Transaction Processing

CPS 116
Introduction to Database Systems

Announcements (November 27)

- Homework #4 due this Thursday
- Project demo period starts on December 7
  - Each project gets a 30-minute with me and Yi
  - Watch for an email this weekend scheduling demo slots
- Final exam on December 15

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;
Concurrent control

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

```
T1: read(A); write(A); read(B); write(B); commit;
T2: read(A); write(A); read(C); write(C); commit;
```

Good versus bad schedules

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

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

Serial schedule

- **Execute transactions in order, with no interleaving of operations**
  - $T_1 \cdot r(A), T_1 \cdot w(A), T_1 \cdot r(B), T_1 \cdot w(B), T_2 \cdot r(A), T_2 \cdot w(A), T_2 \cdot r(C), T_2 \cdot w(C)$
  - $T_2 \cdot r(A), T_2 \cdot w(A), T_2 \cdot r(C), T_2 \cdot w(C), T_1 \cdot r(A), T_1 \cdot w(A), T_1 \cdot r(B), T_1 \cdot w(B)$

- 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

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<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>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<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>
<td>w(B)</td>
</tr>
<tr>
<td>r(C)</td>
<td>r(C)</td>
<td>r(C)</td>
<td>r(C)</td>
</tr>
<tr>
<td>w(C)</td>
<td>w(C)</td>
<td>w(C)</td>
<td>w(C)</td>
</tr>
</tbody>
</table>

Good: no cycle

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

Basic locking is not enough

Add 1 to both $A$ and $B$ (preserve $A = B$)
- Read 100
- Write 100 + 1

Lock-X($A$)
- Read $A$
- Write $A$
- Unlock($A$)

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

$T_1$
- Read 200
- Write 200 + 1

Lock-X($B$)
- Read $B$
- Write $B$
- Unlock($B$)

Two-phase locking (2PL)

All lock requests precede all unlock requests

Phase 1: obtain locks, phase 2: release locks

$T_1$
- Lock-X($A$)
- Lock-X($B$)
- Unlock($A$)
- Unlock($B$)

$T_2$
- Read $A$
- Write $A$
- Unlock($A$)
- Read $B$
- Write $B$
- Unlock($B$)

$T_1$ 2PL guarantees a conflict-serializable schedule

$T_2$
- Read $A$
- Write $A$
- Unlock($A$)
- Read $B$
- Write $B$
- Unlock($B$)

Cannot obtain the lock on $B$ until $T_1$ unlocks

Possible schedule under locking
- But still not conflict-serializable!
### Problem of 2PL

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(a)</td>
<td>r(a)</td>
</tr>
<tr>
<td>w(a)</td>
<td>w(a)</td>
</tr>
<tr>
<td>r(b)</td>
<td>r(b)</td>
</tr>
<tr>
<td>w(b)</td>
<td>w(b)</td>
</tr>
<tr>
<td>Abort</td>
<td>Abort</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:
- 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:
  \( T_i, X, \text{old
d_value_of}_X, \text{new
d_value_of}_X \)
- 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 \( B \))

```plaintext
read(A, a); a = a - 100;
write(A, a); 
read(B, b); b = b + 100;
write(B, b); commit;
```

Memory

<table>
<thead>
<tr>
<th></th>
<th>( A )</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>800</td>
<td>400</td>
</tr>
</tbody>
</table>

Disk

<table>
<thead>
<tr>
<th></th>
<th>( A )</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>800</td>
<td>400</td>
</tr>
</tbody>
</table>

Log

- \( <T_1, \text{start}> \)
- \( <T_1, A, 100, 700> \)
- \( <T_1, B, 400, 500> \)
- \( <T_1, \text{commit}> \)

Steal: can flush before commit

No force: can flush after commit

No restriction (except WAL) on when memory blocks can/should be flushed

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 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)
- 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
- 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)