School of Information Technology and Engineering

Lab #10 by Data Bases

Done by: Nurlan Alinur
Checker: Kuralbayev A.

2025, Almaty

Content

Intro:

In this laboratory report, I carried out a series of tasks related to SQL and database transactions. I executed various SQL commands, documented their outputs, and answered all questions associated with each task. Additionally, I completed the independent exercises and provided written responses to the self-assessment questions. Through these activities, I explored key concepts of database transactions, including their execution, management, and effects on data integrity. This report demonstrates my practical understanding of transactions and their role in maintaining reliable database operations.

Main part:

3.1

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

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

3.2

```sql
BEGIN;
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Bob';
COMMIT;
```

Questions:
a) Alice: 900, Bob: 600
b) Atomicity - both updates succeed or both fail
c) Data corruption - money deducted but not credited

3.3

```
BEGIN;
UPDATE accounts SET balance = balance - 500.00
WHERE name = 'Alice';
SELECT * FROM accounts WHERE name = 'Alice';
-- Oops! Wrong amount, let's undo
ROLLBACK;
SELECT * FROM accounts WHERE name = 'Alice';
```

Questions :

a) 900

b) 600

c) ROLLBACK is used to undo changes if an error occurs or if a transaction cannot be completed successfully.

3.4

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Bob';
-- Oops, should transfer to Wally instead
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Wally';
COMMIT;
```

Questions :

a) Alice: 900.00, Bob: 500.00, Wally: 850.00

b) No, Bob was never credited because the transaction rolled back to the savepoint before his update.

c) SAVEPOINT allows partial rollback within a transaction without undoing all previous changes.

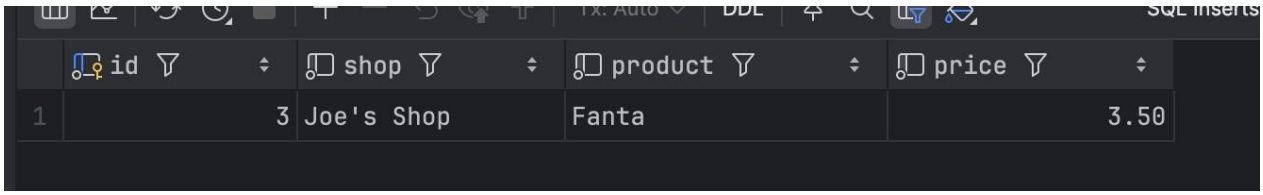
3.5

```sql
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM products WHERE shop = 'Joe''s Shop';
-- Wait for Terminal 2 to make changes and COMMIT
-- Then re-run:
SELECT * FROM products WHERE shop = 'Joe''s Shop';
COMMIT;


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

BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM products WHERE shop = 'Joe''s Shop';
-- Wait for Terminal 2 to make changes and COMMIT
-- Then re-run:
SELECT * FROM products WHERE shop = 'Joe''s Shop';
COMMIT;


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



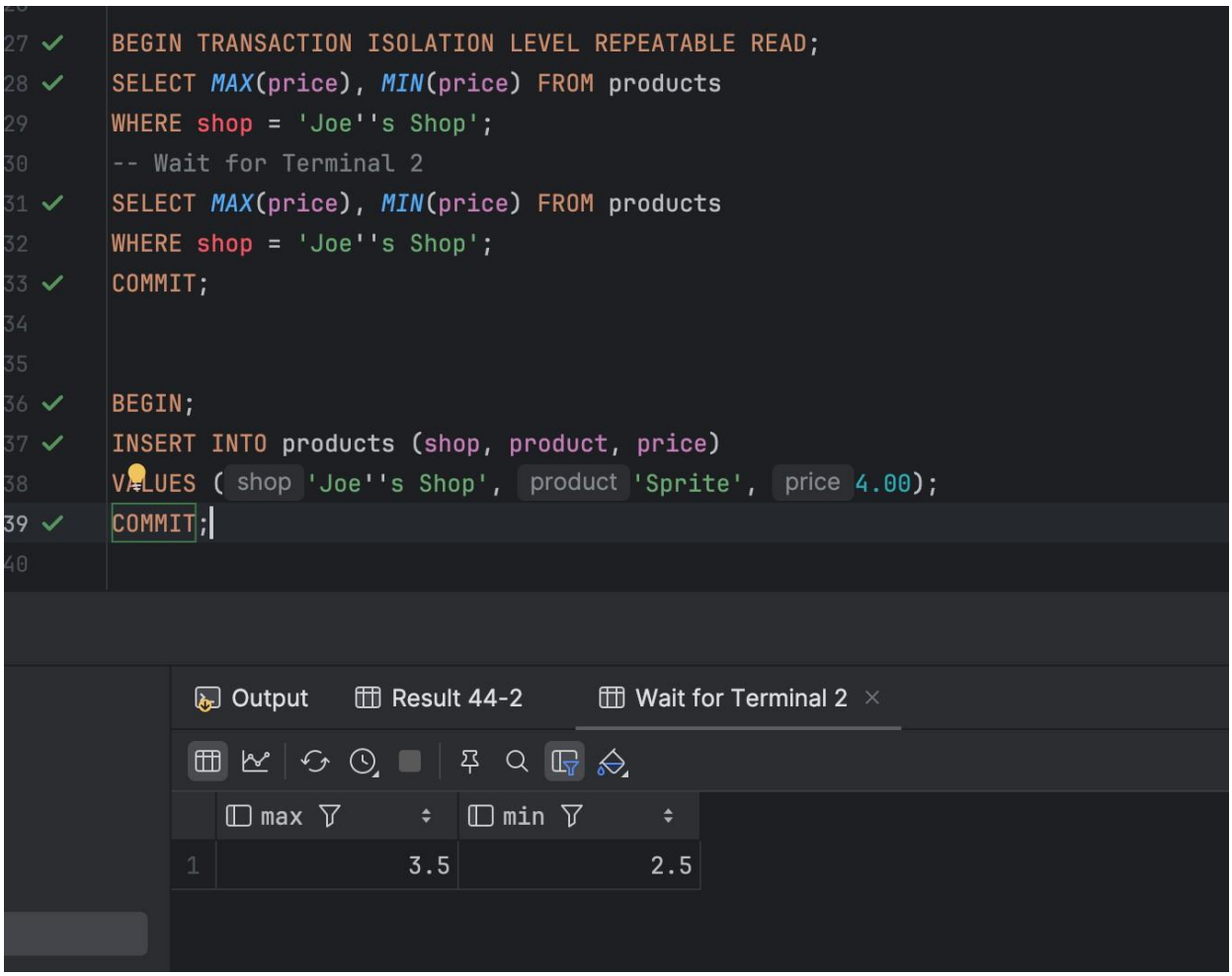
Questions:

a) Scenario A: First sees Coke & Pepsi; after commit sees Fanta.

b) Scenario B: Sees Coke & Pepsi only; second query fails or repeats same snapshot.

c) READ COMMITTED sees new committed data; SERIALIZABLE uses a stable snapshot.

3.6



Questions:

a) No, it sees old values.

b) Seeing new rows appear between reads.

c) SERIALIZABLE.

3.7

```sql
BEGIN TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM products WHERE shop = 'Joe''s Shop';
-- Wait for Terminal 2 to UPDATE but NOT commit
SELECT * FROM products WHERE shop = 'Joe''s Shop';
-- Wait for Terminal 2 to ROLLBACK
SELECT * FROM products WHERE shop = 'Joe''s Shop';
COMMIT;


UPDATE products SET price = 99.99
WHERE product = 'Fanta';
-- Wait here (don't commit yet)
-- Then:
ROLLBACK;
```

ostgres.public.products | Wait for Terminal 2 ...UPDATE but NOT commit | Wait for Terminal 2 to ROLLBACK

Tx: Auto | DDL | SQL Inserts

	id	shop	product	price
1	4	Joe's Shop	Fanta	3.50
2	5	Joe's Shop	Coke	2.50
3	6	Joe's Shop	Pepsi	3.00
4	7	Joe's Shop	Sprite	4.00

Questions:

a) Yes, because dirty reads allow uncommitted data; it's unsafe.

b) Reading uncommitted changes.

c) Because it can return invalid or rolled-back data.

Independent Questions

#1

```sql
BEGIN;
DO $$
BEGIN
    IF(SELECT balance FROM accounts WHERE name = 'Bob') < 200 THEN RAISE EXCEPTION 'Insufficient funds for Bob
    END IF;
    UPDATE accounts SET balance = balance - 200 WHERE name = 'Bob';
    UPDATE accounts SET balance = balance + 200 WHERE name = 'Willy';
END$$;
COMMIT;
```

#2

```
BEGIN;
INSERT INTO products(shop, product, price) VALUES ( shop 'Joe''s Shop', product 'Juice', price 2);
SAVEPOINT savepoint1;
UPDATE products SET price = 3 WHERE product = 'Juice';
SAVEPOINT savepoint2;
DELETE FROM products WHERE product = 'Juice';
ROLLBACK TO savepoint1;
COMMIT;
```

#3

```
--Terminal 1
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT balance FROM accounts WHERE name = 'Alice';
--result 1000
UPDATE accounts SET balance = balance - 200 WHERE name = 'Alice';

COMMIT;
--Terminal 2:
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT balance FROM accounts WHERE name = 'Alice';
--result 1000 (because Terminal 1 has not commited yet)
UPDATE accounts SET balance = balance - 200 WHERE name = 'Alice';

COMMIT;
```

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SELECT balance FROM accounts WHERE name = 'Alice';
--Snapshot = 1000
UPDATE accounts SET balance = balance - 200 WHERE name = 'Alice';
--Don't commit yet
COMMIT;
```

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SELECT balance FROM accounts WHERE name = 'Alice';
--Still sees snapshot 1000
UPDATE accounts SET balance = balance - 200 WHERE name = 'Alice';

COMMIT;
```

```sql
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SELECT balance FROM accounts WHERE name = 'Alice';
--Reads 1000
UPDATE accounts SET balance = balance - 200 WHERE name = 'Alice';
--Not commited yet
COMMIT;
```

```sql
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SELECT balance FROM accounts WHERE name = 'Alice';
--Reads 1000 (snapshot)
UPDATE accounts SET balance = balance - 200 WHERE name = 'Alice';

COMMIT;
```

#4

```sql
CREATE TABLE Sells(
    id SERIAL PRIMARY KEY,
    price NUMERIC
);

INSERT INTO Sells(price) VALUES ( price 100), ( price 200);
--user1 executes MAX:
SELECT MAX(price) FROM Sells; -- Returns 100;
--user2 removes and adds elements
DELETE FROM Sells WHERE price = 100;
DELETE FROM Sells WHERE price = 200;
INSERT INTO Sells(price) VALUES ( price 400), ( price 900);
--user3 executes MIN
SELECT MIN(price) FROM Sells; -- Returns 400
--In this example min > max

--Solution:
BEGIN;
SELECT MAX(price) FROM Sells;
SELECT MIN(price) FROM Sells;
COMMIT;
```

Questions for Self-Assessment

1. ACID properties

   Atomicity: the whole transaction succeeds or fails as one unit.

   Consistency: database rules remain valid after the transaction.

   Isolation: concurrent transactions do not interfere with each other.

   Durability: committed changes survive crashes.

2. Difference between COMMIT and ROLLBACK

   COMMIT makes all changes permanent; ROLLBACK cancels all changes since the transaction began.

3. When to use a SAVEPOINT instead of full ROLLBACK

   When you want to undo only part of a transaction while keeping earlier work.

4. Compare the four isolation levels

   Read Uncommitted: allows dirty reads.

   Read Committed: no dirty reads but non-repeatable reads.

   Repeatable Read: stable row snapshots but phantom reads possible.

   Serializable: fully isolated, no anomalies.

5. What is a dirty read and which level allows it

   Reading uncommitted data from another transaction; allowed in Read Uncommitted.

6. What is a non-repeatable read

   A row read twice returns different values because another transaction updated it in between.

7. What is a phantom read and which levels prevent it

New rows appear between two reads of the same query; only prevented by Serializable.

8. Why choose Read Committed over Serializable in high-traffic applications

   It reduces locking, improves concurrency, and avoids frequent serialization failures.

9. How transactions maintain consistency during concurrent access

   They isolate changes, enforce atomicity, and ensure the database moves from one valid state to another despite concurrency.

10. What happens to uncommitted changes if the system crashes

    They are discarded; only committed changes are preserved.

Conclusion

In this laboratory work I learned how SQL transactions ensure data integrity through the ACID properties and how COMMIT, ROLLBACK, and SAVEPOINT control the flow of changes. I practiced writing safe transactions, handling partial rollbacks, and managing concurrent access. I also observed how different isolation levels affect visibility of data and prevent anomalies such as dirty reads, non-repeatable reads, and phantom reads. Overall, the work showed why transactions are essential in multi-user database systems and how correct isolation settings help maintain consistency and reliability.