Transaction isolation levels define the degree to which a transaction must be isolated from resource or data modifications made by other transactions. They range from the least restrictive, which allows the most concurrency, to the most restrictive, which ensures the highest data integrity. Here are the isolation levels defined by SQL standard, with examples:

1. Read Uncommitted

Definition: The lowest isolation level, where transactions are allowed to read data that has been modified but not yet committed by other transactions. This can lead to dirty reads.

Characteristics:

- Dirty Reads: Allowed.

- Non-Repeatable Reads: Allowed.

- Phantom Reads: Allowed.

Example:

- Transaction T1: Reads the balance of Account A, which is $100.

- Transaction T2: Updates the balance of Account A to $150 but does not commit.

- Transaction T1: Reads the balance of Account A again and sees $150 (dirty read).

- Transaction T2: Rolls back the update.

- Transaction T1: The final balance of Account A should still be $100, but T1 saw an uncommitted $150.

2. Read Committed

Definition: Ensures that any data read during the transaction is committed at the moment it is read. This prevents dirty reads but allows non-repeatable reads.

Characteristics:

- Dirty Reads: Prevented.

- Non-Repeatable Reads: Allowed.

- Phantom Reads: Allowed.

Example:

- Transaction T1: Reads the balance of Account A, which is $100.

- Transaction T2: Updates the balance of Account A to $150 and commits.

- Transaction T1: Reads the balance of Account A again and sees $150 (non-repeatable read).

- Transaction T1: The first read saw $100, but the second read saw $150 due to T2’s committed update.

3. Repeatable Read

Definition: Ensures that if a transaction reads a row, it will see the same data for that row if it reads it again later, even if other transactions modify it in the meantime. This prevents dirty reads and non-repeatable reads but allows phantom reads.

Characteristics:

- Dirty Reads: Prevented.

- Non-Repeatable Reads: Prevented.

- Phantom Reads: Allowed.

Example:

- Transaction T1: Reads the balance of Account A, which is $100.

- Transaction T2: Updates the balance of Account A to $150 and commits.

- Transaction T1: Reads the balance of Account A again and still sees $100 (prevented non-repeatable read).

- Transaction T2: Inserts a new Account B with balance $200 and commits.

- Transaction T1: Performs a query to see all accounts. Initially saw Account A only, now sees both Account A and Account B (phantom read).

4. Serializable

Definition: The highest isolation level, where transactions are executed in a way that ensures they are completely isolated from each other. This level prevents dirty reads, non-repeatable reads, and phantom reads.

Characteristics:

- Dirty Reads: Prevented.

- Non-Repeatable Reads: Prevented.

- Phantom Reads: Prevented.

Example:

- Transaction T1: Reads the balance of Account A, which is $100.

- Transaction T2: Tries to update the balance of Account A but is blocked until T1 completes.

- Transaction T1: Reads the balance of Account A again and sees $100 (prevented non-repeatable read).

- Transaction T2: Tries to insert a new Account B but is blocked until T1 completes.

- Transaction T1: Performs a query to see all accounts, only sees Account A (no phantom read).

- Transaction T1: Commits or rolls back.

- Transaction T2: Proceeds with its updates or inserts.

Summary

|  |  |  |  |
| --- | --- | --- | --- |
| Isolation Level | Dirty Reads | Non-Repeatable Reads | Phantom Reads |
| Read Uncommitted | Allowed | Allowed | Allowed |
| Read Committed | Prevented | Allowed | Allowed |
| Repeatable Read | Prevented | Prevented | Allowed |
| Serializable | Prevented | Prevented | prevented |

- Read Uncommitted: Use when the highest level of concurrency is needed, and occasional dirty reads are acceptable.

- Read Committed: Use when dirty reads are unacceptable but non-repeatable reads and phantom reads are tolerable.

- Repeatable Read: Use when both dirty reads and non-repeatable reads are unacceptable, but phantom reads are tolerable.

- Serializable: Use when the highest level of data integrity is required, and you can afford the reduced concurrency.

The choice of isolation level depends on the specific requirements for data consistency and the performance characteristics of the application.

To demonstrate a transaction involving a user placing an order, updating the product inventory, and reflecting the changes in a relational database, we can set up three tables: `Users`, `Orders`, and `Products`.

Table Definitions

1. Users Table

sql

CREATE TABLE Users (

user\_id INT PRIMARY KEY,

username VARCHAR(50) NOT NULL,

email VARCHAR(100) NOT NULL

);

2. Products Table

sql

CREATE TABLE Products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100) NOT NULL,

price DECIMAL(10, 2) NOT NULL,

stock INT NOT NULL

);

3. Orders Table

sql

CREATE TABLE Orders (

order\_id INT PRIMARY KEY AUTO\_INCREMENT,

user\_id INT NOT NULL,

product\_id INT NOT NULL,

quantity INT NOT NULL,

order\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES Users(user\_id),

FOREIGN KEY (product\_id) REFERENCES Products(product\_id)

);

Sample Data

sql

-- Insert sample users

INSERT INTO Users (user\_id, username, email) VALUES

(1, 'Alice', 'alice@example.com'),

(2, 'Bob', 'bob@example.com');

-- Insert sample products

INSERT INTO Products (product\_id, product\_name, price, stock) VALUES

(1, 'Laptop', 1000.00, 10),

(2, 'Smartphone', 500.00, 20);

Transaction Example

Let's create a Java program to handle the process of a user placing an order. This program will:

1. Check the availability of the product.

2. Create a new order.

3. Update the product's inventory.

4. Commit the transaction if all operations succeed or roll back if any operation fails.

java

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class EcommerceTransaction {

public static void main(String[] args) {

Connection conn = null;

PreparedStatement checkStockStmt = null;

PreparedStatement updateStockStmt = null;

PreparedStatement insertOrderStmt = null;

try {

Class.forName("com.mysql.cj.jdbc.Driver");

conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/ecommerce", "username", "password");

conn.setAutoCommit(false); // Disable auto-commit

int userId = 1;

int productId = 1;

int orderQuantity = 2;

// Check if there is enough stock

checkStockStmt = conn.prepareStatement("SELECT stock FROM Products WHERE product\_id = ?");

checkStockStmt.setInt(1, productId);

ResultSet rs = checkStockStmt.executeQuery();

if (rs.next()) {

int stock = rs.getInt("stock");

if (stock >= orderQuantity) {

// Deduct stock

updateStockStmt = conn.prepareStatement("UPDATE Products SET stock = stock - ? WHERE product\_id = ?");

updateStockStmt.setInt(1, orderQuantity);

updateStockStmt.setInt(2, productId);

updateStockStmt.executeUpdate();

// Insert order record

insertOrderStmt = conn.prepareStatement("INSERT INTO Orders (user\_id, product\_id, quantity) VALUES (?, ?, ?)");

insertOrderStmt.setInt(1, userId);

insertOrderStmt.setInt(2, productId);

insertOrderStmt.setInt(3, orderQuantity);

insertOrderStmt.executeUpdate();

// Commit transaction

conn.commit();

System.out.println("Order placed successfully.");

} else {

System.out.println("Insufficient stock for the order.");

}

}

} catch (SQLException | ClassNotFoundException e) {

if (conn != null) {

try {

conn.rollback(); // Rollback transaction

System.out.println("Transaction failed. Rolled back.");

} catch (SQLException ex) {

ex.printStackTrace();

}

}

e.printStackTrace();

} finally {

if (checkStockStmt != null) {

try {

checkStockStmt.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (updateStockStmt != null) {

try {

updateStockStmt.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (insertOrderStmt != null) {

try {

insertOrderStmt.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (conn != null) {

try {

conn.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

}

}

Explanation

1. Load the JDBC Driver: Register the JDBC driver.

2. Establish a Connection: Connect to the e-commerce database.

3. Disable Auto-commit: Disable auto-commit mode to manage transactions manually.

4. Check Stock: Ensure there is enough stock for the order.

5. Update Stock: Deduct the ordered quantity from the product's inventory.

6. Insert Order: Create an order record with the user ID, product ID, and quantity.

7. Commit or Rollback: Commit the transaction if all operations succeed, or roll back if any operation fails.

8. Clean up Resources: Ensure all resources are closed properly to avoid memory leaks.

This code ensures that the transaction is atomic, consistent, isolated, and durable, adhering to the ACID properties of transactions.