Understanding Transaction Isolation Levels in DBMS

The complete project is available on GitHub This project was created using JetBrains Junie with only 4 prompts, which are included in the git repository under the prompts directory.

1. Introduction

Database transactions are a fundamental concept in database management systems (DBMS) that ensure data integrity and consistency. A transaction is a sequence of operations performed as a single logical unit of work. The ACID properties (Atomicity, Consistency, Isolation, Durability) govern database transactions, with Isolation being particularly important for concurrent operations.

In this article, we'll explore transaction isolation levels in detail, focusing on:

- The anomalies that can occur without proper isolation
- How different isolation levels prevent these anomalies
- How to implement and manage transactions in Spring applications
- Optimistic Concurrency Control (OCC) as an alternative approach

We'll use a banking application as our example, demonstrating how transaction isolation levels affect operations like account balance updates and money transfers. You'll be able to follow along with practical examples using MySQL and a Spring Boot application.

1.1. Understanding the Banking Scenario

Our example application models a simple banking system where:

- Each customer has one or more bank accounts
- Accounts can have deposits and withdrawals (transactions)
- Money can be transferred between accounts
- Multiple users might access the same account simultaneously

This scenario is perfect for demonstrating transaction isolation issues because:

1. Banking operations require strict data consistency (you don't want money to disappear or be created)
2. Concurrent access is common (multiple systems might access the same account)
3. Different operations have different isolation requirements (reading an account balance vs. transferring money)

In our model, we have two main entities:

- `Account`: Represents a bank account with properties like account number, owner name, and balance
- `Transaction`: Represents money movement (deposits, withdrawals, transfers) with properties like amount, type, and timestamp

Without proper transaction isolation, this banking system could face serious issues like: - Showing incorrect balances (dirty reads) - Processing transfers based on outdated balances (non-repeatable reads) - Missing new accounts in reports (phantom reads)

Let's explore how different isolation levels help prevent these problems.

2. Transaction Isolation Levels Explained

2.1. Understanding Transaction Anomalies

Before diving into isolation levels, let's understand the anomalies that can occur when multiple transactions run concurrently without proper isolation:

2.1.1. Dirty Reads

A dirty read occurs when a transaction reads data that has been modified by another transaction that has not yet been committed. If the modifying transaction rolls back, the reading transaction has read invalid data.

2.1.2. Non-repeatable Reads

A non-repeatable read occurs when a transaction reads the same row twice and gets different values each time. This happens when another transaction modifies the row between the first and second read.

2.1.3. Phantom Reads

A phantom read occurs when a transaction re-executes a query that returns a set of rows satisfying a condition, and finds that the set of rows has changed due to another transaction that committed during the execution of the first transaction.

2.2. Isolation Levels and Their Characteristics

Database systems provide different isolation levels to prevent these anomalies:

Isolation Level	Dirty Reads	Non-repeatable Reads	Phantom Reads	Description
READ UNCOMMITTED	Possible	Possible	Possible	Lowest isolation level. Transactions can see uncommitted changes made by other transactions.
READ COMMITTED	Prevented	Possible	Possible	Transactions can only see committed changes made by other transactions.
REPEATABLE READ	Prevented	Prevented	Possible	Transactions see a consistent snapshot of the data as it was at the beginning of the transaction.
SERIALIZABLE	Prevented	Prevented	Prevented	Highest isolation level. Transactions are completely isolated from each other.

2.3. Demonstrating Isolation Levels with MySQL

Let's see these isolation levels in action using MySQL. We'll use a banking scenario with accounts and transactions.

2.3.1. Setting Up the Database

First, let's connect to MySQL and create our database:

```
mysql -u junie -p
Enter password: junie

CREATE DATABASE IF NOT EXISTS isolation_levels;
USE isolation_levels;

-- Create the accounts table
CREATE TABLE accounts (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    account_number VARCHAR(50) NOT NULL UNIQUE,
    owner_name VARCHAR(100) NOT NULL,
    balance DECIMAL(10, 2) NOT NULL,
    version BIGINT DEFAULT 0
);

-- Create the transactions table
CREATE TABLE transactions (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    account_id BIGINT NOT NULL,
    amount DECIMAL(10, 2) NOT NULL,
    description VARCHAR(255) NOT NULL,
    timestamp DATETIME NOT NULL,
    type VARCHAR(10) NOT NULL,
    FOREIGN KEY (account_id) REFERENCES accounts(id)
```

```
);

-- Insert sample accounts
INSERT INTO accounts (account_number, owner_name, balance) VALUES
('ACC001', 'John Doe', 1000.00),
('ACC002', 'Jane Smith', 2000.00);

-- Insert sample transactions
INSERT INTO transactions (account_id, amount, description, timestamp, type) VALUES
((SELECT id FROM accounts WHERE account_number = 'ACC001'), 500.00, 'Initial deposit',
NOW(), 'CREDIT'),
((SELECT id FROM accounts WHERE account_number = 'ACC001'), 200.00, 'ATM withdrawal',
NOW(), 'DEBIT'),
((SELECT id FROM accounts WHERE account_number = 'ACC002'), 1000.00, 'Salary deposit',
NOW(), 'CREDIT'),
((SELECT id FROM accounts WHERE account_number = 'ACC002'), 300.00, 'Bill payment',
NOW(), 'DEBIT');
```

2.3.2. Demonstrating Dirty Reads

To demonstrate dirty reads, we need to set the isolation level to READ UNCOMMITTED:

Session 1

```
-- Start a new session and set isolation level
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;

-- Check initial balance
SELECT * FROM accounts WHERE account_number = 'ACC001';
```

Session 2

```
-- Start another session
START TRANSACTION;

-- Update the balance but don't commit yet
UPDATE accounts SET balance = balance + 500 WHERE account_number = 'ACC001';

-- Don't commit yet!
```

Session 1 (continued)

```
-- Read the balance again - will see the uncommitted change (dirty read)
SELECT * FROM accounts WHERE account_number = 'ACC001';
```

Session 2 (continued)

```
-- Now rollback the transaction
ROLLBACK;
```

Session 1 (continued)

```
-- Read again - the balance is back to the original value
SELECT * FROM accounts WHERE account_number = 'ACC001';

-- End transaction
COMMIT;
```

In this example, Session 1 reads a value that was modified but not committed by Session 2. When Session 2 rolls back, the value read by Session 1 becomes invalid - this is a dirty read.

This problem is resolved by moving to the next isolation level: READ COMMITTED. By using READ COMMITTED isolation level, transactions will only see data that has been committed by other transactions, preventing dirty reads entirely.

2.3.3. Demonstrating Non-repeatable Reads

To demonstrate non-repeatable reads, we'll use READ COMMITTED isolation level:

Session 1

```
-- Set isolation level
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;

-- Read the balance
SELECT * FROM accounts WHERE account_number = 'ACC001';
```

Session 2

```
-- Start another session
START TRANSACTION;

-- Update the balance
UPDATE accounts SET balance = balance + 1000 WHERE account_number = 'ACC001';

-- Commit the change
COMMIT;
```

Session 1 (continued)

```
-- Read the balance again - will see the committed change (non-repeatable read)
SELECT * FROM accounts WHERE account_number = 'ACC001';
```

```
-- End transaction
COMMIT;
```

In this example, Session 1 reads the same row twice but gets different values because Session 2 committed a change in between the reads. This is a non-repeatable read, which can lead to inconsistent data processing within a single transaction.

This problem is resolved by moving to the next isolation level: REPEATABLE READ. By using REPEATABLE READ isolation level, transactions will see a consistent snapshot of the data as it was at the beginning of the transaction, ensuring that repeated reads of the same data will yield the same results throughout the transaction.

2.3.4. Demonstrating Phantom Reads

To demonstrate phantom reads, we'll use REPEATABLE READ isolation level:

Session 1

```
-- Set isolation level
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;

-- Read accounts with balance > 1000
SELECT * FROM accounts WHERE balance > 1000;
```

Session 2

```
-- Start another session
START TRANSACTION;

-- Insert a new account with balance > 1000
INSERT INTO accounts (account_number, owner_name, balance)
VALUES ('ACC004', 'New User', 5000);

-- Commit the change
COMMIT;
```

Session 1 (continued)

```
-- Read accounts with balance > 1000 again
-- In REPEATABLE READ, you won't see the new account (no phantom read)
SELECT * FROM accounts WHERE balance > 1000;

-- But if you explicitly request fresh data with a new transaction:
COMMIT;
START TRANSACTION;
SELECT * FROM accounts WHERE balance > 1000;
-- Now you'll see the new account
```

```
-- End transaction
COMMIT;
```

In REPEATABLE READ isolation level, MySQL prevents phantom reads within the same transaction for most operations. However, there are edge cases where phantom reads can still occur, particularly with range queries and inserts. Also, if you start a new transaction, you'll see the new data.

While REPEATABLE READ in MySQL provides strong protection against phantom reads (better than the SQL standard requires), the only isolation level that fully guarantees protection against phantom reads in all database systems is SERIALIZABLE.

2.3.5. Preventing All Anomalies with SERIALIZABLE

To prevent all anomalies, we can use SERIALIZABLE isolation level:

Session 1

```
-- Set isolation level
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;

-- Read accounts with balance > 1000
SELECT * FROM accounts WHERE balance > 1000;
```

Session 2

```
-- Start another session with SERIALIZABLE
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;

-- Try to insert a new account (this will wait for Session 1 to complete)
INSERT INTO accounts (account_number, owner_name, balance)
VALUES ('ACC005', 'Another User', 6000);

-- This won't complete until Session 1 commits or rolls back
```

Session 1 (continued)

```
-- Read accounts with balance > 1000 again
-- You won't see any changes
SELECT * FROM accounts WHERE balance > 1000;

-- End transaction
COMMIT;
```

Session 2 (continued)

```
-- Now the insert will complete
COMMIT;
```

In SERIALIZABLE isolation level, transactions are completely isolated from each other, preventing all anomalies but potentially reducing concurrency. SERIALIZABLE achieves this by effectively making transactions run one after another (serially) rather than concurrently when they might conflict.

The key differences between REPEATABLE READ and SERIALIZABLE are:

1. REPEATABLE READ allows transactions to execute concurrently even when they might conflict, but takes snapshots to ensure consistent reads
2. SERIALIZABLE detects potential conflicts and forces transactions to wait, ensuring complete isolation
3. REPEATABLE READ may allow phantom reads in some edge cases (especially in databases other than MySQL)
4. SERIALIZABLE guarantees no phantom reads under any circumstances

2.4. Choosing the Right Isolation Level: Performance vs. Consistency

Each isolation level represents a trade-off between data consistency and performance:

2.4.1. READ UNCOMMITTED

- **Performance Impact**: Minimal - highest throughput of all isolation levels
- **When to Use**: Rarely used in production; might be suitable for reporting queries where approximate results are acceptable
- **Risks**: High risk of inconsistent data due to dirty reads

2.4.2. READ COMMITTED

- **Performance Impact**: Low - good performance with reasonable consistency
- **When to Use**: General-purpose operations where dirty reads must be avoided but some inconsistency is tolerable
- **Risks**: Non-repeatable reads and phantom reads can still occur
- **Common Use Cases**: Reading account information, generating non-critical reports

2.4.3. REPEATABLE READ

- **Performance Impact**: Moderate - some overhead for maintaining read consistency
- **When to Use**: When consistent reads within a transaction are important

- **Risks**: Potential for phantom reads in some database systems (less so in MySQL)
- **Common Use Cases**: Financial calculations, balance transfers where consistent reads are critical

2.4.4. SERIALIZABLE

- **Performance Impact**: High - significant reduction in concurrency
- **When to Use**: When absolute data consistency is required, even at the cost of performance
- **Risks**: Deadlocks and timeouts more likely; reduced throughput under high concurrency
- **Common Use Cases**: Critical financial transactions, regulatory compliance scenarios

In our banking application example, you might use different isolation levels for different operations: - READ COMMITTED for viewing account details - REPEATABLE READ for calculating interest - SERIALIZABLE for executing money transfers between accounts

2.5. Cleaning Up the Database

Before moving on to the next section, let's clean up our database by dropping the tables we created:

```
-- Drop tables (transactions first due to foreign key constraint)
DROP TABLE IF EXISTS transactions;
DROP TABLE IF EXISTS accounts;

-- Verify tables are gone
SHOW TABLES;
```

This ensures we start with a clean slate for the Spring application in the next section.

3. Spring Transaction Management

Spring provides a comprehensive transaction management framework that simplifies working with transactions in Java applications. Let's explore how Spring manages transactions and how to configure different isolation levels.

3.1. Spring Transaction Annotations and How They Work

Spring's transaction management is primarily annotation-based, with `@Transactional` being the most important annotation. When you annotate a method with `@Transactional`, Spring creates a proxy around your object that intercepts calls to the annotated methods:

```
@Transactional(isolation = Isolation.READ_COMMITTED)
public void transferMoney(String fromAccount, String toAccount, BigDecimal amount) {
    // Transaction logic here
```

```
}
```

When a client calls this method, here's what happens behind the scenes:

1. The proxy intercepts the method call
2. The proxy starts a new transaction (or joins an existing one, depending on the propagation setting)
3. The proxy sets the appropriate isolation level, timeout, and read-only attributes
4. The proxy invokes the actual method
5. If the method completes normally, the proxy commits the transaction
6. If the method throws an exception, the proxy may roll back the transaction (depending on the exception type and rollback settings)

This declarative approach means you don't need to write explicit transaction management code in your business methods. Spring handles all the transaction boundaries, commit, and rollback operations automatically.

The `@Transactional` annotation can be applied at both class and method levels, with method-level annotations overriding class-level ones. This allows you to set default transaction behavior for all methods in a class while customizing specific methods as needed.

3.2. Key Transaction Attributes

Spring's `@Transactional` annotation supports several attributes that give you fine-grained control over transaction behavior:

The `isolation` attribute sets the transaction isolation level for the method. As we've discussed earlier, this determines how the transaction interacts with other concurrent transactions. For example, setting `isolation = Isolation.SERIALIZABLE` ensures the highest level of isolation but may impact performance under high concurrency.

The `propagation` attribute defines how transactions relate to each other when methods call other methods. The default value, `Propagation.REQUIRED`, means that the method will use an existing transaction if one exists, or create a new one if none exists. Other options like `Propagation.REQUIRES_NEW` always create a new transaction, suspending any existing one.

The `timeout` attribute specifies how long (in seconds) the transaction may run before timing out. This is useful for preventing long-running transactions from holding locks for extended periods. If a transaction exceeds this time limit, Spring will automatically roll it back.

The `readOnly` attribute is a hint to the transaction infrastructure that the transaction will not modify any data. This can enable optimizations in some databases and ORM frameworks. For example, Hibernate can skip dirty checking for read-only transactions, improving performance.

The `rollbackFor` and `noRollbackFor` attributes allow you to specify which exceptions should cause a transaction to roll back or not roll back. By default, runtime exceptions trigger a rollback while checked exceptions do not. These attributes let you customize this behavior for specific exception

types.

3.3. Configuring Transaction Isolation Levels

Here's how to configure different isolation levels in Spring:

```
// READ UNCOMMITTED - allows dirty reads
@Transactional(isolation = Isolation.READ_UNCOMMITTED)
public Account getAccountReadUncommitted(String accountNumber) {
    return accountRepository.findByAccountNumber(accountNumber);
}

// READ COMMITTED - prevents dirty reads
@Transactional(isolation = Isolation.READ_COMMITTED)
public Account getAccountReadCommitted(String accountNumber) {
    return accountRepository.findByAccountNumber(accountNumber);
}

// REPEATABLE READ - prevents dirty and non-repeatable reads
@Transactional(isolation = Isolation.REPEATABLE_READ)
public Account getAccountRepeatableRead(String accountNumber) {
    return accountRepository.findByAccountNumber(accountNumber);
}

// SERIALIZABLE - prevents all anomalies
@Transactional(isolation = Isolation.SERIALIZABLE)
public Account getAccountSerializable(String accountNumber) {
    return accountRepository.findByAccountNumber(accountNumber);
}
```

3.4. Transaction Propagation

Transaction propagation defines how transactions relate to each other when methods are called within a transaction context:

```
// REQUIRED - Uses existing transaction or creates a new one
@Transactional(propagation = Propagation.REQUIRED)
public void methodA() {
    // Transaction logic
    methodB(); // Will use the same transaction
}

// REQUIRES_NEW - Always creates a new transaction
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void methodB() {
    // Always runs in a new transaction
}
```

3.5. Programmatic Transaction Management

In addition to annotations, Spring also supports programmatic transaction management:

```
@Autowired
private PlatformTransactionManager transactionManager;

public void complexTransactionLogic() {
    TransactionTemplate template = new TransactionTemplate(transactionManager);
    template.setIsolationLevel(TransactionDefinition.ISOLATION_SERIALIZABLE);

    template.execute(status -> {
        // Transaction logic here
        return null;
    });
}
```

3.6. Handling Transaction Exceptions

Spring provides a rich exception hierarchy for transaction management:

```
@Transactional
public void transferWithExceptionHandling(String fromAccount, String toAccount,
BigDecimal amount) {
    try {
        // Transaction logic
    } catch (DataAccessException e) {
        // Handle database-related exceptions
        throw new ServiceException("Database error during transfer", e);
    } catch (Exception e) {
        // Handle other exceptions
        throw new ServiceException("Error during transfer", e);
    }
}
```

3.7. Optimistic vs. Pessimistic Locking

Beyond isolation levels, Spring Data JPA provides two additional concurrency control mechanisms: optimistic and pessimistic locking. These approaches address the fundamental problem of concurrent data access from different perspectives.

3.7.1. Optimistic Locking: Trust but Verify

Optimistic locking operates on the assumption that conflicts are rare. Rather than locking resources preemptively, it allows multiple transactions to proceed simultaneously and checks for conflicts only at commit time. This approach is called "optimistic" because it optimistically assumes that

most transactions won't conflict.

Here's how optimistic locking works in Spring Data JPA:

1. A version field is added to the entity class
2. When an entity is read, its current version is recorded
3. When an entity is updated, the version is checked against the database
4. If the version matches, the update proceeds and the version is incremented
5. If the version doesn't match, an OptimisticLockingFailureException is thrown

```
@Entity
public class Account {
    @Id
    @GeneratedValue
    private Long id;

    private String accountNumber;
    private BigDecimal balance;

    @Version
    private Long version; // This field enables optimistic locking

    // Getters and setters
}
```

Optimistic locking is particularly well-suited for: - High-read, low-write scenarios - Applications where conflicts are infrequent - User interfaces that can handle conflict resolution (e.g., by showing a "someone else has modified this data" message) - Environments where holding database locks for extended periods is problematic

3.7.2. Pessimistic Locking: Lock First, Ask Questions Later

Pessimistic locking takes the opposite approach: it assumes conflicts are likely and prevents them by acquiring locks on resources before they're accessed. This approach is "pessimistic" because it assumes the worst-case scenario and takes preventive measures.

Spring Data JPA supports several types of pessimistic locks:

1. **PESSIMISTIC_READ**: Acquires a shared lock that prevents other transactions from updating or deleting the data but allows them to read it
2. **PESSIMISTIC_WRITE**: Acquires an exclusive lock that prevents other transactions from reading, updating, or deleting the data
3. **PESSIMISTIC_FORCE_INCREMENT**: Similar to PESSIMISTIC_WRITE but also increments the version field

Here's how to use pessimistic locking in a repository:

```
public interface AccountRepository extends JpaRepository<Account, Long> {
    // Acquire a pessimistic write lock when finding an account
    @Lock(LockModeType.PESSIMISTIC_WRITE)
    @Query("SELECT a FROM Account a WHERE a.accountNumber = :accountNumber")
    Optional<Account> findByAccountNumberWithPessimisticLock(@Param("accountNumber")
String accountNumber);

    // Acquire a pessimistic read lock
    @Lock(LockModeType.PESSIMISTIC_READ)
    @Query("SELECT a FROM Account a WHERE a.id = :id")
    Optional<Account> findByIdWithPessimisticReadLock(@Param("id") Long id);
}
```

Pessimistic locking is well-suited for: - High-contention scenarios where conflicts are frequent - Critical operations where conflicts must be prevented rather than resolved - Short-lived transactions where lock holding time is minimal - Situations where the cost of conflict resolution is higher than the cost of locking

3.7.3. Choosing Between Optimistic and Pessimistic Locking

The choice between optimistic and pessimistic locking depends on your application's characteristics:

- **Optimistic locking** provides better concurrency and scalability but requires conflict resolution logic
- **Pessimistic locking** prevents conflicts but reduces concurrency and can lead to deadlocks

In our banking application example, you might use: - Optimistic locking for updating customer information or account settings - Pessimistic locking for critical operations like money transfers or balance updates

4. Conclusion and Optimistic Concurrency Control

4.1. Summary of Isolation Levels

Throughout this article, we've explored the four standard transaction isolation levels and their characteristics. Let's summarize what we've learned:

READ UNCOMMITTED is the lowest isolation level, allowing all types of anomalies (dirty reads, non-repeatable reads, and phantom reads) to occur. While it provides maximum concurrency and performance, it does so at the cost of data consistency. In our banking application, this level would be too risky for most operations as it could lead to incorrect balance calculations or duplicate transfers.

READ COMMITTED prevents dirty reads by ensuring that transactions only see committed data

from other transactions. This provides a reasonable balance between performance and consistency for many applications. However, it still allows non-repeatable reads and phantom reads, which can be problematic for complex financial calculations or reports that need consistent data throughout a transaction.

REPEATABLE READ addresses the non-repeatable read issue by ensuring that if a transaction reads a row once, it will get the same data if it reads that row again, regardless of changes made by other transactions. This is achieved by taking a snapshot of the data at the beginning of the transaction. While this level prevents both dirty reads and non-repeatable reads, it may still allow phantom reads in some database systems (though MySQL's implementation provides stronger guarantees).

SERIALIZABLE, the highest isolation level, prevents all types of anomalies by essentially making transactions run one after another when they might conflict. This provides the strongest consistency guarantees but at the cost of reduced concurrency and potential performance issues under high load. In our banking application, we might reserve this level for the most critical operations like large money transfers or end-of-day reconciliation processes.

4.2. Optimistic Concurrency Control (OCC)

Optimistic Concurrency Control represents a fundamentally different approach to managing concurrent access to data compared to traditional isolation levels. While isolation levels focus on controlling how transactions interact with each other through database mechanisms, OCC takes a more application-centric approach.

At its core, OCC operates on the principle that conflicts between concurrent transactions are relatively rare in most applications. Instead of preventing conflicts through locks or isolation mechanisms, OCC allows transactions to proceed without restrictions but verifies at commit time that no conflicts have occurred.

This approach works by tracking the state of data when it's read and then checking whether that state has changed when the transaction attempts to commit. If the data hasn't changed, the commit succeeds; if it has changed, the transaction is rolled back and can be retried.

The primary advantage of OCC is that it allows for high concurrency without the overhead of maintaining locks or complex isolation mechanisms. This makes it particularly well-suited for modern web applications with many users but relatively few conflicts over the same data.

In our banking application, OCC might be used for operations like updating account settings or personal information, where conflicts are unlikely but still need to be handled correctly if they occur. For example, if two bank employees try to update a customer's address simultaneously, OCC would allow both operations to proceed but ensure that the second update doesn't accidentally overwrite the first one without acknowledging the conflict.

4.2.1. Implementing OCC in Spring

Spring Data JPA makes implementing OCC easy with the `@Version` annotation:

```
@Entity
public class Account {
```

```
    @Id
    @GeneratedValue
    private Long id;

    private String accountNumber;
    private BigDecimal balance;

    @Version
    private Long version;

    // Getters and setters
}
```

When multiple transactions try to update the same entity concurrently, Spring will throw an `OptimisticLockingFailureException` if a conflict is detected.

4.2.2. Handling OCC Conflicts

To handle OCC conflicts, you can catch and handle the exception:

```
@Service
public class AccountService {
    @Autowired
    private AccountRepository repository;

    @Transactional
    public void updateBalanceWithRetry(Long accountId, BigDecimal newBalance, int
maxRetries) {
        int retries = 0;
        while (retries < maxRetries) {
            try {
                Account account = repository.findById(accountId).orElseThrow();
                account.setBalance(newBalance);
                repository.save(account);
                return; // Success
            } catch (OptimisticLockingFailureException e) {
                retries++;
                if (retries >= maxRetries) {
                    throw new ServiceException("Failed to update after " + maxRetries
+ " attempts");
                }
                // Wait before retrying
                try {
                    Thread.sleep(100);
                } catch (InterruptedException ie) {
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
```

```
}
```

4.3. Choosing the Right Approach

Selecting the appropriate concurrency control strategy for your application requires careful consideration of several factors. There's no one-size-fits-all solution, and the best approach often involves using different strategies for different parts of your application.

First, assess your application's requirements for data consistency. Some operations, like viewing account balances, might tolerate slight inconsistencies, while others, like executing financial transactions, require absolute consistency. Map these requirements to the appropriate isolation levels or concurrency control mechanisms.

Next, evaluate the performance implications of your choices. Higher isolation levels and pessimistic locking provide stronger consistency guarantees but can significantly impact performance under high concurrency. Consider whether your application can handle the potential performance overhead or if a more optimistic approach would be better.

The nature of your data access patterns also plays a crucial role. Analyze the likelihood of conflicts in your application by considering questions like: How often will multiple users try to update the same data simultaneously? Are certain entities more contended than others? This analysis will help you decide between optimistic and pessimistic approaches.

Finally, there's no substitute for thorough testing under realistic concurrent loads. Theoretical understanding is important, but real-world behavior can sometimes be surprising. Set up test scenarios that mimic your expected production load and verify that your chosen approach maintains both consistency and acceptable performance.

In our banking application example, we might use a mixed approach: - READ COMMITTED isolation for general account viewing - REPEATABLE READ for generating financial reports - SERIALIZABLE for critical money transfers - Optimistic locking for customer profile updates - Pessimistic locking for balance updates

By understanding the trade-offs between different concurrency control approaches and carefully matching them to your application's needs, you can build systems that are both consistent and performant.

5. Building and Running the Application

5.1. Prerequisites

- Java 17 or higher
- Maven 3.6 or higher
- MySQL 8.0 or higher

5.2. Database Setup

Create the database and user:

```
CREATE DATABASE isolation_levels;
CREATE USER 'junie'@'localhost' IDENTIFIED BY 'junie';
GRANT ALL PRIVILEGES ON isolation_levels.* TO 'junie'@'localhost';
FLUSH PRIVILEGES;
```

5.3. Building the Application

```
# Clone the repository
git clone https://github.com/jetbrains/isolation-levels-demo.git
cd isolation-levels-demo

# Build the application
mvn clean package
```

5.4. Running the Application

```
# Run the application
java -jar target/isolation_levels-1.0-SNAPSHOT.jar
```

The application will start on port 8080 and automatically create the necessary tables and sample data.

5.5. API Endpoints

The application provides several REST endpoints to demonstrate transaction isolation levels:

- `GET /api/accounts`: Get all accounts
- `GET /api/accounts/{accountNumber}?isolationLevel=READ_COMMITTED`: Get an account with specified isolation level
- `PUT /api/accounts/{accountNumber}/balance`: Update an account's balance
- `POST /api/transactions/transfer`: Transfer money between accounts

5.6. Example API Calls

```
# Get all accounts
curl -X GET http://localhost:8080/api/accounts

# Get an account with READ_UNCOMMITTED isolation level
```

```
curl -X GET
"http://localhost:8080/api/accounts/ACC001?isolationLevel=READ_UNCOMMITTED"

# Update an account's balance
curl -X PUT http://localhost:8080/api/accounts/ACC001/balance \
  -H "Content-Type: application/json" \
  -d '{"balance": "1500.00"}'

# Transfer money between accounts
curl -X POST http://localhost:8080/api/transactions/transfer \
  -H "Content-Type: application/json" \
  -d '{"fromAccountNumber": "ACC001", "toAccountNumber": "ACC002", "amount":
"500.00"}'
```

6. Conclusion

Understanding transaction isolation levels is crucial for developing robust database applications. By choosing the appropriate isolation level, you can balance data consistency with performance requirements.

Spring's transaction management framework provides a powerful and flexible way to implement transactions in Java applications, with support for different isolation levels, propagation behaviors, and concurrency control mechanisms.

Whether you choose traditional isolation levels or optimistic concurrency control, the key is to understand the trade-offs and select the approach that best fits your application's needs.

7. License

This project is licensed under the MIT License - see the LICENSE file for details.

The MIT License is a permissive license that allows you to use, modify, distribute, and sublicense the code, even for commercial purposes, provided that you include the original copyright notice and license terms in any copy of the software/source code.