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
- Phantom Problem
- Implementing transactions in practice

Practicality of Binary Locks

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

Shared/Exclusive Locks

- Observation: Reads don’t conflict with each other
- Simple 3-tier lock hierarchy:
  - Exclusive/Write Lock → $X_i(A)$
    - Full control
    - No other locks may exist
  - Shared/Read Lock → $S_i(A)$
    - Shared control
    - May exist with other shared locks
    - Unlocked
Shared/Exclusive Locks

<table>
<thead>
<tr>
<th>Requested Link</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)
  - Serializable is often not the default!

Phantom Reads

- **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

- **Dynamic database serializability needs either:**
  - Table locking (prevent insertions) or
  - Predicate locking (lock based on query filters)

Phantom Reads

SERIALIZABLE Level

- **Write Lock → Strict 2PL**
- **Read Lock → Strict 2PL**
- Plus predicate locks and/or table locks to handle phantom problem
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).

Transaction Setup

```java
conn.setTransactionIsolation(Java.sql.Connection.TRANSACTION_SERIALIZABLE);
```

Choose one:
```java
conn.setAutoCommit(true); // if you want each SQL statement to be TR
conn.setAutoCommit(false); // if you want to declare BEGIN/COMMIT yourself
```

DB Transaction Programming in Java

```java
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;"),
    conn.setAutoCommit(true);
    return "success"
} catch (SQLException ex) {
    try {
        conn.execute("ROLLBACK;"),
        conn.setAutoCommit(true);
        return "failed"
    } catch (SQLException e) {
        return "failed"
    }
}
```