Transaction Processing: Concurrency Control

CPS 216
Advanced Database Systems

ACID

• Atomicity
  – Transactions are either done or not done
  – They are never left partially executed
• Consistency
  – Transactions should leave the database in a consistent state
• Isolation
  – Transactions must behave as if they were executed in isolation
• Durability
  – Effects of completed transactions are resilient against failures

Transaction in SQL

• (Implicit beginning of transaction)
  SELECT ...;
  UPDATE ...
  ....
  ROLLBACK | COMMIT;

• ROLLBACK (a.k.a. transaction abort)
  – Will undo the the partial effects of the transaction
  – May be initiated by the DBMS
Concurrency control

- Goal: ensure the “I” (isolation) in ACID

\[
\begin{align*}
T_1: & \quad \text{read}(A); \quad \text{write}(A); \quad \text{read}(B); \quad \text{write}(B); \quad \text{commit}; \\
T_2: & \quad \text{read}(A); \quad \text{write}(A); \quad \text{read}(C); \quad \text{write}(C); \quad \text{commit};
\end{align*}
\]

Good versus bad schedules

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

Serial schedule

- Execute transactions in order, with no interleaving of operations
  - $T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B), T_2.r(A), T_2.w(B), T_2.r(C), T_2.w(C)$
  - $T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C), T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B)$
  - Isolation achieved by definition!
- Problem: no concurrency at all
- Question: how to reorder schedule 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 T₁.r(A) precedes T₂.w(A), then conceptually, T₁ should precede T₂

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

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

Mode of the lock requested

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode of lock(s) currently held by other transactions</td>
<td>S</td>
<td>X</td>
</tr>
</tbody>
</table>

Grant the lock?

Compatibility matrix

Basic locking is not enough

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

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
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 $T_2$
- What’s worse, what if $T_2$ commits before $T_1$?

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)

Deadlocks

Deadlock = cycle in the wait-for graph
Dealing with deadlocks

- Impose an order for locking objects
  – Must know in advance which objects a transaction will access
- Timeout
  – If a transaction has been blocked for too long, just abort
- Prevention
  – Idea: abort more often, so blocking is less likely
  – Wait/die versus wound/wait
- Detection using wait-for graph
  – Idea: deadlock is rare, so only deal with it when it becomes an issue
  – How often do we detect deadlocks?
  – Which transactions do we abort in case of deadlock?

Implementation of locking

- Do not rely on transactions themselves to lock/unlock explicitly
- DBMS inserts lock/unlock requests automatically

SQL transaction isolation levels

- SERIALIZABLE (default)
- Weaker isolations levels
  – READ UNCOMMITTED
  – READ COMMITTED
  – REPEATABLE READ
- Why weaker levels?
READ UNCOMMITTED

- Dirty reads possible (dirty = uncommitted)
- Example: wrong average
  
  T1: 
  UPDATE Account
  SET balance = balance – 200
  WHERE number = 142857;

  ROLLBACK;

  T2: 
  SELECT AVG(balance)
  FROM Account;

- Possible cause

READ COMMITTED

- No dirty reads, but non-repeatable reads possible
- Example: different averages
  
  T1: 
  UPDATE Account
  SET balance = balance – 200
  WHERE number = 142857;

  COMMIT;

  T2: 
  SELECT AVG(balance)
  FROM Account;

- Possible cause

REPEATABLE READ

- Reads repeatable, but may see phantoms
- Example: different average (still!)
  
  T1: 
  INSERT INTO Account
  VALUES(428571, 1000);

  COMMIT;

  T2: 
  SELECT AVG(balance)
  FROM Account;

- Possible cause
Summary of SQL isolation levels

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty reads</th>
<th>Non-repeatable reads</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

- Criticized for definition in terms of anomalies

Concurrency control without locking

- Optimistic (validation-based)
- Timestamp-based
- Multi-version (Oracle)

Optimistic concurrency control

- Locking is pessimistic
  - Use blocking to avoid conflicts
  - Overhead of locking even if contention is low
- Optimistic concurrency control
  - Assume that most transactions do not conflict
  - Let them execute as much as possible
  - If it turns out that they conflict, abort and restart
Sketch of protocol

- Read phase: transaction executes, reads from the database, and writes to a private space
- Validate phase: DBMS checks for conflicts with other transactions; if conflict is possible, abort and restart
  - Requires maintaining a list of objects read and written by each transaction
- Write phase: copy changes in the private space to the database

Pessimistic versus optimistic

- Overhead of locking vs. overhead of validation and copying private space
- Blocking versus aborts and restarts
  - Locking has better throughput for environments with medium-to-high contention
  - Optimistic concurrency control is better when resource utilization is low enough

Timestamp-based

- Associate each database object with a read timestamp and a write timestamp
- Assign a timestamp to each transaction
  - Timestamp order is commit order
- When transaction reads/writes an object, check the object’s timestamp for conflict with a younger transaction; if so, abort and restart
- Problems
  -
Multi-version concurrency control

- Maintain versions for each database object
  - Each write creates a new version
  - Each read is directed to an appropriate version
  - Conflicts are detected in a similar manner as timestamp concurrency control
- In addition to the problems inherited from timestamp concurrency control
  - Pro:
  - Con:
- Oracle uses some variant of this scheme

Summary

- Covered
  - Conflict-serializability
  - 2PL, strict 2PL
  - Deadlocks
  - Overview of other concurrency-control methods
- Not covered
  - View-serializability (overview next)
  - Hierarchical locking (overview next)
  - Predicate locking and tree locking (later in course)

View-serializability

- Some schedules are not conflict-serializable, but they are still equivalent to a serial schedule

- View-serializability allows blind writes
  - That’s it? Forget it!
Hierarchical locking

- A database contains many tables, a table contains many pages/blocks, and a page/block contains many rows…
- Fine-granule locking allows more concurrency
  - Example:
- Coarse-granule locking has lower overhead
  - Example:
- Allow both: need to revise the locking protocol

Next time

Recovery

SQL triggers and programming interface