Database Transactions

A database transaction is a sequence of one or more operations performed as a single logical unit of work. These operations typically include reading, writing, updating, or deleting data in the database. A transaction ensures that all the operations within it are executed successfully or none at all, thereby maintaining the integrity and consistency of the data.

Key Characteristics of a Transaction

It involves multiple steps or operations.

It must be treated as an indivisible unit of work.

It interacts with the database to perform read/write operations.

It can either succeed completely (commit) or fail entirely (rollback).

Problem Statement

In real-world applications, databases often face challenges such as:

1. Partial Updates :

If a system crash occurs during a transaction, some changes might be applied while others are not, leaving the database in an inconsistent state.

Example: In a banking system, if money is deducted from Account A but not added to Account B due to a crash, the accounts will be out of sync.

2. Concurrency Issues :

When multiple users or processes access and modify the same data simultaneously, conflicts may arise.

Example: Two users trying to book the last available seat on a flight could result in double bookings if transactions are not isolated.

3. Data Loss :

After a system failure, committed changes might vanish, leading to data loss and unreliability.

Example: After confirming an order in an e-commerce system, if the system crashes before saving the order details, the customer's order will be lost.

ACID Properties

1. Atomicity

Ensures that a transaction is treated as a single, indivisible unit. Either all operations in the transaction are completed, or none are applied.

Problem Addressed : Prevents partial updates.

Example : In a money transfer between two bank accounts:

- Deduct $100 from Account A.
- Add $100 to Account B.

If the system crashes after deducting $100 but before adding it to Account B, the entire transaction is rolled back to maintain atomicity.

2. Consistency

Ensures that a transaction brings the database from one valid state to another, adhering to predefined rules and constraints.

Problem Addressed : Maintains data integrity.

Example : A rule states that an account balance cannot be negative. If a withdrawal would result in a negative balance, the transaction is aborted to preserve consistency.

3. Isolation

Ensures that concurrent transactions do not interfere with each other. Each transaction operates as if it is the only one running.

Problem Addressed : Prevents concurrency issues like dirty reads, lost updates, or inconsistent data.

Example : Two users try to update the same record simultaneously. Isolation ensures that one transaction completes before the other begins, avoiding conflicts.

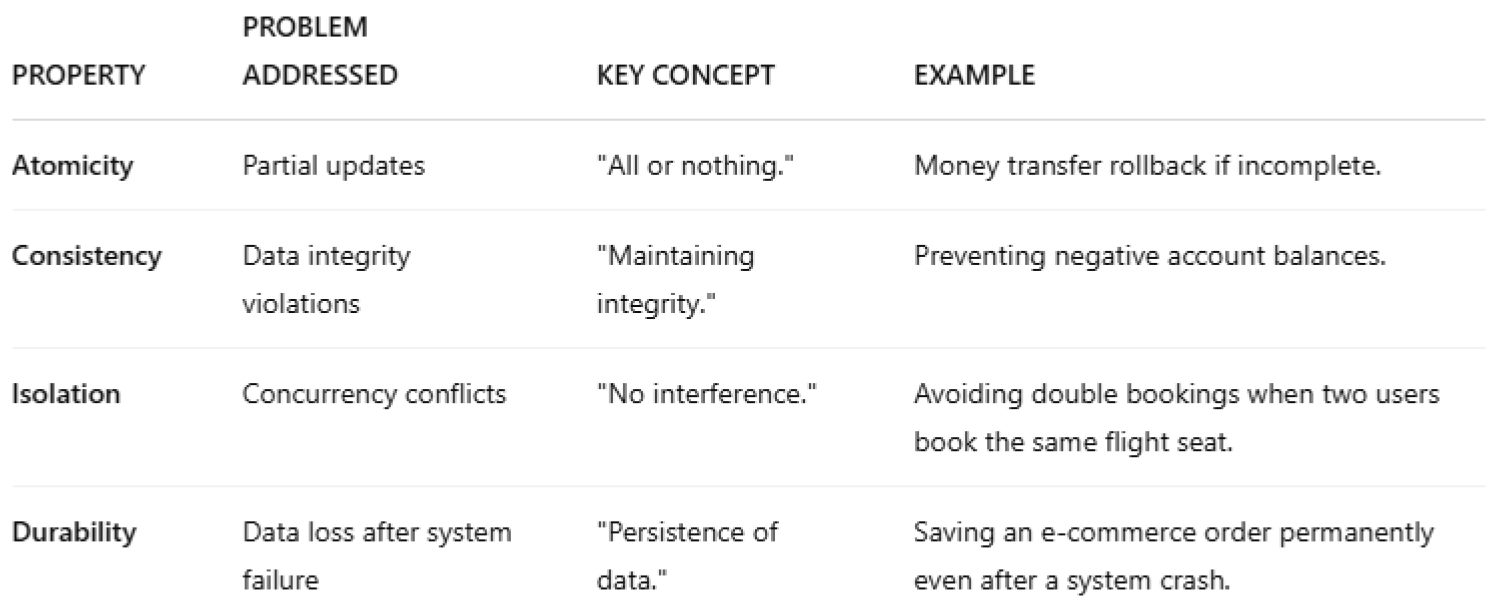
4. Durability

Guarantees that once a transaction is committed, its changes are permanently stored in the database, even if the system crashes afterward

Problem Addressed : Prevents data loss after system failures.

Example : After confirming an order in an e-commerce system, the order details are saved permanently. Even if the server crashes immediately after, the order remains intact.

Summary Table



PROPERTY	PROBLEM ADDRESSED	KEY CONCEPT	EXAMPLE
Atomicity	Partial updates	"All or nothing."	Money transfer rollback if incomplete.
Consistency	Data integrity violations	"Maintaining integrity."	Preventing negative account balances.
Isolation	Concurrency conflicts	"No interference."	Avoiding double bookings when two users book the same flight seat.
Durability	Data loss after system failure	"Persistence of data."	Saving an e-commerce order permanently even after a system crash.

Anomalies in Transactions

Anomalies in transactions, often referred to as concurrency anomalies , are undesirable or incorrect behaviors that can occur when multiple database transactions execute concurrently without proper isolation. These anomalies arise due to improper handling of concurrent access to shared data and can compromise the integrity and consistency of the database.

Types of Transaction Anomalies

1. Dirty Reads

A transaction reads uncommitted (dirty) data written by another transaction that has not yet been committed.

Problem : If the writing transaction is rolled back after the reading transaction has accessed its data, the reading transaction will have worked with invalid or inconsistent data.

Example :

- Transaction T1 updates a record (e.g., changes a user's balance).
- Before T1 commits, Transaction T2 reads the updated value.
- If T1 rolls back, T2 has already read invalid data.

2. Non-Repeatable Reads

A transaction reads the same data multiple times but gets different results because another transaction modifies the data between the reads.

Problem : This inconsistency occurs when the data being read changes during the execution of the transaction.

Example :

- Transaction T1 reads a user's balance (e.g., $5000).
- Meanwhile, Transaction T2 updates the same user's balance to $4000 and commits.
- When T1 reads the balance again, it sees 4000 instead of 5000, leading to confusion or errors.

3. Phantom Reads

A transaction executes the same query multiple times but retrieves different sets of rows because another transaction inserts or deletes rows that match the query criteria.

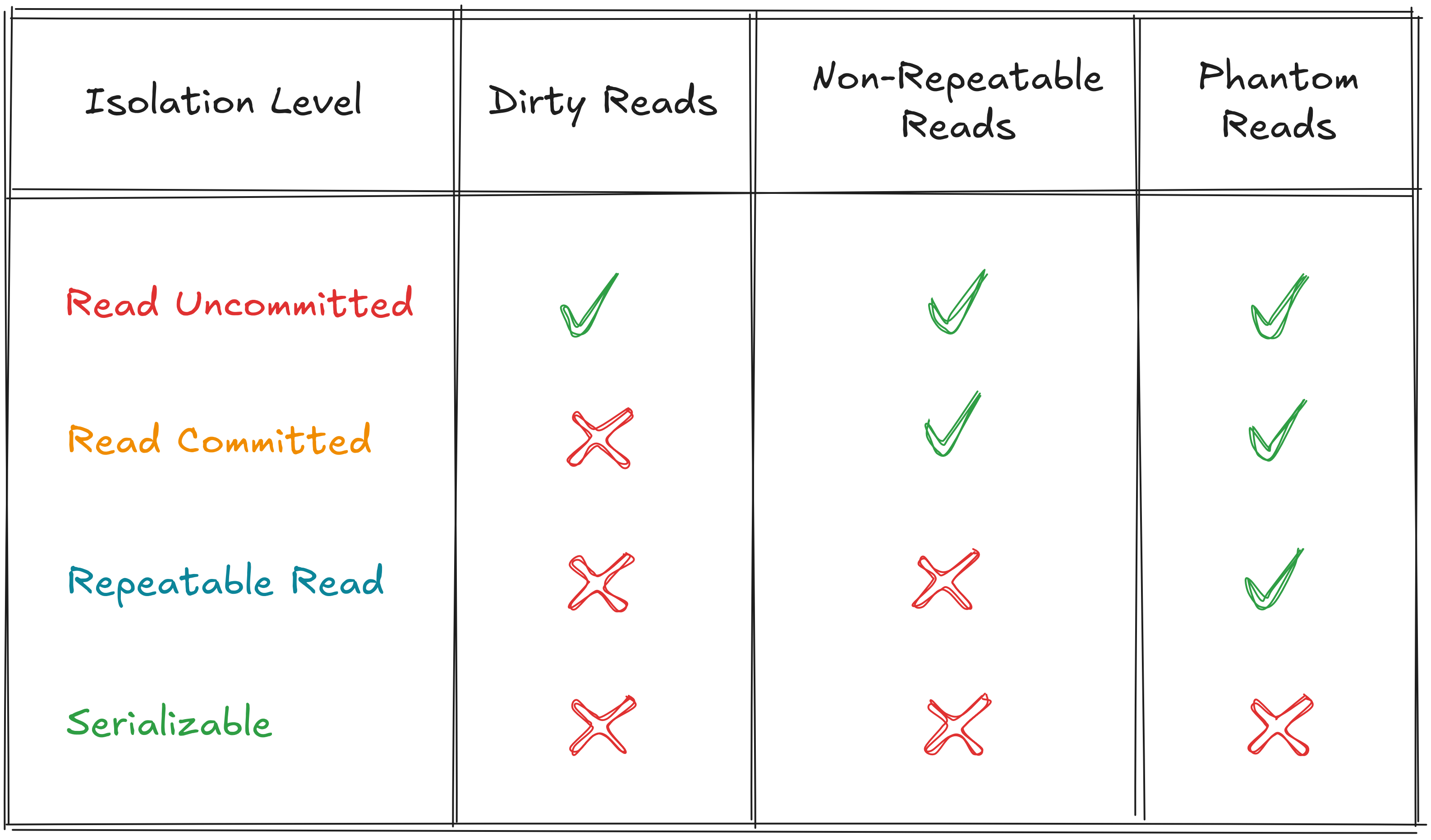
Problem : This anomaly occurs when new or deleted rows "appear" or "disappear" between executions of the same query.

Example :

- Transaction T1 queries all users with a balance greater than $1000 and retrieves two rows.
- Meanwhile, Transaction T2 inserts a new user with a balance of $1500 and commits.
- When T1 repeats the query, it now retrieves three rows, including the newly inserted user.

Isolation levels

To mitigate these anomalies, databases provide isolation levels that control how transactions interact with each other. Each isolation level offers a trade-off between performance and strictness of concurrency control.



Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads
Read Uncommitted	✓	✓	✓
Read Committed	✗	✓	✓
Repeatable Read	✗	✗	✓
Serializable	✗	✗	✗

1. Read Uncommitted

The lowest isolation level. Transactions can read uncommitted data written by other transactions.

Anomalies Addressed : None.

Dirty Reads , Non-Repeatable Reads , and Phantom Reads are all possible at this level.

Allows reading uncommitted (dirty) data, which means a transaction may see intermediate or invalid data that could later be rolled back.

Example :

- Transaction T1 updates a user's balance but does not commit.
- Transaction T2 reads the updated value before T1 commits.
- If T1 rolls back, T2 has already used invalid data.

Use Case : Rarely used in practice because it risks significant inconsistencies.

2. Read Committed

Ensures that a transaction can only read data that has been committed by other transactions.

Anomalies Addressed :

- Prevents Dirty Reads .
- Non-Repeatable Reads and Phantom Reads are still possible

A transaction cannot read uncommitted data.

However, if another transaction modifies and commits data after the first transaction reads it, the first transaction may get different results upon re-reading the same data.

Example :

- Transaction T1 reads a user's balance as $5000.
- Transaction T2 updates the balance to $4000 and commits.
- When T1 reads the balance again, it sees 4000 instead of 5000 (non-repeatable read).

Use Case : Commonly used in scenarios where dirty reads must be avoided, but some inconsistency is tolerable.

3. Repeatable Read

Ensures that once a transaction reads a piece of data, no other transaction can modify that data until the first transaction completes.

Anomalies Addressed :

- Prevents Dirty Reads .
- Prevents Non-Repeatable Reads .
- Phantom Reads are still possible. (Modern Databases tries to mitigate it)

Many modern databases (e.g., MySQL with InnoDB, PostgreSQL) implement Repeatable Read in a way that mitigates Phantom Reads by using techniques like next-key locking or predicate locking

Locks the rows being read to prevent other transactions from modifying them.

However, new rows matching the query criteria can still be inserted by other transactions, leading to phantom reads.

Example :

- Transaction T1 queries all users with a balance greater than $1000 and retrieves two rows.
- Transaction T2 inserts a new user with a balance of $1500 and commits.
- When T1 repeats the query, it retrieves three rows (phantom read).

Use Case : Suitable for scenarios where consistency within a transaction is critical, but some flexibility is allowed for new data.

4. Serializable

The highest isolation level. Transactions are executed in complete isolation, as if they were running sequentially.

Anomalies Addressed :

- Prevents Dirty Reads .
- Prevents Non-Repeatable Reads .
- Prevents Phantom Reads .

Ensures strict isolation by locking all rows involved in a transaction and preventing other transactions from inserting, updating, or deleting rows that match the query criteria.

Transactions are processed one after another, eliminating all concurrency anomalies.

Example :

- Transaction T1 queries all users with a balance greater than $1000 and retrieves two rows.
- Transaction T2 attempts to insert a new user with a balance of $1500 but is blocked until T1 completes.
- T1 completes, and T2 proceeds without causing phantom reads.

Use Case : Used in highly sensitive environments where absolute consistency is required, but at the cost of reduced performance due to strict locking.