Transaction

A **transaction** is a **logical unit of work** that consists of one or more **database operations** (like read, write, update) performed together.

Either **all operations succeed** (commit), or **none of them take effect** (rollback).

Think of transferring money from Account A to B:

* Deduct amount from A
* Add amount to B

If only the first step happens and the second fails, money is lost! Hence both must **succeed together** or **fail together**.

**Types of Transactions**

**a) Read-Only Transaction**

* Performs **SELECT** queries only
* No data modifications

BEGIN TRANSACTION;

SELECT \* FROM employees WHERE department = 'HR';

COMMIT;

**b) Read-Write Transaction**

* Involves **insert**, **update**, or **delete** operations

BEGIN TRANSACTION;

UPDATE accounts SET balance = balance - 500 WHERE account\_id = 1;

UPDATE accounts SET balance = balance + 500 WHERE account\_id = 2;

COMMIT;

**ACID Properties of a Transaction**

| **Property** | **Description** | **Example** |
| --- | --- | --- |
| **A**tomicity | All-or-nothing execution | If one step fails, rollback all |
| **C**onsistency | Brings DB from one valid state to another | No constraint violations |
| **I**solation | Transactions do not interfere | Prevent dirty reads, etc. |
| **D**urability | Changes persist after commit | Even after system crash |

Serializability

**Serializability in Database Management System (DBMS)** is a **concurrency control concept** that ensures **consistency of data** when multiple transactions are executed concurrently. It helps ensure that the final result of executing **concurrent transactions** is the **same as if they were executed serially**, one after the other.

**Serializability** is the **highest level of isolation** in a transaction schedule. A **schedule** (sequence of operations from different transactions) is **serializable** if its **outcome is equivalent to some serial schedule**, i.e., a schedule where transactions are executed **one at a time without overlapping**.

**🔹 Types of Serializability**

There are **two major types** of serializability:

**1. Conflict Serializability**

* Based on **conflicting operations**.
* Two operations conflict if:
  + They belong to **different transactions**.
  + They access the **same data item**.
  + At least one is a **write** operation.
* A schedule is **conflict serializable** if it can be transformed into a **serial schedule** by **swapping non-conflicting operations**.
* ✅ **Easier to check using precedence (dependency) graph**.
* **Example:**
* css
* CopyEdit
* T1: R(X) W(X)
* T2: R(X) W(X)
* Schedule: T1 → T2
* This schedule is conflict serializable if we can reorder T1 and T2 to form a serial order without violating data conflicts.

**2. View Serializability**

* Based on **views** of read and write operations.
* A schedule is **view serializable** if it is **view equivalent** to a serial schedule.
* More **general** than conflict serializability.
* Harder to implement due to **higher computational complexity**.

A schedule is **view equivalent** if:

* **Initial reads** are the same.
* Each **read operation** reads the same value in both schedules.
* The **final writes** to each data item are the same.

**🔹 Precedence Graph (Conflict Serializability Check)**

* **Nodes** = Transactions
* **Edges** = Dependency (T1 → T2 if T1's operation conflicts and precedes T2’s)
* If the graph has **no cycles**, the schedule is **conflict serializable**.

**🔹 Why Serializability is Important**

* Maintains **data consistency** in concurrent transactions.
* Prevents issues like:
  + Lost updates
  + Dirty reads
  + Uncommitted dependencies
* Essential for **ACID** properties (especially Isolation and Consistency).

**Concurrency Control**

**Concurrency** refers to multiple transactions executing **simultaneously** in a database system. Concurrency control ensures that database transactions are performed **correctly and concurrently** without violating the **consistency** of the database.

**Goals of Concurrency Control**

* Ensure **atomicity**, **consistency**, **isolation**, and **durability** (ACID properties).
* Prevent anomalies such as:
  + **Lost Update**
  + **Dirty Read**
  + **Non-repeatable Read**
  + **Phantom Read**

**Problems Without Concurrency Control**

**Lost Update Problem**

Occurs when two transactions read the same value and update it simultaneously, causing one update to be **overwritten**.

**Example**:

Initial value of A = 100

T1: READ A (100)

T2: READ A (100)

T1: A = A + 50 (A = 150)

T2: A = A - 30 (A = 70)

T1: WRITE A (150)

T2: WRITE A (70) ❌ → Lost T1's update

**Dirty Read Problem**

Occurs when one transaction reads data **written by another uncommitted transaction**.

**Example**:

T1: WRITE A = 500

T2: READ A = 500

T1: ROLLBACK

T2: Used incorrect value

**Non-repeatable Read Problem**

Occurs when a transaction reads the same data **twice** but gets **different values** due to updates by other transactions.

**Example**:

T1: READ A = 100

T2: UPDATE A = 200 → COMMIT

T1: READ A = 200 ❌ → Different value

**Concurrency Control Techniques**

**Lock-Based Protocols**

**Locks** are used to control access to data:

* **Shared Lock (S)**: for reading
* **Exclusive Lock (X)**: for writing

**Two-Phase Locking Protocol (2PL)**

* **Growing phase**: transactions acquire all locks.
* **Shrinking phase**: transactions release locks.

✔ Ensures **conflict serializability**

**Timestamp-Based Protocol**

Each transaction is assigned a **timestamp (TS)**:

* Ensures **older transactions** get priority.
* Operations are allowed only if they **preserve timestamp order**.

**Optimistic Concurrency Control (OCC)**

* Based on the assumption that **conflicts are rare**.
* Phases:
  1. **Read**
  2. **Validation**
  3. **Write**

**Serializability**

**Conflict Serializability**

* Based on **reordering** non-conflicting operations.
* Use **precedence graph** to check for cycles.

**View Serializability**

* Based on **read-from** and **write-to** equivalence.
* More general, harder to check.

**Schedule Examples**

**Example 1: Conflict Serializable**

T1: R(X), W(X)

T2: R(Y), W(Y)

→ No conflicts, can run in any order → Serializable

**Example 2: Not Serializable**

T1: R(X) → W(X)

T2: R(X) → W(X)

→ Conflicting writes with cyclic dependency → Not serializable

**Deadlock in Concurrency**

**What is a Deadlock?**

When two or more transactions are waiting **indefinitely** for resources **held by each other**.

**Example:**

T1: LOCK(X), LOCK(Y)

T2: LOCK(Y), LOCK(X)

→ Deadlock

**✅ Deadlock Prevention:**

* **Wait-die** scheme
* **Wound-wait** scheme