CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 12: Transactions (User's View)

Amit Kumar Dhar

IIT Bhilai

The last piece of ”using” a database

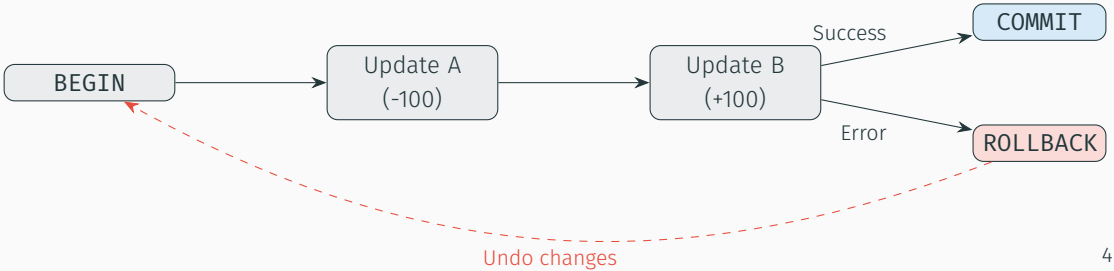
- We can define, query, modify, package, and secure data.
- One thing remains: making **multi-step changes reliable** in the face of errors and concurrency.
- That’s what **transactions** provide.
- Today = the **user’s** view. The **implementation** (locking, MVCC, logging, recovery) is Module III (Weeks 11–12).

Agenda

- What is a transaction?
- The **ACID** properties (in depth)
- `BEGIN` / `COMMIT` / `ROLLBACK` / `SAVEPOINT`
- Autocommit
- Concurrency **anomalies** (dirty / non-repeatable / phantom)
- SQL **isolation levels**
- A bridge to the internals module

What is a transaction?

- A **transaction** is a sequence of operations treated as **one logical unit of work**.
- Canonical example — transfer ₹100 from account A to B:
  1. `UPDATE accounts SET bal = bal - 100 WHERE id = 'A';`
  2. `UPDATE accounts SET bal = bal + 100 WHERE id = 'B';`
- Either **both** happen or **neither** does. That guarantee is the point.



Quick Check

If a transaction consists of two `UPDATE` operations and the system crashes after the first one completes but before the second one starts, what happens to the first update once the system recovers?

Quick Check

If a transaction consists of two `UPDATE` operations and the system crashes after the first one completes but before the second one starts, what happens to the first update once the system recovers?

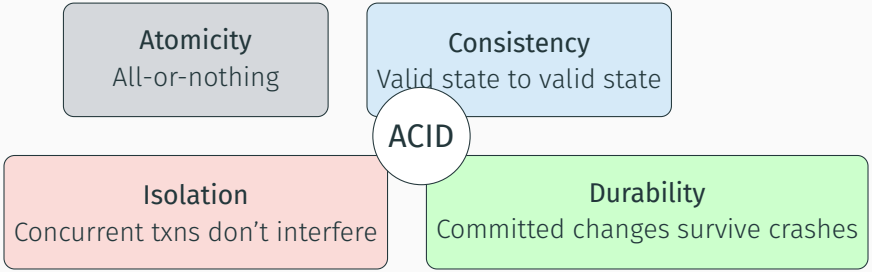
Answer: It is rolled back. A transaction guarantees that either all operations succeed, or none do (Atomicity).

Why we need transactions

- **Failures happen mid-way** — a crash after step 1 loses ₹100.
- **Users run concurrently** — two transfers on the same account can corrupt each other (the lost-update problem from Lecture 1).
- A transaction bundles operations so the DBMS can guarantee correctness despite crashes and concurrency.

ACID

The four guarantees a DBMS makes for each transaction:



Let's take them one at a time.

Atomicity

- A transaction is **indivisible**: all its effects apply, or none do.
- If anything fails before `COMMIT`, the DBMS **rolls back** every change.
- Implemented via **logging/undo** (Week 12) — but as a user you just get the guarantee.
- The transfer never leaves money half-moved.

Consistency

- A transaction moves the database from **one valid state to another**.
- ”Valid” = all constraints hold (PK, FK, CHECK, and app-level invariants like ”total money is conserved”).
- The DBMS enforces declared constraints; the **application** is responsible for logical consistency (writing correct transactions).

Isolation

- Concurrent transactions must appear to run **as if one at a time** (**serializable**).
- Intermediate states of one transaction are **invisible** to others.
- Without isolation: dirty reads, lost updates, and other anomalies.
- In practice, databases offer **weaker** isolation levels for performance — we'll see the trade-offs.

Durability

- Once the DBMS says **committed**, the change **survives** power loss, crashes, restarts.
- Achieved by forcing the **write-ahead log** to stable storage before acknowledging commit (`fsync`).
- ”It’s committed” is a promise the database keeps — even if the server dies one millisecond later.

Quick Check

Which ACID property ensures that an application’s business invariants (like ”balances cannot be negative”) are maintained, provided the constraints are properly defined?

Quick Check

Which ACID property ensures that an application's business invariants (like "balances cannot be negative") are maintained, provided the constraints are properly defined?

Answer: Consistency. The DBMS enforces declared constraints to ensure the database moves from one valid state to another.

Transaction control statements

```sql
BEGIN;                          -- start a transaction
UPDATE accounts SET bal = bal - 100 WHERE id = 'A';
UPDATE accounts SET bal = bal + 100 WHERE id = 'B';
COMMIT;                         -- make it permanent
```

- `BEGIN` / `START TRANSACTION` — open.
- `COMMIT` — apply everything, durably.
- `ROLLBACK` — undo everything since `BEGIN`.

ROLLBACK

```sql
BEGIN;
UPDATE products SET stock = stock - 1 WHERE prod_id = 101;
-- ...business check fails...
ROLLBACK;                          -- as if nothing happened
```

- Discards all changes since `BEGIN`.
- Triggered manually, by an error, or by a lost connection.
- This is exactly what a program's `except` → `conn.rollback()` does (Lab 4).

SAVEPOINT — partial rollback

```
BEGIN;
INSERT INTO orders ...;
SAVEPOINT after_order;
INSERT INTO order_items ...;   -- suppose this fails
ROLLBACK TO after_order;       -- undo just the items, keep the order
COMMIT;
```

- A **savepoint** is a named checkpoint within a transaction.
- Roll back to it without abandoning the whole transaction.

Quick Check

What is the main advantage of using a `SAVEPOINT` in a transaction?

Quick Check

What is the main advantage of using a `SAVEPOINT` in a transaction?

Answer: It allows you to roll back a specific part of a transaction (e.g., a failed insert) without abandoning the entire transaction's progress up to that savepoint.

Autocommit

- By default, many tools run in **autocommit**: each statement is its own transaction, committed immediately.
- `sqlite3` (Python) and `psql` wrap single statements automatically unless you open a transaction.
- To group statements, **explicitly** `BEGIN ... COMMIT` (or use the driver's transaction API).
- **Gotcha:** forgetting this means each of your two transfer updates commits separately — no atomicity!

Transactions in code (recall Lab 4)

```
try:
    with conn:                  # BEGIN ... COMMIT on success / ROLLBACK on erro
        conn.execute("UPDATE accounts SET bal = bal - 100 WHERE id='A'")
        conn.execute("UPDATE accounts SET bal = bal + 100 WHERE id='B'")
except Exception:
    print("transfer failed, rolled back")
```

- The `with conn:` block gives you atomicity in Python.

Quick Check

What is the danger of relying on ”Autocommit” when performing a multi-step operation like transferring money between accounts?

Quick Check

What is the danger of relying on ”Autocommit” when performing a multi-step operation like transferring money between accounts?

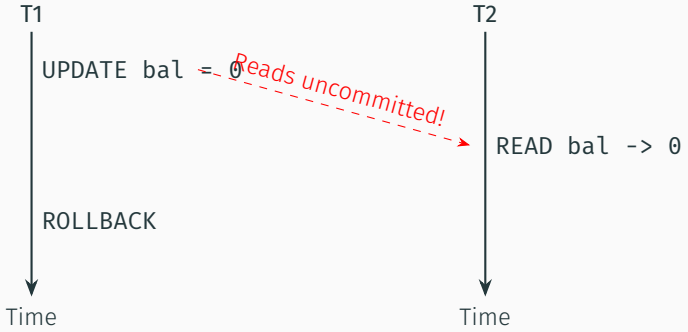
Answer: Each statement is treated as a separate transaction. If one fails, the others are already committed, breaking the Atomicity of the multi-step operation.

The need for isolation levels

- Perfect isolation (**serializable**) can be **slow** — it limits concurrency.
- SQL defines **weaker levels** that allow more concurrency but permit certain **anomalies**.
- You choose the trade-off per transaction.
- First, meet the anomalies.

Anomaly 1: dirty read

- T1 modifies a row; **before it commits**, T2 reads the modified (uncommitted) value; then T1 **rolls back**.
- T2 acted on data that **never officially existed**.



Anomaly 2: non-repeatable read

- T2 reads a row twice; **between** the reads, T1 commits an **update** to it.
- T2 gets **different values** for the same row within one transaction.

```
T2: READ price -> 799
T1: UPDATE price = 899; COMMIT
T2: READ price -> 899    (changed under T2's feet)
```

Anomaly 3: phantom read

- T2 runs a query with a **WHERE**; T1 **inserts a new row** matching it and commits; T2 re-runs the query and sees an **extra ("phantom") row**.

```
T2: SELECT count(*) ... WHERE cat_id=1  -> 5
T1: INSERT a new cat_id=1 product; COMMIT
T2: SELECT count(*) ... WHERE cat_id=1  -> 6   (phantom)
```

- Differs from non-repeatable read: it's about **sets of rows**, not one row.

The four SQL isolation levels

Level	Dirty read	Non-repeatable	Phantom
READ UNCOMMITTED	possible	possible	possible
READ COMMITTED	no	possible	possible
REPEATABLE READ	no	no	possible*
SERIALIZABLE	no	no	no

*Postgres's Repeatable Read (snapshot) also prevents phantoms in practice.

Quick Check

If transaction T2 reads a row, and then transaction T1 modifies that row and commits, and T2 reads the row again and sees the new value, what anomaly has occurred? Which isolation level is required to prevent it?

Quick Check

If transaction T2 reads a row, and then transaction T1 modifies that row and commits, and T2 reads the row again and sees the new value, what anomaly has occurred? Which isolation level is required to prevent it?

Answer: This is a **non-repeatable read**. It can be prevented by using the **REPEATABLE READ** or **SERIALIZABLE** isolation levels.

Setting the isolation level

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- or per transaction:
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

- PostgreSQL default: READ COMMITTED; supports up to SERIALIZABLE (via Serializable Snapshot Isolation).
- SQLite is simpler: one writer at a time, so it behaves close to SERIALIZABLE for writes.

Choosing a level (the trade-off)

- **Higher isolation** → fewer anomalies, but **less concurrency** (more blocking / more aborts to retry).
- **Lower isolation** → more throughput, but your app must tolerate anomalies.
- Default (**READ COMMITTED**) is a common, sensible middle ground.
- Money/inventory correctness may demand **SERIALIZABLE** (with retry logic).

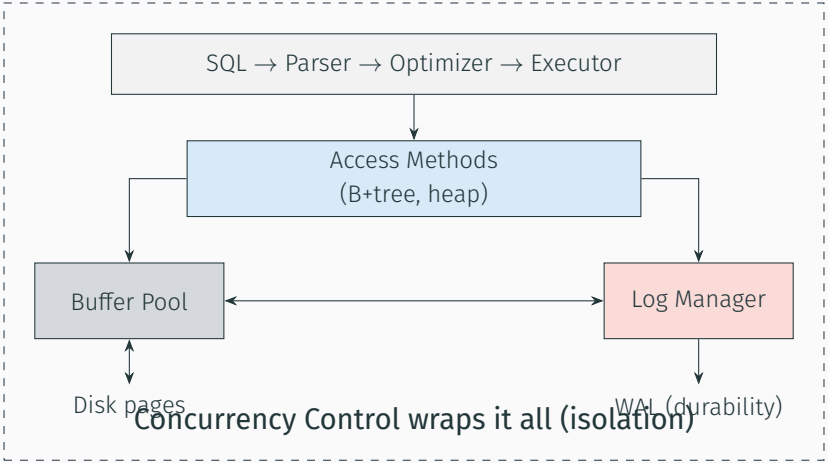
We'll test this ourselves (Lab 11)

- In Week 11 you'll open **two concurrent Postgres sessions** and:
  - reproduce dirty/non-repeatable/phantom reads,
  - change isolation levels and watch anomalies appear/disappear,
  - observe **lost updates** and how locking/MVCC prevent them.
- Seeing the anomalies live makes the table above stick.

Bridge to Module III (internals)

- **How** does the DBMS actually deliver ACID?
  - **Atomicity & Durability** → **logging & recovery** (Week 12): the write-ahead log, undo/redo, ARIES.
  - **Isolation** → **concurrency control** (Week 11): 2-phase locking, timestamps, **MVCC** (how Postgres gives each transaction a snapshot).
- We'll even **build a small WAL** in Lab 12.

The full DBMS picture (revisited)



- Every component maps to a lecture/lab in Weeks 7–12.

Pessimistic vs optimistic concurrency

- **Pessimistic** — take **locks** up front; others wait. Good when conflicts are frequent. (Two-phase locking — Week 11.)
- **Optimistic** — don't lock; **check at commit** whether anyone conflicted; if so, **abort and retry**. Good when conflicts are rare. (MVCC / SSI — Week 11.)
- PostgreSQL's SERIALIZABLE is optimistic — expect occasional retry-worthy aborts.

Deadlocks (a preview)

- Two transactions each hold a lock the other needs → both wait forever.

```
T1: locks A, wants B
T2: locks B, wants A          -> deadlock
```

- The DBMS **detects** the cycle and **aborts one** transaction (a victim), which the app should **retry**.
- Prevention heuristic: acquire locks in a **consistent order**. Details in Week 11.

Quick Check

How does an optimistic concurrency control system (like PostgreSQL's SERIALIZABLE) handle a conflict between two transactions?

Quick Check

How does an optimistic concurrency control system (like PostgreSQL's SERIALIZABLE) handle a conflict between two transactions?

Answer: It does not use locks to block them; instead, it checks for conflicts at commit time. If a conflict occurred, it aborts one of the transactions, which the application must then retry.

Read-only transactions

```
BEGIN TRANSACTION READ ONLY;
SELECT ... ; SELECT ... ;      -- a consistent snapshot across several queries
COMMIT;
```

- Declaring read-only lets the DBMS optimize and gives a **stable snapshot** for multi-query reports (no non-repeatable reads within the transaction).

The retry pattern (for SERIALIZABLE)

```
for attempt in range(3):
    try:
        with conn:                          # BEGIN ... COMMIT
            run_business_logic(conn)
        break                               # success
    except SerializationFailure:
        continue                            # conflict: retry
```

- At high isolation, transactions may abort due to conflicts.
- Robust apps **wrap transactions in a retry loop** — expected, not exceptional.

Keep transactions short

- A transaction holds resources (locks, snapshot) until it ends.
- **Long** transactions → more blocking, more version bloat, more deadlocks.
- Best practices:
  - do slow work (user input, network) **outside** the transaction,
  - open the transaction late, commit early,
  - don't leave transactions idle-open.

Distributed transactions (glimpse)

- Spanning multiple databases/services needs **two-phase commit (2PC)**: a coordinator asks all participants to *prepare*, then *commit*.
- 2PC is correct but **blocking** and slow; modern systems often prefer **sagas** (compensating actions) for cross-service workflows.
- We revisit distribution in Week 13.

Transactions vs the CAP trade-off (glimpse)

- Single-node ACID is ”easy”; **distributed** systems face **CAP**: under a network partition you trade **consistency** vs **availability**.
- This is *why* many NoSQL systems relax ACID to **BASE** (Week 13).
- Understand strong single-node transactions first — then the trade-offs make sense.

Common mistakes

- Relying on autocommit when you needed a multi-statement transaction.
- Expecting SERIALIZABLE behaviour at READ COMMITTED.
- Long-running transactions holding locks / bloating snapshots.
- Not handling **serialization failures** (retry) at SERIALIZABLE.
- Assuming SQLite and Postgres isolate identically.

Summary — key takeaways

- A **transaction** is one logical unit of work with **ACID** guarantees.
- `BEGIN`/`COMMIT`/`ROLLBACK`/`SAVEPOINT` control it; beware **autocommit**.
- Weaker **isolation levels** trade correctness (dirty/non-repeatable/phantom) for concurrency.
- ACID is delivered by **logging/recovery** and **concurrency control** — the heart of Module III.

Exercises

1. Write a transfer transaction; force a failure between the two updates and confirm (via rollback) that no money is lost.
2. In two `psql` sessions, reproduce a non-repeatable read at READ COMMITTED, then prevent it at REPEATABLE READ.
3. Explain which ACID property each of the seven Lecture-1 file problems maps to.
4. When would you accept READ COMMITTED over SERIALIZABLE? Give an example.

Quick reference — transactions

```
BEGIN;  ... ; COMMIT;                  -- atomic unit
ROLLBACK;                              -- undo since BEGIN
SAVEPOINT s;  ROLLBACK TO s;           -- partial undo
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION READ ONLY;
```

Property	Delivered by
Atomicity	logging / undo (Week 12)
Consistency	constraints + correct app logic
Isolation	concurrency control (Week 11)
Durability	write-ahead log + fsync (Week 12)

End of Module I

- You can now **use** a relational database with real fluency: model, query (deeply), program against, secure, and transact.
- **Next — Week 5:** we switch to **design**: the **ER model** for turning requirements into good schemas, then **normalization** (Week 6).
- Then Module III: we go **inside** the engine and start **building** it.

Any Questions?