**Transaction Processing**

CPS 116
Introduction to Database Systems

---

**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

---

**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>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>r(B)</td>
<td>r(B)</td>
</tr>
<tr>
<td>w(B)</td>
<td>w(B)</td>
<td>w(B)</td>
</tr>
</tbody>
</table>

---

**Serial schedule**

- Execute transactions in order, with no interleaving of operations
  - $T_1.r(A), T_1.w(A), T_2.r(B), T_1.w(B), T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C), T_1.r(B), T_1.w(B)$
  - 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 T₁.r(A) precedes T₂.w(A), then conceptually, T₁ should precede T₂

Conflict-serializable schedule

- A schedule is conflict-serializable if 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</th>
<th>Grant the lock?</th>
</tr>
</thead>
<tbody>
<tr>
<td>S Y X</td>
<td>Yes</td>
</tr>
<tr>
<td>X N N</td>
<td>No</td>
</tr>
</tbody>
</table>

Compatibility matrix

Basic locking is not enough

Add 1 to both A and B (preserve A=B)
- T₁
  - lock-X(A)
  - t(A)
  - lock-X(B)
  - t(B)
- T₂
  - lock-X(A)
  - t(A)
  - Write 101
  - w(A)
  - unlock(A)
  - Possible schedule under locking
  - But still not conflict-serializable!
- T₃
  - lock-X(B)
  - t(B)
  - Read 200
  - w(B)
  - unlock(B)

Two-phase locking (2PL)

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

Precedence graph

- A node for each transaction
- A directed edge from Tᵢ to Tⱼ if an operation of Tᵢ precedes and conflicts with an operation of Tⱼ 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 the lock requested</th>
<th>Grant the lock?</th>
</tr>
</thead>
<tbody>
<tr>
<td>S Y X</td>
<td>Yes</td>
</tr>
<tr>
<td>X N N</td>
<td>No</td>
</tr>
</tbody>
</table>

Compatibility matrix

Two-phase locking (2PL)

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

Add 1 to both A and B (preserve A=B)
- T₁
  - lock-X(A)
  - t(A)
  - lock-X(B)
  - t(B)
- T₂
  - lock-X(A)
  - t(A)
  - Read 101
  - w(A)
  - Unlock(A)
  - Possible schedule under locking
  - But still not conflict-serializable!
- T₃
  - lock-X(B)
  - t(B)
  - Unlock(B)

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>$T_1$</th>
<th>$T_2$</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>

- $T_2$ 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

### 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: \( T_i, X, \text{old}_\text{value}_X, \text{new}_\text{value}_X \)
- 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 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 \( B \))

<table>
<thead>
<tr>
<th>Disk</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A = 800 )</td>
<td>( &lt;T_1, \text{start}&gt; )</td>
</tr>
<tr>
<td>( B = 400 )</td>
<td>( &lt;T_1, A, 800, 700&gt; )</td>
</tr>
<tr>
<td>( A = 700 )</td>
<td>( &lt;T_1, B, 400, 500&gt; )</td>
</tr>
<tr>
<td>( B = 500 )</td>
<td>( &lt;T_1, \text{commit}&gt; )</td>
</tr>
</tbody>
</table>

Memory

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

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 \( T \text{ begin-checkpoint} S \)
  - Flush all modified memory blocks at your leisure
  - Log \( T \text{ end-checkpoint begin-checkpoint_ location} \)
  - 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 \( T \text{ start-checkpoint} S \)
- 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 \( T \text{ commit \mid abort} \), remove \( T \) from \( U \)
  - For a log record \( T, X, \text{old}, \text{new} \), issue write\( (X, \text{new}) \)
= 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)