**🔹 What are transactions?**

A transaction is basically a unit of work — a group of operations that should either all happen together or not at all. It’s a way to keep things consistent. Like, if you're transferring money between two accounts, you don’t want to debit one and forget to credit the other. So, transactions are the mechanism that gives us this "all or nothing" behavior.

Scenario:  
You transfer ₹1,000 from your savings to your checking account. The system debits savings successfully but crashes before crediting checking. Without a transaction, you just lost ₹1,000 in thin air. With a transaction, everything rolls back. Your money’s safe.

**🔹 What are ACID properties?**

ACID is a set of rules that makes sure transactions are reliable and databases don’t go haywire:

* Atomicity: Either the whole transaction happens, or none of it does. No half-done stuff.
* Consistency: The database must remain in a valid state before and after the transaction.
* Isolation: Transactions run as if they're the only one happening, even if others are running at the same time.
* Durability: Once a transaction is done, it's permanent — power cuts or crashes won't undo it.

Scenario:  
Imagine buying a concert ticket online. Hundreds of people are rushing to book the last few seats. With ACID, you're guaranteed that:

* The system won’t sell the same seat twice (Isolation).
* Your payment and seat booking both happen or none (Atomicity).
* Your booking is valid and doesn’t break any rules like double booking or seat overflow (Consistency).
* Once it says "Booking Confirmed", it stays confirmed (Durability).

**🔹 Suppose you do not have transactions. Is that system useful? Why?**

Yeah, it might still "work", but without transactions, you’re operating without a safety net. You could end up in a state where only part of an operation finishes — that’s dangerous in critical systems.

Scenario:  
Let’s say your app updates both the user profile and their settings. If it updates the profile but crashes before the settings are saved, now you’ve got inconsistent data. Users might get a broken experience — like seeing a name updated but not preferences, which can cause all sorts of UX bugs.

When it’s okay:  
For non-critical operations — like saving app usage logs or telemetry — losing a few entries isn’t the end of the world. So, transactions might be overkill there.

**🔹 What properties does your file system have?**

A typical file system like ext4, NTFS, or APFS provides some helpful behaviors like:

* Durability through journaling.
* Basic atomicity for whole-file replacements (not within-file writes).
* Minimal isolation or consistency — those are on you to handle.

Scenario:  
You’re saving a file in MS Word and your laptop crashes mid-save. Some file systems can recover the last autosaved state, thanks to journaling. But if you were writing a custom binary file and your app didn't flush properly, it might be corrupted. File systems don’t offer full-blown database-like guarantees.

**🔹 Suppose you do not have "A" (Atomicity). What happens? When is it okay?**

Without atomicity, you can end up with half-completed transactions. That’s dangerous in multi-step operations.

Scenario:  
Say you’re processing an order: reduce inventory, create invoice, send confirmation email. If the inventory updates but the rest fails, your stock is off, the customer gets no email, and there’s no invoice — total chaos.

When it’s okay:  
In systems where partial work is fine — like app logging or analytics tracking. If one out of five log entries fail, it’s not ideal, but the system doesn’t break.

**🔹 Suppose you do not have "C" (Consistency). What happens? When is it okay?**

Without consistency, your data might violate business rules or constraints.

Scenario:  
You insert an order that references a non-existent user. Now your database has a foreign key violation — invalid data. That’s what happens when the DB doesn't check or enforce constraints.

When it’s okay:  
In eventually consistent systems (like many NoSQL databases), temporary inconsistency is allowed. For example, when uploading a social media post and fetching comments asynchronously — it’s okay if comments show up a bit later. As long as it gets consistent *eventually*, users are happy.

**🔹 Suppose you do not have "I" (Isolation). What happens? When is it okay?**

Without isolation, transactions can interfere with each other and create race conditions.

Scenario:  
Two users try to withdraw ₹500 from an account that only has ₹800. Without isolation, both might see ₹800, approve the withdrawal, and leave the account with -₹200 — oops.

When it’s okay:  
In low-stakes read-heavy apps — like viewing blog posts or scrolling through public profiles. If two users view slightly outdated data, no big deal.

**🔹 Suppose you do not have "D" (Durability). What happens? When is it okay?**

If durability’s missing, data can vanish even after you’re told it’s saved.

Scenario:  
You update your profile picture and see “Saved successfully.” But then the system crashes before writing to disk, and next time you log in, the old pic’s still there. That’s a trust-breaker.

When it’s okay:  
Durability isn’t a must in volatile data stores like Redis (when used as a cache). If the system crashes and loses the cache, it can regenerate from the primary DB later. Temporary, non-critical data is fine without full durability.