Introduction to Data Management

Isolation Levels

Jonathan Leang

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

August 11, 2019
Recap

- Schedules under 2PL are conflict serializable
  - Locking phase → unlocking phase

- Conflict serializable schedules follow the isolation principle of ACID
  - No dirty read (WR)
  - No unrepeatable read (RW)
  - No lost update (WW)

- Schedules under strict 2PL additionally provide recoverability
  - Locking phase → unlock with commit or rollback
Outline

- Shared/Exclusive locks
- Isolation levels
- Implementing transactions in practice
Practicality of Binary Locks

- Binary Locks → full control or no control
- Leads to excessive deadlocking
Thrashing

Throughput (txns/sec TPS) vs. # of active transactions

Thrashing
Observation: Reads don’t conflict with each other

Simple 3-tier lock hierarchy:
- Exclusive/Write Lock $\rightarrow X_i(A)$
  - Full control
  - No other locks may exist
- Shared/Read Lock $\rightarrow S_i(A)$
  - Shared control
  - May exist with other shared locks
- Unlocked
# Shared/Exclusive Locks

<table>
<thead>
<tr>
<th>Requested Lock</th>
<th>unlocked</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Practicality of Serializability

- **Easy to reason about**
  - Application programming is easier under serializability assumptions

- **Expensive to use**
  - Slow
  - Resource intensive

- **Applications often don’t need serializability**
  - Application functionality may not depend on serializability
  - Financial/User experience cost is low enough for tradeoff considerations
Isolation Levels

- **SET TRANSACTION ISOLATION LEVEL ...**
  - READ UNCOMMITTED
  - READ COMMITED
  - REPEATABLE READ
  - SERIALIZABLE
  - SNAPSHOT ISOLATION
  - ...

- Default isolation level and configurability depends on the DBMS (read the docs)
### READ UNCOMMITTED

- **W**rites → **S**trict 2PL write locks
- **R**eads → **N**o locks needed
- Dirty reads are possible

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A) W(A)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>ABORT</td>
<td></td>
</tr>
<tr>
<td>U(A)</td>
<td></td>
</tr>
</tbody>
</table>
READ COMMITTED

- Writes → Strict 2PL write locks
- Reads → Short-duration read read locks
  - Acquire lock before reading and release lock after (not 2PL)
- Dirty reads are prevented

T1  | T2
---|---
X(A) W(A) | 
R(A) | 
COMMIT | 
ABORT U(A) | 

T1  | T2
---|---
X(A) W(A) | 
S(A) blocked... | 
...granted S(A) | 
ABORT U(A) | 
R(A) | 
COMMIT U(A) |
**Unrepeatable reads are possible**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A) blocked…</td>
<td>S(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>…granted X(A)</td>
<td>U(A)</td>
</tr>
<tr>
<td><strong>W(A)</strong></td>
<td>S(A) blocked…</td>
</tr>
<tr>
<td>COMMIT U(A)</td>
<td>…granted S(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT U(A)</td>
</tr>
</tbody>
</table>
### REPEATABLE READ

- **W**rites → Strict 2PL write locks
- **R**eads → Strict 2PL read locks
- **Unrepeatable reads are prevented**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A) blocked...</td>
<td>S(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>...granted X(A)</td>
<td>U(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>S(A) blocked...</td>
</tr>
<tr>
<td>COMMIT U(A)</td>
<td>...granted S(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT U(A)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A) blocked...</td>
<td>S(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>...granted X(A)</td>
<td>COMMIT U(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMIT U(A)</td>
</tr>
</tbody>
</table>
REPEATABLE READ

- Writes $\rightarrow$ Strict 2PL write locks
- Reads $\rightarrow$ Strict 2PL read locks
- **Phantom reads are possible...**
Conflict serializability implying serializability assumes a **static database**

- Conflicts only matter for the same element
- Inserting a new element (tuple-level granularity) means that the conflict model no longer is able to encapsulate it

```
SELECT * FROM Table;
R(A)
R(B)

I(C)

INSERT INTO Table VALUES (C...);

SELECT * FROM Table;
R(A)
R(B)
R(C)
```
- Dynamic database serializability needs either:
  - Table locking (prevent insertions) or
  - Predicate locking (lock based on query filters)
- Write Lock $\rightarrow$ Strict 2PL
- Read Lock $\rightarrow$ Strict 2PL
- Plus predicate locks and/or table locks
### Isolation Level Summary

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Dirty Read</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Unrepeatable Read</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Phantom Read</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>No Anomalies</td>
</tr>
</tbody>
</table>
Applying Transaction Logic

- Applications generally need to:
  - Check/Set isolation levels
  - Specify operations as transactions

- Common mistakes/misconceptions:
  - You do not need to implement locking. The DBMS takes care of it.
  - You must **close all explicit transactions** with COMMIT or ROLLBACK. Not doing so will cause the application to hang (wait due to unfinished locking).
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
conn.setTransactionIsolation(Connection.TRANSACTION_READ_UNCOMMITTED);
conn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
conn.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
conn.setAutoCommit(true);
conn.setAutoCommit(false);
try {
    // Each Instance hold a unique conn
    PreparedStatement q = conn.prepareStatement("SELECT ...");
    PreparedStatement i = conn.prepareStatement("INSERT ...");
    // Make sure the statements don’t execute separately
    conn.setAutoCommit(false);
    conn.execute("BEGIN TRANSACTION;");
    ResultSet rs = q.executeQuery();
    while (rs.next()) { ... } // Read out tuples from the ResultSet
    i.executeUpdate();
    conn.execute("COMMIT;"刑法3);
    conn.setAutoCommit(true);
    return "success";
} catch (SQLException ex) {
    try {
        conn.execute("ROLLBACK;"刑法3);
        conn.setAutoCommit(true);
        return "failed";
    } catch (SQLException e) {
        return "failed";
    }
}