Announcements
- Homework #4 due next Thursday (Dec. 2)
- Last call for student presentation on Dec. 2 (on databases for small devices)
  - Allows your lowest homework grade to be dropped
  - Need one more volunteer (talk to me right after class)
- Course project demo period coming up (Dec. 3-6)
  - Sign-up will start in a week
  - Early demos are encouraged

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
  - Isolation achieved by definition!
  - Problem: no concurrency at all
  - Question: how to reorder operations to allow more concurrency

Transaction Processing

CPS 116
Introduction to Database Systems

Concurrency control

Good versus bad schedules

<table>
<thead>
<tr>
<th>Good!</th>
<th>Bad!</th>
<th>Good! (But why?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>r($A$)</td>
<td>r($A$)</td>
<td>r($A$)</td>
</tr>
<tr>
<td>w($A$)</td>
<td>w($A$)</td>
<td>r($A$)</td>
</tr>
<tr>
<td>r($B$)</td>
<td>w($B$)</td>
<td>r($B$)</td>
</tr>
<tr>
<td>w($B$)</td>
<td>w($B$)</td>
<td>w($B$)</td>
</tr>
</tbody>
</table>
Conflicting operations

- Two operations on the same data item conflict if at least one of the operations is a write:
  - \text{r}(X) \text{ and } \text{w}(X) conflict
  - \text{w}(X) \text{ and } \text{r}(X) conflict
  - \text{w}(X) \text{ and } \text{w}(X) conflict
  - \text{r}(X) \text{ and } \text{r}(X) do not conflict
  - \text{r}(X) \text{ and } \text{w}(Y) do not conflict

- Order of conflicting operations matters:
  - E.g., if \text{T}_1.\text{r}(A) precedes \text{T}_2.\text{w}(A), then conceptually, \text{T}_1 should precede \text{T}_2.

Precedence graph

- A node for each transaction
- A directed edge from \text{T}_i to \text{T}_j if an operation of \text{T}_i precedes and conflicts with an operation of \text{T}_j in the schedule

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.

<table>
<thead>
<tr>
<th>Mode of lock(s) currently held by other transactions</th>
<th>Grant the lock?</th>
</tr>
</thead>
<tbody>
<tr>
<td>S \quad Yes \quad No</td>
<td></td>
</tr>
<tr>
<td>X \quad No \quad No</td>
<td></td>
</tr>
</tbody>
</table>

Two-phase locking (2PL)

- All lock requests preceed 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)</td>
<td>r(B)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(B)</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: \( T_i, X, old\_value\_of\_X, new\_value\_of\_X \)
- A transaction \( T_i \) is committed when its commit log record \( T_i, commit \) 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 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

Transaction \( T_1 \) (balance transfer of $100 from \( A \) to \( B \))

- Read \( A \), \( a = a - 100 \);
- Write \( A \), \( a \);
- Read \( B \), \( b = b + 100 \);
- Write \( B \), \( b \);

Memory

\[
\begin{align*}
A &= 800 \\
B &= 400
\end{align*}
\]

Log

\[
\begin{align*}
<T_1, start> &
<T_1, A, 800, 700> \\
<T_1, B, 400, 500> &
<T_1, commit>
\end{align*}
\]

Disk

\[
\begin{align*}
A &= 800 & 700 \\
B &= 400 & 500
\end{align*}
\]

No force: can flush before commit

Steal: can flush after commit

No restriction on when memory blocks can/should be flushed

Checkpointing

- Naive 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 \( begin\text{-}checkpoint \)
  - Flush all modified memory blocks at your leisure
  - Log \( end\text{-}checkpoint \)
  - 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 \( start\text{-}checkpoint \)
- Initially, let \( U \) be \( S \)
- Scan forward from that start-checkpoint to end of the log
  - For a log record \( T, start \), add \( T \) to \( U \)
  - For a log record \( T, commit | abort \), remove \( T \) from \( U \)
  - For a log record \( T, X, old, new \), issue \( write(X, new) \)
  - \( \Rightarrow \) Basically repeats history!

Recovery: undo phase

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