Transaction Processing

CPS 196.3
Introduction to Database Systems

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

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

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

A  B  C

Good versus bad schedules

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

Serial schedule

- Execute transactions in order, with no interleaving of operations
  - T₁.r(A), T₁.w(A), T₁.r(B), T₁.w(B), T₂.r(A), T₂.w(A), T₂.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 t/w(Y) do not

- Order of conflicting operations matters
  - If T₁.r(A) precedes T₂.w(A), then conceptually, T₁ should precede T₂
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

- Add 1 to both A and B (preserves $A = B$)
- Multiply both A and B by 2 (preserves $A = B$)
- A ≠ B!

Two-phase locking (2PL)

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

Problem of 2PL

- $T_2$ 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 $T_1$ commits before $T_2$?
  - 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)?
- 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:
  \( \langle T_i, X, old\_value\_of\_X, new\_value\_of\_X \rangle \)
- A transaction \( T_i \) is committed when its commit log record \( \langle T_i, commit \rangle \) 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 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 $B$)
read($A, a$); $a = a - 100$
write($A, a$);
read($B, b$); $b = b + 100$
write($B, b$);
commit;

Memory
$A = 800, 700$
$B = 400, 500$

Disk
$A = 800, 700$
$B = 400, 500$

Log
$<T_1, \text{start}>$
$<T_1, A, 800, 700>$
$<T_1, B, 400, 500>$
$<T_1, \text{commit}>$

Steal: can flush before commit
No force: 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
    $<\begin{\text{begin-checkpoint} S}>$
  - Flush all modified memory blocks at your leisure
  - Log $<\begin{\text{end-checkpoint} \begin{\text{begin-checkpoint-location} S}>$
  - 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 $<\begin{\text{start-checkpoint} S}>$
- 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 $<T, X, \text{old}|\text{new}>$, issue write($X, \text{old}$)
- 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 (backword)