CS290f – Lecture 13
Database Crash Recovery

Scalable Internet Services, Fall 2006

Thorsten von Eicken
Department of Computer Science
University of California at Santa Barbara

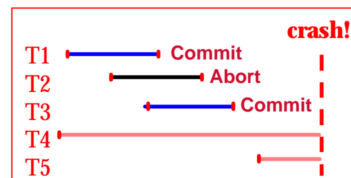
Some database material from J. Hellerstein @UCB

Review: The ACID properties

- **Atomicity**
  - All actions in the Xact happen, or none happen.
- **Consistency**
  - If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **Isolation**
  - Execution of one Xact is isolated from that of other Xacts.
- **Durability**
  - If a Xact commits, its effects persist.
- **Recovery Manager helps with:**
  - Atomicity & Durability
  - and also used for Consistency-related rollbacks

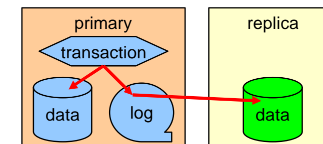
Motivation from DB point of view

- **Atomicity:**
  - Transactions may abort ("Rollback").
- **Durability:**
  - What if DBMS stops running? (Causes?)
- Desired state after system restarts:
  - **T1 & T3 should be durable.**
  - **T2, T4 & T5 should be aborted (effects not seen).**



Motivation from scalable internet services point of view

- **"Everything that matters" is in the database**
  - Database contains customer data and service data
  - Loss of database or corruption of database is fatal
- **Logging, replication, and scaling are related**
  - Need to understand logging to understand the others



- **Logging – basic idea**
  - Make updates to data in-place
  - Also log actions in a manner that enables redo/undo
  - Sequential log is fast, and can be pruned

Guarantee Atomicity & Durability

- **Assumptions:**
  - System may crash, but the *disk is durable*
  - The only *atomicity* guarantee is that *a disk block write* is *atomic (true?)*
- **Approach to guarantee A and D**
  - control how the disk and memory interact
  - store enough information during normal processing to recover from failures
  - develop algorithms to recover the database state
- **Obvious naïve solutions exist that work, but they are too slow**
  - E.g. shadow copy
    - Make a copy of the database
    - do the changes on the copy
    - do an atomic switch of the *dbpointer* at commit time
  - Goal is to do this as efficiently as possible

Simple solution

- **Only updates from committed transaction are written to disk**
- **Updates from a transaction are forced to disk before commit**
  - Minor problem: how to guarantee that all updates from a transaction make it to the disk atomically?
    - Remember we are only guaranteed an atomic *block write*
    - What if some updates make it to disk, and other don't?
  - Can use something like shadow copying/shadow paging
    - Write copies of block, including indirect/index blocks
    - Update "master" block last, switches all subordinate pages atomically
- **No atomicity/durability problem arise.**
  - But very sequential operation

Assumptions

- **Concurrency control is in effect.**
  - Strict 2PL, in particular
- **Updates are happening "in place".**
  - i.e. data is overwritten on (or deleted from) the actual page copies (not private copies)
- **Want to separate buffer pool from DB operation**
  - Database operates on in-memory buffers
  - Buffer manager decides when to read or write specific pages from/to disk
    - Including page replacement policy

STEAL vs. NO-STEAL

- **STEAL:**
  - The buffer manager can steal a (memory) page from the database
    - it can write an arbitrary page to the disk and use that page for something else from the disk
    - the database system doesn't control the buffer replacement policy
  - Problem?
    - page might contain dirty writes, i.e., writes/updates by a transaction that hasn't committed
  - But, we must allow steal for performance reasons.
- **NO STEAL:**
  - Not allowed. More control, but less flexibility for the buffer manager.

FORCE vs. NO FORCE

- **FORCE:**
  - The database system *forces* all the updates of a transaction to disk before committing
  - Why?
    - To make its updates permanent before committing
  - Problem?
    - Most probably random I/Os, so poor response time and throughput
    - Interferes with the disk controlling policies
- **NO FORCE:**
  - Don't do the above. Desired.
  - Problem:
    - Guaranteeing durability becomes hard
  - We might still have to *force* some pages to disk, but minimal.

Preferred Policy: Steal/No-Force

- This combination is most complicated but allows for highest performance.

- **No-Force – (complicates enforcing Durability)**
  - What if system crashes before a modified page written by a committed transaction makes it to disk?
  - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.
- **Steal – (complicates enforcing Atomicity)**
  - What if the Xact that performed udpates aborts?
  - What if system crashes before Xact is finished?
  - Must remember the old value of P (to support UNDOing the write to page P).

Buffer management summary

	No Steal	Steal
No Force		Fastest
Force	Slowest	

Performance Implications

	No Steal	Steal
No Force	No UNDO REDO	UNDO REDO
Force	No UNDO No REDO	UNDO No REDO

Logging/Recovery Implications

Basic Idea: Logging

- **Record REDO and UNDO information, for every update, in a log**
  - Sequential writes to log (put it on a separate disk)
  - Minimal info (diff) written to log, so multiple updates fit in a single log page
- **Log: An ordered list of REDO/UNDO actions**
  - Log record contains:
    - <XID, pageID, offset, length, old data, new data>
  - and additional control info (which we'll see soon)

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 force all log records for a Xact before commit. (i.e. transaction is not committed until all of its log records including its "commit" record are on the stable log.)
- **#1 (with UNDO info) helps guarantee Atomicity.**
- **#2 (with REDO info) helps guarantee Durability.**
- **Allows implementation of Steal/No-Force**
  - *See database book or database course for details!*

Normal Execution of a Xact

- **Series of reads & writes, followed by commit or abort**
  - We will assume that disk write is atomic.
    - In practice, additional details to deal with non-atomic writes.
- **Strict 2PL**
- **STEAL, NO-FORCE buffer management, with Write-Ahead Logging**

LOG	DISK	MEMORY
undo/redo info linked by prevLSN	pages master record	xact table …

Transaction Commit

- **Write commit record to log.**
- **All log records up to Xact's commit record are flushed to disk.**
  - Note that log flushes are sequential, synchronous writes to disk.
  - Many log records per log page.
- **Commit() returns.**
- **Write end record to log.**

Simple Transaction Abort

- **For now, consider an explicit abort of a Xact**
  - No crash involved.
- **Need to "play back" the log in reverse order, UNDOing updates**
  - Get lastLSN of Xact from Xact table.
  - Write an Abort log record before starting to rollback operations
  - Can follow chain of log records backward via the prevLSN field.
  - Write a "CLR" (compensation log record) for each undone operation.

Abort, cont.

- **To perform UNDO, must have a lock on data!**
  - No problem!
- **Before restoring old value of a page, write a CLR:**
  - You continue logging while you UNDO!!
  - CLR contains REDO info
  - CLRs never Undone
    - Undo needn't be idempotent (>1 UNDO won't happen)
    - But they might be Redone when repeating history (=1 UNDO guaranteed)
- **At end of all UNDOs, write an "end" log record.**

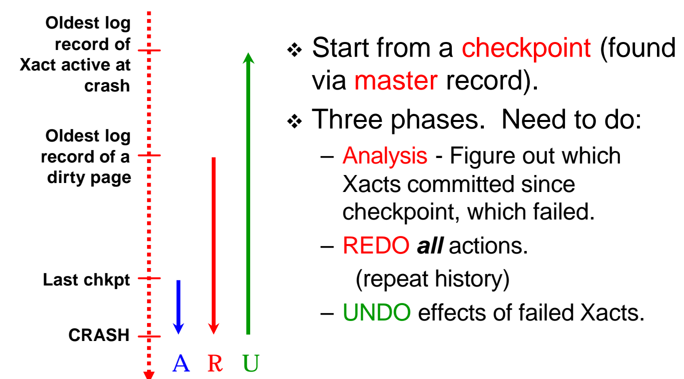
Checkpointing

- **Conceptually, keep log around for all time**
  - Obviously this has performance/implementation problems…
- **Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash. Write to log:**
  - begin_checkpoint record: Indicates when chkpt began.
  - end_checkpoint record: Contains current Xact table and dirty page table. This is a `fuzzy checkpoint':
    - Other Xacts continue to run; so these tables accurate only as of the time of the begin_checkpoint record.
    - No attempt to force dirty pages to disk; effectiveness of checkpoint limited by oldest unwritten change to a dirty page.
  - Store LSN of most recent chkpt record in a safe place (master record).

Hot backup

- **Idea: consistent backup of DB while it's running**
- **Traditional "cold backup"**
  - Shut DB down, backup files, start DB back up
  - Fast backup: take filesystem snapshot, copy snapshot to archive while DB runs again
- **Hot backup**
  - Halt DB writes to table space
  - All modifications go into log only
  - After backup, flush buffers to disk, apply log "redo" to table space (approximately)

Crash Recovery: Big Picture



Physical vs. Logical Logging

- **Physical logging is easy**
  - byte-oriented diffs
  - REDO/UNDO the same logic: apply the diff one way or the other
  - Can get messy if a logical operation (e.g. INSERT) has physical side-effects
    - e.g. file structures, indexes
- **Logical logging can be much more compact**
  - e.g. INSERT TUPLE vs. all the index page modification
  - But can be complicated! (Precludes page-oriented recovery.)
  - MySQL produces a logical (binary) log for backup and replication
  - InnoDB seems to use a physical log for atomicity/durability