ACID Properties
Transaction Management must fulfill four requirements:
1. **Atomicity**: every action within a transaction either succeeds or fails as a whole.
   - Only actions of committed transactions must be visible.
2. **Consistency**: concurrent execution must leave the database in a consistent state.
   - Concurrent execution must leave the database in a consistent state.
3. **Isolation**: each transaction is protected from the effects of other concurrent transactions.
   - Net effect is that of some sequential execution.
4. **Durability**: once a transaction commits, the effects of the transaction persist.
   - Conversely, if a transaction aborts, there are no effects.

Recovery Manager
- **Crash recovery**: ensure that atomicity is preserved if the system crashes while one or more transactions are still incomplete.
- Main idea is to keep a log of operations; every action is logged before its page updates reach disk (Write-Ahead Log or WAL).

The Recovery Manager guarantees Atomicity & Durability.
- "One of hardest components of a DBMS to design and implement", pg. 580.
- One reason: need calls to it from all over the storage manager.

Motivation
- **Atomicity**: transactions may abort – must rollback their actions.
- **Durability**: what if DBMS stops running – e.g., power failure?

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

Assumptions
- **Concurrency control is in effect**
  - Strict 2PL
- **Updates are happening "in place"**
  - Data overwritten on (deleted from) the disk.
  - Centralized system, with one buffer pool for all system disks.
  - No pages in buffer overlay those pages on disk to define the database state.
- A simple scheme is needed.
  - A protocol that is too complex is difficult to implement.
  - Performance is also an important issue.

Handling the Buffer Pool
- **Force** every write to disk?
  - Poor response time - disk is slow!
  - But provides durability.
- **Want to be lazy** about writes to disk, but not too lazy!
- Note that one transaction can use more pages than can fit in the buffer manager, so DB needs to support spillage to disk.
- Need to be able to write out a page changed by an uncommitted transaction.
Stealing a page (see text, pg. 541)

- The same capability of writing a page with uncommitted data is used for “stealing” a page
- Scenario:
  - Transaction T1 has a lot of pages in buffer, with uncommitted changes
  - Transaction T2 needs a buffer page, steals it from T1 by having T1’s page written to disk, then using that buffer slot
  - With stealing going on, how can we ensure atomicity?
  - One controlling mechanism is page pinning
  - Only an unpinned buffer page can be stolen…
  - Another mechanism involves the log’s LSNs (log sequence numbers), covered soon

Lifetime of a page: page pinning in action

- Read by T1 and pinned (see pg. 319), S lock on row
- Read by T2 and pinned/share, S lock on row
- Read access finished by T1, unpinned by T1, still pinned by T2
- Read access finished by T2, unpinned, now fully unpinned
  - Note: no logging for reads
- Write access requested by T3, page is pinned exclusive, T3 gets X lock on row C, changes row, logs action, gets LSN back, puts in page header, page unpinned
  - Page now has 2 rows with S locks, one with X lock, is unpinned, so could be stolen

Steal and Force

- STEAL
  - Not easy to enforce atomicity when steal is possible
  - To steal frame F: current (unpinned) page P is written to disk; some transaction holds lock on row A of P
    - What if holder of the lock on A aborts?
      - Note the disk page holding A has the new value row, needs undoing
      - Must remember the old value of A at or before steal time (to support UNDOing the write to row A)
  - NO FORCE (lazy page writes)
    - What if system crashes before a modified page is written to disk?
      - Write as little as possible in a convenient place to support REDOing modifications

The Log

- The following actions are recorded in the log:
  - Ti writes an object: the old value and the new value.
    - Log record must go to disk before the changed page!
  - Ti commits/aborts: a log record indicating this action.
- Log records are chained together by Xact id, so it’s easy to undo a specific Xact.
- Log is often duplexed and archived on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

Logging

- Essential function for recovery
  - Record REDO and UNDO information, for every update
  - Example: Ti updates A from 10 to 20
    - Undo: know how to change 20 back to 10 if find 20 in disk page and know Ti aborted
    - Redo: know how to change 10 to 20 if see 10 in the disk page and know Ti committed.
  - Writes to log must be sequential, stored on a separate disk
  - Minimal information (summary of changes) written to log, since writing the log can be a performance problem

Logging

- What is in the Log
  - Ordered list of REDO/UNDO actions
  - Update log record contains:
    - <prevLSN, transID, pageID, offset, length, old data, new data>
  - Old data is called the before image
  - New data called the after image
  - The prevLSN provides the LSN of the transaction’s previous log record, so it’s easy to scan backwards through log records as needed in UNDO processing
Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  1. Must force the log record for an update before the corresponding data page gets to disk
  2. Must write all log records for transaction before commit returns

- Property 1 guarantees Atomicity
- Property 2 guarantees Durability

- We focus on the ARIES algorithm
  - Algorithms for Recovery and Isolation Exploiting Semantics

How Logging is Done

- Each log record has a unique Log Sequence Number (LSN)
  - LSNs always increasing
  - Works similar to “record locator”
- 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 largest LSN flushed so far
- WAL: Before a page is written, flush its log record such that pageLSN ≤ flushedLSN

- Log Records
  - Possible log entry types:
    - Update
    - Commit
    - Abort
    - End (signifies end of commit or abort)
    - Compensation Log Records (CLRs)
    - for UNDO actions

- LogRecord fields:
  - prevLSN
  - transID
  - entryType
  - pageID
  - length
  - offset
  - before-image
  - after-image

- Normal Execution of Transactions
  - 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
  - Strict 2PL
  - STEAL, NO-FORCE buffer management, with Write-Ahead Logging

- Transaction Commit
  - Write commit record to log for transaction T
  - All log records up to lastLSN of T are flushed.
  - Guarantees that flushedLSN ≥ lastLSN
  - Note that log flushes are sequential, synchronous writes to disk
  - Does NOT mean that page writes are propagated to data disk!
  - Commit() returns.
  - Write end record to log

- Other Log-Related State
  - Transaction Table:
    - One entry per active transaction
    - Contains transID, status (running/commited/aborted), and lastLSN (most recent LSN for transaction)
  - A dirty page is one whose disk and buffer images differ
    - So a dirty page becomes clean if page write, it stays in buffer
    - Once clean, can be deleted from dirty page table
    - And is clean if it gets read back into buffer even with uncommitted data in it
  - 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 (spec’s what part of log relates to redos for this page)
    - Earliest recLSN - important milestone for recovery (spec’s what part of log relates to redos for whole system)
  - Both the above are stored in RAM, hence volatile!
**Example: A Committing transaction**

R1(A, 50) W1(A, 20) C1
- R1(A): Transaction started, entered into Transaction table, page read into buffer, pinned, data used, unpinned (no logging).
- W1(A): page found in buffer, pinned, log record written:
  - prevLSN = null, transID = 1, entryType = update, etc.
  - Page now dirty, pageLSN=222, entered into dirty page table, unpinned
  - TxTable entry now has lastLSN = 222.
- C1: Log record (LSN223) for commit has prevLSN=222, Log is pushed so LSN 223 record is on disk. Now transaction is committed.
- Transaction status in TxTable is changed to committed.
- Log record for End (LSN224) is written, has prevLSN=223.
- Note: dirty page can still hang around in buffer pool: its content defines the database state for that page.
- Sometime later, dirty page written to disk, page considered clean, dropped from dirty page table.

**Simple Transaction Abort**

- First, consider an explicit abort of a transaction.
- No crash involved, have good transaction table.
- Need to “play back” the log in reverse order, UNDOing updates.
  - Get lastLSN of transaction from transaction table.
  - Find that log record, undo one page change.
  - 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?
  - For each update UNDO, write a CLR record in the log…

**Checkpointing**

- Periodically, the DBMS creates a checkpoint.
  - minimize time taken to recover in the event of a system crash
  - Checkpoint logging:
    - begin_checkpoint record: Indications when checkpoint began.
    - end_checkpoint record: Contains current transaction table and dirty page table as of begin_checkpoint time.
    - So the earliest recLSN is known at recovery time, and the set of live transactions, very useful for recovery.
    - Other transactions continue to run; tables accurate only as of the time of the begin_checkpoint record = fuzzy checkpoint.
    - No attempt to force dirty pages to disk!

**Example: An aborting transaction**

R1(A, 50) W1(A, 20) A1
- R1(A): Transaction started, entered into Transaction table, page read into buffer, pinned, data used, unpinned (no logging).
- W1(A): page found in buffer, pinned, log record written:
  - prevLSN = null, transID = 1, entryType = update, etc.
  - Page now dirty, pageLSN=222, entered into dirty page table, unpinned.
  - TxTable entry now has lastLSN = 222.
- A1: Log record (LSN223) for abort has prevLSN=222. Then undo actions are started.
  - Undo W1(A): use lastLSN of TxTable to locate log entry for write.
  - Write CLR record to log, with LSN 224.
  - Find page in buffer, pin, apply before image (50), so A=50 again, unpin.
  - Transaction status in TxTable is changed to aborted.
  - Log record for End (LSN224) is written, has prevLSN=224.
  - Note: dirty page can still hang around in buffer pool: its content defines the database state for that page.

**ARIES Overview**

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

- DB
  - Each with a pageLSN

- RAM
  - Transaction Table
  - lastLSN
  - status
  - Dirty Page Table
  - recLSN
  - flushedLSN
Crash Recovery: Big Picture

Start from a checkpoint (found in master record)

Three phases:

**ANALYSIS**: Find which transactions committed or failed since checkpoint
**REDO** all actions (repeat history)
**UNDO** effects of failed transactions

---

Oldest log rec. of Xact active at crash

Smallest rec LSN in dirty page table after Analysis

Last chkpt

CRASH

A R U