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
    - For example, when some statement in the transaction violates a database constraint

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>
<tr>
<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>
</tr>
</tbody>
</table>

Serial schedule
- Execute transactions in order, with no interleaving of operations
  - $T_1$, $r(A)$, $T_1$, $w(A)$, $T_1$, $r(B)$, $T_1$, $w(B)$, $T_2$, $r(C)$, $T_2$, $w(C)$
  - $T_2$, $r(A)$, $T_2$, $w(A)$, $T_2$, $r(C)$, $T_2$, $w(C)$, $T_1$, $r(A)$, $T_1$, $w(A)$, $T_1$, $r(B)$, $T_1$, $w(B)$
  - 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 conflict
  - r(w(X)) and w(Y) do not conflict
- 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

Two-phase locking (2PL)
- All lock requests precede all unlock requests
  - Phase 1: obtain locks, phase 2: release locks

Basic locking is not enough
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 $T_1$?
  - 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)

Deadlocks

Deadlock = cycle in the wait-for graph

- $T_1$ is waiting for $T_2$
- $T_2$ is waiting for $T_1$

- Deadlock!

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

Transactions Streams of operations

Insert lock/unlock requests Operations with lock/unlock requests

Lock table

Lock info for each object, including locks currently held and the request queue

Scheduler

Serialized schedule with no lock/unlock operations

SQL transaction isolation levels

- SERIALIZABLE (default)
- Weaker isolations levels
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
- Why weaker levels?
  - Increase performance by eliminating overhead and allowing higher degree of concurrency
**READ UNCOMMITTED**

- Dirty reads possible (dirty = uncommitted)
- Example: wrong average
  
  T1:  
  ```sql
  UPDATE Account 
  SET balance = balance – 200 
  WHERE number = 142857; 
  SELECT AVG(balance) 
  FROM Account; 
  ROLLBACK; 
  COMMIT; 
  ```
  
  T2:  
  ```sql
  UPDATE Account 
  SET balance = balance – 200 
  WHERE number = 142857; 
  COMMIT; 
  ```
  
  Possible cause
  - Non-strict locking protocol, or no read lock

**READ COMMITTED**

- No dirty reads, but non-repeatable reads possible
- Example: different averages
  
  T1:  
  ```sql
  SELECT AVG(balance) 
  FROM Account; 
  COMMIT; 
  ```
  
  T2:  
  ```sql
  UPDATE Account 
  SET balance = balance – 200 
  WHERE number = 142857; 
  COMMIT; 
  ```
  
  Possible cause
  - Locking is not two-phase

**REPEATABLE READ**

- Reads repeatable, but may see phantoms
- Example: different average (still!)
  
  T1:  
  ```sql
  INSERT INTO Account 
  VALUES(428571, 1000); 
  COMMIT; 
  ```
  
  T2:  
  ```sql
  SELECT AVG(balance) 
  FROM Account; 
  COMMIT; 
  ```
  
  Possible cause
  - Insertion did not acquire any lock (what to acquire?)

**Summary of SQL isolation levels**

<table>
<thead>
<tr>
<th>Isolation level / anomaly</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>No</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 versus 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
  - Even reads require writes (of object timestamps)
  - Ensuring recoverability is hard (plenty of dirty reads)

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: Reads are never blocked
  - Con: Multiple versions need to be maintained
- 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
  - Hierarchical locking
  - Predicate locking and tree locking

Next time

- Recovery
- SQL triggers and programming interface