): there may be unlocked dirty versions (from aborted transactions) on disk.
- All transactions read the commit list
  - to read a version, a transaction checks that it was written by a committed transaction (or itself)

- Instant recovery since there’s no Restart algorithm!
- Garbage collect versions of aborted transactions in the background.

Shadow Paging

- Each file is managed via a page table P
  - Each transaction T updates the file via a private page table
  - Commit T by replacing the public page table by a private one
  - Example: suppose DB has two files, “a” and “b”

Shadow Paging in Practice

- Used in the Gemstone OO DBMS.
- Not good for TPC
  - count disk updates per transaction
  - how to do record level locking?

4. Log-based Recovery

- Logging is the most popular mechanism for implementing recovery algorithms.
- The recovery manager implements
  - Commit - by writing a commit record to the log and flushing the log (satisfies the Redo Rule)
  - Abort - by using the transaction’s log records to restore before-images
  - Restart - by scanning the log and undoing and redoing operations as necessary
- The algorithms are fast since they use sequential log I/O in place of random database I/O. They greatly affect TP and Restart performance.
Implementing Commit

- Every commit requires a log flush.
- If you can do $K$ log flushes per second, then $K$ is your maximum transaction throughput
- Group Commit Optimization - when processing commit, if the last log page isn’t full, delay the flush to give it time to fill
- If there are multiple data managers on a system, then each data mgr must flush its log to commit
  - If each data mgr isn’t using its log’s update bandwidth, then a shared log saves log flushes
  - A good idea, but rarely supported commercially

Implementing Abort

- To implement Abort($T$), scan $T$’s log records and install before images.
- To speed up Abort, back-chain each transaction’s update records.

Transaction Descriptors

<table>
<thead>
<tr>
<th>Transaction</th>
<th>last log record</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_7$</td>
<td>$P_m$ backpointer</td>
</tr>
<tr>
<td>$T_i$</td>
<td>$P_k$ null pointer</td>
</tr>
</tbody>
</table>

Group Commit Optimization - when processing commit, if the last log page isn’t full, delay the flush to give it time to fill

Satisfying the Undo Rule

- To implement the Write-Ahead Log Protocol, tag each cache slot with the log sequence number (LSN) of the last update record to that slot’s page.

<table>
<thead>
<tr>
<th>Page</th>
<th>Dirty Bit</th>
<th>Cache Address</th>
<th>Pin Count</th>
<th>LSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{77}$</td>
<td>1</td>
<td>812</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$P_{31}$</td>
<td>1</td>
<td>10101</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- Cache manager won’t flush a page $P$ until $P$’s last updated record, pointed to by LSN, is on disk.
- $P$’s last log record is usually stable before Flush($P$), so this rarely costs an extra flush
- LSN must be updated while latch is held on $P$’s slot

Checkpoints

- Problem - Prevent Restart from scanning back to the start of the log
- A checkpoint is a procedure to limit the amount of work for Restart
- Commit-consistent checkpointing
  - Stop accepting new update, commit, and abort operations
  - Make list of [active transaction, pointer to last log record]
  - Flush all dirty pages
  - Append a checkpoint record to log; include the list
  - Resume normal processing
- Database and log are now mutually consistent

Restart Algorithm (rev 2)

- No need to redo records before last checkpoint, so
  - Starting with the last checkpoint, scan forward in the log.
  - Redo all update records. Process all aborts.
  - Maintain list of active transactions (initialized to content of checkpoint record).
  - After you’re done scanning, abort all active transactions
- Restart time is proportional to the amount of log after the last checkpoint.
- Reduce restart time by checkpointing frequently.
- Thus, checkpointing must be cheap.
Fuzzy Checkpointing

- Make checkpoints cheap by avoiding synchronized flushing of dirty cache at checkpoint time.
  - Stop accepting new update, commit, and abort operations
  - Make a list of all dirty pages in cache
  - Make list of [active transaction, pointer to last log record]
  - Append a checkpoint record to log: include the list
  - Resume normal processing
  - Initiate low priority flush of all dirty pages
- Don’t checkpoint again until all of the last checkpoint’s dirty pages are flushed
- Restart begins at second-to-last (penultimate) checkpoint.
- Checkpoint frequency depends on disk bandwidth

Operation Logging

- Record locking requires (at least) record logging.
  - Suppose records x and y are on page P
  - w1[x] w2[y] abort1 commit2 (not strict w.r.t. pages)
- But record logging implies Restart must read a page before updating it
  - Only log after-image of an insertion
  - Only log fields being updated
- Now Restart can’t blindly redo.
  - E.g., it must not insert a record twice

LSN-based logging

- Each database page P’s header has the LSN of the last log record whose operation updated P.
- Restart compares log record and page LSN before redoing the log record’s update U.
  - Redo the update only if LSN(P) < LSN(U)
- Undo is a problem. If U’s transaction aborts and you undo U, what LSN to put on the page?
  - Suppose T1 and T2 update records x and y on P
  - w1[x] w2[y] c2 a1 (what LSN does a2 put on P?)
  - not LSN before w1[x] (which says w2[x] didn’t run)
  - not w2[y] (which says w1[x] wasn’t aborted)
  - not a1 (have to latch all of T1’s pages while aborting)

Logging Undo’s

- Log the undo(U) operation, and use its LSN on P
  - CLR = Compensation Log Record = a logged undo
  - Do this for all undo’s (during normal abort or recovery)
- This preserves the invariant that the LSN on each page P exactly describes P’s state relative to the log.
  - P contains all updates to P up to and including the LSN on P, and no updates with larger LSN.
- So every aborted transaction’s log is a palindrome of update records and undo records.
- Restart processes Commit and Abort the same way
  - It redoes the transaction’s log records.
  - It only aborts active transactions after the forward scan

Logging Undo’s (cont’d)

- Tricky issues
  - Multi-page updates (it’s best to avoid them)
  - Restart grows the log by logging undos.
    Each time it crashes, it has more log to process
- Optimization - CLR points to the transaction’s log record preceding the corresponding “do”.
  - Splices out undone work
  - Avoids undoing undone work during abort
  - Avoids growing the log due to aborts during Restart

Restart Algorithm (rev 3)

- Starting with the last checkpoint, scan forward in the log.
  - Maintain list of active transactions (initialized to content of checkpoint record).
  - Redo an update record U for page P only if LSN(P) < LSN(U).
  - After you’re done scanning, abort all active transactions.
    Log undos while aborting. Log an abort record when you’re done aborting.
- This style of record logging, logging undo’s, and replaying history during restart was popularized in the ARIES algorithm by Mohan et al at IBM.
Analysis Pass

• Log flush record after a flush occurs (to avoid redo)
• To improve redo efficiency, pre-analyze the log
  – Requires accessing only the log, not the database
• Build a Dirty Page Table that contains list of dirty pages and oldestLSN that must be redone
  – Flush(P) says to delete P from Dirty Page Table
  – Include Dirty Page Table in checkpoint records
  – Start at checkpoint record, scan forward building the table
• Also build list of active txns with lastLSN

Analysis Pass (cont’d)

• Start redo at oldest oldestLSN in Dirty Page Table
  – Then scan forward in the log, as usual
  – Only redo records that might need it, that is, LSN(redo) ≥ oldestLSN, hence no later flush record
• Also use Dirty Page Table to guide page prefetching
  • Prefetch pages in oldestLSN order in Dirty Page Table

Logging B-Tree Operations

• To split a page
  – log records deleted from the first page (for undo)
  – log records inserted to the second page (for redo)
  – they’re the same records, so long them once!
• This doubles the amount of log used for inserts
  – log the inserted data when the record is first inserted
  – if a page has N records, log N/2 records, every time a page is split, which occurs once for every N/2 insertions

Shared Disk System

• Can cache a page in two processes that write-lock different records
• Only one process at a time can have write privilege
• Use a global lock manager
• When setting a write lock on P, may need to refresh the cached copy from disk (if another process recently updated it)
• Use version number on the page and in the lock
• When setting the lock, a process tells the lock manager its cached version number. After update, flush the page to server and increment version number in the lock.

User-level Optimizations

• If checkpoint frequency is controllable, then run some experiments
• Partition DB across more disks to reduce restart time (if Restart is multithreaded)
• Increase resources (e.g. cache) available to restart program.

5. Media Failures

• A media failure is the loss of some of stable storage.
• Most disks have MTBF over 10 years
• Still, if you have 10 disks ...
• So shadowed disks are important
  – Writes go to both copies. Handshake between Writes to avoid common failure modes (e.g. power failure)
  – Service each read from one copy
• To bring up a new shadow
  – Copy tracks from good disk to new disk, one at a time
  – A Write goes to both disks if the track has been copied
  – A read goes to the good disk, until the track is copied
RAID

- RAID - redundant array of inexpensive disks
  - Use an array of N disks in parallel
  - A stripe is an array of the 8th block from each disk
  - A stripe is partitioned as follows:
    ![Diagram of RAID stripes]
    - M data blocks
    - N-M error correction blocks
  - Each stripe is one logical block, which can survive a single-disk failure.

Where to Use Disk Redundancy?

- Preferable for both the DB and log
- But at least for the log
  - In an undo algorithm, it’s the only place that has certain before images
  - In a redo algorithm, it’s the only place that has certain after images
- If you don’t shadow the log, it’s a single point of failure

Archiving

- An archive is a database snapshot used for media recovery.
  - Load the archive and redo the log
- To take an archive snapshot
  - write a start-archive record to the log
  - copy the DB to an archive medium
  - write an end-archive record to the log
    (or simply mark the archive as complete)
- So, the end-archive record says that all updates before the start-archive record are in the archive
- Can use the standard LSN-based Restart algorithm to recover an archive copy relative to the log.

Archiving (cont’d)

- To archive the log, use 2 pairs of shadowed disks. Dump one pair to archive (e.g. tape) while using the other pair for on-line logging. (I.e. ping-pong to avoid disk contention)
  - Optimization - only archive committed pages and purge undo information from log before archiving
- To do incremental archive, use an archive bit in each page.
  - Each page update sets the bit.
  - Archive only copies pages with the bit set, and clears it.
- To reduce media recovery time
  - rebuild archive from incremental copies
  - partition log to enable fast recovery of a few corrupted pages