Announcements (December 1)

- Homework #4 due next Tuesday (Dec. 6)
- Project demo period will start next Tuesday
  - Watch for an email tomorrow about scheduling
- 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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$:</td>
<td>$T_2$:</td>
<td></td>
</tr>
<tr>
<td>read(A);</td>
<td>read(A);</td>
<td></td>
</tr>
<tr>
<td>write(A);</td>
<td>write(A);</td>
<td></td>
</tr>
<tr>
<td>read(B);</td>
<td>read(C);</td>
<td></td>
</tr>
<tr>
<td>write(B);</td>
<td>write(C);</td>
<td></td>
</tr>
<tr>
<td>commit;</td>
<td>commit;</td>
<td></td>
</tr>
</tbody>
</table>

Good versus bad schedules

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>r(A)</td>
<td>r(A)</td>
<td>r(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td>r(A)</td>
<td>w(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>w(A)</td>
<td>w(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>w(B)</td>
<td>r(A)</td>
<td>r(B)</td>
<td>r(B)</td>
</tr>
<tr>
<td>r(A)</td>
<td>r(B)</td>
<td>r(C)</td>
<td>r(C)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(B)</td>
<td>w(B)</td>
<td>w(C)</td>
</tr>
<tr>
<td>r(C)</td>
<td>w(B)</td>
<td>w(C)</td>
<td>w(C)</td>
</tr>
</tbody>
</table>

Serial schedule

- Execute transactions in order, with no interleaving of operations
  - $T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B), T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C)$
  - $T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C), T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B)$
- 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 $w(X)$ conflict
  - $r(X)$ and $r(X)$ do not
  - $r/w(X)$ and $r/w(Y)$ do not

- Order of conflicting operations matters
  - E.g., if $T_1.r(A)$ precedes $T_2.w(A)$, then conceptually, $T_1$ should precede $T_2$

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

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
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

Basic locking is not enough

Possible schedule under locking

Two-phase locking (2PL)

- All lock requests precede all unlock requests
  - Phase 1: obtain locks, phase 2: release locks
Problem of 2PL

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

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:
  \( \langle T_i, X, \text{old}_X, \text{new}_X \rangle \)
- A transaction \( T_i \) is committed when its commit log record \( \langle T_i, \text{commit} \rangle \) 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 been 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 \( B \))

\begin{align*}
\text{read}(A, a) &; a = a - 100; \\
\text{write}(A, a) &; \\
\text{read}(B, b) &; b = b + 100; \\
\text{write}(B, b) &; \text{commit;}
\end{align*}

No force: can flush after commit

Steal: can flush before commit

No restriction on when memory blocks can/should be flushed

Checkpointing

- Naïve approach:
  - Stop accepting new transactions (lame!)
  - Finish all active transactions
  - Take a database dump
  - Now safe to truncate the log
- Fuzzy checkpointing
  - Determine \( S \), the set of currently active transactions, and log
    \( \langle \text{begin-checkpoint}, S \rangle \)
  - Flush all modified memory blocks at your leisure
  - Log \( \langle \text{end-checkpoint}, \text{begin-checkpoint}\_\text{location} \rangle \)
  - 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 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 $(T, X, \text{old, new})$, issue write($X$, new)

$F$ Basically repeats history!

Recovery: undo phase

- Scan log backward
  - Undo the effects of transactions in $U$
  - That is, for each log record $(T, X, \text{old, new})$ where $T$ is in $U$, issue write($X$, old), and log this operation too (part of the repeating-history paradigm)
  - Log $(T, \text{abort})$ when all effects of $T$ have been undone
  
$F$ 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)