Announcements

- Homework #4 due on December 1
  - Can be used to replace your lowest homework grade
- December 1
  - Stream data processing
  - Final review (if time permits)
- December 3
  - Semantic Web
  - Course evaluation
- Course project demo December 8-12
- Final 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

<table>
<thead>
<tr>
<th>Good versus bad schedules</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
</tr>
<tr>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
</tr>
<tr>
<td>w(B)</td>
</tr>
<tr>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
</tr>
<tr>
<td>r(C)</td>
</tr>
<tr>
<td>w(C)</td>
</tr>
</tbody>
</table>

Serial schedule

- Execute transactions in order, with no interleaving of operations
  - T1, r(A), T1, w(A), T2, r(B), T1, w(B), T2, r(A), T2, w(A), T2, w(B), T1, r(C), T2, w(C)
  - T2, r(A), T2, w(A), T2, r(C), T2, w(C), T1, r(A), T1, w(A), T1, r(B), T1, 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
  - 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

<table>
<thead>
<tr>
<th>Mode of lock(s) currently held by other transactions</th>
<th>Compatibility matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>S Yes No</td>
<td>X Yes No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grant the lock?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes No</td>
</tr>
</tbody>
</table>

**Two-phase locking (2PL)**

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

**Basic locking is not enough**

- Add 1 to both $A$ and $B$ (preserves $A=B$)
  - lock-X(A)
  - lock-X(B)
  - read 100
  - write 100+1
  - unlock(A)
  - unlock(B)

- Possible schedule under locking
  - But still not conflict-serializable!

- Multiplying both $A$ and $B$ by 2 (preserves $A=B$)
  - lock-X(A)
  - lock-X(B)
  - read 101
  - write 101*2
  - unlock(A)
  - unlock(B)
Problem of 2PL

\[ T_1 \quad | \quad T_2 \]
\[ r(A) \quad w(A) \quad r(A) \quad w(A) \quad r(A) \quad w(A) \quad r(B) \quad w(B) \quad r(B) \quad w(B) \quad \text{Abort}^1 \]

- \( T_2 \) has read uncommitted data written by \( T_1 \)
- If \( T_1 \) aborts, then \( T_2 \) must abort as well
- Cascading aborts possible if other transactions have read data written by \( T_2 \)
- Even worse, what if \( T_2 \) commits before \( T_1 \)?
  - Schedule is not recoverable if the system crashes right after \( T_2 \) 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

\[
\text{CPU} \quad \text{Memory} \quad X \quad \text{Disk} \\
/X/ \quad \text{Y} \\
\]

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)?
- Media fails; data on disk corrupted
  - How do we reconstruct the database (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

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

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

Memory
- \( A = 800 \)
- \( B = 400 \)

Disk
- \( A = 700 \)
- \( B = 500 \)

Log
- \( \langle T_1, \text{start} \rangle \)
- \( \langle T_1, A, 800, 700 \rangle \)
- \( \langle T_1, B, 400, 500 \rangle \)
- \( \langle T_1, \text{commit} \rangle \)

Steal: can flush before commit
No force: can flush after 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 \( \{ \text{begin-checkpoint} \} \)
  - Flush all modified memory blocks at your leisure
  - Log \( \{ \text{end-checkpoint begin-checkpoint_location} \} \)
  - 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 \( \{ T, \text{commit} | \text{abort} \} \), remove \( T \) from \( U \)
  - For a log record \( \{ T, X, old, new \} \), issue write(\( X \), new)

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)