Lecture 10. Transaction

What is a Transaction?

> Transaction = a **single logical unit of work** consisting of one or more SQL statements.

- All operations inside a transaction are treated as one package:
- Either **all succeed** → changes are saved
- Or **any fails** → all changes aare undone

PostgreSQL transaction follow ACID properties (Atomicity, Consistency, Isolation, Durability)

ACID Properties

Atomoicity

“All or nothing”

If **any operations** in the transaction fails:

  - Whole transaction is **rolled back**
  - **No partial changes** remain in the database

Consistency

Transaction moves database from **one valid state to another**

All constraint must hold:

- primary/unique keys
- foreign keys, referential integrity
- CHECK constraint, buisiness rules

Example: likes count in a blog must always match actual rows in “likes” table

Isolation

- Concurrent transactions must **not interfere** with each other.
- A running transaction behaves **as if it is alone in the system**, from its point of view.
- Intermediate (uncommitted) changes of one transaction are **not visible** to others.

Durability

Once a transaction is **committed**, its changes are:

- **Permanent** (written to non-volatile storage / WAL).
- Survive crashes, power loss, restarts.

Transaction Lifecycle

Phases:

1. **BEGIN** - start transaction
2. **EXXECUTE** - run SQL queries (SELECT/INSERT/UPDATE/DELETE, etc.).
3. **COMMIT** - save all changes if everything OK
4. **ROLLBACK** - undo all changes if error / decisions to cancel

PostgreSQL treats **every SQL statement** as running in a transaction:

- Without explicit `BEGIN`, each statement runs in its own implicit transaction (`BEGIN` + `COMMIT`)

Basic Transaction Control Statement

BEGIN

Start a transaction:

```
BEGIN;
--or
BEGIN TRANSACTION;
--or
BEGIN WORK;
```

Everything after `BEGIN` until `COMMIT` / `ROLLBACK` is **one transaction block**.

COMMIT

Apply (save) changes:

```
COMMIT;
--or
COMMIT WORK;
--or
COMMIT TRANSACTIONS;
```

After `COMMIT` :

- Changes are visible to other sessions
- Durability guarantee kicks in

ROLLBACK

Undo all changes made since `BEGIN` :

```
ROLLBACK;
--or
ROLLBACK WORK;
--or
ROLLBACK TRANSACTION;
```

Use when:

- An error occurs.
- You decide not to apply the changes.

Isolation Levels

PostgreSQL supports standard **transaction isolation levels**:

- **Read Uncommitted**
  - Can *theoretically* read uncommitted data.
  - In PostgreSQL, this behaves like **Read Committed** (dirty reads are still not allowed).
- **Read Committed** (Default)
  - Each query sees **only committed data** at the moment that query starts.
  - If another transaction commits in between your queries, your next query can see its changes.
- **Repeatable Read**
  - Transaction sees a **snapshot** of the database taken at the start of the transaction.
  - The same row read twice in the same transaction returns **the same data**, even if others commit updates.
- **Serializable**
  - Strictest level.
  - Guarantees transactions behave **as if executed one by one in some serial order**.
  - Prevents subtle anomalies at the cost of more blocking or rollbacks.

Concurrency Anomalies

1. **Dirty Read**
   - Transaction T1 reads data written by T2 **that is not committed yet**.
   - If T2 rolls back, T1 has seen “ghost” data.
   - PostgreSQL avoids this even at Read Uncommitted.
2. **Non-Repeatable Read**

- T1 reads a row.
- T2 commits an update to that row.
- T1 reads the same row again → gets **different values**.

3. **Phantom Read**
   - T1 runs a query with some WHERE condition (e.g. `salary > 1000`) and gets N rows.
   - T2 inserts/deletes rows matching that condition and commits.
   - T1 runs the **same query** again → **different set of rows** (new “phantoms” appear/disappear).
4. **Lost Update**
   - T1 and T2 both read the same value, then both write based on that old value.
   - One update overwrites the other → one update is “lost”.
5. **Write Skew**
   - T1 and T2 read overlapping data and both decide to update **different** rows based on a shared condition.
   - Individually each transaction is “valid”, but combined they violate a business rule.

Setting Isolation Level

Global per session:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
--operations
COMMIT;
```

Or:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
--operations
```

```
COMMIT;
```

Isolation level affects **visibility** and **allowed anomalies**.

Savepoints

> **Savepoint** = checkpoint **inside** a transaction

Use when you want **partial rollback**

Creating a savepoint:

```
BEGIN;

-- some operations
SAVEPOINT sp1;

-- more operations

ROLLBACK TO sp1;  -- undo only changes after sp1

COMMIT;          -- commit everything before sp1 + after rollback fixes
```

Key points:

- `SAVEPOINT name;` marks a position.
- `ROLLBACK TO name;`:
  - discards changes from after `SAVEPOINT` to the rollback point,
  - **keeps earlier work in the same transaction**.
- After `ROLLBACK TO`, the savepoint still exists and can be reused.
- `RELEASE SAVEPOINT name;`:
  - removes the savepoint; cannot roll back to it anymore.
  - Also automatically releases all savepoints defined after it.

Benefits:

- More flexible error handling (rollback only part of transaction).
- Better modularity (complex transaction broken into steps).
- Useful for financial/booking systems where some steps may fail but you don't want to discard everything.

When to Use Transactions

- **Financial applications** (banking, e-commerce payments).
- **Inventory management** (adjusting stock, orders).
- **Reservation systems** (airlines, hotels).
- Any **multi-user application** updating shared data.

Advantages and Disadvantages

Advantages

- **ACID guarantees** → data integrity and consistency.
- **Error handling via ROLLBACK**.
- **Improved concurrency control** with proper isolation.
- Combine multiple operations into one unit → can also be more efficient.

Disadvantages

- **Complexity & overhead**:
    - Application logic more complex.
    - Extra work for the DB engine (locking, logging).
- **Reduced concurrency** (depending on isolation level and locks).
- **More resource usage** for long or complex transactions.

- **Harder debugging** (many operations and dependencies inside one transaction).