

CMSC724: Recovery

Amol Deshpande

April 15, 2008

1 Concepts

Recovery Mechanism

- 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**
 - * Data is not lost once its on disk
 - * Typically ensured through redundancy (e.g. RAID) and/or wide-area replication

Options

- ATOMIC:
 - A transaction's updates become visible on disk all at once
 - BUT disks only support atomic single-block writes
 - Can be done through "shadow paging"
 - * Make a copy of the page being updated and operate on that (System R)
 - Not desirable
 - * Storage space, sequentiality lost etc...
- 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

Recovery Mechanism

- 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 paging
 - A page might contain updates of two transactions ?
 - * Use page-level locking etc. . .
 - * For more details, see [Repeating History Beyond Aries; Mohan; VLDB 1991](#)

Recovery Mechanism

- Desired option: STEAL, NO FORCE
- STEAL: Issues
 - Dirty data might be written on disk
 - 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)
 - * Otherwise: dirty data on disk, but no UNDO log record
- NO FORCE: Issues
 - Data from committed transactions might not make it to disk
 - Use REDO logs
 - The REDO log records must make it disk before the transaction is “committed”
- Either case: log must be on the stable storage

Log Records

- Physical vs Logical logging:
 - Physical: Before and after copies of the data
 - Logical: 100 was added to $t1.a$, diffs
- Must be careful with logical log records
 - More compact, but **not idempotent**
 - Can't be applied twice
- Physical more common
- Why okay to write log but not the pages ?
 - Logs written sequentially on a separate disk
 - The change, and hence the log record, smaller

Checkpoints

- Don't want to start from the beginning of LOG everytime
- Use checkpoints to record the state of the DBMS at any time
- Simplest option:
 - Stop accepting new transactions, finish all current transactions
 - Write all memory contents to disk, all log records to LOG disk
 - Write a checkpoint record
 - Start processing again
- Not acceptable
 - Need to allow checkpoint to happen during normal processing
 - Typically dirty data written to disk as well

2 Simple Log-based Recovery

Simple Log-based Recovery

- Each action generates a *log* record (before/after copies)
- **Write Ahead Logging (WAL)**: Log records make it to disk before corresponding data page
- *Strict Two-Phase Locking*
 - Locks held till the end of transaction
 - 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 redo-list one by one
- Example: $(T_1, A, 10, 20)$, $(T_1, Abort)$, $(T_2, A, 10, 30)$, $(T_2, commit)$
- Must do UNDO before REDO
- This is because no CLR's (later)

3 ARIES

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 and apply - no need to touch any other pages
 - Requires stricter locking behaviour
 - * Hence not used for *indexes*
- Logical Undos
 - More compact, allow higher concurrency
 - May not be idempotent: Shouldn't undo twice
- Compensation Log Records (CLR's)
 - 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
 - $\text{Min}(\text{RecLSN of all dirty pages}) \rightarrow$ 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
- Latches vs Locks
 - Latches used for physical consistency
 - Latches are 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 (**repeating 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.

4 Miscellaneous

Recovery: Miscellaneous

- Basic ARIES Protocol not that complex
 - Fuzzy checkpoints, support for nested transactions etc complicate matters
- [WAL in PostgreSQL](#)
 - Note the line: “.. If *fsync* is off then this setting is irrelevant, since updates will not be forced out at all.”
- Postgres Storage Manager
 - Allowed **time-travel**: kept all copies of the data around
 - Used a small NVRAM (flash memory) in interesting ways to simplify recovery
 - Eventually taken out of Postgres (because of lack of a proper implementation?)