Transaction

- **Definition**: A sequence of database operations (like reads and writes) treated as a single, logical unit of work. It's "all or nothing."
- **Goal**: To ensure data consistency and integrity, especially when multiple operations are involved (e.g., transferring funds requires debiting one account and crediting another).
- **Key Operations**: COMMIT (make changes permanent), ROLLBACK (undo changes).

ACID Properties

- **Definition**: Fundamental properties guaranteeing reliable transaction processing. Crucial for maintaining data integrity.
- **Properties**:
    - **Atomicity**: Ensures transactions are indivisible. Either all operations within the transaction succeed, or none are applied. Prevents partial updates.
    - **Consistency**: Ensures a transaction transitions the database from one valid state to another, preserving all predefined rules and constraints (like foreign keys, unique constraints).
    - **Isolation**: Ensures that concurrent transactions do not interfere with each other inappropriately. The effect should be the same as if transactions ran sequentially. This prevents anomalies like dirty reads.
    - **Durability**: Ensures that once a transaction is committed, its changes are permanent and survive system failures (crashes, power outages). Often achieved using mechanisms like Write-Ahead Logging (WAL).

Concurrency Anomalies & Isolation Levels

- **Problem**: When multiple transactions run concurrently without sufficient isolation, anomalies can occur.
- **Common Anomalies**:
    - **Dirty Read**: Reading data that has been modified by another transaction but *not yet committed*. (Risk: Reading data that might be rolled back).
    - **Non-Repeatable Read**: Reading the same data twice within a transaction and getting different values because another transaction modified and committed the data in between. (Risk: Inconsistent view of data within a single transaction).
    - **Phantom Read**: Re-executing a query within a transaction returns a different *set* of rows because another transaction inserted/deleted rows matching the criteria and committed. (Risk: Set of data appears to change unexpectedly).
- **Isolation Levels**: Control the trade-off between consistency and concurrency by defining which anomalies are prevented. Common levels (from least to most strict):

1. **Read Uncommitted**: Allows all anomalies. Rarely used.

2. **Read Committed**: Prevents Dirty Reads. Default for many databases (e.g., PostgreSQL, SQL Server). Still allows Non-Repeatable and Phantom Reads.

3. **Repeatable Read**: Prevents Dirty Reads and Non-Repeatable Reads. Still allows Phantom Reads. Default for MySQL (InnoDB).

4. **Serializable**: Prevents all anomalies. Provides the highest level of isolation, effectively executing transactions as if they were serial. Can significantly impact concurrency.

Concurrency Control Mechanisms

- **Purpose**: Techniques used by DBMS to manage concurrent access and prevent anomalies, ensuring some level of serializability.
- **Lock-Based Protocols**: Common approach where transactions acquire locks on data before accessing it.
    - **Shared Lock (S-lock / Read Lock)**: Allows multiple transactions to read concurrently.
    - **Exclusive Lock (X-lock / Write Lock)**: Allows only one transaction to write (or read). Prevents other locks.
    - **Two-Phase Locking (2PL)**: A protocol ensuring serializability (though not necessarily preventing deadlocks). Transactions have a "growing" phase (acquiring locks) and a "shrinking" phase (releasing locks). *Strict 2PL* (holding write locks until commit/abort) is common as it prevents cascading rollbacks.
    - **Deadlock**: A situation where two or more transactions are blocked indefinitely, each waiting for a lock held by the other. Databases have mechanisms to detect and resolve deadlocks (often by aborting one transaction).
- **Timestamp-Based Protocols**: Assigns timestamps to transactions and uses them to determine execution order, potentially aborting transactions that violate the order. Less common than locking in many traditional RDBMSs.
- **Optimistic Concurrency Control (OCC)**: Assumes conflicts are rare. Transactions proceed without locking, and checks are performed at commit time. If a conflict is detected, the transaction rolls back.
- **Multiversion Concurrency Control (MVCC)**: Allows reads and writes to proceed concurrently by maintaining multiple versions of data. Reads access a consistent snapshot without blocking writes. Used by databases like PostgreSQL and Oracle.

Schedules & Serializability

- **Schedule**: The order in which operations from concurrent transactions are executed.
- **Serial Schedule**: Transactions run one after another. Consistent but slow.
- **Concurrent Schedule**: Operations are interleaved. Faster but risks anomalies.
- **Serializability**: The goal for concurrent schedules. Ensures the outcome is equivalent to *some* serial execution, maintaining consistency. Conflict-serializability (based on ordering conflicting operations) is the most common type.

Recoverability

- **Recoverable**: Ensures transactions only commit *after* reading data from already committed transactions. Prevents committing based on "dirty" reads.
- **Cascadeless**: Prevents cascading rollbacks. Ensures transactions only read data from committed transactions. (Strict 2PL achieves this).

Database Recovery Management

- **Purpose**: Restore database consistency after failures.
- **Key Techniques**:
    - **Logging (Write-Ahead Logging - WAL)**: Changes are written to a log file *before* being applied to the database itself. Essential for Undo (rollback) and Redo (durability).
    - **Backup and Restore**: Creating periodic copies of the database to restore from in case of major failures (like media failure).