НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування та спеціалізованих комп'ютерних систем

Лабораторна робота №3

з дисципліни
«Бази даних і засоби управління»

Виконав: студент III курсу
ФПМ групи КВ-94
Орел Б.В.
Перевірив: доц. Петрашенко А. В.

Київ – 2021

Метою роботи є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

Завдання роботи полягає у наступному:

1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL.
3. Розробити тригер бази даних PostgreSQL.
4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

Варіант 19

У другому завданні проаналізувати індекси BTree, BRIN.

Умова для тригера – before insert, delete.

Завдання 1

Інформація про модель та структуру бази даних

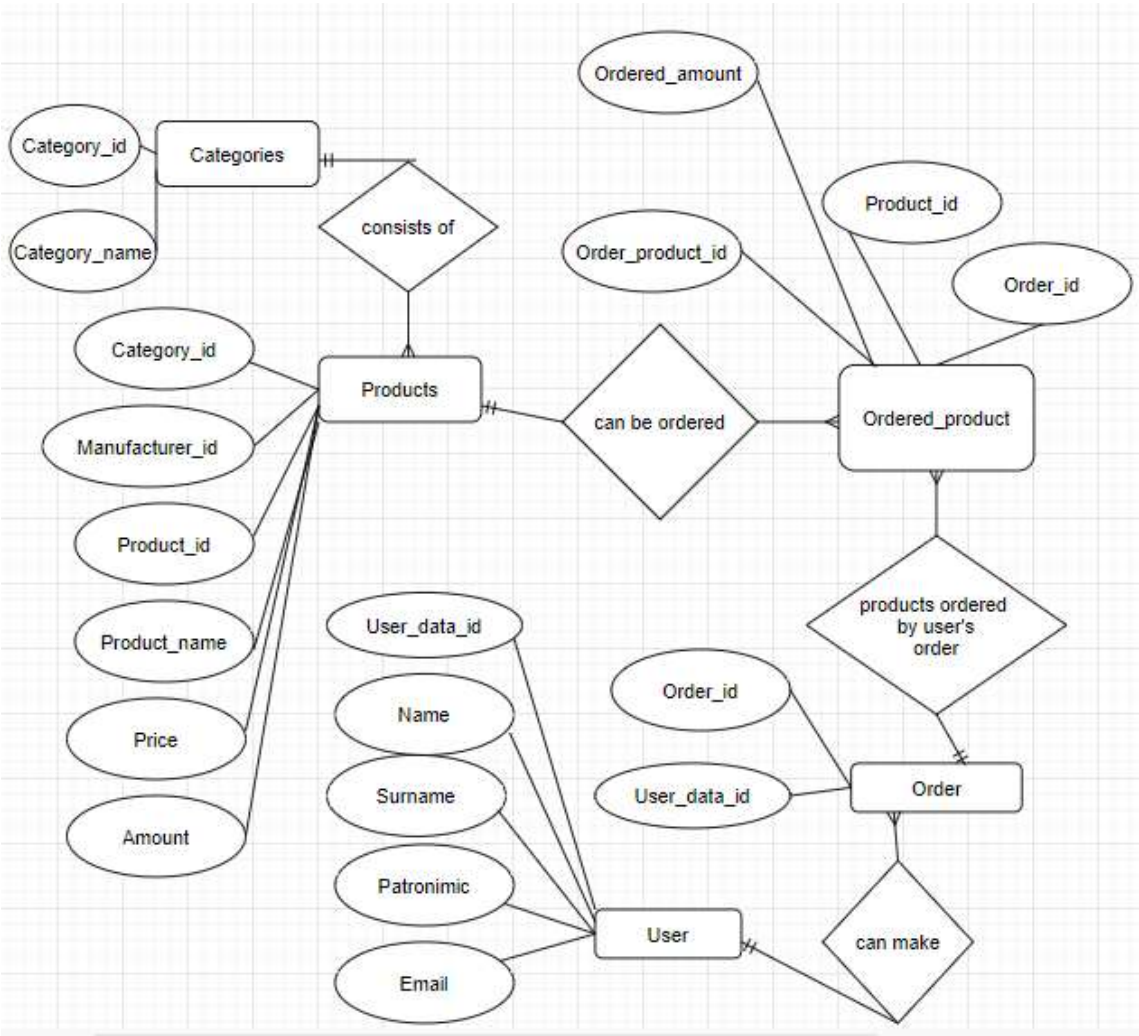


Рисунок 1 - Концептуальна модель предметної області “Магазин”.

Нижче (Рис. 2) наведено логічну модель бази даних:

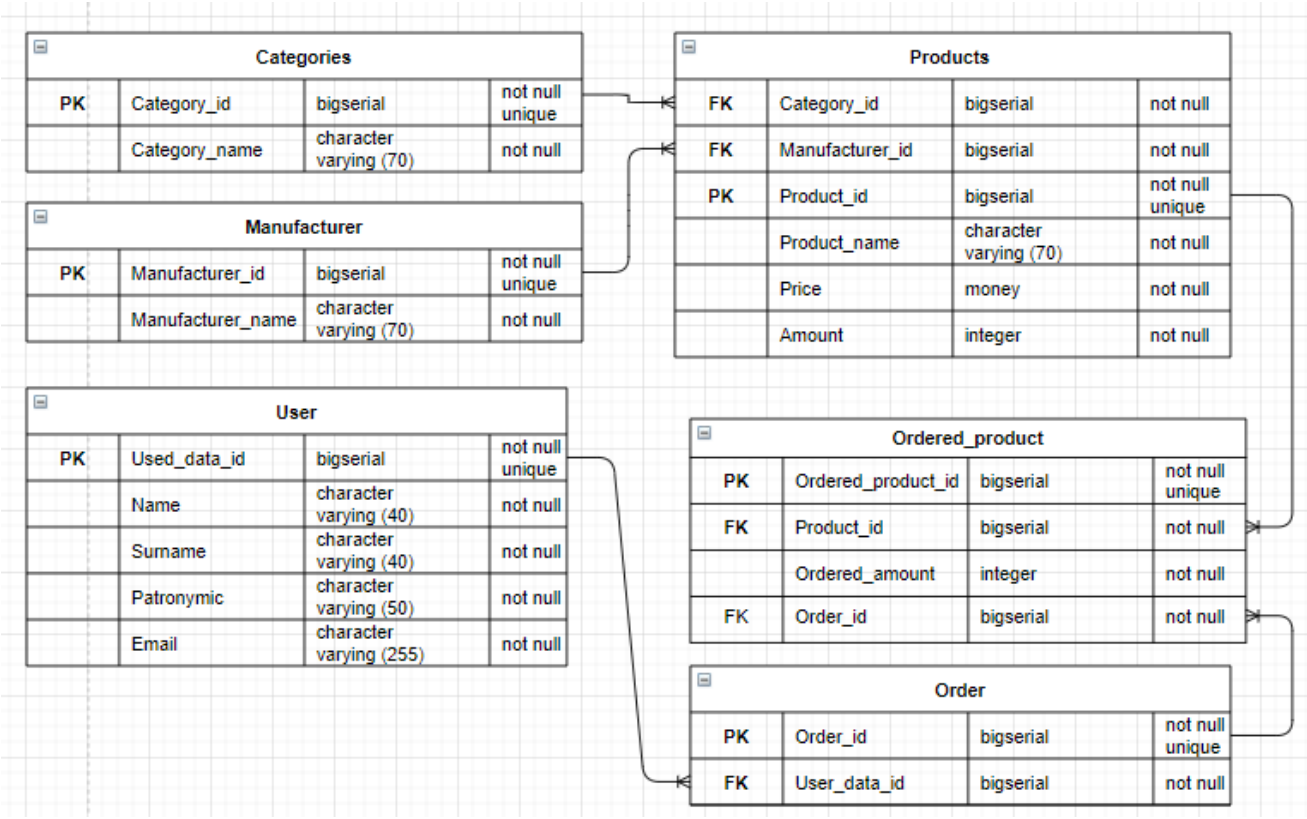


Рисунок 2 - Логічна модель предметної області “Магазин ”.

Для перетворення модуля “Model” програми, створеного в 2 лабораторній роботі, у вигляд об’єктно-реляційної моделі було використано бібліотеку “peewee”

Код сутносних класів програми:

```python

database_proxy = peewee.DatabaseProxy()


class Shop_table(peewee.Model):
    class Meta:
        database = database_proxy
        schema = f"Shop"


class Categories(Shop_table):
    Category_name = peewee.CharField(max_length=70, null=False)


class Manufacturer(Shop_table):
    Manufacturer_name = peewee.CharField(max_length=70, null=False)



class Products(Shop_table):
    Category_id = peewee.ForeignKeyField(Categories, backref="categories")
    Manufacturer_id = peewee.ForeignKeyField(Manufacturer, backref="manufacturer")
    Product_name = peewee.CharField(max_length=70, null=False)
    Price = peewee.DecimalField(null=False)
    Amount = peewee.DecimalField(null=False)


class User(Shop_table):
    Name = peewee.CharField(max_length=40, null=False)
    Surname = peewee.CharField(max_length=40, null=False)
    Patronymic = peewee.CharField(max_length=40, null=False)
    Email = peewee.CharField(max_length=255, null=False)


class Order(Shop_table):
    User_data_id = peewee.ForeignKeyField(User, backref="user")

class Ordered_product(Shop_table):


    Product_id = peewee.ForeignKeyField(Products, backref="products")
    Ordered_amount = peewee.DecimalField(null=False)
    Order_id = peewee.ForeignKeyField(Order, backref="order")

```

Програма працює ідентично програмі з лабораторної роботи 2, за виключенням незначних текстових змін. Інтерфейс модуля «model» не було змінено.

Приклад отримання усіх даних з таблиці «Manufacturer».

Manufacturer.select()

Завдання 2

BTree

Для дослідження індексу була створена таблиця, яка має дві колонки: числову і текстову. Вони проіндексовані як BTree. У таблицю було занесено1000000 записів.

Створення таблиці та її заповнення:

```
CREATE TABLE "test_btree"(
"id" bigserial PRIMARY KEY,
"test_text" varchar(255));
```

```
[bohdan@bohdan ~]$ psql --user user1 --password
Пароль:
psql (13.4)
Введите "help", чтобы получить справку.

user1=> DROP TABLE IF EXISTS "test_btree";
CREATE TABLE "test_btree"(
"id" bigserial PRIMARY KEY,
"test_text" varchar(255)
);
```

```
INSERT INTO "test_btree"("test_text")
SELECT
substr(characters, (random() * length(characters) + 1)::integer, 10)
FROM
(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM'
)) as symbols(characters),
generate_series(1, 1000000) as q;
```

```
user1=> INSERT INTO "test_btree"("test_text")
SELECT
substr(characters, (random() * length(characters) + 1)::integer, 10)
FROM
(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as symbols(characters),
generate_series(1, 1000000) as q;
INSERT 0 1000000
user1=>
```

Вибір даних без індексу:

```
SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0 OR "test_text" LIKE 'b%';
SELECT COUNT(*), SUM("id") FROM "test_btree" WHERE "test_text" LIKE 'b%' GROUP BY "id" % 2;
```

```
user1=> SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0 OR "test_text" LIKE 'b%';
SELECT COUNT(*), SUM("id") FROM "test_btree" WHERE "test_text" LIKE 'b%' GROUP BY "id" % 2;
 count
--------
 500000
(1 строка)

Время: 201,177 мс
 count
--------
 509699
(1 строка)

Время: 224,738 мс
 count |    sum
-------+------------
  9630 | 4821923222
  9699 | 4845619675
(2 строки)

Время: 150,244 мс
```

Сворюємо індекс:
DROP INDEX IF EXISTS "test_btree_test_text_index";
CREATE INDEX "test_btree_test_text_index" ON "test_btree" USING btree ("test_text");

```
user1=> DROP INDEX IF EXISTS "test_btree_test_text_index";
CREATE INDEX "test_btree_test_text_index" ON "test_btree" USING btree ("test_text");
ЗАМЕЧАНИЕ:  индекс "test_btree_test_text_index" не существует, пропускается
DROP INDEX
Время: 0,343 мс
CREATE INDEX
Время: 2081,886 мс (00:02,082)
user1=>
```

Вибір даних з створеним індексом:

```
user1=> SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0 OR "test_text" LIKE 'b%';
SELECT COUNT(*), SUM("id") FROM "test_btree" WHERE "test_text" LIKE 'b%' GROUP BY "id" % 2;
 count
--------
 500000
(1 строка)

Время: 147,508 мс
 count
--------
 509699
(1 строка)

Время: 164,806 мс
 count |    sum
-------+------------
  9630 | 4821923222
  9699 | 4845619675
(2 строки)

Время: 140,613 мс
user1=>
```

BRIN

Для дослідження індексу була створена таблиця, яка має дві колонки: t_data типу timestamp without time zone (дата та час (без часового поясу)) і t_number типу integer. Колонка t_data проіндексована як BRIN. У таблицю занесено 1000000 записів.

Створення таблиці та її заповнення:
DROP TABLE IF EXISTS "test_brin";
CREATE TABLE "test_brin"(
"id" bigserial PRIMARY KEY,
"test_time" timestamp
);
INSERT INTO "test_brin"("test_time")
SELECT

(timestamp '2021-01-01' + random() * (timestamp '2020-01-01' - timestamp '2022-01-01'))
FROM
(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as symbols(characters),
generate_series(1, 1000000) as q;

```
user1=> CREATE TABLE "test_brin"(
"id" bigserial PRIMARY KEY,
"test_time" timestamp
);
INSERT INTO "test_brin"("test_time")
SELECT
(timestamp '2021-01-01' + random() * (timestamp '2020-01-01' - timestamp '2022-01-01'))
FROM
(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as symbols(characters),
generate_series(1, 1000000) as q;
CREATE TABLE
Время: 9,005 мс
INSERT 0 1000000
Время: 5558,990 мс (00:05,559)
user1=>
```

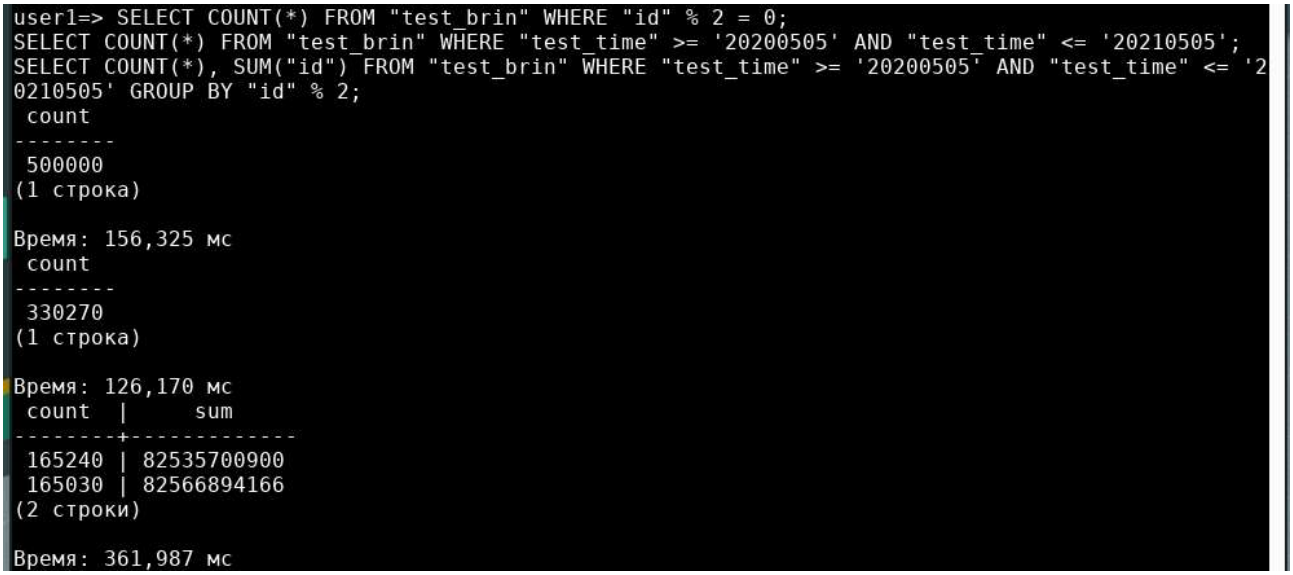
Вибір даних без індексу:
SELECT COUNT(*) FROM "test_brin" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_brin" WHERE "test_time" >= '20200505' AND "test_time" <= '20210505';
SELECT COUNT(*), SUM("id") FROM "test_brin" WHERE "test_time" >= '20200505' AND "test_time" <= '20210505' GROUP BY "id" % 2;

```
SELECT COUNT(*), SUM("id") FROM "test_brin" WHERE "test_time" >= '20200505' AND "test_time" <= '2
0210505' GROUP BY "id" % 2;
 count
--------
 500000
(1 строка)

Время: 195,243 мс
 count
--------
 330270
(1 строка)

Время: 136,224 мс
 count  |     sum
--------+-------------
 165240 | 82535700900
 165030 | 82566894166
(2 строки)

Время: 337,518 мс
```

Сворюємо індекс:
DROP INDEX IF EXISTS "test_brin_test_time_index";
CREATE INDEX "test_brin_test_time_index" ON "test_brin" USING brin ("test_time");

```
user1=> DROP INDEX IF EXISTS "test_brin_test_time_index";
CREATE INDEX "test_brin_test_time_index" ON "test_brin" USING brin ("test_time");
ЗАМЕЧАНИЕ:  индекс "test_brin_test_time_index" не существует, пропускается
DROP INDEX
Время: 0,517 мс
CREATE INDEX
Время: 273,225 мс
user1=>
```

Вибір даних з створеним індексом:



Завдання 3

Розробити тригер бази даних PostgreSQL.
Умова для тригера – before insert, delete.
Таблиці:

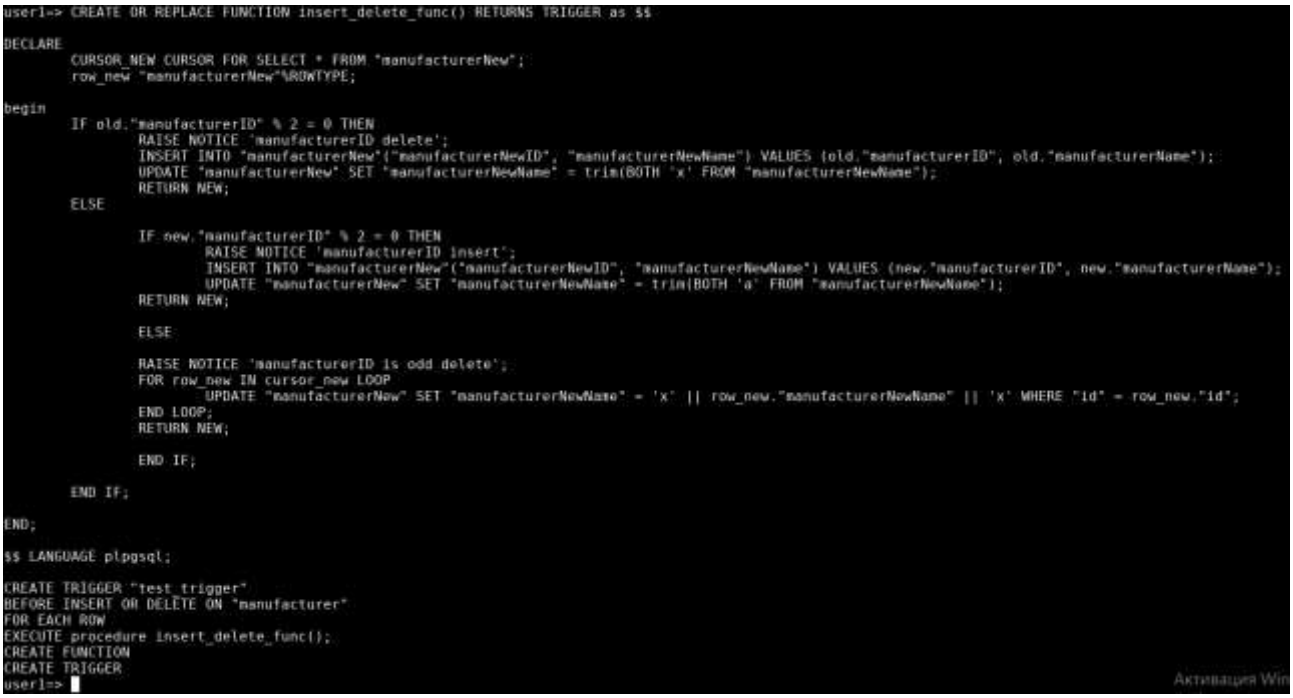
```
DROP TABLE IF EXISTS "manufacturer";
CREATE TABLE "manufacturer"(
        "manufacturerID" bigserial PRIMARY KEY,
        "manufacturerName" varchar(255)
);


DROP TABLE IF EXISTS "manufacturerNew";
CREATE TABLE "manufacturerNew"(
        "id" bigserial PRIMARY KEY,
        "manufacturerNewID" bigint,
        "manufacturerNewName" varchar(255)
);
```

```
user1=> DROP TABLE IF EXISTS "manufacturer";
CREATE TABLE "manufacturer"(
        "manufacturerID" bigserial PRIMARY KEY,
        "manufacturerName" varchar(255)
);


DROP TABLE IF EXISTS "manufacturerNew";
CREATE TABLE "manufacturerNew"(
        "id" bigserial PRIMARY KEY,
        "manufacturerNewID" bigint,
        "manufacturerNewName" varchar(255)
);
ЗАМЕЧАНИЕ:  таблица "manufacturer" не существует, пропускается
DROP TABLE
CREATE TABLE
ЗАМЕЧАНИЕ:  таблица "manufacturerNew" не существует, пропускается
DROP TABLE
CREATE TABLE
user1=>
```

Тригер:

```
CREATE OR REPLACE FUNCTION insert_delete_func() RETURNS TRIGGER as
$$

DECLARE
        CURSOR_NEW CURSOR FOR SELECT * FROM "manufacturerNew";
        row_new "manufacturerNew"%ROWTYPE;

begin
        IF old."manufacturerID" % 2 = 0 THEN
                RAISE NOTICE 'manufacturerID delete';
                INSERT INTO "manufacturerNew"("manufacturerNewID",
"manufacturerNewName") VALUES (old."manufacturerID",
old."manufacturerName");
                UPDATE "manufacturerNew" SET "manufacturerNewName" =
trim(BOTH 'x' FROM "manufacturerNewName");
                RETURN NEW;
        ELSE

                IF new."manufacturerID" % 2 = 0 THEN
                        RAISE NOTICE 'manufacturerID insert';
                        INSERT INTO "manufacturerNew"("manufacturerNewID",
"manufacturerNewName") VALUES (new."manufacturerID",
new."manufacturerName");
                        UPDATE "manufacturerNew" SET "manufacturerNewName" =
trim(BOTH 'a' FROM "manufacturerNewName");
                RETURN NEW;
```

```
        ELSE

        RAISE NOTICE 'manufacturerID add x';
        FOR row_new IN cursor_new LOOP
            UPDATE "manufacturerNew" SET "manufacturerNewName" =
'x' || row_new."manufacturerNewName" || 'x' WHERE "id" = row_new."id";
        END LOOP;
        RETURN NEW;

        END IF;

    END IF;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER "test_trigger"
BEFORE INSERT OR DELETE ON "manufacturer"
FOR EACH ROW
EXECUTE procedure insert_delete_func();
```

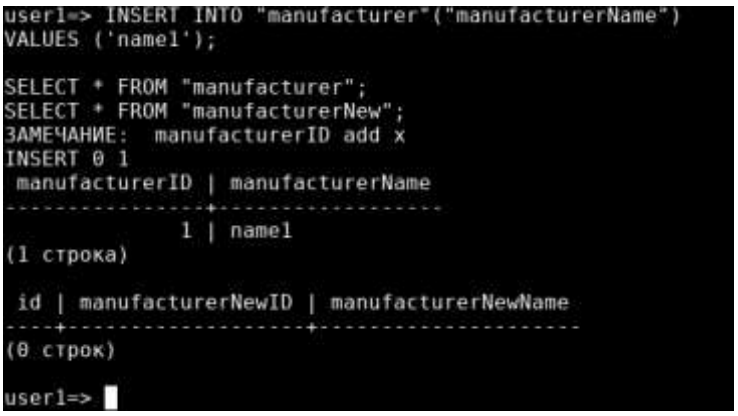


Принцип роботи:
Тригер спрацьовує після видалення з таблиці чи при вставці у таблицю manufacturer. Якщо значення ідентифікатора запису, який видаляється або

вставляється, парне, то цей запис заноситься у додаткову таблицю manufacturerNew. Також, з кожного значення «manufacturerNewName» видаляються символи «a» на початку і кінці при вставці в таблицю manufacturer та видаляються «x» при видаленні з основної таблиці за париним індексом в таблицю manufacturerNew, при видаленні з таблиці manufacturer. Якщо значення ідентифікатора непарне, то до кожного значення «manufacturerNewName» у таблиці manufacturerNew додається “x” на початку і кінці.

Занесемо тестові дані до таблиці:
INSERT INTO "manufacturer"("manufacturerName")
VALUES ('name1');

SELECT * FROM "manufacturer";
SELECT * FROM "manufacturerNew";



«x» нікуди не додався адже в таблиці manufacturerNew ще немає рядків

Занесемо рядок з парним індексом до таблиці:
INSERT INTO "manufacturer"("manufacturerName")
VALUES ('aaauseraaa2');
SELECT * FROM "manufacturer";
SELECT * FROM "manufacturerNew";

Видалилися літери «a» у таблиці manufacturerNew

Занесемо рядок з непарним індексом до таблиці:

INSERT INTO "manufacturer"("manufacturerName")
VALUES ('name3');
SELECT * FROM "manufacturer";
SELECT * FROM "manufacturerNew";

```
user1=> INSERT INTO "manufacturer"("manufacturerName")
VALUES ('name3');
SELECT * FROM "manufacturer";
SELECT * FROM "manufacturerNew";
ЗАМЕЧАНИЕ:  manufacturerID add x
INSERT 0 1
 manufacturerID | manufacturerName
----------------+------------------
              1 | name1
              2 | aaauseraaa2
              3 | name3
(3 строки)

 id | manufacturerNewID | manufacturerNewName
----+-------------------+---------------------
  1 |                 2 | xuseraaa2x
(1 строка)

user1=>
```

Додалися літери «x» у таблиці manufacturerNew на початок та кінець слова

Додамо ще декілька записів:
INSERT INTO "manufacturer"("manufacturerName")
VALUES ('name4aaaa'), ('name5'), ('aaaname6');
SELECT * FROM "manufacturer";
SELECT * FROM "manufacturerNew";

```
user1=> INSERT INTO "manufacturer"("manufacturerName")
VALUES ('name4aaaa'), ('name5'), ('aaaname6');
SELECT * FROM "manufacturer";
SELECT * FROM "manufacturerNew";
ЗАМЕЧАНИЕ:  manufacturerID insert
ЗАМЕЧАНИЕ:  manufacturerID add x
ЗАМЕЧАНИЕ:  manufacturerID insert
INSERT 0 3
 manufacturerID | manufacturerName
----------------+------------------
              1 | name1
              2 | aaauseraaa2
              3 | name3
              4 | name4aaaa
              5 | name5
              6 | aaaname6
(6 строк)

 id | manufacturerNewID | manufacturerNewName
----+-------------------+---------------------
  1 |                 2 | xxuseraaa2xx
  2 |                 4 | xname4x
  3 |                 6 | name6
(3 строки)

user1=>
```

Як ми бачимо парні записи були занесені у додаткову таблицю без літер «а», а також при додаванні непарних у додатковій таблиці з’явилися літери «x» на записах у додатковій таблиці.

При видаленні парного запису літери «x» видаляються з полів додаткової таблиці, при видаленні непарного додаються на початок та кінець кожного поля додаткової таблиці при цьому значення з основної таблиці не видаляються :

DELETE FROM "manufacturer" WHERE "manufacturerID" = 3;
SELECT * FROM "manufacturer";
SELECT * FROM "manufacturerNew";

```
user1=> DELETE FROM "manufacturer" WHERE "manufacturerID" = 3;
ЗАМЕЧАНИЕ:  manufacturerID add x
DELETE 0
user1=> SELECT * FROM "manufacturer";
SELECT * FROM "manufacturerNew";
 manufacturerID | manufacturerName
----------------+------------------
              1 | name1
              2 | aaauseraaa2
              3 | name3
              4 | name4aaaa
              5 | name5
              6 | aaaname6
(6 строк)

 id | manufacturerNewID | manufacturerNewName
----+-------------------+---------------------
  1 |                 2 | xxxuseraaa2xxx
  2 |                 4 | xxname4xx
  3 |                 6 | xname6x
(3 строки)

user1=>
```

DELETE FROM "manufacturer" WHERE "manufacturerID" = 2;
SELECT * FROM "manufacturer";
SELECT * FROM "manufacturerNew";

```
user1=> DELETE FROM "manufacturer" WHERE "manufacturerID" = 2;
SELECT * FROM "manufacturer";
SELECT * FROM "manufacturerNew";
ЗАМЕЧАНИЕ:  manufacturerID delete
DELETE 0
 manufacturerID | manufacturerName
----------------+------------------
              1 | name1
              2 | aaauseraaa2
              3 | name3
              4 | name4aaaa
              5 | name5
              6 | aaaname6
(6 строк)

 id | manufacturerNewID | manufacturerNewName
----+-------------------+---------------------
  1 |                 2 | useraaa2
  2 |                 4 | name4
  3 |                 6 | name6
  4 |                 2 | aaauseraaa2
(4 строки)

user1=>
```

Завдання 4

Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

Самі транзакції особливих пояснень не вимагають, транзакція — це N (N≥1) запитів до БД, які успішно виконуються всі разом або зовсім не виконуються. Ізольованість транзакції показує те, наскільки сильно вони впливають одне на одного паралельно виконуються транзакції.

Вибираючи рівень транзакції, ми намагаємося дійти консенсусу у виборі між високою узгодженістю даних між транзакціями та швидкістю виконання цих транзакцій.

Варто зазначити, що найвищу швидкість виконання та найнижчу узгодженість має рівень read uncommitted. Найнижчу швидкість виконання та найвищу узгодженість — serializable.

При паралельному виконанні транзакцій можливі виникненя таких проблем:

1. **Втрачене оновлення**

Ситуація, коли при одночасній зміні одного блоку даних різними транзакціями, одна зі змін втрачається.

2. **«Брудне» читання**

Читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (відкотиться).

3. **Неповторюване читання**

Ситуація, коли при повторному читанні в рамках однієї транзакції, раніше прочитані дані виявляються зміненими.

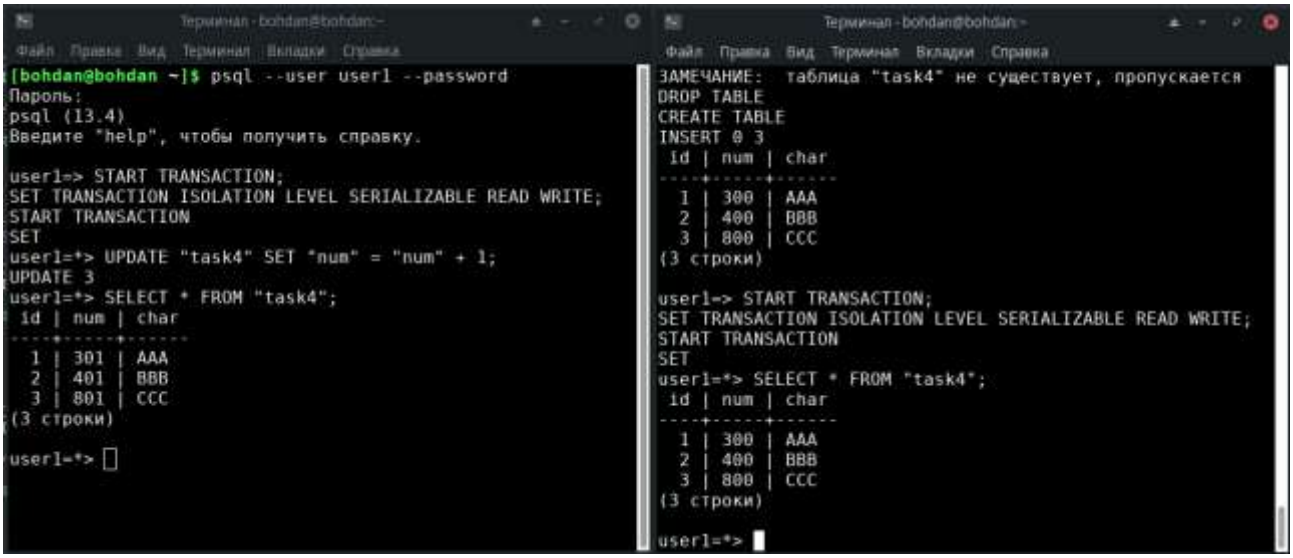
4. **Фантомне читання**

Ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків.

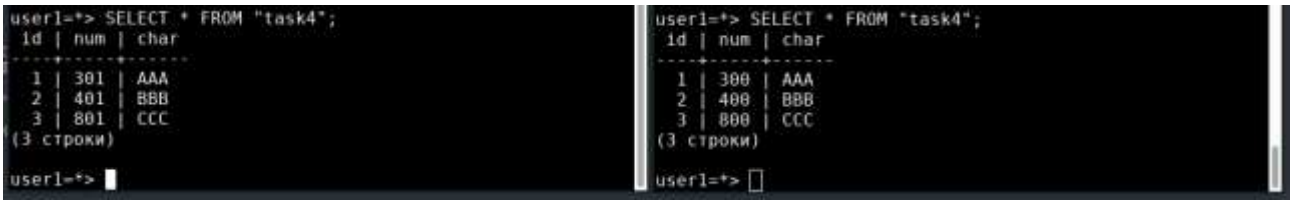
Стандарт SQL-92 визначає наступні рівні ізоляції:

1. **Serializable (впорядкованість)**

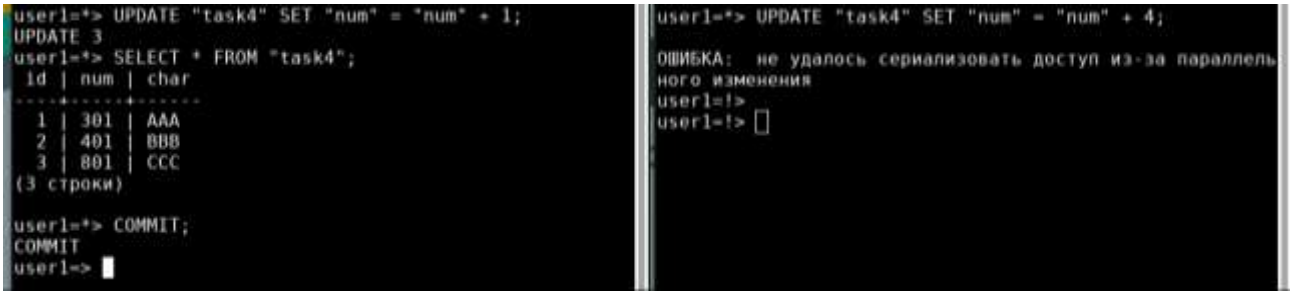
Найбільш високий рівень ізольованості; транзакції повністю ізолюються одна від одної. На цьому рівні результати паралельного виконання транзакцій для бази даних у більшості випадків можна вважати такими, що збігаються з послідовним виконанням тих же транзакцій (по черзі в будь-якому порядку).



Як бачимо, дані у транзаціях ізольовано.

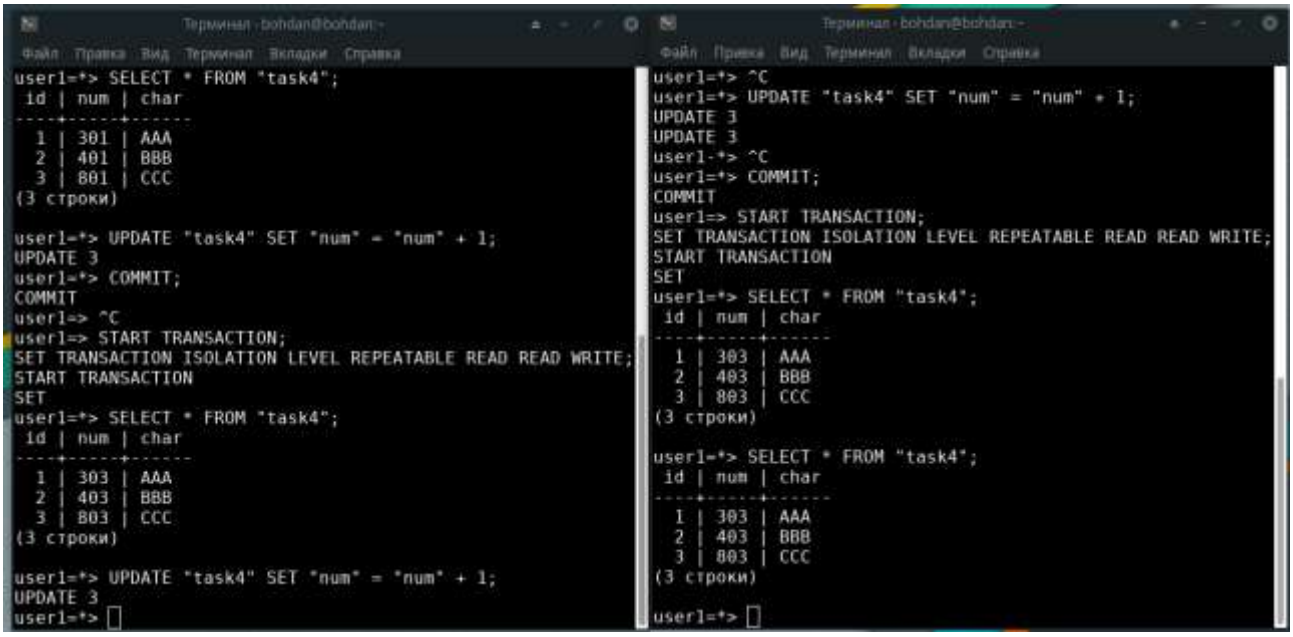


Тепер при оновлені даних в T2(частина фото зправа) бачимо, що T2 блокується поки T1 не не зафіксує зміни або не відмінить їх.

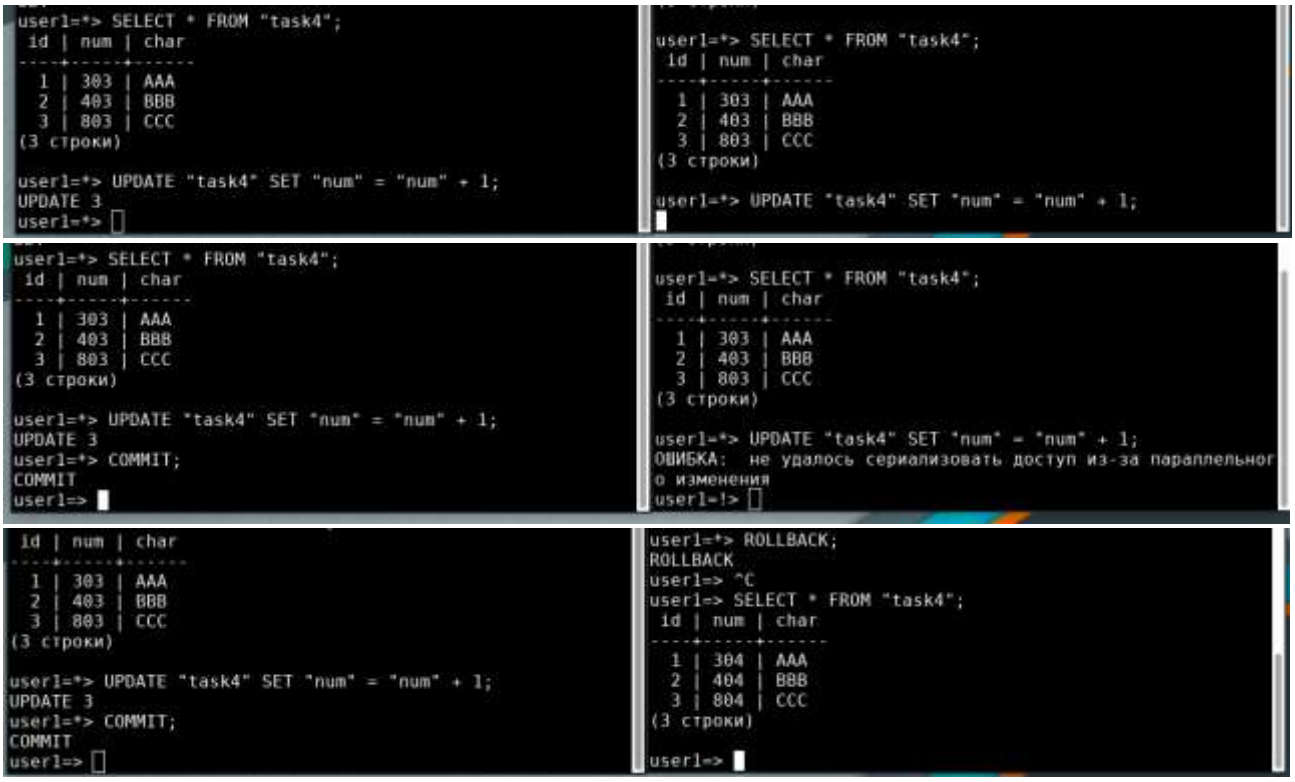


2. Repeatable read (повторюваність читання)

Рівень, при якому читання одного і того ж рядку чи рядків в транзакції дає однаковий результат. (Поки транзакція не закінчена, ніякі інші транзакції не можуть змінити ці дані).

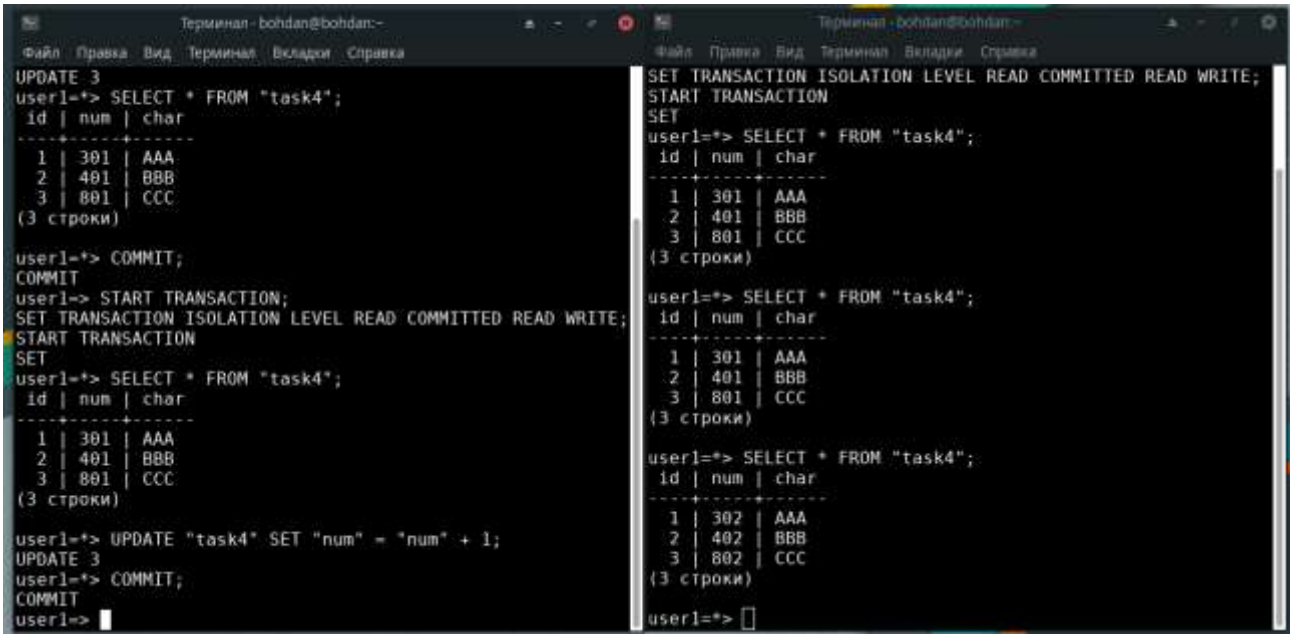


Тепер транзакція T2(зправа) буде чекати поки T1 не не зафіксує зміни або не відмінить їх.



3. Read committed (читання фіксованих даних)

Прийнятий за замовчуванням рівень для PostgreSQL. Закінчене читання, при якому відсутнє «брудне» читання (тобто, читання одним користувачем даних, що не були зафіксовані в БД командою COMMIT). Проте, в процесі роботи однієї транзакції інша може бути успішно закінчена, і зроблені нею зміни зафіксовані. В підсумку, перша транзакція буде працювати з іншим набором даних. Це проблема неповторюваного читання.



4. Read uncommitted (читання незафіксованих даних)

Найнижчий рівень ізоляції, який відповідає рівню 0. Він гарантує тільки відсутність втрачених оновлень. Якщо декілька транзакцій одночасно намагались змінювати один і той же рядок, то в кінцевому варіанті рядок буде мати значення, визначений останньою успішно виконаною транзакцією. У PostgreSQL READ UNCOMMITTED розглядається як READ COMMITTED.

Посилання на репозиторій: https://github.com/OrelBogdan/lab3_bd