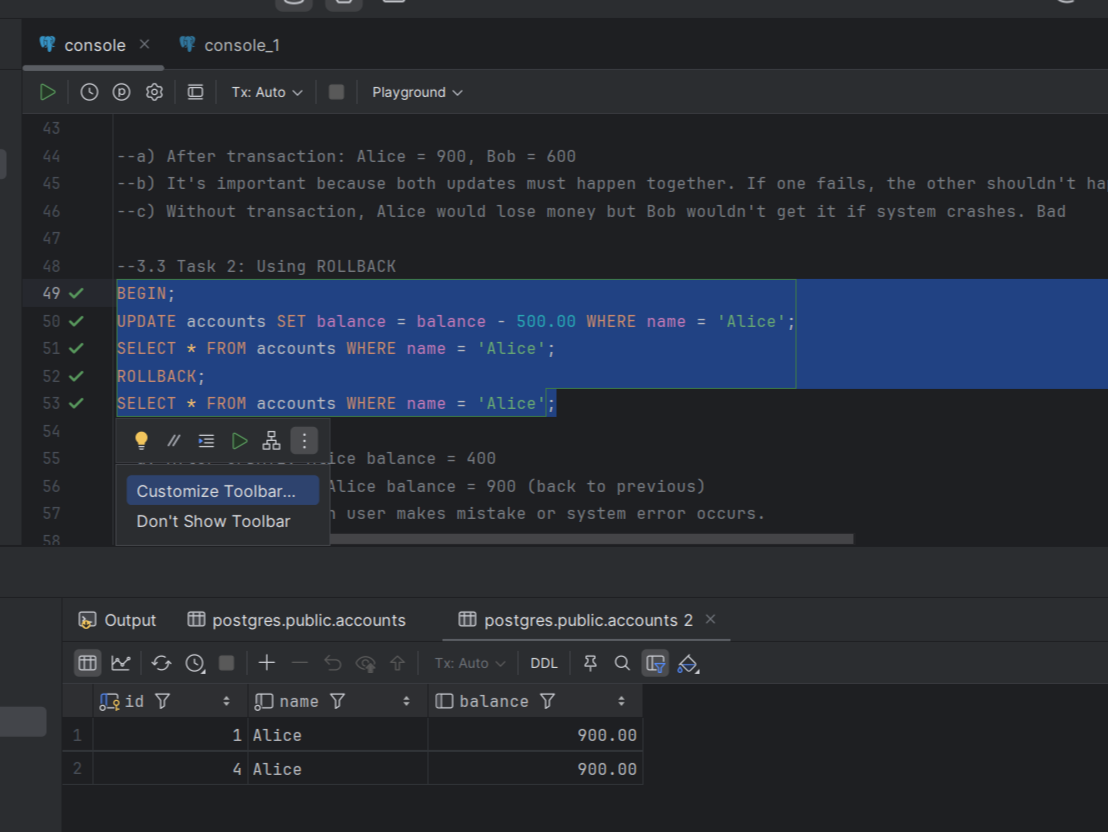
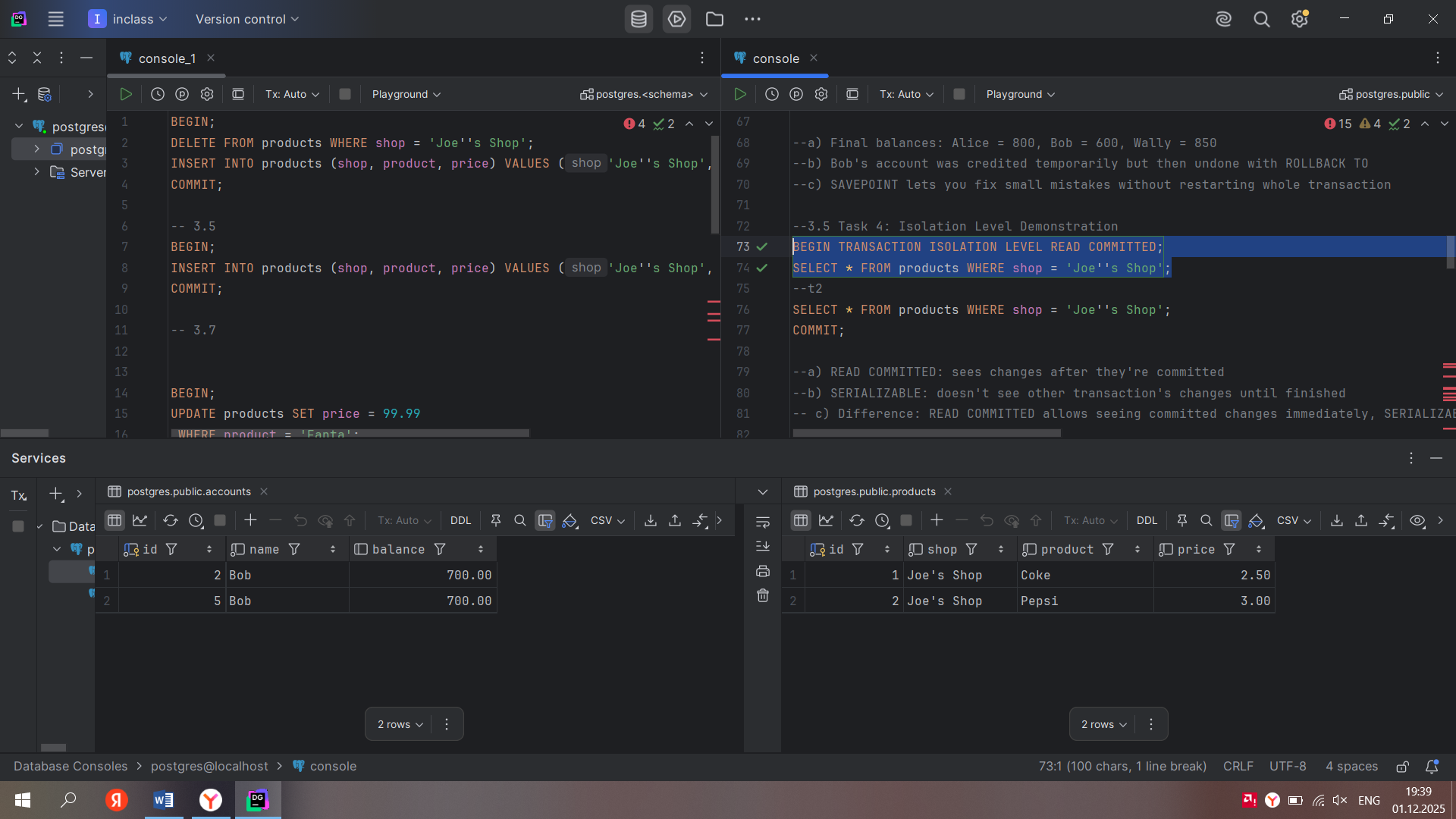
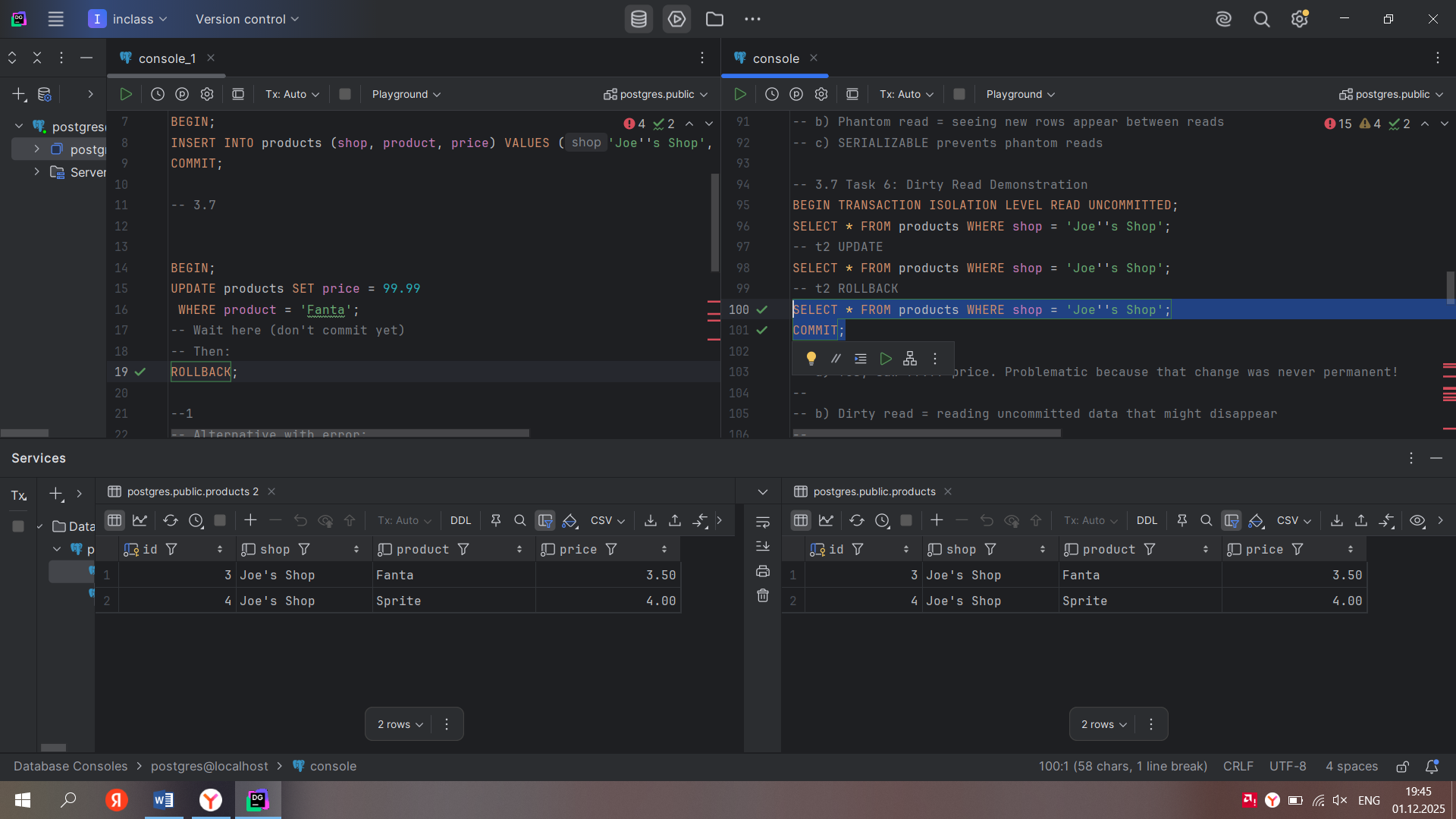
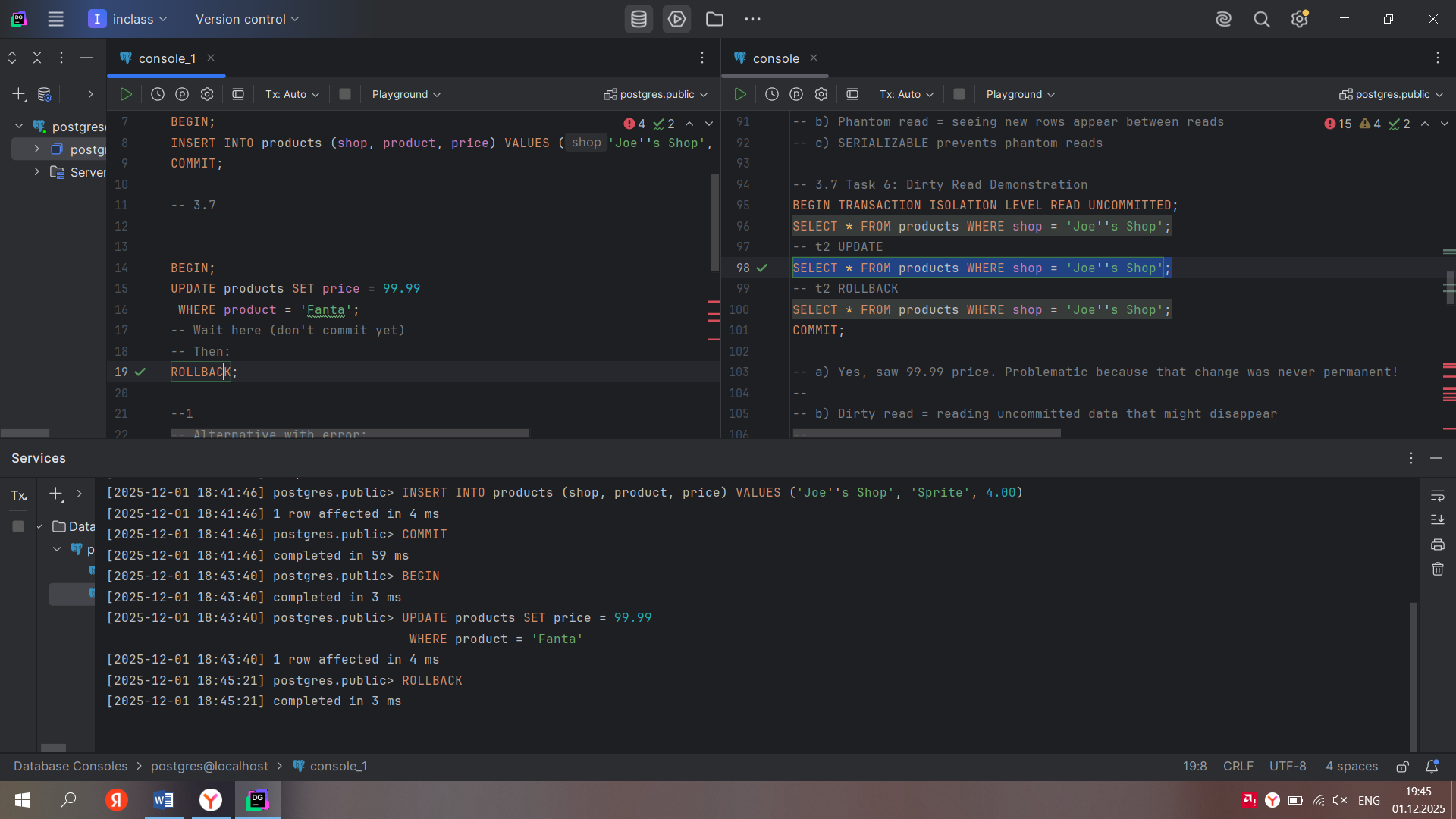
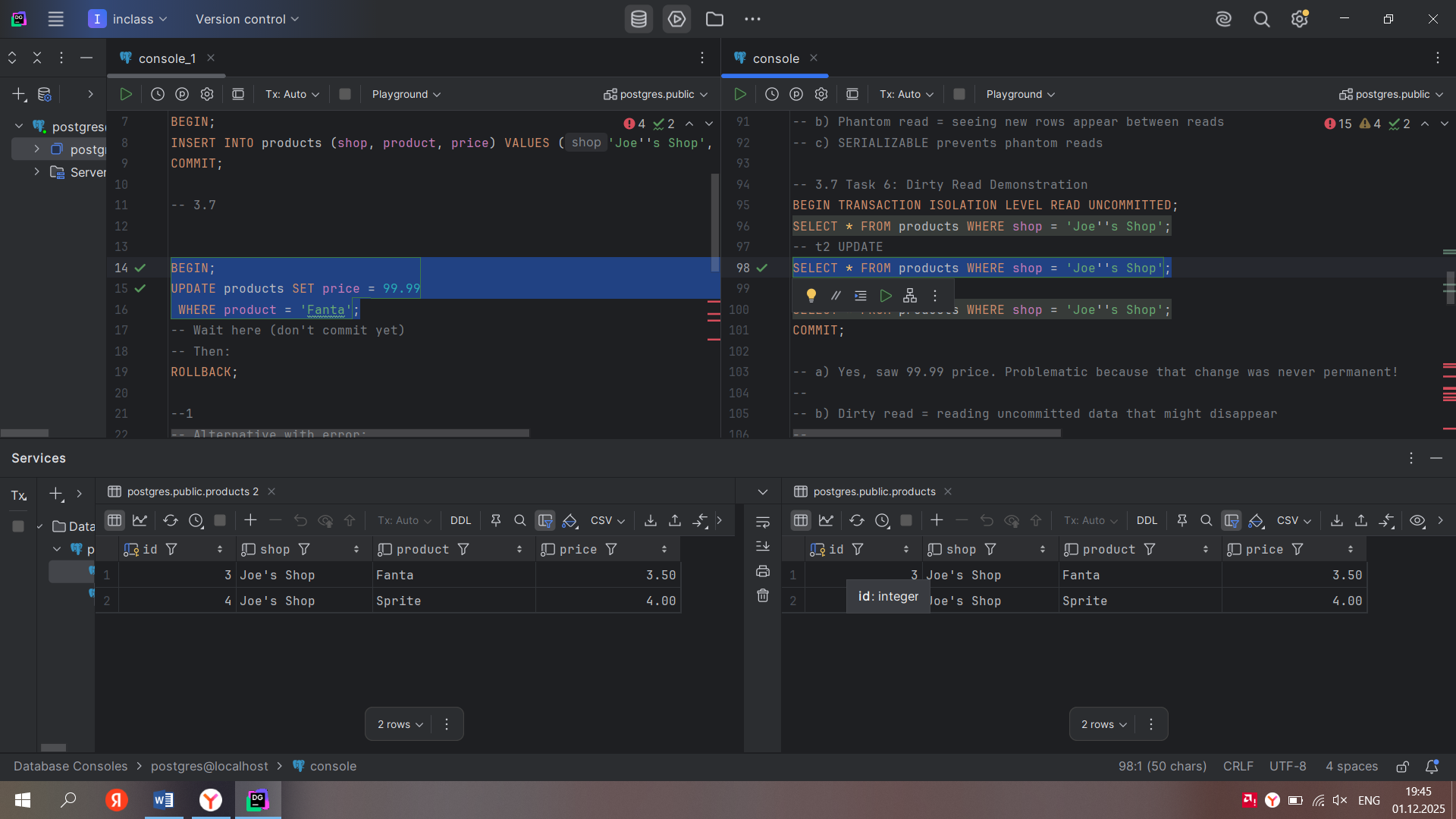
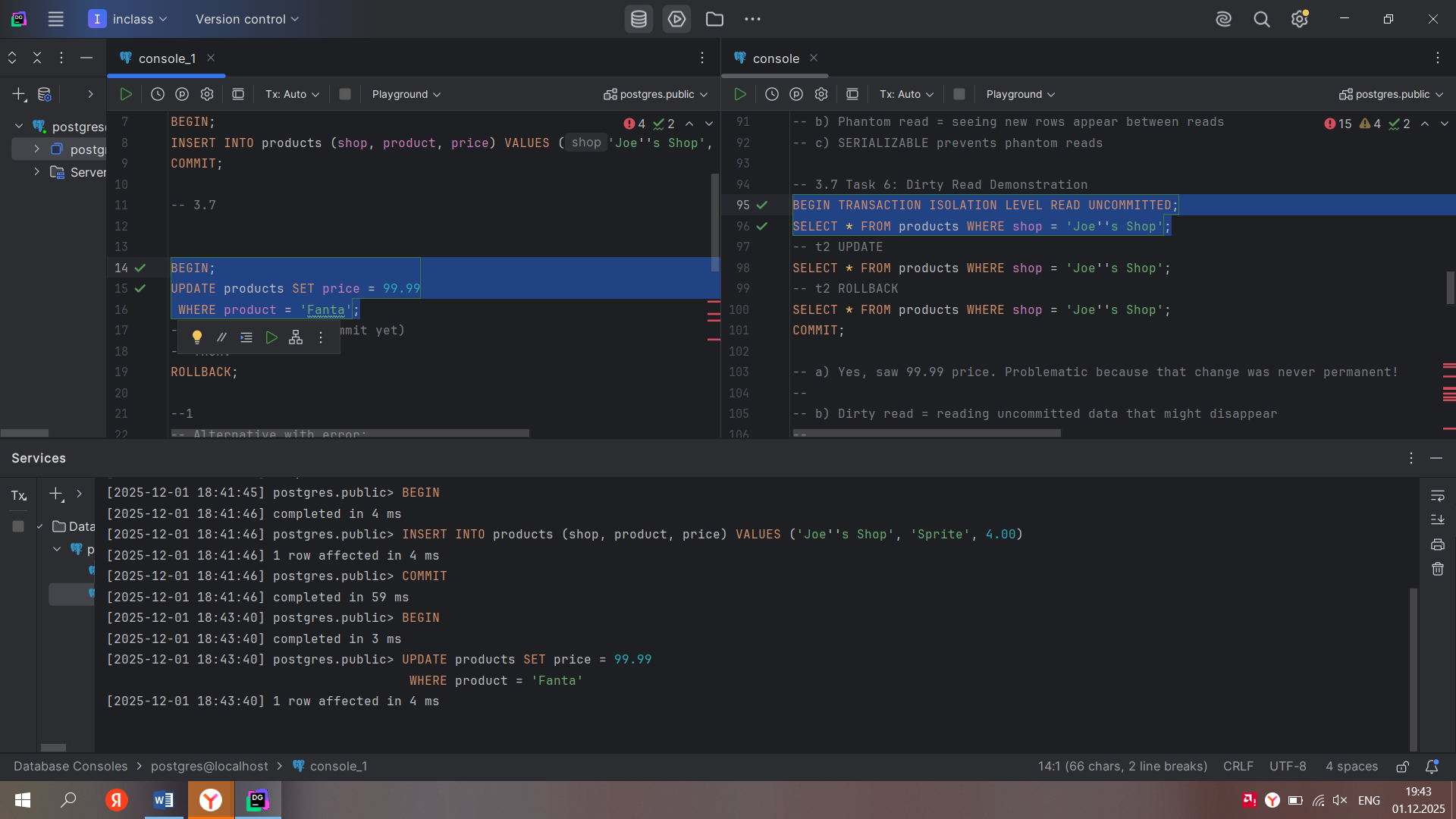
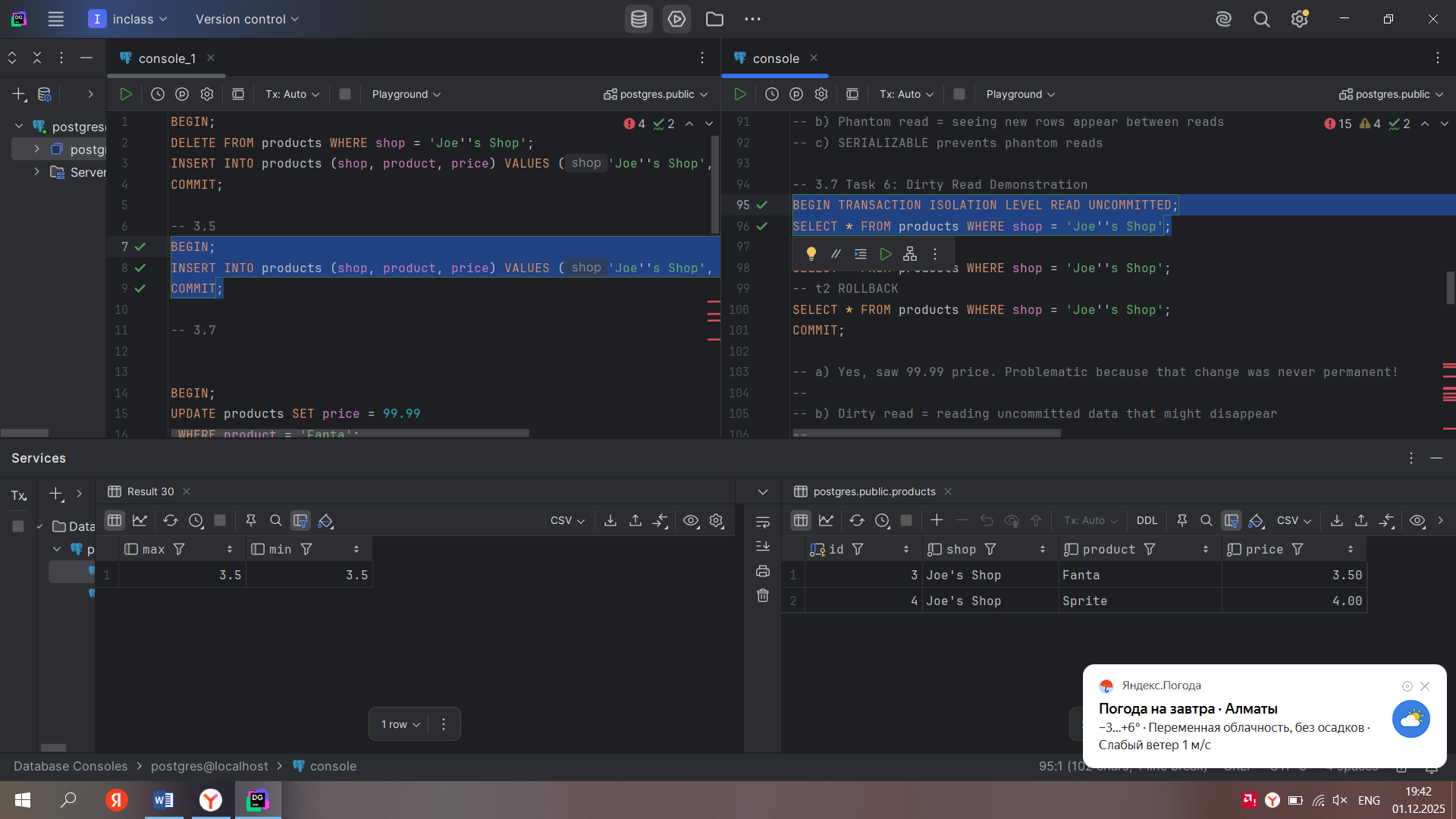
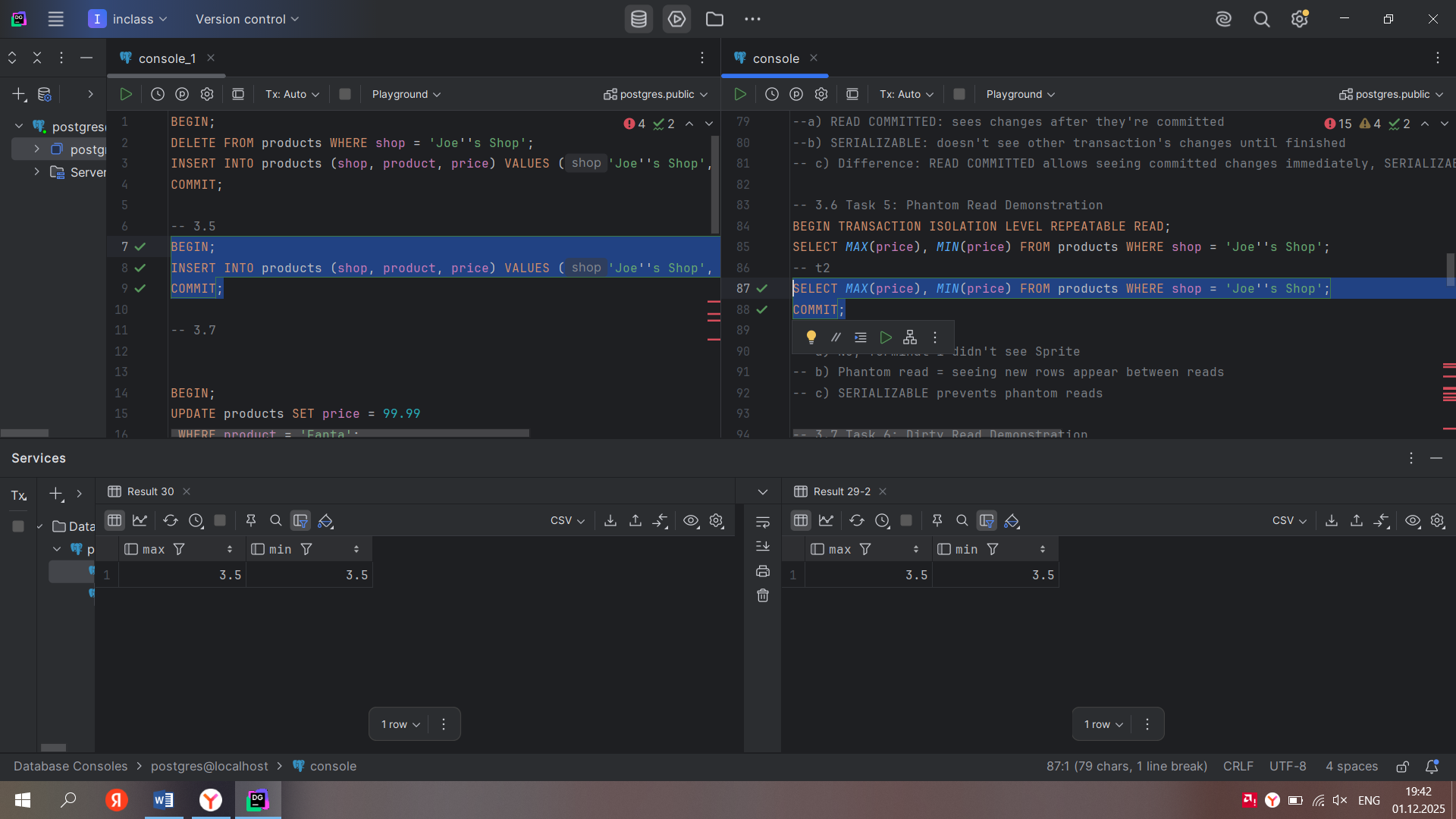
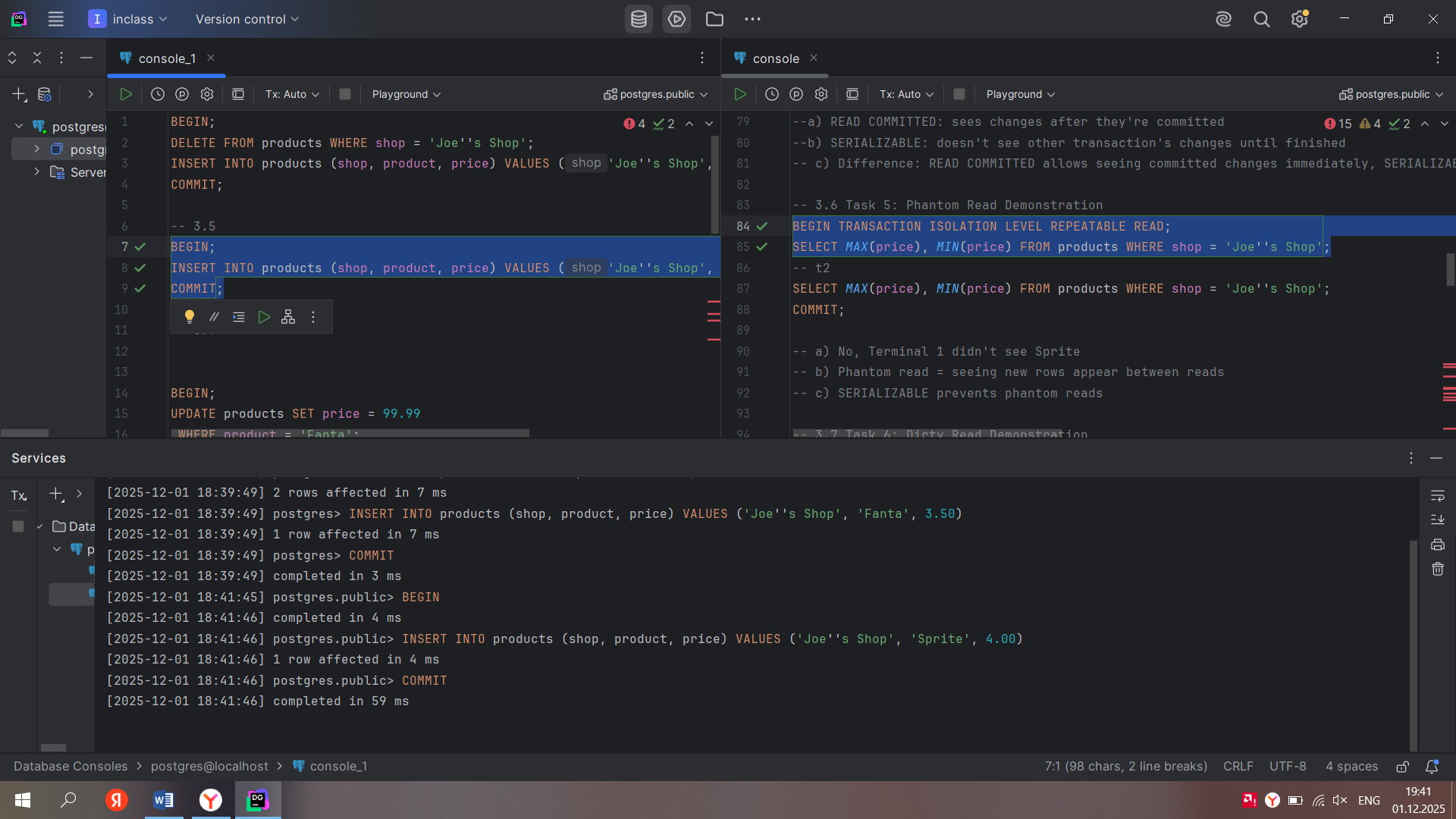
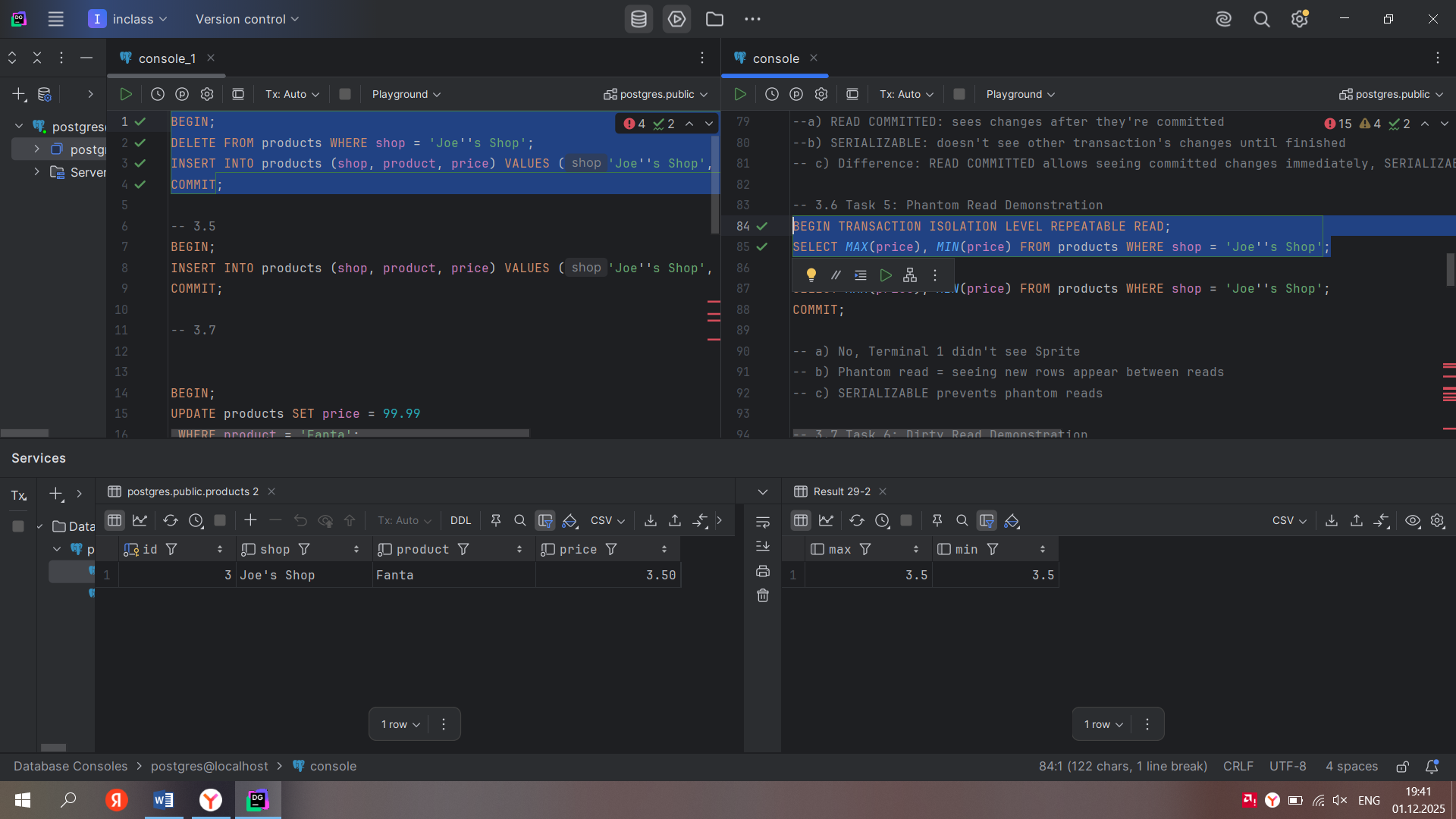
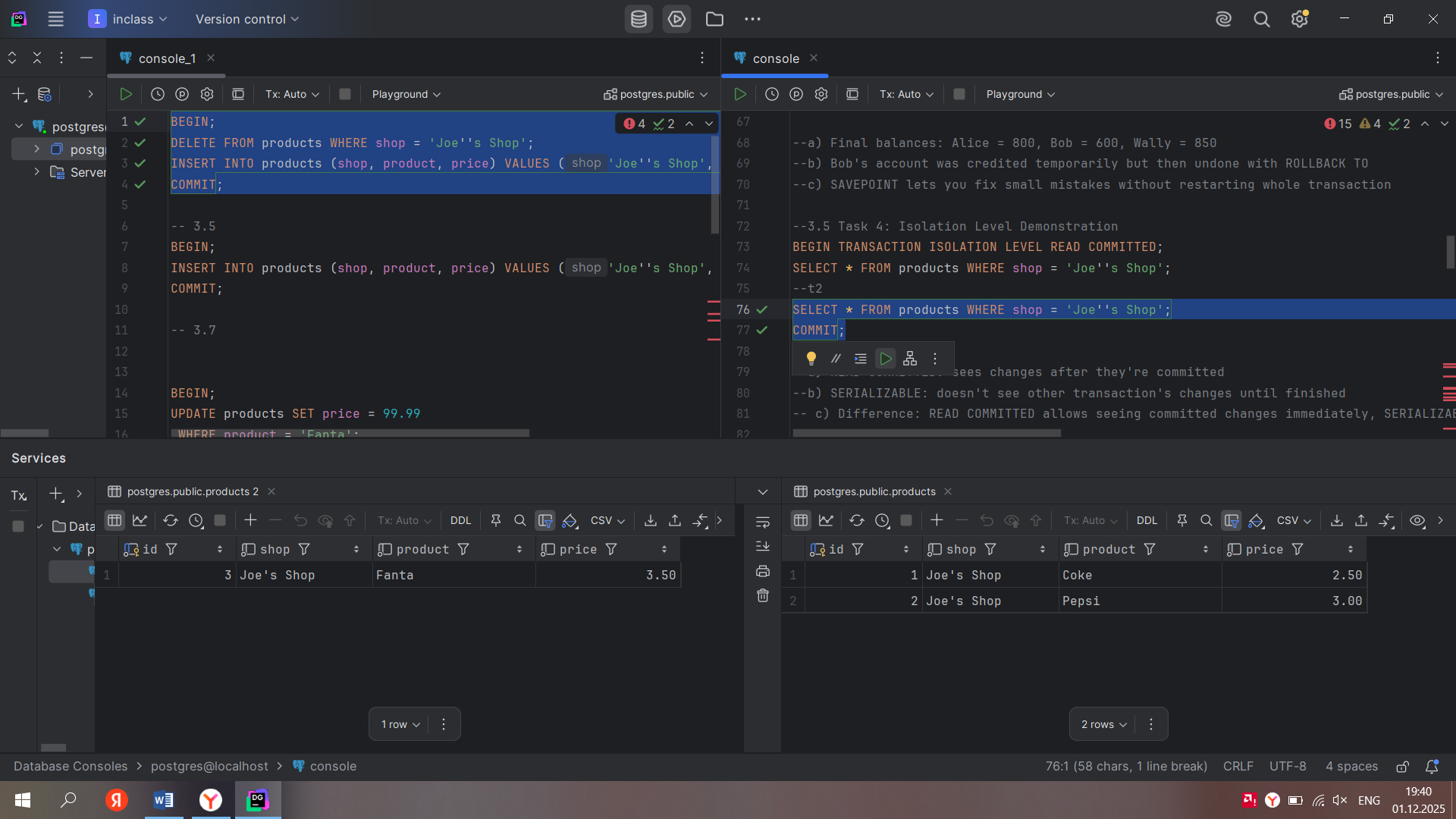
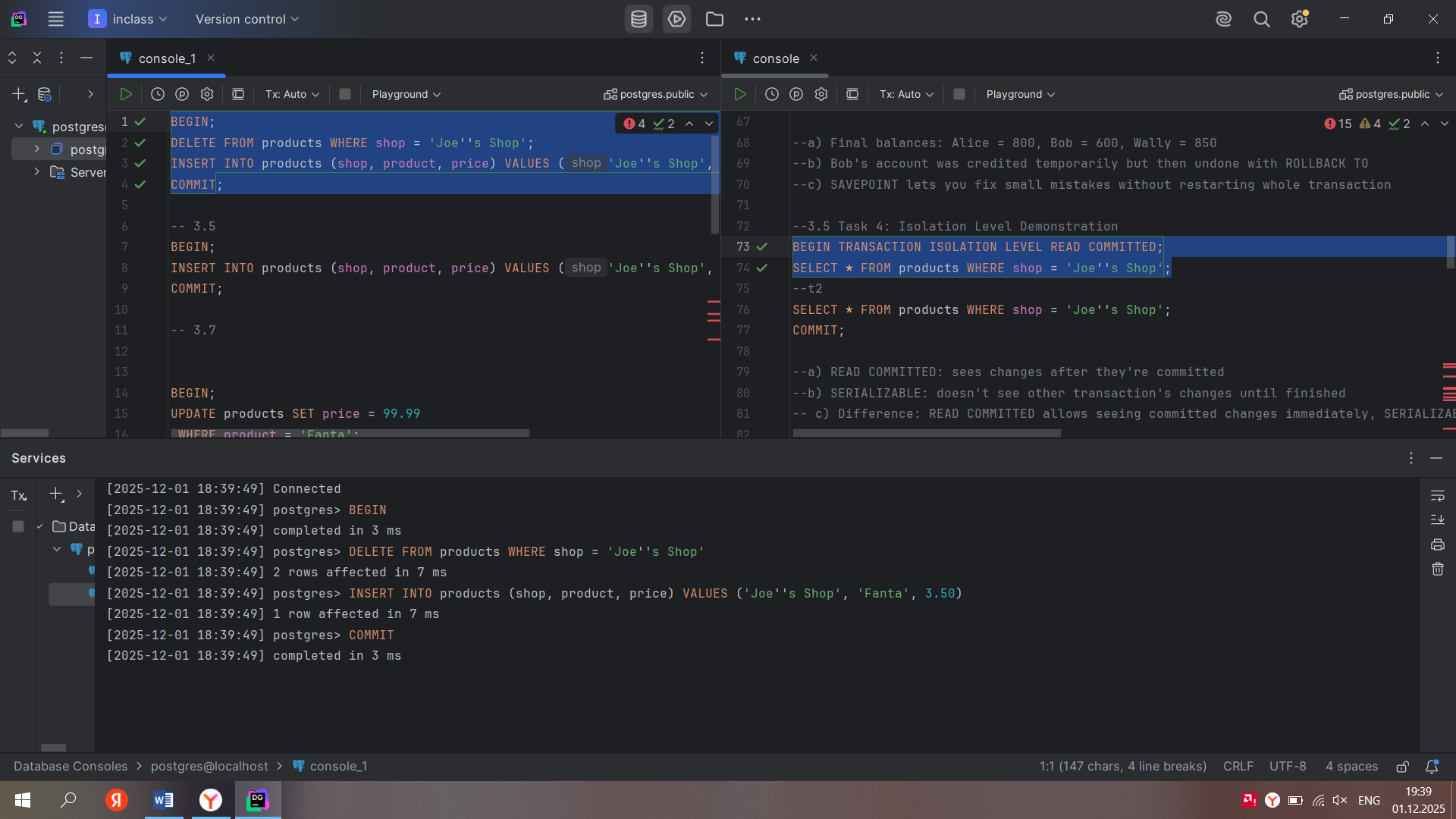
--URMANOVA SAMIRA 24B032087  
  
--3  
--3.1 Setup: Create Test Database  
CREATE TABLE accounts  
(  
 id SERIAL PRIMARY KEY,  
 name VARCHAR(100) NOT NULL,  
 balance DECIMAL(10, 2) DEFAULT 0.00  
);  
CREATE TABLE products  
(  
 id SERIAL PRIMARY KEY,  
 shop VARCHAR(100) NOT NULL,  
 product VARCHAR(100) NOT NULL,  
 price DECIMAL(10, 2) NOT NULL  
);  
  
-- Insert test data  
INSERT INTO accounts (name, balance) VALUES  
 ('Alice', 1000.00),  
 ('Bob', 500.00),  
 ('Wally', 750.00);  
  
-- Insert test data  
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);  
  
--3.2 Task 1: Basic Transaction with COMMIT  
BEGIN;  
UPDATE accounts SET balance = balance - 100.00 WHERE name = 'Alice';  
UPDATE accounts SET balance = balance + 100.00 WHERE name = 'Bob';  
COMMIT;  
  
--a) After transaction: Alice = 900, Bob = 600  
--b) It's important because both updates must happen together. If one fails, the other shouldn't happen.  
--c) Without transaction, Alice would lose money but Bob wouldn't get it if system crashes. Bad  
  
--3.3 Task 2: Using ROLLBACK  
BEGIN;  
UPDATE accounts SET balance = balance - 500.00 WHERE name = 'Alice';  
SELECT \* FROM accounts WHERE name = 'Alice';  
ROLLBACK;  
SELECT \* FROM accounts WHERE name = 'Alice';  
  
--a) After UPDATE: Alice balance = 400  
--b) After ROLLBACK: Alice balance = 900 (back to previous)  
--c) Use ROLLBACK when user makes mistake or system error occurs.  
  
--3.4 Task 3: Working with SAVEPOINTs  
BEGIN;  
UPDATE accounts SET balance = balance - 100.00 WHERE name = 'Alice';  
SAVEPOINT my\_savepoint;  
UPDATE accounts SET balance = balance + 100.00 WHERE name = 'Bob';  
ROLLBACK TO my\_savepoint;  
UPDATE accounts SET balance = balance + 100.00 WHERE name = 'Wally';  
COMMIT;  
  
--a) Final balances: Alice = 800, Bob = 600, Wally = 850  
--b) Bob's account was credited temporarily but then undone with ROLLBACK TO  
--c) SAVEPOINT lets you fix small mistakes without restarting whole transaction  
  
--3.5 Task 4: Isolation Level Demonstration  
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;  
SELECT \* FROM products WHERE shop = 'Joe''s Shop';  
--t2  
SELECT \* FROM products WHERE shop = 'Joe''s Shop';  
COMMIT;  
  
--a) READ COMMITTED: sees changes after they're committed  
--b) SERIALIZABLE: doesn't see other transaction's changes until finished  
-- c) Difference: READ COMMITTED allows seeing committed changes immediately, SERIALIZABLE doesn't  
  
-- 3.6 Task 5: Phantom Read Demonstration  
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
SELECT *MAX*(price), *MIN*(price) FROM products WHERE shop = 'Joe''s Shop';  
-- t2  
SELECT *MAX*(price), *MIN*(price) FROM products WHERE shop = 'Joe''s Shop';  
COMMIT;  
  
-- a) No, Terminal 1 didn't see Sprite  
-- b) Phantom read = seeing new rows appear between reads  
-- c) SERIALIZABLE prevents phantom reads  
  
-- 3.7 Task 6: Dirty Read Demonstration  
BEGIN TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;  
SELECT \* FROM products WHERE shop = 'Joe''s Shop';  
-- t2 UPDATE  
SELECT \* FROM products WHERE shop = 'Joe''s Shop';  
-- t2 ROLLBACK  
SELECT \* FROM products WHERE shop = 'Joe''s Shop';  
COMMIT;  
  
-- a) Yes, saw 99.99 price. Problematic because that change was never permanent!  
--  
-- b) Dirty read = reading uncommitted data that might disappear  
--  
-- c) Avoid READ UNCOMMITTED because you might use wrong data  
  
-- 4. Independent Exercises  
BEGIN;  
-- Check if Bob has enough money  
SELECT balance FROM accounts WHERE name = 'Bob';  
-- If balance >= 200, then proceed  
UPDATE accounts SET balance = balance - 200.00 WHERE name = 'Bob' AND balance >= 200.00;  
-- If row was updated (Bob had enough money)  
UPDATE accounts SET balance = balance + 200.00 WHERE name = 'Wally';  
COMMIT;  
  
  
  
  
-- Exercise 2  
BEGIN;  
INSERT INTO products (shop, product, price) VALUES ('My Shop', 'Juice', 5.00);  
SAVEPOINT sp1;  
UPDATE products SET price = 6.00 WHERE product = 'Juice';  
SAVEPOINT sp2;  
DELETE FROM products WHERE product = 'Juice';  
ROLLBACK TO sp1;  
COMMIT;  
  
-- Final state: Juice exists with price 5.00  
  
-- Exercise 3  
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;  
SELECT balance FROM accounts WHERE name = 'Alice';  
-- Suppose balance = 1000  
UPDATE accounts SET balance = balance - 300 WHERE name = 'Alice';  
-- Don't commit yet  
  
-- With SERIALIZABLE:  
--  
-- Second transaction would fail or wait;Prevents over-withdrawal  
  
-- Exercise 4  
-- Sally runs:  
SELECT *MAX*(price) FROM Sells;  
-- Joe changes prices here  
SELECT *MIN*(price) FROM Sells;  
-- Sally might see MAX < MIN  
  
--With transactions (solution):  
  
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
SELECT *MAX*(price), *MIN*(price) FROM Sells;  
COMMIT;  
-- Now sees consistent view  
  
-- 5. Questions for Self-Assessment  
--1. \*\*ACID examples:\*\*  
 - \*\*Atomic:\*\* Money transfer – either both accounts update or neither does.  
 - \*\*Consistent:\*\* Total money stays the same before and after a transfer.  
 - \*\*Isolated:\*\* Two transfers happening at the same time don’t interfere.  
 - \*\*Durable:\*\* After COMMIT, changes stay even if the power goes out.  
  
2. \*\*COMMIT vs ROLLBACK:\*\* COMMIT saves all changes, ROLLBACK undoes them.  
  
3. \*\*SAVEPOINT use:\*\* You’d use SAVEPOINT for fixing a small mistake in a big transaction without starting over.  
  
4. \*\*Isolation levels comparison:\*\*  
 - \*\*READ UNCOMMITTED:\*\* Can see uncommitted data (dirty reads allowed).  
 - \*\*READ COMMITTED:\*\* Only see committed data (no dirty reads).  
 - \*\*REPEATABLE READ:\*\* Same data every time you read (no dirty or non-repeatable reads).  
 - \*\*SERIALIZABLE:\*\* Like running one transaction at a time (highest isolation).  
  
5. \*\*Dirty read:\*\* Reading uncommitted data that might be rolled back. \*\*READ UNCOMMITTED\*\* allows it.  
  
6. \*\*Non-repeatable read:\*\* Reading the same data twice and getting different results. Example: checking your balance, then someone transfers out money, then checking again and seeing a different balance.  
  
7. \*\*Phantom read:\*\* New rows appear between two reads. \*\*SERIALIZABLE\*\* prevents it.  
  
8. \*\*READ COMMITTED vs SERIALIZABLE:\*\* READ COMMITTED is faster and has less locking, so it’s better for busy apps where speed matters.  
  
9. \*\*Transactions and consistency:\*\* Transactions make sure the database stays correct even when many users are changing data at the same time.  
  
10. \*\*Uncommitted changes after a crash:\*\* They disappear because the database automatically does a ROLLBACK.

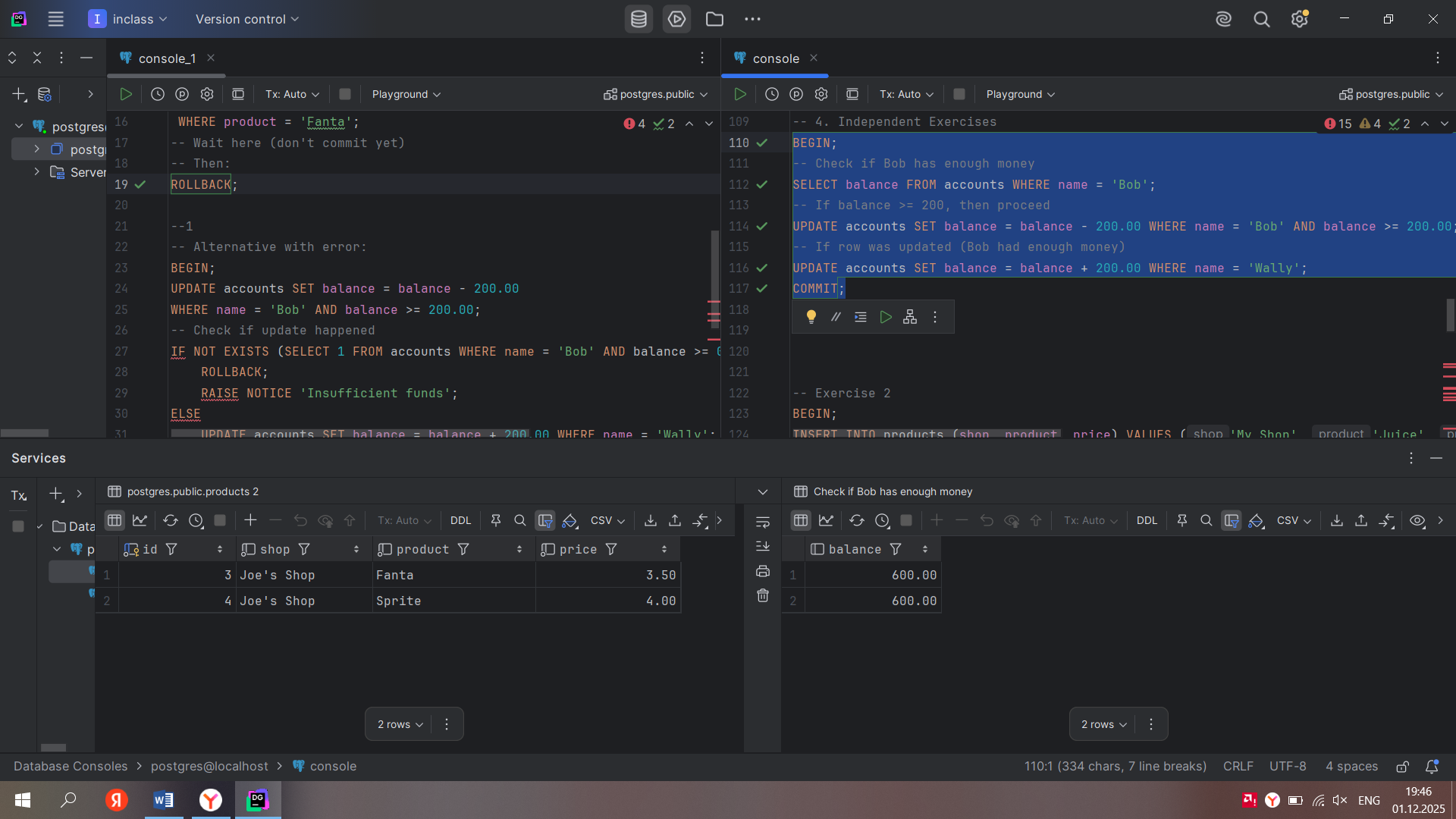




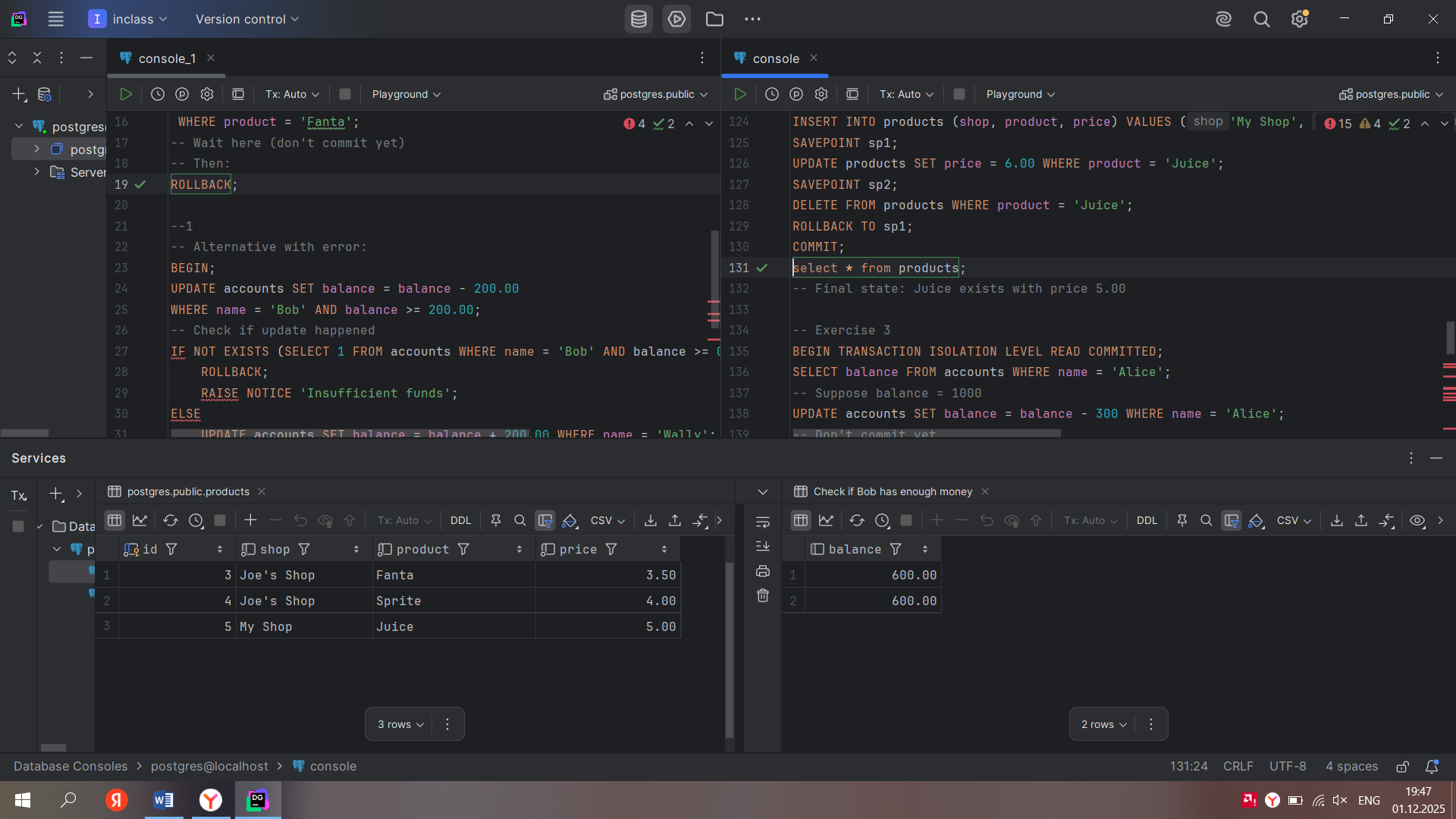


Independent

Ex 1



Ex 2



Ex 3

