Crash Recovery

CSCI 220: Database Management and Systems Design

Slides adapted from
Simon Miner
Gordon College

Practice Quiz: Equivalence

- Order the following from most to least "strict":
  result equivalence, conflict equivalence, view equivalence
- Draw a schedule that shows:
  - Two transactions which **are not serial**, and which **<u>are not</u> conflict serializable**
  - Two transactions which **are not serial**, but which **<u>are</u> conflict serializable**

Today you will learn…

- How do databases recover from crashes?
  - Transaction rollbacks (common)
  - System failures (more rare)

Review

Ensuring Data Integrity

- Issues related to preserving data integrity
  - Concurrency control
  - Crash control
- *Transactions* are a key concept at the heart of these matters
- Database is in a *consistent* state if there are no contradictions between the data within it
  - Temporary inconsistencies occur by necessity, but must not be allowed to persist
  - Example: transfer of funds between bank accounts

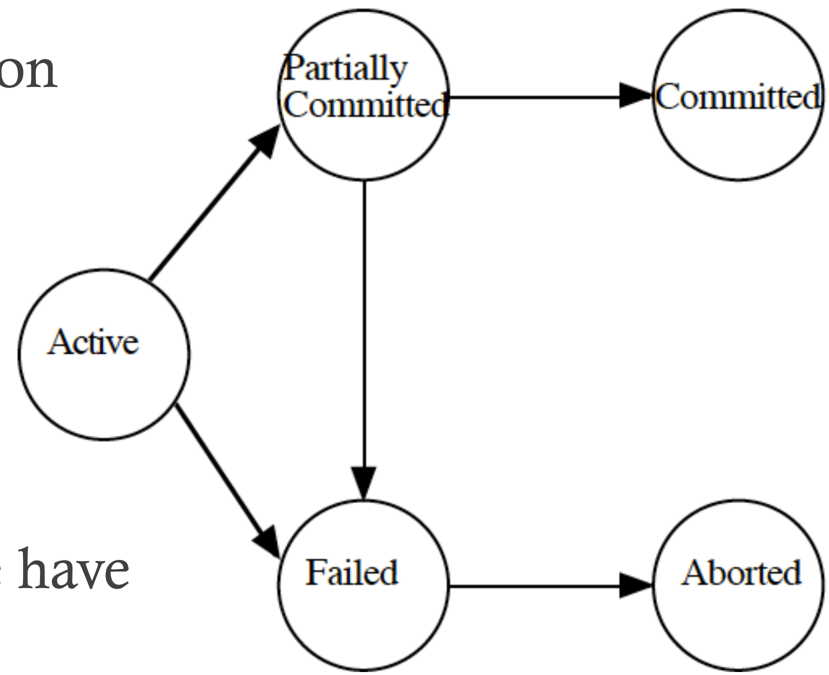
Transactions are Atomic and Preserve Consistency

- A transaction is an atomic operation (unit of work) involving a series of processing steps including:
  - One or more reads and/or writes
  - Data computations can happen during a transaction, but the database is mostly concerned with reads and writes
- If the database is in a consistent state at the start of the transaction, it will be in a consistent state at the end of the transaction

ACID

- **A**tomicity: either all of the transaction completes, or none of it completes
  - If any part of the transaction fails, all effects of it must be removed from the database
- **C**onsistency: database ends the transaction in a consistent state (provided it started that way)
- **I**solation: concurrently executing transactions must be unaware of each other (as if they ran serially)
  - It should look to one as if the other has not started or has already completed
- **D**urability: a transaction’s effects must persist in the database after it completes

Transaction States



- Active: from the time a transaction starts until it fails or reach its last statement
- Partially committed: last statement executed, but changes to database are not yet permanent (SQL commit)
- Committed: changes to database have been made permanent
- Failed: logic error or user abort has precluded completion, and transaction's changes must be undone (SQL rollback)
- Aborted: all effects of the transaction have been removed

Crash Recovery

Causes of Data Corruption

- Logical errors related to incoming data
  - Aborted operations (both programmatic and interactive)
- Transaction failures (i.e. from rollback, deadlock, etc.)
- System crashes
  - Power failure
  - Hardware failure (i.e. failed CPU)
  - Software failure (i.e. operating system crash)
  - Network communication failure
  - Human error
  - Security breach or cyber-attack
- Disk failures that destroy the medium storing the data
- External catastrophes (i.e., fire, flood, etc.)

Storage Types and Data Loss

- Volatile storage: main memory
  - Subject to data loss at any time from many factors (i.e. power, hardware, software failure, etc.)
- Non-volatile storage: disk
  - Not as prone to data corruption
  - Still susceptible to power failures during writes, disk failures, and external catastrophes
- "Stable" storage:
  - Write-once media: CDs, DVDs, Blu-ray, etc.
    - Susceptible to damage and degradation
  - RAID: susceptible to individual disk failures

Mitigating Storage Failures

- Use a redundant RAID configuration (e.g., RAID 10)
  - Monitor the integrity of the RAID by checking S.M.A.R.T. status and performing disk scrubbing
  - Promptly replace failed/failing disks
- Perform regular backups
  - Protect data against non-volatile storage failure and inadvertent data erasure (i.e. human error, ransomware)
    - Rare, but will occur eventually
  - Backups are essential but not enough
    - Need fast restoration of changes since the last backup
  - **Test your backups!**

Crash Recovery Measures

- Restore the system to a consistent state after an aborted operation or crash
- Ensure the durability property of transactions – that commits "stick"
  - Each transaction assigned a unique identifier (i.e. serial number)
  - Keep some record of incoming transactions
- Deal with in-process transactions when the system failed

Transaction Processing Log

- Track details of each transaction
  - Transaction start message
  - Details of changes made to the database
  - Transaction end message
- Used to recover from a crash
- Can be used for database replication

Transaction End Messages

- **Commit entry:** indicates successful completion of a transaction
  - This transaction's changes to the database should persist
- **Abort entry:** indicates the transaction failed
  - None of this transaction's changes should persist
- If the system crashes while a transaction is in progress, the end message **will be missing.** After crash recovery completes:
  - No changes from that transaction should persist
  - If possible, the transaction can be restarted

Protect the Log!

- The transaction processing log needs to be protected against corruption
  - Write it to stable storage
  - Keep multiple copies of the log in different locations
- Ensure the log data is written before the actual changes are written to the database
  - System typically buffers log entries until a block of them can be written
    - Actual database updates written after the log buffer is flushed
  - Sometimes it might be necessary to write out data block before the logging block is full
    - A forced write of a partial log buffer
- Ensure that a crash that occurs while the log block is being written does not corrupt previous log entries

Crash Recovery Schemes

- Incremental Log with Deferred Updates
  - No changes are made to the database until after the transaction commits and the commit entry is written to the log
- Incremental Log with Immediate Updates
  - Changes are made to the database during the transaction, but only after a log entry is written that includes the initial values of the things changed (so they can be recovered if necessary)
- Shadow Paging
  - Two copies of the relevant database data are kept during the transaction – both original and modified values. Once the transaction commits, the modified values permanently replace the original ones. (No log required.)

Storage Types

- Data only persists after it is written to nonvolatile storage
- For performance, data is buffered in memory
- For durability, the memory buffers for database data files and log files must be flushed at certain times in a transaction's lifecycle

Session memory space

Database buffer memory space

Database data files

Log files

Incremental Log with Deferred Updates

Incremental Log with Deferred Updates

- Example: A transaction to transfer $50 from checking to savings (with initial balances of $1000 and $2000, respectively.

SQL	Log Entries

update checking_accounts
set balance = balance – 50
where account_no = 127;
 	
update savings_accounts
set balance = balance + 50
where account_no = 253;
	T1234 starts
	T1234 writes 950 to balance of
	checking_accounts record 127
	T1234 writes 2050 to balance of
	savings_accounts record 253
	T1234 commits

- Once transaction partially commits (e.g. commit log entry is written), actual updates to the database occur
  - If the transaction fails or aborts, no changes have been made to the database

Deferred Update Recovery

- If the system crashes during a transaction:
  - If the crash occurs **before the commit log entry** is written:
    - Ignore (or restart) the transaction when the system is restored
  - If the crash occurs **after the commit log entry** is written:
    - (Re)write values from the log to the database (no harm in writing the same values to the database a second time)
- This *redo log* approach has the following recovery algorithm:
  - for each transaction with a commit record in the log
    - Write each new value for the transaction in the log to the database
- Checkpoint: periodic automated flush of buffers to disk
  - Causes committed transactions to be reflected in non-volatile storage
  - DBMS writes a checkpoint to the log
  - Only transactions after the checkpoint need to be applied after a crash

Deferred Update Tradeoffs

- Changes aren't reflected in the database until they are committed
- This incurs memory overhead
  - A transaction needs to keep a copy of the data it modifies, since it hasn't yet been written to disk
  - Cannot support transactions that don't fit in memory
- The major benefit is simpler recovery, since uncommitted transactions can be ignored

Incremental Log with Immediate Updates

Incremental Log with Immediate Updates

- Since database updates happen during the course of a transaction, log entries (written before the updates) must contain both old and new values

SQL	Log Entries

update checking_accounts
set balance = balance – 50
where account_no = 127;
 	
update savings_accounts
set balance = balance + 50
where account_no = :253;
	T1234 starts
	T1234 writes 950 to balance of checking_accounts record 127 (old value was 1000)
	T1234 writes 2050 to balance of savings_accounts record 253 (old value was 2000)
	T1234 commits

- If the transaction fails or aborts, all database updates must be undone by writing the original values back to the database

Immediate Update Recovery

- *Redo* and *undo* log approach to crash recovery
  - for each transaction with a start record in the log
    - if its commit record is also in the log
      - redo: write each new value for the transaction in the log to the database
    - else
      - undo: rewrite each old value for the transaction in the log to the database
- Order is critical
  - Undo operations must happen first (from newest to oldest)
  - Redo operations can happen afterward (from oldest to newest)
- Use checkpoints to minimize undo/redo work

Immediate Update Tradeoffs

- Longer log entries: both old and new values stored
- Every database write requires the corresponding log entry to be written to disk/stable storage (not just on commit)
- Failed transactions must be “cleaned up”
  - Crash recovery requires processing every transaction, not just the ones that committed
- The major benefit is support for large transactions that don't fit in memory

Shadow Paging

Shadow Paging

- Maintain two copies of the active portion of the database
  - Current version: reflects all changes since start of current transaction
  - Shadow version: state of database before current transaction began
- If transaction fails or aborts, current version is discarded
- If transaction commits, current version replaces shadow version

Shadow Paging Recovery

- Crash recovery is automatic, since changes are only made to the current version, simply revert to the shadow version
- Major benefit: in a single-user environment, a log isn't needed!

Shadow Paging Drawbacks

- Hard to maintain with lots of concurrent transactions
- Larger storage overhead than log-based approaches
  - Entire pages are duplicated (e.g., 8KB per page)
- Data fragmentation occurs quickly
  - Data is moved to different places on disk when it is changed
- Old shadow copies must be cleaned up after a commit
  - Garbage collection

Summary

- Multiple transactions can conflict with each other
  - Conflicts be efficiently detected using precedence graphs
  - Non-conflicting transactions are "Conflict Serializable"
- When there is a conflict, one of the transactions must be rolled back
  - Crash recovery must be aware of these ordinary transaction failures
- Different techniques can be used to implement crash recovery

Further Reading

- PostgreSQL Manual:
  Reliability and the Write-Ahead Log (WAL)
  - https://www.postgresql.org/docs/16/wal.html
  - WAL is an "immediate update" approach