Announcements (November 20)
- Homework #4 due next Tuesday
- Project demo period starts on December 8
- Final exam on December 13

Review
- ACID
  - Atomicity: TX’s are either completely done or not done at all
  - Consistency: TX’s should leave the database in a consistent state
  - Isolation: TX’s must behave as if they are executed in isolation
  - Durability: Effects of committed TX’s are resilient against failures
- SQL transactions
  - Begins implicitly
  - SELECT ...
  - UPDATE ...
  - ROLLBACK | COMMIT;

Concurrency control
- Goal: ensure the "I" (isolation) in ACID

Serial schedule
- Execute transactions in order, with no interleaving of operations
  - \( T_1.r(A), T_1.w(A), T_2.r(B), T_2.w(B), T_2.r(A), T_2.w(A), T_1.r(C), T_2.w(C) \)
  - \( T_1.r(A), T_2.w(A), T_2.r(C), T_2.w(C), T_2.r(A), T_1.w(A), T_1.r(B), T_1.w(B) \)
- Isolation achieved by definition!
- Problem: no concurrency at all
- Question: how to reorder operations to allow more concurrency
Conflicting operations
- Two operations on the same data item conflict if at least one of the operations is a write
  - r(X) and w(X) conflict
  - w(X) and r(X) conflict
  - w(X) and w(Y) conflict
  - r(X) and r(X) do not conflict
  - r/w(X) and r/w(Y) do not conflict
- Order of conflicting operations matters
  - E.g., if \( T_1, r(A) \) precedes \( T_2, w(A) \), then conceptually, \( T_1 \) should precede \( T_2 \)

Conflict-serializable schedule
- A schedule is conflict-serializable iff its precedence graph has no cycles
- A conflict-serializable schedule is equivalent to some serial schedule (and therefore is "good")
  - In that serial schedule, transactions are executed in the topological order of the precedence graph
  - You can get to that serial schedule by repeatedly swapping adjacent, non-conflicting operations from different transactions

Basic locking is not enough
- Add 1 to both \( A \) and \( B \) (preserves \( A = B \))
- Read 100
- Write 100 + 1
- \( w(A) \)
- \( \text{unlock}(A) \)
Possible schedule under locking
- But still not conflict-serializable!
- Read 200
- Write 200 + 1
- \( \text{unlock}(B) \)

Precedence graph
- A node for each transaction
- A directed edge from \( T_i \) to \( T_j \) if an operation of \( T_i \) precedes and conflicts with an operation of \( T_j \) in the schedule
- Good: no cycle
- Bad: cycle

Locking
- Rules
  - If a transaction wants to read an object, it must first request a shared lock (S mode) on that object
  - If a transaction wants to modify an object, it must first request an exclusive lock (X mode) on that object
  - Allow one exclusive lock, or multiple shared locks

Compatibility matrix

Two-phase locking (2PL)
- All lock requests precede all unlock requests
  - Phase 1: obtain locks, phase 2: release locks
  - 2PL guarantees a conflict-serializable schedule
  - Cannot obtain the lock on \( B \) until \( T_1 \) unlocks
Problem of 2PL

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>w(A)</td>
<td>r(B)</td>
</tr>
<tr>
<td>r(A)</td>
<td>w(A)</td>
<td></td>
</tr>
<tr>
<td>r(B)</td>
<td>w(B)</td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td>Abort</td>
<td></td>
</tr>
</tbody>
</table>

- T₂ has read uncommitted data written by T₁.
- If T₁ aborts, then T₂ must abort as well.
- Cascading aborts possible if other transactions have read data written by T₂.
- Even worse, what if T₂ commits before T₁?
  - Schedule is not recoverable if the system crashes right after T₂ commits.

Strict 2PL

- Only release locks at commit/abort time.
  - A writer will block all other readers until the writer commits or aborts.
- Used in most commercial DBMS (except Oracle).

Recovery

- Goal: ensure “A” (atomicity) and “D” (durability) in ACID.
- Execution model: to read/write X
  - The disk block containing X must be first brought into memory.
  - X is read/written in memory.
  - The memory block containing X, if modified, must be written back (flushed) to disk eventually.

Failures

- System crashes in the middle of a transaction T; partial effects of T were written to disk.
  - How do we undo T (atomicity)?
- System crashes right after a transaction T commits; not all effects of T were written to disk.
  - How do we complete T (durability)?

Naïve approach

- Force: When a transaction commits, all writes of this transaction must be reflected on disk.
  - Without force, if system crashes right after T commits, effects of T will be lost.
  - Problem: Lots of random writes hurt performance.
- No steal: Writes of a transaction can only be flushed to disk at commit time.
  - With steal, if system crashes before T commits but after some writes of T have been flushed to disk, there is no way to undo these writes.
  - Problem: Holding on to all dirty blocks requires lots of memory.

Logging

- Log:
  - Sequence of log records, recording all changes made to the database.
  - Written to stable storage (e.g., disk) during normal operation.
  - Used in recovery.
- Hey, one change turns into two—bad for performance?
  - But writes are sequential (append to the end of log).
  - Can use dedicated disk(s) to improve performance.
Undo/redo logging rules

- Record values before and after each modification: $(T_i, X, \text{old\_value\_of\_X}, \text{new\_value\_of\_X})$
- A transaction $T_i$ is committed when its commit log record $(T_i, \text{commit})$ is written to disk
- Write-ahead logging (WAL): Before $X$ is modified on disk, the log record pertaining to $X$ must be flushed
  - Without WAL, system might crash after $X$ is modified on disk but before its log record is written to disk—no way to undo
- No force: A transaction can commit even if its modified memory blocks have not be written to disk (since redo information is logged)
- Steal: Modified memory blocks can be flushed to disk anytime (since undo information is logged)

Undo/redo logging example

$T_1$ (balance transfer of $100$ from $A$ to $B$)
read($A, a$); $a = a - 100$;
write($A, a$);
read($B, b$); $b = b + 100$;
write($B, b$);

$A = 800, B = 400$

Checkpointing

- Where does recovery start?
- Naïve approach:
  - Stop accepting new transactions (lame!)
  - Finish all active transactions
  - Take a database dump
- Fuzzy checkpointing
  - Determine $S$, the set of currently active transactions, and log $(\text{begin-checkpoint } S)$
  - Flush all blocks (dirty at the time of the checkpoint) at your leisure
  - Log $(\text{end-checkpoint } \text{begin-checkpoint\_location})$
  - Between begin and end, continue processing old and new transactions

Recovery: analysis and redo phase

- Need to determine $U$, the set of active transactions at time of crash
- Scan log backward to find the last end-checkpoint record and follow the pointer to find the corresponding $(\text{start-checkpoint } S)$
- Initially, let $U$ be $S$
- Scan forward from that start-checkpoint to end of the log
  - For a log record $(T, \text{start})$, add $T$ to $U$
  - For a log record $(T, \text{commit | abort})$, remove $T$ from $U$
  - For a log record $(T, X, \text{old}, \text{new})$, issue write($X, \text{new}$)
- $\Rightarrow$ Basically repeats history!

Recovery: undo phase

- Scan log backward
  - Undo the effects of transactions in $U$
    - That is, for each log record $(T, X, \text{old}, \text{new})$ where $T$ is in $U$, issue write($X, \text{old}$), and log this operation too (part of the repeating-history paradigm)
    - Log $(T, \text{abort})$ when all effects of $T$ have been undone
  - An optimization
    - Each log record stores a pointer to the previous log record for the same transaction; follow the pointer chain during undo

Summary

- Concurrency control
  - Serial schedule: no interleaving
  - Conflict-serializable schedule: no cycles in the precedence graph; equivalent to a serial schedule
  - 2PL: guarantees a conflict-serializable schedule
  - Strict 2PL: also guarantees recoverability
- Recovery: undo/redo logging with fuzzy checkpointing
  - Normal operation: write-ahead logging, no force, steal
  - Recovery: first redo (forward), and then undo (backward)