# Transactions definition

Transactionality refers to the ability to perform a group of operations as a single unit of work that must either be completed entirely or not at all. In the context of database operations, transactions ensure data integrity by providing features like Atomicity, Consistency, Isolation, and Durability (ACID properties).

The ACID properties are a set of four properties that ensure the reliability and consistency of transactions in database systems. ACID stands for Atomicity, Consistency, Isolation, and Durability. Let's delve into each property with examples:

1. \*\*Atomicity\*\*:

Atomicity ensures that either all operations within a transaction are successfully completed, or none of them are. In other words, a transaction is treated as a single unit of operation, and it either succeeds completely or fails completely.

\*\*Example\*\*: Consider a banking transaction where you transfer money from one account to another. The transaction involves two steps: deducting the amount from one account and crediting it to another. If the system fails after deducting from one account but before crediting to the other, atomicity ensures that the deducted amount is rolled back, and the transaction is aborted, leaving both accounts unchanged.

2. \*\*Consistency\*\*:

Consistency ensures that the database remains in a valid state before and after the transaction. It guarantees that all data integrity constraints are enforced, and the database transitions from one valid state to another valid state.

\*\*Example\*\*: In a university database, if a student enrolls in a course, consistency ensures that the student's enrollment does not violate any constraints such as prerequisite courses, maximum credit limits, or available seats in the course. If any of these constraints are violated, the transaction is rolled back to maintain consistency.

3. \*\*Isolation\*\*:

Isolation ensures that the concurrent execution of transactions does not result in interference or inconsistency. Each transaction appears to execute in isolation, without being affected by other concurrently executing transactions.

\*\*Example\*\*: Consider two transactions T1 and T2. T1 transfers $100 from account A to account B, and T2 transfers $50 from account B to account C. Isolation ensures that if these transactions are executed concurrently, the final result should be the same as if they were executed sequentially, without any interference. For instance, even if T1 completes its transfer from A to B, T2 should not see this intermediate state until T1 commits its changes.

4. \*\*Durability\*\*:

Durability ensures that once a transaction is committed, its effects persist even in the event of system failures. Once a transaction is successfully completed, the changes made by that transaction are permanently stored in the database and cannot be lost.

\*\*Example\*\*: Suppose a customer places an order on an e-commerce website, and the order is successfully processed and stored in the database. Even if the system crashes immediately after the order is placed, durability ensures that the order information remains intact and is not lost when the system is restarted.

In summary, the ACID properties provide a framework for ensuring the reliability, consistency, and durability of transactions in database systems, which are crucial for maintaining data integrity and system reliability.

Example :

import org.springframework.transaction.annotation.Isolation;

@Transactional(isolation = Isolation.READ\_COMMITTED)

public void performTransactionalOperation() {

// Logic within the transaction

}

Transactional Isolation :

**### 1. READ UNCOMMITTED:**

- \*\*Real-Life Example\*\*: Imagine you're booking tickets for a popular concert online. In READ UNCOMMITTED isolation level:

- Another user is currently updating the available tickets for the same concert.

- Even though their transaction is not yet committed, you can still see the updated ticket count while booking.

- However, if their transaction rolls back (due to cancellation or error), your booking might end up with an incorrect ticket count, leading to inconsistencies.

**### 2. READ COMMITTED:**

- \*\*Real-Life Example\*\*: Continuing with the concert ticket booking scenario:

- In READ COMMITTED isolation level:

- You only see the updated ticket count if the other user's transaction has been committed.

- This prevents you from booking tickets based on uncommitted changes, avoiding potential inconsistencies.

- However, if the other user commits their transaction between the time you check the ticket count and proceed to book, the available tickets may have changed, resulting in a non-repeatable read.

**### 3. REPEATABLE READ:**

- \*\*Real-Life Example\*\*: Consider you're managing inventory for an online store:

- In REPEATABLE READ isolation level:

- When you start updating the inventory for a particular product, the database ensures that the inventory remains consistent throughout your transaction.

- Even if another user adds new products to the inventory during your transaction, you won't see those new products until your transaction completes.

- However, if the other user updates the quantity of existing products in the inventory, you might encounter phantom reads as the inventory changes during your transaction.

**### 4. SERIALIZABLE:**

- \*\*Real-Life Example\*\*: Let's say you're transferring funds between two bank accounts:

- In SERIALIZABLE isolation level:

- Your transaction is completely isolated from other transactions.

- Even if another user is concurrently transferring funds from the same accounts, their transaction won't affect yours, and vice versa.

- This ensures the highest level of consistency, but it may lead to decreased concurrency and performance due to locking.

These real-life examples demonstrate how different isolation levels in database transactions can affect data consistency and concurrency in various scenarios. It's essential to understand these behaviors and choose the appropriate isolation level based on your application's requirements to balance consistency and performance effectively.

## i dont understand non-repeatable read. can you please explain this more clearly?

Of course! Let's dive deeper into the concept of non-repeatable reads with a clear explanation and an example:

### Non-Repeatable Read:

Non-repeatable read is a phenomenon that occurs when a transaction reads the same data twice within the same transaction but gets different results each time. This inconsistency arises because another transaction modifies the data between the two reads, leading to different values being retrieved.

#### Explanation:

Imagine a scenario where Transaction A reads a row of data, then Transaction B updates or deletes that same row, and finally Transaction A reads the row again. If Transaction A retrieves different values or finds that the row has been deleted during the second read compared to the first read, it's known as a non-repeatable read.

#### Example:

Let's illustrate this with a simple example involving an online bookstore:

1. \*\*Initial State\*\*:

- Transaction A begins by reading the number of available copies of a specific book from the database. Let's say it finds there are 10 copies available.

2. \*\*Concurrent Transaction\*\*:

- Meanwhile, Transaction B occurs, where someone purchases 5 copies of the same book. The database is updated accordingly, and now there are only 5 copies available.

3. \*\*Second Read by Transaction A\*\*:

- Transaction A continues its process and reads the number of available copies of the book again. However, this time it finds there are only 5 copies available, as Transaction B's changes have been committed.

In this scenario, Transaction A experiences a non-repeatable read because it retrieves different values for the same data (the number of available book copies) within the same transaction. The inconsistency arises because Transaction B modifies the data between Transaction A's two reads.

Non-repeatable reads can occur in database systems with isolation levels lower than SERIALIZABLE, as they allow concurrent transactions to modify data that is being read by other transactions. Higher isolation levels, such as SERIALIZABLE, prevent non-repeatable reads by ensuring complete isolation between transactions.

# Can you please explain repeatable read and phantom read. I didnt get that so please explain it properly with a good example?

Certainly! Let's delve into the concepts of repeatable read and phantom read with clear explanations and examples:

### Repeatable Read:

In a repeatable read scenario, a transaction retrieves the same set of rows multiple times within the same transaction, and the data remains consistent across all reads, regardless of any modifications made by other transactions.

#### Explanation:

Imagine a scenario where Transaction A reads a set of rows from a table, and then Transaction B inserts new rows or updates existing rows in the same table. Despite these changes, when Transaction A reads the same set of rows again later in the transaction, it sees the data as it was at the beginning of the transaction, without being affected by Transaction B's modifications.

#### Example:

Consider a bank application where a user checks their account balance and then performs multiple transactions within the same session:

1. \*\*Initial State\*\*:

- Transaction A begins by checking the user's account balance, which is $1000.

2. \*\*Concurrent Transaction\*\*:

- Meanwhile, Transaction B occurs, where another user transfers $200 to the first user's account, updating the balance to $1200.

3. \*\*Second Read by Transaction A\*\*:

- Transaction A continues and checks the user's account balance again. Despite Transaction B's update, Transaction A still sees the balance as $1000, the value it read at the beginning of the transaction.

In this scenario, Transaction A experiences a repeatable read because it retrieves the same set of rows (the user's account balance) multiple times within the same transaction, and the data remains consistent across all reads, unaffected by concurrent transactions.

### Phantom Read:

A phantom read occurs when a transaction retrieves a set of rows based on a certain condition, and then another transaction inserts new rows that match the same condition. When the first transaction retrieves the same set of rows again later in the transaction, it finds additional rows that were not present in the initial read.

#### Explanation:

Imagine a scenario where Transaction A queries a table based on a certain condition and retrieves a set of rows. Meanwhile, Transaction B inserts new rows into the same table that match the condition. When Transaction A queries the table again later in the transaction, it finds additional rows that were inserted by Transaction B, leading to the phenomenon known as a phantom read.

#### Example:

Consider an online shopping application where a user views a list of available products and then proceeds to checkout:

1. \*\*Initial State\*\*:

- Transaction A queries the database for available products and retrieves a list of 10 products.

2. \*\*Concurrent Transaction\*\*:

- Meanwhile, Transaction B occurs, where a new product is added to the database that matches Transaction A's query criteria.

3. \*\*Second Read by Transaction A\*\*:

- Transaction A re-queries the database for available products. This time, it retrieves a list of 11 products, including the newly added product, even though it was not present in the initial read.

In this scenario, Transaction A experiences a phantom read because it retrieves a set of rows based on a condition, and later, when it retrieves the same set of rows again within the same transaction, it finds additional rows that were not present initially, due to Transaction B's insertion.

# By default which isolation has been used in @transaction?

In Spring's `@Transactional` annotation, if you don't explicitly specify an isolation level, the default isolation level used is \*\*ISOLATION\_DEFAULT\*\*.

The `ISOLATION\_DEFAULT` value means that the default isolation level configured in the underlying database will be used. This typically depends on the default isolation level set by the database management system (DBMS) being used.

For most popular relational databases like MySQL, PostgreSQL, and Oracle, the default isolation level is usually \*\*READ COMMITTED\*\*. However, it's always a good practice to check the specific default isolation level for your chosen database system, as it can vary depending on the configuration and version.

If you need to explicitly set the isolation level in your Spring `@Transactional` annotation, you can do so using the `isolation` attribute, as shown in previous examples.

# Transactional propagation :

In the context of Java Persistence API (JPA) and Hibernate, transaction propagation refers to how a transaction initiated in one method or component is propagated (or carried over) to other methods or components within the same transactional context. This ensures that related database operations are either all committed or all rolled back as a single unit of work.

There are typically several transaction propagation behaviors defined in JPA and Hibernate, such as ***REQUIRED, REQUIRES\_NEW, SUPPORTS, MANDATORY, NEVER, and NESTED.*** Let's discuss some of these with examples:

1. **REQUIRED**: This is the default propagation behavior. If a transaction exists, the method will join that transaction. If no transaction exists, a new transaction will be started.

@Transactional(propagation = Propagation.REQUIRED)

public void parentMethod() {

// This method starts or joins an existing transaction

childMethod();

}

@Transactional(propagation = Propagation.REQUIRED)

public void childMethod() {

// This method will run within the same transaction as the parentMethod

}

1. **REQUIRES\_NEW:** Always creates a new transaction, regardless of any existing transaction. Isolates the method execution from any other ongoing transactions. Commits or rolls back independently of other transactions.

@Transactional(propagation = Propagation.REQUIRED)

public void parentMethod() {

// This method starts a new transaction

childMethod();

}

@Transactional(propagation = Propagation.REQUIRES\_NEW)

public void childMethod() {

// This method will run within its own transaction

}

1. **NESTED**: It's similar to REQUIRES\_NEW but with a twist. We start a new transaction, but this new transaction is nested within the existing transaction.It's like saying, "I'm starting a new group to work on my transfer, but if anything goes wrong within my group, it won't affect the main group's work." So, if our transfer fails, it will roll back independently of the main transaction.

@Transactional(propagation = Propagation.REQUIRED)

public void parentMethod() {

// This method starts a new nested transaction

childMethod();

}

@Transactional(propagation = Propagation.NESTED)

public void childMethod() {

// This method will run within a nested transaction

}

1. **SUPPORTS:** Joins an existing transaction if one is active, otherwise proceeds without a transaction. Cannot create a new transaction. Commits or rolls back based on the existing transaction or lack thereof.

@Transactional(propagation = Propagation.SUPPORTS)

public void readBalance(Account account) {

System.out.println("Balance: " + account.getBalance());

}

1. **MANDATORY:** Requires an existing active transaction. Throws an exception if no transaction is active. Joins and participates in the existing transaction. Commits or rolls back based on the overall outcome.

@Transactional(propagation = Propagation.MANDATORY)

1. **NOT\_SUPPORTED :** Suspends any existing transaction. Proceeds without a transaction. Changes made within the method are committed immediately, not waiting for a transaction commit.

# By default spring boot follows which propagation ?

In Spring Boot, the default transaction propagation behavior is `REQUIRED`. This means that if a method marked with `@Transactional` is called and there's already an ongoing transaction in progress, the method will join that transaction. If there's no ongoing transaction, a new transaction will be started.

So, if you don't explicitly specify the propagation behavior in your `@Transactional` annotation, Spring Boot will use `REQUIRED` as the default behavior.

# What annotations are important in case of transactionality in spring boot ?

In Spring Boot, when working with transaction management, several annotations are important for defining transactional behavior:

1. \*\*@Transactional\*\*: This annotation is used to mark methods or classes where transactions should be applied. When applied at the method level, it specifies that the method is transactional and will be wrapped in a transaction. When applied at the class level, it indicates that all public methods in the class are transactional. This annotation can also be configured with various attributes to control transactional behavior, such as isolation level, propagation behavior, rollback rules, etc.

2. \*\*@EnableTransactionManagement\*\*: This annotation is used at the configuration class level to enable Spring's annotation-driven transaction management capability. It activates Spring's ability to detect `@Transactional` annotations and manage transactions accordingly.

3. \*\*@TransactionalEventListener\*\*: This annotation is used to declare a method as an event listener for transactional events. It allows you to listen to different transactional phases (before commit, after commit, after rollback, etc.) and perform specific actions accordingly.

4. \*\*@Transactional(propagation = Propagation.REQUIRED)\*\*: This is an attribute of the `@Transactional` annotation, which defines the propagation behavior of the transaction. It specifies whether the method should join an existing transaction or start a new one if no transaction exists. Other propagation behaviors include `REQUIRES\_NEW`, `NESTED`, etc.

These annotations are important for managing transactional behavior in Spring Boot applications. They provide a declarative way to define transactional boundaries and manage the transactional aspects of your application's business logic.

# Explain the concept of pessimistic and optimistic locking in the context of transactions.

In the context of transactions and concurrency control, pessimistic and optimistic locking are two different approaches used to manage access to shared resources (usually database records) to ensure data consistency and integrity.

### Pessimistic Locking:

\*\*Concept:\*\*

Pessimistic locking, also known as "lock-based concurrency control," involves acquiring locks on resources before accessing them to prevent other transactions from modifying the same resources concurrently. When a transaction wants to access a resource, it first acquires a lock on that resource, ensuring exclusive access until the lock is released.

\*\*Implementation:\*\*

In databases, pessimistic locking is typically implemented using locks at the database level. For example, when a transaction wants to update a database record, it acquires a write lock on that record, preventing other transactions from reading or writing to the same record until the lock is released.

\*\*Usage:\*\*

Pessimistic locking is commonly used in scenarios where conflicts between concurrent transactions are likely, and data integrity is critical. For example, in banking applications, when transferring funds between accounts, pessimistic locking ensures that only one transaction can update the account balances at a time to prevent inconsistencies.

\*\*Drawbacks:\*\*

- Pessimistic locking can lead to reduced concurrency and performance since it blocks other transactions from accessing resources even if they don't conflict.

### Optimistic Locking:

\*\*Concept:\*\*

Optimistic locking is based on the assumption that conflicts between concurrent transactions are infrequent. Instead of acquiring locks before accessing resources, optimistic locking allows multiple transactions to access and modify resources concurrently. However, before committing changes, each transaction checks whether the resource has been modified by other transactions since it was last read.

\*\*Implementation:\*\*

In databases, optimistic locking is typically implemented using versioning or timestamp mechanisms. Each record in the database is associated with a version number or timestamp. When a transaction reads a record, it also retrieves its version number or timestamp. Before committing changes, the transaction verifies that the version number or timestamp of the record has not changed since it was last read. If the record has been modified by another transaction, the current transaction aborts or retries the operation.

\*\*Usage:\*\*

Optimistic locking is suitable for scenarios where conflicts between concurrent transactions are rare, and the overhead of acquiring and releasing locks for every access is significant. It is often used in web applications where the probability of concurrent updates to the same data is low.

\*\*Drawbacks:\*\*

- Optimistic locking may lead to increased overhead due to the need to check for conflicts before committing changes. In cases of frequent conflicts, it can result in transaction retries and decreased throughput.

### Comparison:

- \*\*Concurrency Control:\*\* Pessimistic locking ensures exclusive access to resources, while optimistic locking allows concurrent access with conflict resolution.

- \*\*Performance:\*\* Pessimistic locking may lead to reduced concurrency and performance due to lock contention, while optimistic locking can incur overhead due to conflict detection.

- \*\*Use Cases:\*\* Pessimistic locking is suitable for scenarios with frequent conflicts and critical data integrity requirements, while optimistic locking is suitable for scenarios with infrequent conflicts and high concurrency requirements.

In summary, pessimistic locking ensures data consistency by preventing concurrent access to resources, while optimistic locking allows concurrent access with conflict resolution mechanisms to maintain data integrity. The choice between pessimistic and optimistic locking depends on the specific requirements of the application and the likelihood of conflicts between concurrent transactions.