

CMSC 724: Recovery

Amol Deshpande

University of Maryland, College Park

March 6, 2007

*Adapted from Joe Hellerstein's Notes (<http://redbook.cs.berkeley.edu/redbook3/lec4.html>)

Recovery

- To guarantee Atomicity and Durability
 - Abort/Rollbacks, System Crashes etc..
 - Reasons for crashes
 - Transaction failures: logical errors, deadlocks
 - System crash: power failures, operating system bugs etc
 - Disk failure: head crashes
 - We will assume *STABLE STORAGE* for now
 - Data is not lost
 - Typically ensured through redundancy (e.g. RAID)

Recovery

- STEAL:
 - The buffer manager can steal a memory page for replacement purposes
 - The page might contain dirty writes
- FORCE:
 - Before committing a transaction, force its updates to disk
- Easiest option: NO STEAL, FORCE
 - NO STEAL, so atomicity easier to guarantee
 - No serious durability issues because of FORCE
 - Issues:
 - How to force all updates to disk atomically ? Can use shadow copying.
 - A page might contain updates of two transactions ? Can use page level locking etc. . .

Recovery

- Desired option: STEAL, NO FORCE
- STEAL:
 - Dirty data might be written on disk
 - Need to use UNDO logs so we can rollback that action
 - The UNDO log records must be on disk before the page can be written (Write-Ahead Logging)
- NO FORCE:
 - Data from committed transaction might not make it to disk
 - Use REDO logs
 - The REDO log records must make it disk before the transaction is “committed”

Simple Log-based Recovery

- Each action generates a *log* record (before/after copies)
- Write Ahead Logging: Log records make it to disk before corresponding data page
- *Strict* Two-Phase Locking
 - Locks held till the end
 - Once a lock is released, not possible to undo
- Normal Processing: UNDO (rollback)
 - Go backwards in the log, and restore the updates
 - Locks are already there, so not a problem
- Normal Processing: Checkpoints
 - Halt the processing
 - Dump dirty pages to disk
 - Log: (*checkpoint list-of-active-transactions*)

Simple Log-based Recovery: Restart

- Analysis:
 - Go back into the log till the checkpoint
 - Create *undo-list*: $(T_i, Start)$ after the checkpoint but no (T_i, End)
 - Create *redo-list*: (T_i, End) after the checkpoint
- **Undo before Redo:**
 - Undo all transactions on the undo-list one by one
 - Redo all transactions on the undo-list one by one
 - E.g.
 $(T_1, A, 10, 20), (T_1, Abort), (T_2, A, 10, 30), (T_2, commit)$
 - Must do UNDO before REDO
 - This is because no CLR's

ARIES

- *Log-based Recovery*
 - Every database action is logged
 - Even actions performed during *undo* (also called *rollback*) are logged
- Log records:
 - (LSN, Type, TransID, PrevLSN, PageID, UndoNextLSN (CLR Only), Data)
 - LSN = Log Sequence Number
 - Type = Update | Compensation Log Record | Commit related | Non-transaction related (OS stuff)
 - Allows logical logging
 - More compact, Allows higher concurrency (*indexes*)

ARIES: Logs

- Physical Undos or Redos (also called page-oriented)
 - Store before and after copies
 - Easier to manage, apply - no need to touch any other pages
 - Requires stricter locking behaviour
- Logical Undos
 - More compact, allow higher concurrency
 - May not be idempotent: Shouldn't undo twice
- CLRs
 - Redo-only; Typically generated during abort/rollback
 - Contain an UndoNextLSN - can skip already undone records.
- ARIES does “Physiological” logging
 - Physical REDO: Page oriented redo recovery
 - Supports logical UNDO, but allows physical UNDO also

ARIES: Other Data Structures

- With each page:
 - page_LSN: LSN of last log record that updated the page
- Dirty pages table: (*PageID*, *RecLSN*)
 - RecLSN (recovery LSN): Updates made by log records before RecLSN are definitely on disk
 - Min(RecLSN of all dirty pages) → where the REDO Pass starts
- Transaction Table: (*TransID*, *State*, *LastLSN*, *UndoNxtLSN*)
 - State: Commit state
 - UndoNxtLSN: Next record to be processed during rollback

ARIES: Assumptions/Setup

- STEAL, NO FORCE
- In-place updating
- Write-ahead Logging (WAL)
 - Log records go to the stable storage before the corresponding page (at least UNDO log records)
 - May have to flush log records to disk when writing a page to disk
- Log records flushed in order
- Strict 2 Phase Locking
 - High concurrency locks can be used instead
- Latches vs Locks
 - Latches used for physical consistency
 - Latches shorter duration

ARIES: What it does

- Normal processing:
 - Write log records for each action
- Normal processing: Rollbacks/Partial Rollbacks
 - Supports “savepoints”, and partial rollbacks
 - Write CLRs when undoing
 - Allows logical undos
 - Can release some locks when partial rollback completed
- Normal processing: Checkpoints
 - Store some state to disk
 - Dirty pages table, active transactions etc. . .
 - No need to write the dirty pages to disk: They are continuously being written in background
 - Checkpoint records the progress of that process
 - Called **fuzzy checkpoint**

ARIES: Restart Recovery

- **Redo before Undo**
- Analysis pass
 - Bring dirty pages table, transactions up to date
- Redo pass: Repeats history
 - Forward pass
 - Redo everything including transactions to be aborted
 - Otherwise page-oriented redo would be in trouble
- Undo pass: Undo loser transactions
 - Backwards pass
 - Undo simultaneously
 - Use CLRs to skip already undone actions

ARIES: Advanced

- Selective and deferred restart
- Fuzzy image copies
- Media recovery
- High concurrency lock modes (for increment/decrement operations)
- Nested Top Actions:
 - Transactions within transactions
 - E.g. Split a B+-Tree page; Increase the Extent size etc. . .
 - Use a dummy CLR to skip undoing these if the enclosing transaction is undone.