EECS 647: Introduction to Database Systems

Instructor: Luke Huan

Spring 2009
Today’s Topic

- Guarantee conflict-serializable 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>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>X</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Compatibility matrix

Mode of the lock requested

Grant the lock?
Basic locking is not enough

Possible schedule under locking
But still not conflict-serializable!

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

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

2PL guarantees a conflict-serializable schedule

Cannot obtain the lock on B until $T_1$ unlocks
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 are 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
Next ...

- A few examples
## Non-2PL, $A=1000$, $B=2000$, Output = ?

<table>
<thead>
<tr>
<th>Lock_X(A)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Lock_S(A)</td>
</tr>
<tr>
<td>A: = A - 50</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Unlock(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A)</td>
</tr>
<tr>
<td></td>
<td>Unlock(A)</td>
</tr>
<tr>
<td></td>
<td>Lock_S(B)</td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>Unlock(B)</td>
</tr>
<tr>
<td></td>
<td>PRINT(A + B)</td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>B := B + 50</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
</tbody>
</table>
**2PL, A= 1000, B=2000, Output =?**

<table>
<thead>
<tr>
<th>Action</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_X(A)</td>
<td>Lock_S(A)</td>
</tr>
<tr>
<td>Read(A)</td>
<td>Lock_S(B)</td>
</tr>
<tr>
<td>A := A-50</td>
<td>Read(A)</td>
</tr>
<tr>
<td>Write(A)</td>
<td>Lock_S(B)</td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Unlock(A)</td>
<td>B := B +50</td>
</tr>
<tr>
<td>Write(B)</td>
<td>Unlock(A)</td>
</tr>
<tr>
<td>Unlock(B)</td>
<td>Unlock(B)</td>
</tr>
<tr>
<td></td>
<td>PRINT(A+B)</td>
</tr>
</tbody>
</table>
Strict 2PL, $A = 1000$, $B=2000$, Output =?

<table>
<thead>
<tr>
<th>Action 1</th>
<th>Action 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_X(A)</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td>Lock_S(A)</td>
</tr>
<tr>
<td>$A := A-50$</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>$B := B +50$</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>Unlock(B)</td>
<td>Lock_S(B)</td>
</tr>
<tr>
<td></td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>PRINT(A+B)</td>
</tr>
<tr>
<td></td>
<td>Unlock(A)</td>
</tr>
<tr>
<td></td>
<td>Unlock(B)</td>
</tr>
</tbody>
</table>
Lock Management

- Lock and unlock requests handled by Lock Manager

- LM keeps an entry for each currently held lock.

- Entry contains:
  - List of xacts currently holding lock
  - Type of lock held (shared or exclusive)
  - Queue of lock requests
Lock Management, cont.

- When lock request arrives:
  - Does any other xact hold a conflicting lock?
    - If no, grant the lock.
    - If yes, put requestor into wait queue.

- Lock upgrade:
  - Shared lock can request to upgrade to exclusive
Deadlocks

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other.

- Two ways of dealing with deadlocks:
  - prevention
  - detection

- Many systems just punt and use Timeouts
  - What are the dangers with this approach?
Deadlock Detection

- Create and maintain a “waits-for” graph
- Periodically check for cycles in graph
Example:

T1: S(A), S(D), S(B)
T2: X(B) X(C)
T3: S(D), S(C), X(A)
T4: X(B)

Deadlock!
Deadlock Prevention

- Assign priorities based on timestamps.
- Say Ti wants a lock that Tj holds

  Two policies are possible:
  
  **Wait-Die**: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  
  **Wound-wait**: If Ti has higher priority, Tj aborts; otherwise Ti waits

- Why do these schemes guarantee no deadlocks?

- **Important detail**: If a transaction re-starts, make sure it gets its original timestamp. -- Why?
Summary I

- Correctness criterion for isolation is “serializability”.
  - In practice, we use “conflict serializability,” which is somewhat more restrictive but easy to enforce.

- Two Phase Locking and Strict 2PL: Locks implement the notions of conflict directly.
  - The lock manager keeps track of the locks issued.
  - **Deadlocks** may arise; can either be prevented or detected.
Recovery Motivation

- Atomicity:
  - Transactions may abort (“Rollback”).

- Durability:
  - What if DBMS stops running? (Causes?)

- Desired state after system restarts:
  - T1 & T3 should be durable.
  - T2, T4 & T5 should be aborted (effects not seen).
Handling 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:
  \[ T_i, X, \text{old\_value\_of\_X}, \text{new\_value\_of\_X} \]
- 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 $100 from A to B)$

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

Steal: can flush before commit

No force: can flush after commit

No restriction on when memory blocks can/should be flushed
Summary

- Goal:
  - Support Concurrency with Isolation

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Isolation</th>
<th>Recover-ability</th>
<th>Deadlock</th>
<th>Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strict 2PL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strict 2PL with priority</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)