🧱 Module 5: Transactions, Isolation, and Serializability

💥 Why Transactions Matter

Databases are like cash registers. If you ring up an item, scan a coupon, and charge a card — all those steps **must succeed together**, or none should happen at all.

That's a **transaction**: a group of operations treated as a single unit.

Definition:

A **transaction** is a sequence of operations performed as a single logical unit of work — either all of it happens, or none of it does.

✅ The ACID Properties (Explained Simply)

Property	What It Means	Real-Life Analogy
Atomicity	All or nothing	You either submit your online order — or it cancels. No half-orders.
Consistency	Data must remain valid	You can't have a product with negative inventory.
Isolation	Transactions shouldn't interfere	Two people shouldn't double-book the same hotel room.
Durability	Once committed, it stays committed	Even if the system crashes, your bank deposit remains.

🧪 **Let's See a Transaction in Action (E-Commerce Example)**

💳 **Problem: Buying a Product**

1. Decrease product inventory
2. Charge the customer
3. Create order record

If the **payment fails**, but steps 1 and 3 still happen — you've just lost inventory **without payment**. That's why we use a **transaction**.

SQL Pseudo-Code

sql

CopyEdit

```
BEGIN TRANSACTION;

UPDATE Products
SET Quantity = Quantity - 1
WHERE ProductID = 123;

INSERT INTO Orders (...) VALUES (...);

-- Payment API fails here...

ROLLBACK; -- Everything undone
```

If the payment **succeeds**, we'd COMMIT; instead.

😬 What Are Dirty Reads?

A **dirty read** happens when one transaction reads data that another transaction has **changed but not yet committed**.

💸 Real-Life Analogy:

You check your friend's Venmo balance while they're mid-transfer. It shows $100 — but in 5 seconds, it'll be $40. You read a "dirty" (temporary) value.

Dirty reads can lead to:

- Inconsistent reports
- Incorrect decisions
- Security issues

🔐 Isolation Levels (And Why They Matter)

SQL databases offer **different levels** of isolation to balance speed and safety. Think of them like privacy settings for your transactions:

Level	What It Allows	Danger
Read Uncommitted	Sees dirty reads	Fastest, but unsafe
Read Committed	Only sees committed changes	Default in many systems
Repeatable Read	Same row returns same data	Prevents non-repeatable reads
Serializable	Full isolation	Safest, but slowest

Analogy:

- **Read Uncommitted** = peeking at someone's diary while they're still writing
- **Serializable** = you wait until they're completely done, locked room and all

---

🔁 Serializability: Making Concurrency Safe

Serializability means transactions produce the same result as if they'd run **one at a time**, even if they were executed simultaneously.

🧠 Why It Matters:

Without it, we risk **lost updates**, **phantom reads**, or **contradictory states** — like two people booking the last seat on a plane.

Demo Scenario:

- Alice transfers $100 to Bob.
- System deducts $100 from Alice
- At the same time, another transaction reads Alice's balance and sends $50.

If both go through simultaneously without proper isolation... she could overdraw her account.

💼 Real-Life Use Cases

💸 Money Transfers:

- Ensure both **debit and credit** happen, or **neither**.
- Atomic, isolated, and durable.

🛒 E-Commerce Carts:

- Prevent **double purchases**
- Keep stock consistent
- Clean up abandoned carts only after timeout or failed checkout

---

🧠 Recap: Why ACID and Isolation Matter

Concept	What It Prevents
Atomicity	Partial transactions (e.g., money lost)
Consistency	Invalid data (e.g., negative inventory)
Isolation	Conflicting operations or dirty reads
Durability	Data loss from system crashes
Serializability	Inconsistent results from simultaneous actions