

```
-- 3.1

DROP TABLE IF EXISTS accounts CASCADE;
DROP TABLE IF EXISTS products CASCADE;

CREATE TABLE accounts (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    balance DECIMAL(10,2) DEFAULT 0.00
);

CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    shop VARCHAR(100) NOT NULL,
    product VARCHAR(100) NOT NULL,
    price DECIMAL(10,2) NOT NULL
);

INSERT INTO accounts (name, balance) VALUES
    ('Alice', 1000.00),
    ('Bob', 500.00),
    ('Wally', 750.00);

INSERT INTO products (shop, product, price) VALUES
    ('Joe''s Shop', 'Coke', 2.50),
    ('Joe''s Shop', 'Pepsi', 3.00);

Query returned successfully in 153 msec.
```

```
-- 3.2

BEGIN;

UPDATE accounts SET balance = balance - 100 WHERE name = 'Alice';
UPDATE accounts SET balance = balance + 100 WHERE name = 'Bob';
COMMIT;

SELECT * FROM accounts;
```

	id [PK] integer	name character varying (100)	balance numeric (10,2)
1	3	Wally	750.00
2	1	Alice	700.00
3	2	Bob	800.00

-- a) Final balances:

-- Alice: $1000 - 100 = 900$

-- Bob: $500 + 100 = 600$

--b) Both UPDATE statements must be in one transaction because

-- a money transfer is one logical action. If one part succeeds and the other fails, the database becomes inconsistent.

--c) If the system crashed between the two UPDATE statements (without a transaction),

-- Alice would lose 100 but Bob would not receive money. This creates an incorrect state.

-- 3.3

BEGIN;

UPDATE accounts SET balance = balance - 500 WHERE name = 'Alice';

SELECT * FROM accounts WHERE name = 'Alice'; -- before rollback

ROLLBACK;

SELECT * FROM accounts WHERE name = 'Alice'; -- after rollback

	id [PK] integer	name character varying (100)	balance numeric (10,2)
1	1	Alice	1000.00

--a) After the UPDATE but before ROLLBACK, Alice's balance = $1000 - 500 = 500$.

--b) After ROLLBACK, Alice's balance returns to 1000.00.

--c) You use ROLLBACK when:

-- you updated the wrong data,

-- wrong amount,

-- validation failed,

-- unexpected error happened.

--It safely undoes all temporary changes.

-- 3.4

BEGIN;

UPDATE accounts SET balance = balance - 100 WHERE name = 'Alice';

SAVEPOINT my_savepoint;

UPDATE accounts SET balance = balance + 100 WHERE name = 'Bob';

-- wrong transfer → undo Bob

ROLLBACK TO my_savepoint;

-- correct transfer

UPDATE accounts SET balance = balance + 100 WHERE name = 'Wally';

COMMIT;

SELECT * FROM accounts;

	id [PK] integer	name character varying (100)	balance numeric (10,2)
1	2	Bob	500.00
2	1	Alice	900.00
3	3	Wally	850.00

--a) Final balances:

-- Alice: 1000 - 100 = 900

-- Bob: 500 (unchanged in final state)

-- Wally: 750 + 100 = 850

--b) Bob was credited temporarily, but this update was undone with ROLLBACK TO SAVEPOINT. Therefore, Bob ends up unchanged.

--c) SAVEPOINT allows undoing only part of a transaction instead of cancelling everything. Convenient when one step is wrong, but the rest is correct.

-- 3.5 TASK 4 – ISOLATION LEVEL DEMO (RUN IN 2 TERMINALS)

-- TERMINAL 1:

BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT * FROM products WHERE shop='Joe''s Shop';

-- SELECT * again after Terminal 2 COMMIT;

COMMIT;

```
-- TERMINAL 2:  
  
BEGIN;  
  
DELETE FROM products WHERE shop='Joe''s Shop';  
  
INSERT INTO products(shop,product,price)  
VALUES ('Joe''s Shop','Fanta',3.50);  
  
COMMIT;  
  
COMMIT
```

Query returned successfully in 100 msec.

--Scenario A — READ COMMITTED:

--a) Terminal 1 sees:

--Before Terminal 2 commits → Coke, Pepsi

--After Terminal 2 commits → Fanta

--READ COMMITTED always shows the latest committed data.

--Scenario B — SERIALIZABLE:

--b) Terminal 1 only sees Coke, Pepsi.

--It does NOT see new changes during the transaction.

--c) Difference:

-- READ COMMITTED: every SELECT sees newly committed data.

-- SERIALIZABLE: transaction behaves as if it runs alone; no other changes are visible.

-- 3.6

-- TERMINAL 1:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
  
SELECT MAX(price), MIN(price) FROM products WHERE shop='Joe''s Shop';  
  
-- SELECT MAX(price), MIN(price) again after Terminal 2 insert  
  
COMMIT;
```

-- TERMINAL 2:

```
BEGIN;  
  
INSERT INTO products(shop,product,price)
```

```
VALUES ('Joe''s Shop','Sprite',4.00);
```

```
COMMIT;
```

```
COMMIT
```

```
Query returned successfully in 126 msec.
```

--a) No, Terminal 1 does NOT see the new product inserted by Terminal 2.

-- REPEATABLE READ freezes the result set for the entire transaction.

--b) A phantom read happens when new rows appear in the result of the same query during a transaction.

-- Example: MAX price changes because someone inserts a new row.

--c) SERIALIZABLE is the only isolation level that prevents phantom reads completely.

-- 3.7

-- TERMINAL 1:

```
BEGIN TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

```
SELECT * FROM products WHERE shop='Joe''s Shop';
```

-- SELECT * again while Terminal 2 UPDATED but NOT committed;

-- SELECT * after Terminal 2 ROLLBACK

```
COMMIT;
```

-- TERMINAL 2:

```
BEGIN;
```

```
UPDATE products SET price=99.99 WHERE product='Fanta';
```

```
ROLLBACK;
```

--a) Yes, Terminal 1 sees the price 99.99. This is a dirty read because Terminal 2 has not committed the change yet and later rolls it back. Terminal 1 works with data that never truly existed.

--b) A dirty read means reading uncommitted (temporary) changes from another transaction.

--c) READ UNCOMMITTED should be avoided because it can show incorrect, temporary, or inconsistent data. This can lead to wrong results and broken business logic.

-- 4. INDEPENDENT EXERCISE 1

-- Transfer \$200 from Bob to Wally IF Bob has enough money

```
DO $$
```

```

BEGIN
  IF (SELECT balance FROM accounts WHERE name='Bob') >= 200 THEN
    BEGIN
      UPDATE accounts SET balance = balance - 200 WHERE name='Bob';
      UPDATE accounts SET balance = balance + 200 WHERE name='Wally';
      RAISE NOTICE 'Transfer successful';
    END;
  ELSE
    RAISE NOTICE 'Transfer failed: insufficient funds';
  END IF;
END $$;

SELECT * FROM accounts;

```

	id [PK] integer	name character varying (100)	balance numeric (10,2)
1	1	Alice	900.00
2	2	Bob	300.00
3	3	Wally	1050.00

```

-- 4. INDEPENDENT EXERCISE 2

-- SAVEPOINT DEMO WITH INSERT → UPDATE → DELETE → ROLLBACK

BEGIN;
  INSERT INTO products(shop,product,price)
  VALUES ('Demo Shop','Tea',1.00);
  SAVEPOINT sp1;
  UPDATE products SET price=2.50 WHERE product='Tea';
  SAVEPOINT sp2;
  DELETE FROM products WHERE product='Tea';
  ROLLBACK TO sp1;
  COMMIT;
  SELECT * FROM products;

```

	id [PK] integer	shop character varying (100)	product character varying (100)	price numeric (10,2)
1	4	Joe's Shop	Fanta	3.
2	5	Joe's Shop	Sprite	4.
3	6	Demo Shop	Tea	1.

-- 4. INDEPENDENT EXERCISE 3

-- Simultaneous withdrawals (conceptual example only)

-- TERMINAL 1:

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
UPDATE accounts SET balance = balance - 300 WHERE name='Alice';
```

```
COMMIT;
```

-- TERMINAL 2:

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
UPDATE accounts SET balance = balance - 300 WHERE name='Alice';
```

```
COMMIT;
```

-- Under SERIALIZABLE, second transaction would fail.

-- 4. INDEPENDENT EXERCISE 4

-- Demonstrate MAX < MIN problem without transactions

-- BAD SESSION 1:

```
SELECT MAX(price) FROM products WHERE shop='Joe"s Shop';
```

	max numeric
1	4.00

-- Meanwhile SESSION 2 deletes rows

-- BAD RESULT: MAX < MIN possible

-- GOOD (WITH TRANSACTION):

```
BEGIN;  
SELECT MAX(price), MIN(price) FROM products WHERE shop='Joe''s Shop';  
COMMIT;
```

	max numeric	min numeric
1	4.00	3.50

-- 5.

/*

1. ACID Properties:

Atomic – all or nothing (bank transfer).

Consistent – DB always stays valid (constraints).

Isolated – transactions don't see each other's work.

Durable – changes remain after crash.

2. COMMIT saves changes permanently,

ROLLBACK cancels all changes.

3. SAVEPOINT is used to undo only part of a transaction.

4. Levels:

- Read Uncommitted – dirty reads
- Read Committed – no dirty reads
- Repeatable Read – no non-repeatable reads
- Serializable – fully isolated

5. Dirty read = seeing uncommitted data (allowed in READ UNCOMMITTED).

6. Non-repeatable read:

You read a row twice and it changed in between.

7. Phantom read:

A query returns NEW ROWS added by another transaction.

Prevented only by SERIALIZABLE.

8. READ COMMITTED is faster and used in high-load apps.

9. Transactions maintain consistency when many users access the DB at same time.

10. Uncommitted changes are LOST when system crashes.

In this lab work, I learned that SQL transactions help keep the database safe and reliable, especially when many operations happen at the same time.

I understood the ACID properties:

atomicity — all steps are done together,

consistency — the data stays valid,

isolation — parallel operations don't interfere,

durability — committed changes are not lost.

I also learned how to use the main transaction commands: **BEGIN**, **COMMIT**, **ROLLBACK**,
and **SAVEPOINT**.

*/