# Concurrency control

# Oleksii Nosov

- Software Engineer at EPAM Systems

CODEUS_

# Agenda

- Anomalies
- Isolation levels
- Pessimistic locking
- Optimistic locking
- A bit about MVCC
- Deadlock
- Row level locking

# Anomalies

# Dirty read


**”A transaction reads data written by a concurrent uncommitted transaction.”**

It happens when one transaction reads data that has been modified by another transaction that has not yet been committed.

In other words, a transaction reads uncommitted or *dirty* data.

**This is completely prevented in PostgreSQL**

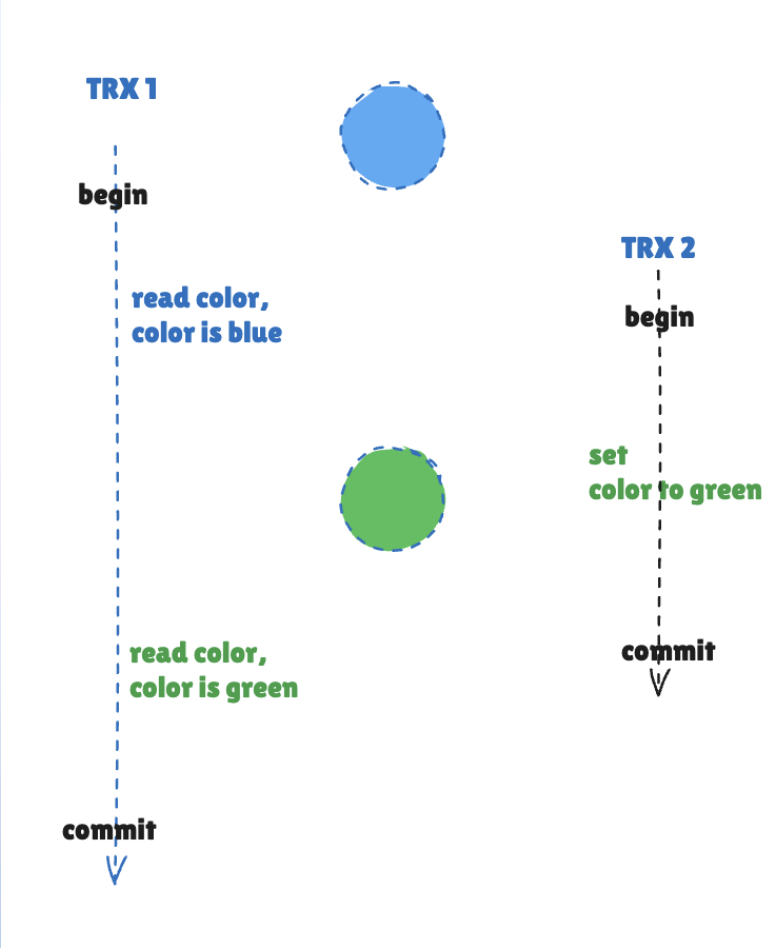
# Non repeatable read

Dirty read

non repeatable read

**”A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).”**

This phenomenon can occur when a transaction reads the same row or tuple multiple times during its execution, but the data values change or vanishes (is deleted) between the reads due to concurrent modifications by other transactions.



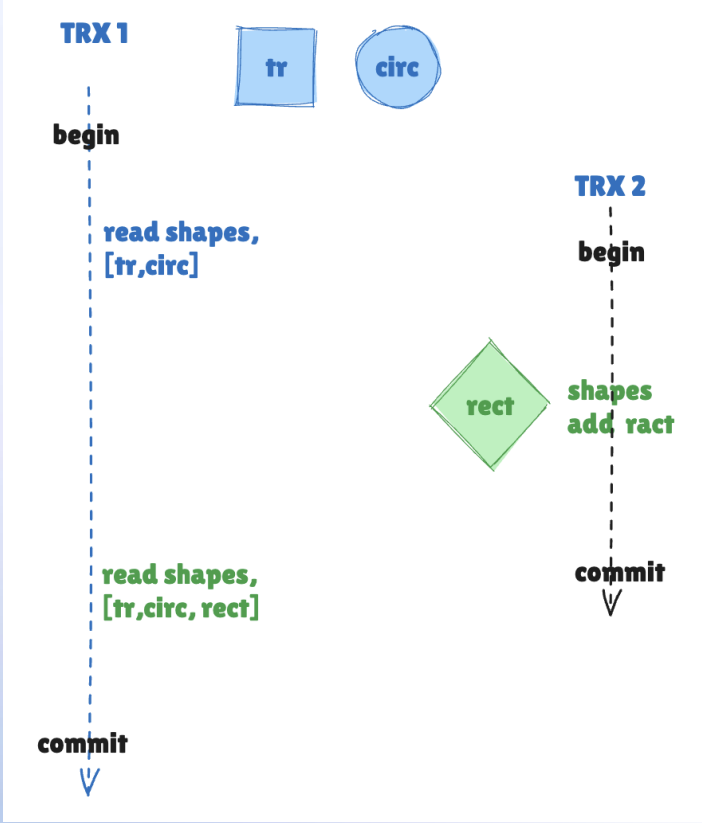

CODEUS_

# Phantom read


**“A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.”**

This occurs when a transaction retrieves a set of rows based on a condition, and between consecutive reads, another transaction inserts that satisfy the same condition.

As a result, the second read includes additional rows or misses previously retrieved rows, leading to an inconsistent result set.



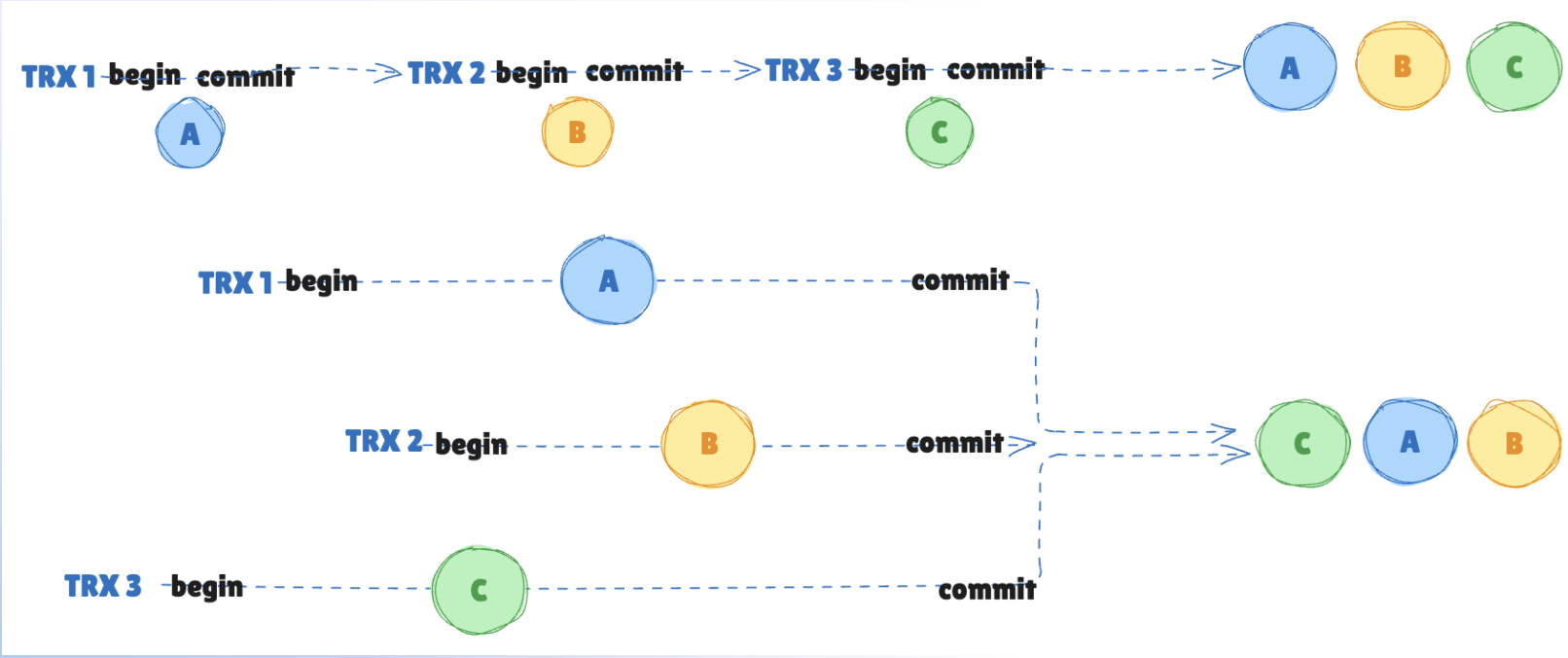

CODEUS

# Serialization


**“The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.”**

This occurs when the outcome of executing a group of transactions concurrently is inconsistent with the outcome of executing the same transactions sequentially in all possible orderings.

In other words, the final result of a set of concurrent transactions is not equivalent to any possible serial execution of those transactions.




CODEUS

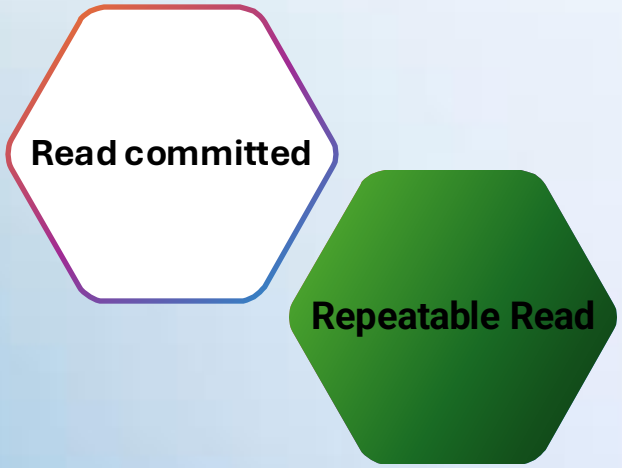
# Isolation Levels

# Read committed


**Read Committed (default):**

- Provides read consistency by ensuring that a transaction only sees data that has been committed at the time the query begins.
- **Prevents dirty reads** by requiring data to be committed before it becomes visible to other transactions.
- **Allows non-repeatable** reads and **phantom reads**, as concurrent transactions may modify the data between reads.
- Offers a good balance between consistency and concurrency and is suitable for many applications.

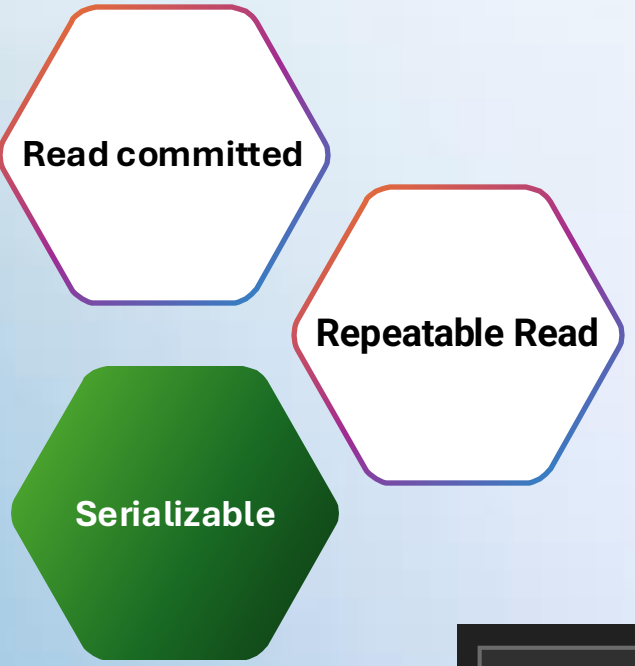
# Repeatable read




**Repeatable Read:**

- Provides a higher level of read consistency than Read Committed.
- Ensures that a transaction sees a consistent snapshot of the database as of the transaction's start time.
- **Prevents dirty reads and non-repeatable** reads by acquiring read locks on accessed data, preventing concurrent modifications.
- **Allows phantom reads**, as concurrent transactions may insert new rows that match the query criteria.
- Provides a stronger guarantee of data consistency but can result in increased concurrency issues due to acquired locks.

# Serializable




- Provides the highest level of isolation and guarantees serializability of transactions.
- Ensures that concurrent transactions appear as if they were executed serially, without any concurrency anomalies.
- **Prevents dirty reads, non-repeatable reads, and phantom reads.**
- May result in increased locking and potential serialization conflicts, leading to more blocking and reduced concurrency.

| Isolation Level | Dirty Read | Nonrepeatable Read | Phantom Read | Serialization Anomaly |
|---|---|---|---|---|
| Read uncommitted | Allowed, but not in PG | Possible | Possible | Possible |
| Read committed | Not possible | Possible | Possible | Possible |
| Repeatable read | Not possible | Not possible | Allowed, but not in PG | Possible |
| Serializable | Not possible | Not possible | Not possible | Not possible |

# Pessimistic locking

# Pessimistic locking

**Core Mechanisms**

It takes a cautious approach by assuming that conflicts between transactions are likely to occur, and it prevents conflicts by acquiring locks on database objects (rows or tables)

These locks can be either shared (read) locks or exclusive (write) locks based on the transaction type.

In the lock release phase, a transaction releases the locks it holds on database objects. This typically happens at the end of the transaction (a commit or a rollback).

**SELECT FOR UPDATE**

1. The primary mechanism for explicit pessimistic locking
2. Locks rows against concurrent updates until your transaction completes
3. Example: SELECT * FROM accounts WHERE id = 1 FOR UPDATE;

**ROW-LEVEL LOCKS**

1. PostgreSQL supports granular row-level locking to prevent concurrent modifications
2. These locks are automatically acquired during UPDATE, DELETE, and SELECT FOR UPDATE operations
3. Different lock modes exist: FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE, FOR KEY SHARE

# Optimistic locking

# Optimistic locking

**Core Mechanisms**

This concurrency control technique takes an optimistic approach by assuming that conflicts between transactions are rare, and it allows transactions to proceed without acquiring locks on database objects during the execution of the entire transaction.

Conflicts are validated, detected and resolved only at the time of committing the transaction.

**Benefits**

**1.HIGHER CONCURRENCY**

1. No locks held during user think-time or between operations
2. Better performance in read-heavy workloads with few conflicts

**2.DEADLOCK PREVENTION**

1. Avoids deadlocks entirely since no locks are acquired

**3.APPLICATION-LEVEL CONTROL**

1. Gives applications explicit control over conflict resolution
2. Can implement custom retry or merging strategies

**VERSION COLUMNS**

The most common implementation uses a version counter column

1. Incremented with each update to track modifications

Example: UPDATE users SET name = 'Alice', version = version + 1 WHERE id = 1 AND version = 5;

2. If no rows are updated, a conflict has occurred

**TIMESTAMP-BASED TRACKING**

Using updated_at timestamp columns instead of numeric versions

•Example: UPDATE products SET stock = stock - 1, updated_at = NOW() WHERE id = 101 AND updated_at = '2023-10-15 14:30:00';

**HASH OR CHECKSUM VERIFICATION**

•Comparing a hash of the record's values to detect any changes

•Useful when tracking multiple columns would be cumbersome

# Dead lock

# Dead lock

**Core Mechanisms**

Deadlocks occur in PostgreSQL when two or more transactions are waiting for each other to release locks, creating a circular dependency that prevents any of them from proceeding.

**Transaction 1** locks resource **A** and needs resource **B**
**Transaction 2** locks resource **B** and needs resource **A**
Both wait indefinitely for the other to release their lock

**Consistent Lock Ordering:**
Always acquire locks in the same order across all transactions
Example: Always lock lower IDs before higher IDs

**Acquire All Locks Upfront:** Get all needed locks at the beginning of a transaction
Example: SELECT * FROM accounts WHERE id IN (1, 2, 3) FOR UPDATE;

**Reduce Transaction Duration:**
Keep transactions short to minimize the lock overlap window
Move read-only operations before or after the transaction

# Row level locking

# Row level locking

## Core Mechanisms

Row-level locking is a fundamental concurrency control mechanism in PostgreSQL that allows multiple transactions to work on different rows of the same table simultaneously.

PostgreSQL locks individual rows rather than entire tables Provides better concurrency than table-level locks

## Automatic Lock Acquisition

UPDATE, DELETE, and SELECT FOR UPDATE automatically acquire row locks

NSERT acquires locks on new rows being created

Regular SELECT queries don't acquire row locks (uses MVCC snapshots instead)

## Lock Modes

**1.FOR UPDATE**

1. Strongest row lock, prevents other transactions from modifying or locking rows
2. Syntax: SELECT * FROM accounts WHERE id = 1 FOR UPDATE;
3. Used when you plan to update rows and need to prevent concurrent modifications

**2.FOR NO KEY UPDATE**

1. Similar to FOR UPDATE but less restrictive
2. Doesn't block FOR KEY SHARE locks
3. Useful when you're modifying non-key columns

**3. FOR SHARE**

1. Allows concurrent read locks but blocks writers
2. Multiple transactions can hold FOR SHARE locks simultaneously
3. Prevents UPDATE, DELETE, and SELECT FOR UPDATE from other transactions
4. Syntax: SELECT * FROM inventory WHERE quantity < 10 FOR SHARE;

**4. FOR KEY SHARE**

1. Least restrictive lock mode
2. Only blocks modifications to the key columns
3. Useful for foreign key checks

CODEUS_

# Thank you

- Author: Oleksii Nosov
- Date: April 2025
- Join Codeus community in Discord
- Join Codeus community in LinkedIn