НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ
Кафедра системного програмування та спеціалізованих
комп'ютерних систем

Лабораторна робота №2

з дисципліни: «Бази даних і засоби управління»

Тема: «Засоби оптимізації роботи СУБД PostgreSQL»

Виконала: студентка III курсу
ФПМ групи КВ-13
Щербина Н. І.
Телеграм
Перевірив: Петрашенко А.В.

Київ 2023

Лабораторна робота № 2.

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

Завдання роботи полягає у наступному:

1. Перетворити модуль “Модель” з шаблону MVC РГР 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 24

24	*GIN, BRIN*	*after update, insert*

Додаткова інформація

URL репозиторію з вихідним кодом та звітом - github.

Відомості про предметну галузь з лабораторної роботи №1

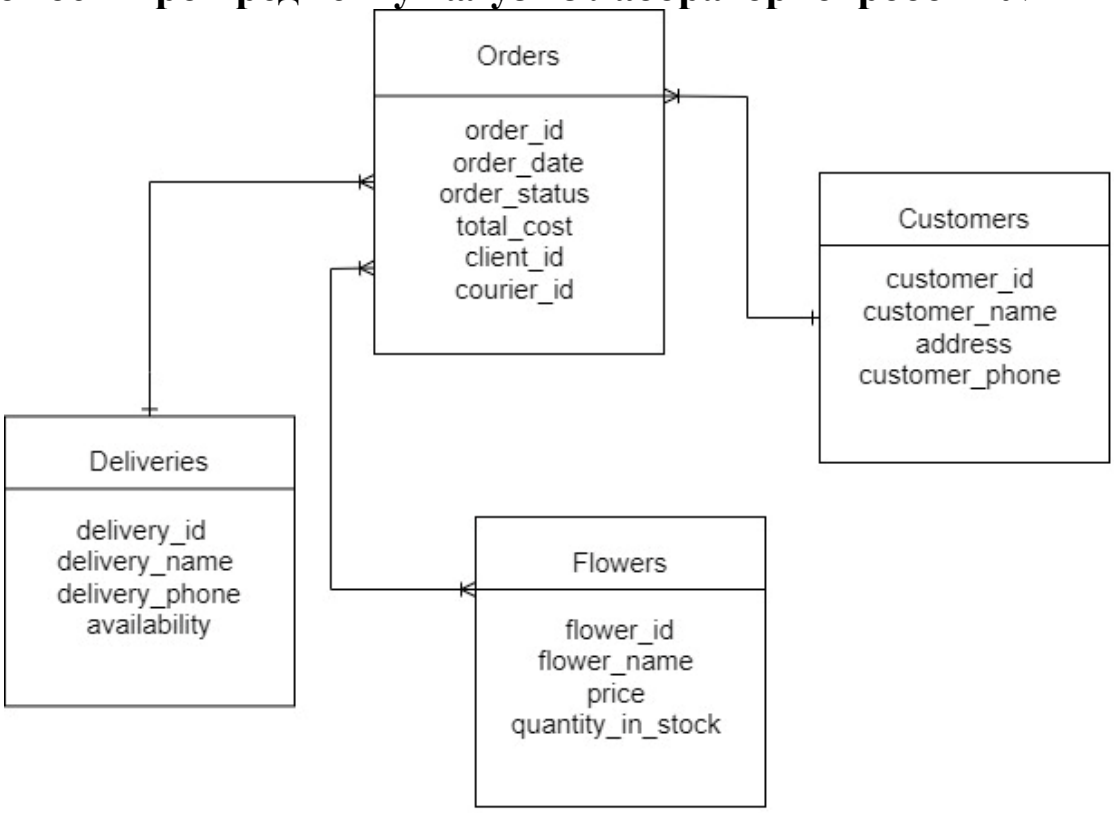


Рисунок 1 – Концептуальна модель предметної області «*Система управління замовленнями та доставкою квітів*».
Нотація: «Пташина лапка». Модель побудована засобами програми draw.io

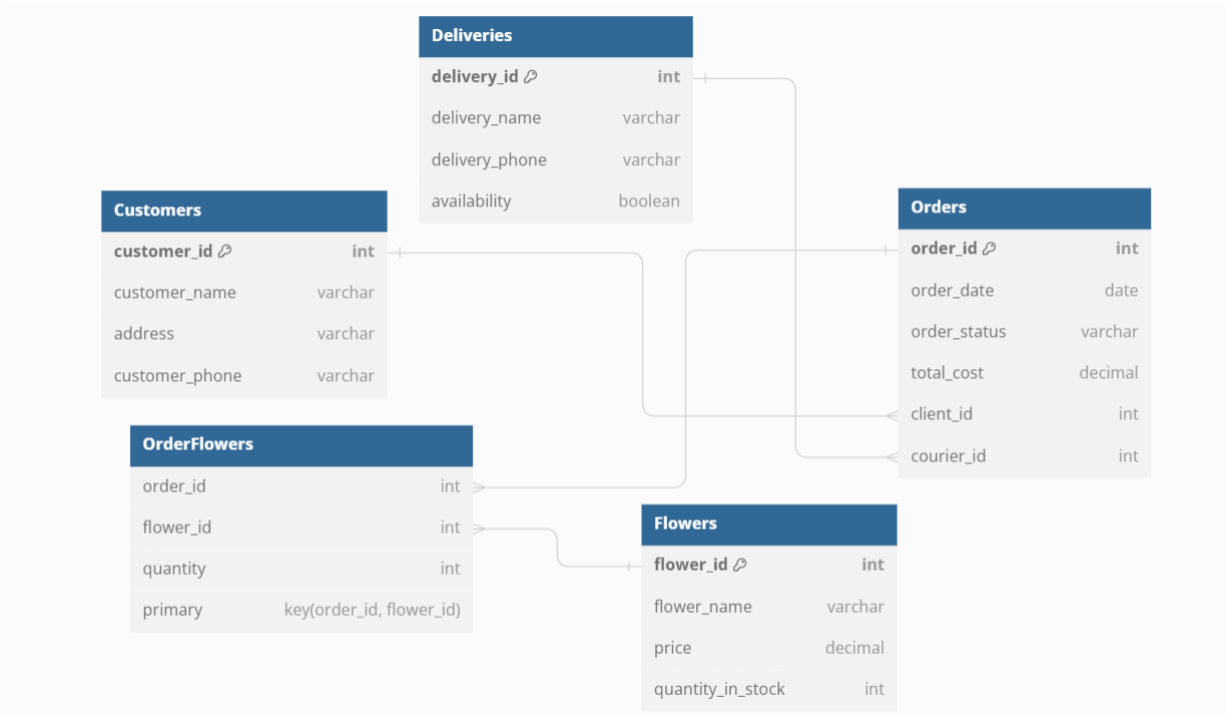


Рисунок 2 – Логічна модель БД « *Система управління замовленнями та доставкою квітів* »
Нотація: Модель побудована засобами dbdiagram.io.

Опис предметної галузі

Під час оформуванні обраної предметної галузі «*Система управління замовленнями та доставкою квітів*» було сформовано наступні сутності:

- Клієнт (Customers):

Призначення: Представляє інформацію про клієнтів, які роблять замовлення квітів.

Атрибути: customer_id (ідентифікатор клієнта), customer_name (ім'я клієнта), address (адреса клієнта), customer_phone (номер телефону клієнта).

- Замовлення (Orders):

Призначення: Зберігає дані про замовлення квітів, включаючи інформацію про клієнта, статус замовлення та загальну вартість.

Атрибути: order_id (ідентифікатор замовлення), order_date (дата замовлення), order_status (статус замовлення), total_cost (загальна вартість замовлення), client_id (ідентифікатор клієнта), courier_id (ідентифікатор кур'єра).

- Кур'єр (Deliveries):

Призначення: Представляє інформацію про кур'єрів, які доставляють замовлення квітів.

Атрибути: delivery_id (ідентифікатор кур'єра), delivery_name (ім'я кур'єра), delivery_phone (номер телефону кур'єра), availability

- Квіти (Flowers):

Призначення: Містить інформацію про різні види квітів, які доступні для замовлення.

Атрибути: flower_id (ідентифікатор квіту), flower_name (назва квіту), price (ціна квіту), quantity_in_stock (кількість квітів на складі).

Зв'язки:

Клієнт може розміщувати багато замовлень, але кожне замовлення належить лише одному клієнту. (1:N - Клієнт і Замовлення)

Замовлення може містити багато продуктів, і це працює в іншу сторону. (N:M - Замовлення і Квіти)

Кожне замовлення обслуговується одним кур'єром, але кур'єр може доставляти багато замовлень. (1:N - Замовлення і Кур'єр)

Завдання №1

Для наочності показано на прикладі таблиць вставку, вилучення, редагування даних.

- Редагування

  На прикладі таблиці **Customers** було відредаговано рядок з id 6:

```
customers_id customer_name address            customer_phone
============ ============= ================== ==================
1            Олександр     вул. Квіткова 65   123-456-7890
2            Валерія       вул. Травнева 42/5 987-654-3210
3            Олександр     вул. Сумська 25    0123456789
4            Катерина      вул. Ковалева 6    0987654321
5            7ac5847c7     b4439550f          194be128b
6            aa            aaa                645165
7            aa            aa                 aaa

Enter id of row that you want to UPDATE
'n' => next 15 rows
'b' => previous 15 rows
'r' => return to menu
6
If you don't want to UPDATE column -> write as it was
Input data separated by comma
Table: customers. Input: customer_name->text, address->text, customer_phone->t
Олександра, вул. Книжна 85, 0156864219
UPDATED successfully
```

Після виконання цього запиту таблиця Customers буде виглядати наступним чином:

```
 customers_id customer_name address            customer_phone
============ ============= ================== ==================
1            Олександр     вул. Квіткова 65   123-456-7890
2            Валерія       вул. Травнева 42/5 987-654-3210
3            Олександр     вул. Сумська 25    0123456789
4            Катерина      вул. Ковалева 6    0987654321
5            7ac5847c7     b4439550f          194be128b
6            Олександра     вул. Книжна 85     0156864219
7            aa            aa                 aaa
```

- Вилучення

  Для демонстрації використано таблицю **Deliveries,** було видалено рядок з id 10:

```
deliveries_id delivery_name delivery_phone availability
============= ============= ============== ============
1             Василь        111-222-3333   True
2             Степан        444-555-6666   False
5             Nadypyshka    090            False
7             aaa           aaa            True
8             a             a              True
9             a             a              True
10            934dc03a7     64b3858f0      False
11            82fad4739     9c53abbfc      False
12            43e89fddb     fe48ec751      False
13            58f40a48d     e9372b6ab      True
14            9b7ee1d1c     8cc40195f      True
15            Stas          0671234567     True
16            Max           0123456789     False
17            5c35cf9cc     6832fb1ab      True
18            6ee3206cc     8d7c4d16e      True

Enter id of row that you want to DELETE
'n' => next 15 rows
'b' => previous 15 rows
'r' => return to menu
10
The row DELETED successfully
```

Після успішного виконання цього запиту таблиця **Deliveries** виглядає так:

deliveries_id	delivery_name	delivery_phone	availability
1	Василь	111-222-3333	True
2	Степан	444-555-6666	False
5	Nadypyshka	090	False
7	aaa	aaa	True
8	a	a	True
9	a	a	True
11	82fad4739	9c53abbfc	False
12	43e89fddb	fe48ec751	False
13	58f40a48d	e9372b6ab	True
14	9b7ee1d1c	8cc40195f	True
15	Stas	0671234567	True
16	Max	0123456789	False
17	5c35cf9cc	6832fb1ab	True
18	6ee3206cc	8d7c4d16e	True
19	edfe45902	5fbcbd5d4	False

- Вставка

Для показу роботи використано таблицю **Flowers:**

Початкова таблиця

flowers_id	flower_name	price	quantity_in_stock
1	Тюльпани	10.99	100
2	Рожі	12.99	75
8	xxx	12.0	6
9	04f709d82	7081.0	958
10	624b1f3ed	11072.0	17
11	c4a80b968	24752.0	35
12	92f9eb3cb	22240.0	462
13	3018161e3	6679.0	637
14	aaa	15.0	10
15	QQQ	1.0	2
16	Піони	20.0	60

Процес занесення даних відповідних значень:

```
Table: flowers. Input: flower_name->text, price->numeric, quantity_in_stock
Троянди, 25, 45
Successfully INSERTED data into table:
  flowers_id     flower_name       price     quantity_in_stock
Data INSERTED successfully
```

Таблиця після внесених нового рядка:

flowers_id	flower_name	price	quantity_in_stock
1	Тюльпани	10.99	100
2	Рожі	12.99	75
8	xxx	12.0	6
9	04f709d82	7081.0	958
10	624b1f3ed	11072.0	17
11	c4a80b968	24752.0	35
12	92f9eb3cb	22240.0	462
13	3018161e3	6679.0	637
14	aaa	15.0	10
15	QQQ	1.0	2
16	Піони	20.0	60
17	Троянди	25.0	45

Класи ORM реалізовані в модулі Model.py

```python
from sqlalchemy import Column, Integer, String, Date, Float, Boolean,
ForeignKey, exc
from sqlalchemy.orm import relationship
from config import Session, engine, base
import distutils.util

session = Session()

def connection():
    try:
        base.metadata.create_all(engine)
        print("Successfully CONNECTED to database FLowers")

    except (Exception, exc.DBAPIError) as _ex:
        print("Impossible to CONNECTED to database FLowers\n", _ex)
        session.rollback()

# Definition of classes corresponding to database tables using SQLAlchemy ORM
# Each class represents a table in the database
class Deliveries(base):
            __tablename__ = 'deliveries'
```

```
            deliveries_id = Column(Integer, primary_key=True)
            delivery_name = Column(String)
            delivery_phone = Column(String)
            availability = Column(Boolean)

            def __init__(self, delivery_name, delivery_phone, availability,
deliveries_id=-1):
                self.delivery_name = delivery_name
                self.delivery_phone = delivery_phone
                self.availability = availability
                if deliveries_id != -1:
                    self.deliveries_id = deliveries_id

            def __repr__(self):
                return "{:^12}{:^20}{:^20}{:^15}".format(self.deliveries_id,
self.delivery_name, self.delivery_phone,
                                                        self.availability)

            def __str__(self):
                return
f"{'deliveries_id':^12}{'delivery_name':^20}{'delivery_phone':^20}{'availabil
ity':^15}"


class Flowers(base):
    __tablename__ = 'flowers'

    flowers_id = Column(Integer, primary_key=True)
    flower_name = Column(String)
    price = Column(Float)
    quantity_in_stock = Column(Integer)

    def __init__(self, flower_name, price, quantity_in_stock, flowers_id=-1):
        self.flower_name = flower_name
        self.price = price
        self.quantity_in_stock = quantity_in_stock
        if flowers_id != -1:
            self.flowers_id = flowers_id

    def __repr__(self):
        return "{:^15}{:^15}{:^15}{:^15}".format(
            self.flowers_id, self.flower_name, self.price,
self.quantity_in_stock
        )

    def __str__(self):
        return
f"{'flowers_id':^15}{'flower_name':^15}{'price':^15}{'quantity_in_stock':^15}
"

class Customers(base):
    __tablename__ = 'customers'

    customers_id = Column(Integer, primary_key=True)
    customer_name = Column(String)
    address = Column(String)
    customer_phone = Column(String)

    def __init__(self, customer_name, address, customer_phone, customers_id=-
1):
        self.customer_name = customer_name
        self.address = address
        self.customer_phone = customer_phone
```

```
        if customers_id != -1:
            self.customers_id = customers_id

    def __repr__(self):
        return "{:^12}{:^20}{:^20}{:^15}".format(
            self.customers_id, self.customer_name, self.address,
self.customer_phone
        )

    def __str__(self):
        return
f"{'customers_id':^12}{'customer_name':^20}{'address':^20}{'customer_phone':^
15}"


class Orders(base):
    __tablename__ = 'orders'

    orders_id = Column(Integer, primary_key=True)
    order_date = Column(Date)
    order_status = Column(String)
    total_cost = Column(Integer)
    client_id = Column(Integer, ForeignKey('customers.customers_id'))
    courier_id = Column(Integer, ForeignKey('deliveries.deliveries_id'))

    def __init__(self, order_date, order_status, total_cost, client_id,
courier_id, orders_id=-1):
        self.order_date = order_date
        self.order_status = order_status
        self.total_cost = total_cost
        self.client_id = client_id
        self.courier_id = courier_id
        if orders_id != -1:
            self.orders_id = orders_id

    def __repr__(self):
        return "{:^12}{:^12}{:^20}{:^15}{:^15}{:^15}".format(
            self.orders_id, self.order_date, self.order_status,
            self.total_cost, self.client_id, self.courier_id
        )

    def __str__(self):
        return
f"{'orders_id':^12}{'order_date':^12}{'order_status':^20}{'total_cost':^15}{'
client_id':^15}{'courier_id':^15}"

class OrderFlowers(base):
    __tablename__ = 'orderflowers'

    orderflowers_id = Column(Integer, primary_key=True)
    flowers_id = Column(Integer, ForeignKey('flowers.flowers_id'))
    quantity = Column(Integer)

    flower = relationship("Flowers")

    def __init__(self, flowers_id, quantity, orderflowers_id=-1):
        self.flowers_id = flowers_id
        self.quantity = quantity
        if orderflowers_id != -1:
            self.orderflowers_id = orderflowers_id

    def __repr__(self):
        return "{:^15}{:^15}{:^15}".format(
            self.orderflowers_id, self.flowers_id, self.quantity
```

```
        )

    def __str__(self):
        return f"{'orderflowers_id':^15}{'flowers_id':^15}{'quantity':^15}"


def insert(choice: int, data: list) -> bool:
    if len(data) < 2:
        return False
    else:
        try:
            if choice == 1:
                elem = Deliveries(*data)
            elif choice == 2:
                elem = Orders(*data)
            elif choice == 3:
                elem = Flowers(*data)
            elif choice == 4:
                elem = OrderFlowers(*data)
            elif choice == 5:
                elem = Customers(*data)
            session.add(elem)
            session.commit()
            print("Successfully INSERTED data into table:")
            print(elem)
            return True
        except (Exception, exc.DBAPIError) as _ex:
            print("Impossible to INSERT data into table\n", _ex)
            session.rollback()
            return False

tables = {
    'deliveries': Deliveries,
    'orders': Orders,
    'flowers': Flowers,
    'customers': Customers,
    'orderFlowers': OrderFlowers,
}
def select_by_key(table: str, key_name: str, key_val: str) -> list:
    try:
        table_obj = tables[table]
        return session.query(table_obj).filter(getattr(table_obj, key_name)
== key_val).all()
    except (Exception, exc.DBAPIError) as _ex:
        print(f"Impossible to SELECT data from table {table} by key
{key_name}\n", _ex)
        session.rollback()
        return []

def select_by_table(table: str, quantity: str = '100', offset: str = '0') ->
list:
    try:
        table_obj = tables[table]
        key_name = table_obj.__tablename__ + '_id'  # Пример получения имени
столбца с ID
        return session.query(table_obj).order_by(getattr(table_obj,
key_name).asc()).offset(offset).limit(quantity).all()
    except (Exception, exc.DBAPIError) as _ex:
        print(f"Impossible to SELECT data from table {table}. Exception:
{_ex}")
        session.rollback()
        return []
```

```
def delete(table: str, key_name: str, key_val: str) -> bool:
    try:
        if table == 'deliveries':
            table_obj = Deliveries
        elif table == 'orders':
            table_obj = Orders
        elif table == 'flowers':
            table_obj = Flowers
        elif table == 'orderFlowers':
            table_obj = OrderFlowers
        elif table == 'customers':
            table_obj = Customers

        session.query(table_obj).filter(getattr(table_obj, key_name) ==
key_val).delete()
        return True
    except (Exception, exc.DBAPIError) as _ex:
        print(f"Impossible to DELETE data from table {table}", _ex)
        session.rollback()
        return False

def update(choice: int, data: list, id1: int, id2: int = 0) -> bool:
    if len(data) < 2:
        return False
    else:
        try:
            if choice == 1:

session.query(Deliveries).filter_by(deliveries_id=f"{id1}").update(
                    {Deliveries.delivery_name: data[0],
Deliveries.delivery_phone: data[1],
                     Deliveries.availability:
distutils.util.strtobool(data[2])})
            elif choice == 2:
                 session.query(Orders).filter_by(orders_id=f"{id1}").update(
                {Orders.order_date: data[0], Orders.order_status:
distutils.util.strtobool(data[1]), Orders.total_cost: data[2],
                 Orders.client_id: data[3], Orders.courier_id: data[4]})
            elif choice == 3:
                session.query(Flowers).filter_by(flowers_id=f"{id1}").update(
                    {Flowers.flower_name: data[0], Flowers.price: data[1],
Flowers.quantity_in_stock: data[2]})
            elif choice == 4:

session.query(OrderFlowers).filter_by(orderFlowers_id=f"{id1}").update(
                    {OrderFlowers.flowers_id: data[0], OrderFlowers.quantity:
distutils.util.strtobool(data[1])})
            elif choice == 5:

session.query(Customers).filter_by(customers_id=f"{id1}").update(
                    {Customers.customer_name: data[0], Customers.address:
data[1], Customers.customer_phone: data[2]})
            session.commit()
            return True
        except Exception as _ex:
            print("Impossible to UPDATE data into table", _ex)
            return False
```

Завдання №2

- Теоритичні відомості про GIN:

GIN — це індекс, спрямований на оптимізацію пошуку складених даних, які містяться в рядках. Він призначений для роботи зі складними структурами, такими як текстові рядки або документи, де пошукові запити можуть шукати окремі фрагменти або слова у цих складених значеннях. GIN зберігає пари ключів та списки рядків, де ці ключі з'являються. Оскільки один ключ може мати декілька співпадінь у різних рядках, це дозволяє швидко здійснювати пошук, особливо у випадках, коли одні й ті ж ключі зустрічаються багато разів. GIN спеціалізується на полі типу tsvector, що робить його ефективним для пошуку текстової інформації в базі даних.

- Демонстрація роботи:

Створення та заповнення таблиці БД:

```
DROP TABLE IF EXISTS "another_gin_test";
CREATE TABLE "another_gin_test"(
  "id" bigserial PRIMARY KEY,
  "string" text,
  "gin_vector" tsvector
);
INSERT INTO "another_gin_test"("string")
SELECT substr(characters, (random() * length(characters) + 1)::integer, 10)
FROM (VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as
symbols(characters),
generate_series(1, 1000000) as q;

UPDATE "another_gin_test" SET "gin_vector" = to_tsvector("string");
```

Запит на тестування:

```
SELECT COUNT(*) FROM "another_gin_test" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "another_gin_test" WHERE ("gin_vector" @@
to_tsquery('zxcv'));
SELECT SUM("id") FROM "another_gin_test" WHERE ("gin_vector" @@
to_tsquery('QWERTYUIOP')) OR ("gin_vector" @@ to_tsquery('zxcv'));
SELECT MIN("id"), MAX("id") FROM "another_gin_test" WHERE ("gin_vector" @@
to_tsquery('zxcv')) GROUP BY "id" % 2;
```

Створення індексу:

```
DROP INDEX IF EXISTS "another_gin_index";
CREATE INDEX "another_gin_index" ON "another_gin_test" USING
gin("gin_vector");
```

Для кращого аналізу та виконання запитів використано було команду **EXPLAIN ANALYZE**. Вона дозволяє отримати план виконання запиту та його реальний час виконання.

Скріни виконання запитів

Результат до індексації:

```
postgres=# SELECT COUNT(*) FROM "another_gin_test" WHERE "id" % 2 = 0;
 count
--------
 500000
(1 Ё фюъ)


postgres=# SELECT COUNT(*) FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('zxcv'));
 count
-------
     0
(1 Ё фюъ)


postgres=# SELECT SUM("id") FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('QWERTYUIOP')) OR ("gin_vector" @@ to_tsquery('zxcv'));
     sum
-------------
 14436457931
(1 Ё фюъ)


postgres=# SELECT MIN("id"), MAX("id") FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('zxcv')) GROUP BY "id" % 2;
 min | max
-----+-----
(0 Ё фъ|т)
```

```
postgres=# EXPLAIN ANALYZE SELECT COUNT(*) FROM "another_gin_test" WHERE "id" % 2 = 0;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=22568.42..22568.43 rows=1 width=8) (actual time=145.197..148.279 rows=1 loops=1)
   ->  Gather  (cost=22568.21..22568.42 rows=2 width=8) (actual time=76.009..148.271 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=21568.21..21568.22 rows=1 width=8) (actual time=36.474..36.474 rows=1 loops=3)
               ->  Parallel Seq Scan on another_gin_test  (cost=0.00..21563.00 rows=2083 width=0) (actual time=0.011..30.820 rows=166667 loops=3)
                     Filter: ((id % '2'::bigint) = 0)
                     Rows Removed by Filter: 166667
 Planning Time: 0.077 ms
 Execution Time: 148.304 ms
(10 Ё фъ|т)


postgres=# EXPLAIN ANALYZE SELECT COUNT(*) FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('zxcv'));
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=125693.42..125693.43 rows=1 width=8) (actual time=452.171..457.932 rows=1 loops=1)
   ->  Gather  (cost=125693.21..125693.42 rows=2 width=8) (actual time=451.970..457.925 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=124693.21..124693.22 rows=1 width=8) (actual time=418.035..418.036 rows=1 loops=3)
               ->  Parallel Seq Scan on another_gin_test  (cost=0.00..124688.00 rows=2083 width=0) (actual time=418.032..418.033 rows=0 loops=3)
                     Filter: (gin_vector @@ to_tsquery('zxcv'::text))
                     Rows Removed by Filter: 333333
 Planning Time: 0.119 ms
 Execution Time: 457.969 ms
(10 Ё фъ|т)


postgres=# EXPLAIN ANALYZE SELECT SUM("id") FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('QWERTYUIOP')) OR ("gin_vector" @@ to_tsquery('zxcv'));
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=230932.51..230932.52 rows=1 width=32) (actual time=1108.026..1113.905 rows=1 loops=1)
   ->  Gather  (cost=230932.29..230932.50 rows=2 width=32) (actual time=1107.739..1113.889 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=229932.29..229932.30 rows=1 width=32) (actual time=1045.866..1045.867 rows=1 loops=3)
               ->  Parallel Seq Scan on another_gin_test  (cost=0.00..229896.33 rows=14382 width=8) (actual time=2.060..1044.174 rows=9630 loops=3)
                     Filter: ((gin_vector @@ to_tsquery('QWERTYUIOP'::text)) OR (gin_vector @@ to_tsquery('zxcv'::text)))
                     Rows Removed by Filter: 323703
 Planning Time: 0.087 ms
```

Створення відповідного індексу:

```
postgres=# CREATE INDEX "another_gin_index" ON "another_gin_test" USING gin("gin_vector");
CREATE INDEX
```

Результат після індексації:

```
postgres=# EXPLAIN ANALYZE SELECT COUNT(*) FROM "another_gin_test" WHERE "id" % 2 = 0;
                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=22568.42..22568.43 rows=1 width=8) (actual time=156.141..159.326 rows=1 loops=1)
   ->  Gather  (cost=22568.21..22568.42 rows=2 width=8) (actual time=74.076..159.318 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=21568.21..21568.22 rows=1 width=8) (actual time=36.372..36.372 rows=1 loops=3)
               ->  Parallel Seq Scan on another_gin_test  (cost=0.00..21563.00 rows=2083 width=0) (actual time=0.010..30.444 rows=166667 loops=3)
                     Filter: ((id % '2'::bigint) = 0)
                     Rows Removed by Filter: 166667
 Planning Time: 0.906 ms
 Execution Time: 159.353 ms
(10 Ё фь|т)


postgres=# EXPLAIN ANALYZE SELECT COUNT(*) FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('zxcv'));
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=11758.51..11758.52 rows=1 width=8) (actual time=0.015..0.015 rows=1 loops=1)
   ->  Bitmap Heap Scan on another_gin_test  (cost=71.00..11746.01 rows=5000 width=0) (actual time=0.013..0.013 rows=0 loops=1)
         Recheck Cond: (gin_vector @@ to_tsquery('zxcv'::text))
         ->  Bitmap Index Scan on another_gin_index  (cost=0.00..69.75 rows=5000 width=0) (actual time=0.012..0.012 rows=0 loops=1)
               Index Cond: (gin_vector @@ to_tsquery('zxcv'::text))
 Planning Time: 0.770 ms
 Execution Time: 0.063 ms
(7 Ё фь|т)


postgres=# EXPLAIN ANALYZE SELECT SUM("id") FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('QWERTYUIOP')) OR ("gin_vector" @@ to_tsquery('zxcv'));
                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=24145.90..24145.91 rows=1 width=32) (actual time=55.265..58.414 rows=1 loops=1)
   ->  Gather  (cost=24145.68..24145.89 rows=2 width=32) (actual time=37.810..58.401 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=23145.68..23145.69 rows=1 width=32) (actual time=12.531..12.532 rows=1 loops=3)
               ->  Parallel Bitmap Heap Scan on another_gin_test  (cost=357.76..23109.72 rows=14382 width=8) (actual time=1.502..11.181 rows=9630 loops=3)
                     Recheck Cond: ((gin_vector @@ to_tsquery('QWERTYUIOP'::text)) OR (gin_vector @@ to_tsquery('zxcv'::text)))
                     Heap Blocks: exact=8688
                     ->  BitmapOr  (cost=357.76..357.76 rows=34667 width=0) (actual time=3.425..3.425 rows=0 loops=1)
                           ->  Bitmap Index Scan on another_gin_index  (cost=0.00..270.75 rows=29667 width=0) (actual time=3.401..3.401 rows=28890 loops=1)
                                 Index Cond: (gin_vector @@ to_tsquery('QWERTYUIOP'::text))
                           ->  Bitmap Index Scan on another_gin_index  (cost=0.00..69.75 rows=5000 width=0) (actual time=0.022..0.022 rows=0 loops=1)
                                 Index Cond: (gin_vector @@ to_tsquery('zxcv'::text))
 Planning Time: 0.111 ms
 Execution Time: 58.459 ms
```

На основі аналізу результатів виконання запитів до таблиці "**another_gin_test**" до та після індексації можна зробити наступні висновки:

Попередня продуктивність запитів:

1. Запит SELECT COUNT(*) FROM "another_gin_test" WHERE "id" % 2 = 0; виконувався за близько **148 мс.**
2. Запит SELECT COUNT(*) FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('zxcv')); займав приблизно **458 мс**.
3. Запит SELECT SUM("id") FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('QWERTYUIOP')) OR ("gin_vector" @@ to_tsquery('zxcv')); виконувався за **1.11 с.**
4. Запит SELECT MIN("id"), MAX("id") FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('zxcv')) GROUP BY "id" % 2; займав близько **447 мс.**

Після індексації змінилися так:

1. Запит SELECT COUNT(*) FROM "another_gin_test" WHERE "id" % 2 = 0; тепер виконується приблизно за **159 мс**.
2. Запит SELECT COUNT(*) FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('zxcv')); тепер займає лише **0.063 мс**.

3. Запит SELECT SUM("id") FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('QWERTYUIOP')) OR ("gin_vector" @@ to_tsquery('zxcv')); тепер виконується за **58.5 мс**.
4. Запит SELECT MIN("id"), MAX("id") FROM "another_gin_test" WHERE ("gin_vector" @@ to_tsquery('zxcv')) GROUP BY "id" % 2; тепер займає близько **0.118 мс**.

Висновки:

Після створення відповідного GIN-індексу швидкість запитів, особливо тих, які використовують функцію **to_tsquery** для пошуку в текстових полях, дуже значно покращилась. Операції фільтрації за умовою та групування демонструють істотне покращення швидкості після індексації. Загалом, створення індексу позитивно позначилося на продуктивності запитів, зокрема тих, які використовують повнотекстовий пошук та операції фільтрації. Відчутний приріст швидкості сприяє більш ефективному використанню ресурсів та покращенню продуктивності бази даних.

- Теоритичні відомості про BRIN:

BRIN — це індек, який групує дані у блоки та підтримує інформацію про їх згортку, щоб ефективно працювати з великими обсягами впорядкованих даних. BRIN дозволяє швидше виконувати запити до великих обсягів даних, зменшуючи обсяг пам'яті, потрібної для створення індексу.

- Демонстрація роботи:

Створення та заповнення таблиці БД:

```
DROP TABLE IF EXISTS "another_brin_test";
CREATE TABLE "another_brin_test" (
    "id" bigserial PRIMARY KEY,
    "another_time" timestamp
);
INSERT INTO "another_brin_test"("another_time")
SELECT
    (timestamp '2023-01-01' + random() * (timestamp '2022-01-01' - timestamp
'2024-01-01'))
FROM
    (VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as
symbols(characters),
    generate_series(1, 1000000) as q;
```

Запит на тестування:

```
SELECT COUNT(*) FROM "another_brin_test" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "another_brin_test" WHERE "another_time" >= '20230505'
AND "another_time" <= '20240505';
SELECT COUNT(*), SUM("id") FROM "another_brin_test" WHERE "another_time" >=
'20230505' AND "another_time" <= '20240505' GROUP BY "id" % 2;
```

Створення індексу:

```
DROP INDEX IF EXISTS "another_brin_time_index";
```

```
CREATE INDEX "another_brin_time_index" ON "another_brin_test" USING brin
("another_time");
```

Використовувалась також команда **EXPLAIN ANALYZE.**

Скріни виконання запитів

Результат до індексації:

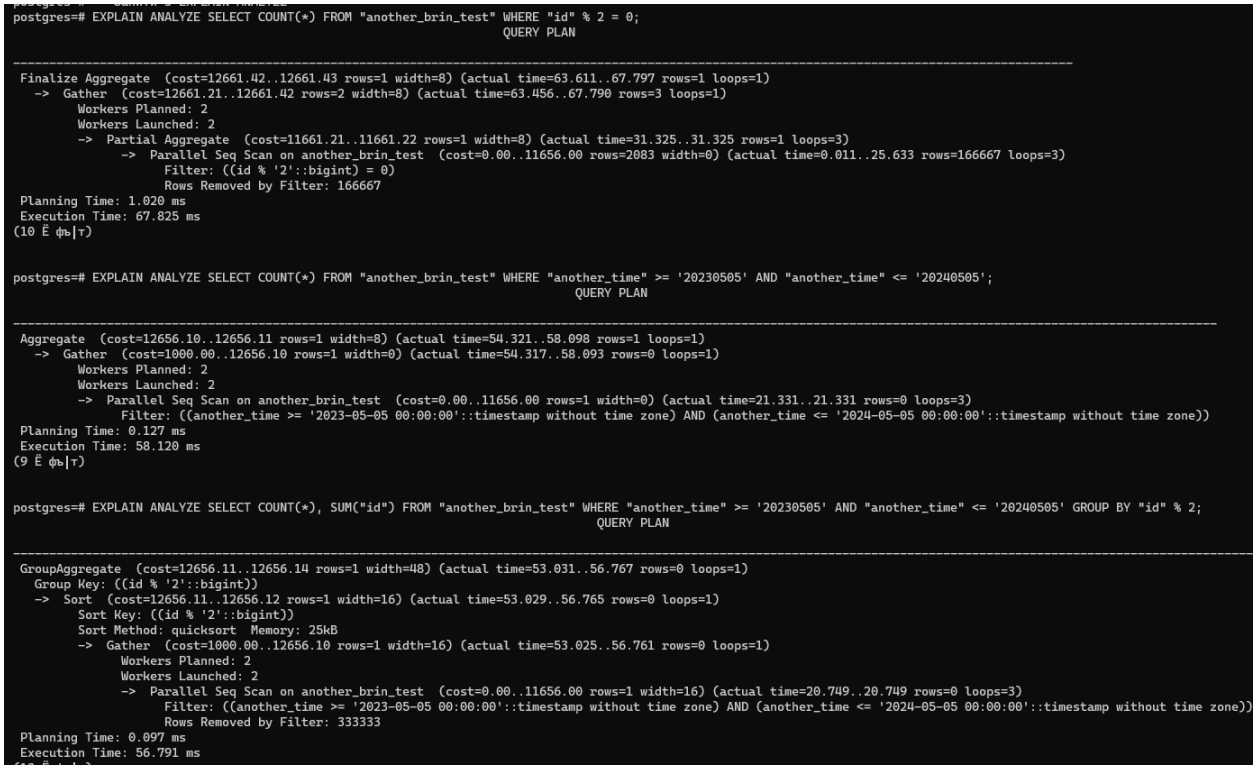
```
postgres=# SELECT COUNT(*) FROM "another_brin_test" WHERE "id" % 2 = 0;
 count
--------
 500000
(1 Ё фмЬ)


postgres=# SELECT COUNT(*) FROM "another_brin_test" WHERE "another_time" >= '20230505' AND "another_time" <= '20240505';
 count
-------
     0
(1 Ё фмЬ)


postgres=# SELECT COUNT(*), SUM("id") FROM "another_brin_test" WHERE "another_time" >= '20230505' AND "another_time" <= '20240505' GROUP BY "id" % 2;
 count | sum
-------+-----
(0 Ё фЬ|т)


postgres=# EXPLAIN ANALYZE SELECT COUNT(*) FROM "another_brin_test" WHERE "id" % 2 = 0;
                                                                  QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=12662.11..12662.12 rows=1 width=8) (actual time=67.298..71.934 rows=1 loops=1)
   ->  Gather  (cost=12661.90..12662.11 rows=2 width=8) (actual time=67.135..71.926 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=11661.90..11661.91 rows=1 width=8) (actual time=34.143..34.144 rows=1 loops=3)
               ->  Parallel Seq Scan on another_brin_test  (cost=0.00..11656.69 rows=2084 width=0) (actual time=0.012..28.263 rows=166667 loops=3)
                     Filter: ((id % '2'::bigint) = 0)
                     Rows Removed by Filter: 166667
 Planning Time: 0.081 ms
 Execution Time: 71.960 ms
(10 Ё фЬ|т)


postgres=# EXPLAIN ANALYZE SELECT COUNT(*) FROM "another_brin_test" WHERE "another_time" >= '20230505' AND "another_time" <= '20240505';
                                                                        QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=12662.11..12662.12 rows=1 width=8) (actual time=55.859..59.708 rows=1 loops=1)
   ->  Gather  (cost=12661.90..12662.11 rows=2 width=8) (actual time=55.741..59.702 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=11661.90..11661.91 rows=1 width=8) (actual time=23.580..23.581 rows=1 loops=3)
               ->  Parallel Seq Scan on another_brin_test  (cost=0.00..11656.69 rows=2084 width=0) (actual time=23.578..23.578 rows=0 loops=3)
                     Filter: ((another_time >= '2023-05-05 00:00:00'::timestamp without time zone) AND (another_time <= '2024-05-05 00:00:00'::timestamp without time zone))
                     Rows Removed by Filter: 333333
 Planning Time: 0.060 ms
 Execution Time: 59.737 ms
(10 Ё фЬ|т)
```

```
postgres=# EXPLAIN ANALYZE SELECT COUNT(*) FROM "another_brin_test" WHERE "another_time" >= '20230505' AND "another_time" <= '20240505';
                                                                        QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=12662.11..12662.12 rows=1 width=8) (actual time=55.859..59.708 rows=1 loops=1)
   ->  Gather  (cost=12661.90..12662.11 rows=2 width=8) (actual time=55.741..59.702 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=11661.90..11661.91 rows=1 width=8) (actual time=23.580..23.581 rows=1 loops=3)
               ->  Parallel Seq Scan on another_brin_test  (cost=0.00..11656.69 rows=2084 width=0) (actual time=23.578..23.578 rows=0 loops=3)
                     Filter: ((another_time >= '2023-05-05 00:00:00'::timestamp without time zone) AND (another_time <= '2024-05-05 00:00:00'::timestamp without time zone))
                     Rows Removed by Filter: 333333
 Planning Time: 0.060 ms
 Execution Time: 59.737 ms
(10 Ё фЬ|т)


postgres=# EXPLAIN ANALYZE SELECT COUNT(*), SUM("id") FROM "another_brin_test" WHERE "another_time" >= '20230505' AND "another_time" <= '20240505' GROUP BY "id" % 2;
                                                                        QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize HashAggregate  (cost=13167.27..13242.28 rows=5001 width=48) (actual time=55.871..59.792 rows=0 loops=1)
   Group Key: ((id % '2'::bigint))
   Batches: 1  Memory Usage: 217kB
   ->  Gather  (cost=12677.53..13125.59 rows=4168 width=48) (actual time=55.857..59.777 rows=0 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial HashAggregate  (cost=11677.53..11708.79 rows=2084 width=48) (actual time=22.922..22.922 rows=0 loops=3)
               Group Key: (id % '2'::bigint)
               Batches: 1  Memory Usage: 121kB
               Worker 0:  Batches: 1  Memory Usage: 121kB
               Worker 1:  Batches: 1  Memory Usage: 121kB
               ->  Parallel Seq Scan on another_brin_test  (cost=0.00..11661.90 rows=2084 width=16) (actual time=22.912..22.912 rows=0 loops=3)
                     Filter: ((another_time >= '2023-05-05 00:00:00'::timestamp without time zone) AND (another_time <= '2024-05-05 00:00:00'::timestamp without time zone))
                     Rows Removed by Filter: 333333
 Planning Time: 0.091 ms
 Execution Time: 59.956 ms
(16 Ё фЬ|т)
```

Створення відповідного індексу:

```
postgres=# CREATE INDEX "another_brin_time_index" ON "another_brin_test" USING brin ("another_time");
CREATE INDEX
```

Результат після індексації:



Після проведення експериментів, результати виконання запитів в базі даних показали певні відмінності у часі виконання до та після створення BRIN-індексу "another_brin_time_index" для колонки "another_time" у таблиці "another_brin_test".

Початкові запити показали такі результати:

1. Запит SELECT COUNT(*) FROM "another_brin_test" WHERE "id" % 2 = 0; виконувався приблизно за **71.96 мс.**
2. Запит SELECT COUNT(*) FROM "another_brin_test" WHERE "another_time" >= '20230505' AND "another_time" <= '20240505'; виконувався приблизно за **59.74 мс.**
3. Запит SELECT COUNT(*), SUM("id") FROM "another_brin_test" WHERE "another_time" >= '20230505' AND "another_time" <= '20240505' GROUP BY "id" % 2; не повертав результатів через відсутність записів, що відповідають умові дати.

Після створення BRIN-індексу "another_brin_time_index" час виконання запитів зазначено нижче:

1. Запит SELECT COUNT(*) FROM "another_brin_test" WHERE "id" % 2 = 0; виконується за **67.83 мс.**
2. Запит SELECT COUNT(*) FROM "another_brin_test" WHERE "another_time" >= '20230505' AND "another_time" <= '20240505'; виконується за **58.12 мс.**

3. Запит SELECT COUNT(*), SUM("id") FROM "another_brin_test" WHERE "another_time" >= '20230505' AND "another_time" <= '20240505' GROUP BY "id" % 2; виконується за **56.79 мс.**

Загалом, створення BRIN-індексу "another_brin_time_index" для колонки "another_time" у таблиці "another_brin_test" призвело до покращення швидкодії деяких запитів, зменшивши час їх виконання.

Завдання №3

Для перевірки роботи тригера були створені дві таблиці.

```
DROP TABLE IF EXISTS "trigger_test";
CREATE TABLE "trigger_test"(
"trigger_testID" bigserial PRIMARY KEY, "trigger_testName" text);
DROP TABLE IF EXISTS "trigger_test_log";
CREATE TABLE "trigger_test_log"(
"id" bigserial PRIMARY KEY, "trigger_test_log_ID" bigint,
"trigger_test_log_name" text);
```

Внесено наступні дані:

```
INSERT INTO "trigger_test"("trigger_testName") VALUES ('trigger_test1'),
('trigger_test2'), ('trigger_test3'), ('trigger_test4'), ('trigger_test5'),
('trigger_test6'), ('trigger_test7'), ('trigger_test8'), ('trigger_test9'),
('trigger_test10');
```

Команди, що запускають тригер на виконання:

```
CREATE TRIGGER "before_delete_update_trigger"
BEFORE DELETE OR UPDATE ON "trigger_test"
FOR EACH ROW EXECUTE procedure before_delete_update_func();
```

Відповідний лістінг тригера:

```
CREATE OR REPLACE FUNCTION before_delete_update_func()
RETURNS TRIGGER as $trigger$ DECLARE
CURSOR_LOG CURSOR FOR SELECT * FROM "trigger_test_log";
row_ "trigger_test_log"%ROWTYPE;
BEGIN
IF old."trigger_testID" % 2 = 0 THEN
IF old."trigger_testID" % 3 = 0 THEN
RAISE NOTICE 'trigger_testID is multiple of 2 and 3';
FOR row_ IN CURSOR_LOG LOOP
UPDATE "trigger_test_log" SET "trigger_test_log_name" = '_' ||
row_."trigger_test_log_name" || '_log' WHERE "id" = row_."id";
END LOOP;
RETURN OLD;
ELSE
RAISE NOTICE 'trigger_testID is even';
INSERT INTO "trigger_test_log"
("trigger_test_log_ID","trigger_test_log_name")
VALUES (old."trigger_testID", old."trigger_testName");
UPDATE "trigger_test_log" SET "trigger_test_log_name" =
trim(BOTH '_log' FROM "trigger_test_log_name");
RETURN NEW;
```

```
END IF;
ELSE
RAISE NOTICE 'trigger_testID is odd';
FOR row_ IN CURSOR_LOG LOOP
UPDATE "trigger_test_log" SET "trigger_test_log_name" = '_' ||
row_."trigger_test_log_name" || '_log' WHERE "id" = row_."id";
END LOOP;
RETURN OLD;
END IF;
END;
$trigger$ LANGUAGE plpgsql;
```

Скріни зі змінами у таблицях бази даних

Початкові дані:

```
SELECT * FROM "trigger_test";
```

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	2	trigger_test2
3	3	trigger_test3
4	4	trigger_test4
5	5	trigger_test5
6	6	trigger_test6
7	7	trigger_test7
8	8	trigger_test8
9	9	trigger_test9

```
SELECT * FROM "trigger_test_log";
```

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text

Після виконання запиту на оновлення даних в таблиці:

```
UPDATE "trigger_test" SET "trigger_testName" = "trigger_testName" || '_log'
WHERE "trigger_testID" % 2 = 0;
```

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	3	trigger_test3
3	5	trigger_test5
4	7	trigger_test7
5	9	trigger_test9
6	2	trigger_test2_log
7	4	trigger_test4_log
8	6	trigger_test6
9	8	trigger_test8_log

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text
1	1	2	trigger_test2
2	2	4	trigger_test4
3	3	8	trigger_test8

За результатами виконання запиту на оновлення даних у таблиці, спостерігається виконання гілок алгоритму тригера. Одна гілка відповідає за парні рядки (з умовою для парних записів) і вона виконується, підтверджуючи зміни. Однак, для рядка з номером 6, тригер виконав іншу вкладену гілку алгоритму, що призвело до повернення до попереднього стану (OLD) цього рядка. При запиті на оновлення даних, потрібно враховувати, що необхідно повертати новий стан, тоді як при запиті на видалення - старий стан.

Після виконання запиту на видалення записів з таблиці:

```
DELETE FROM "trigger_test" WHERE "trigger_testID" % 3 = 0;
```

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	2	trigger_test2
3	4	trigger_test4
4	5	trigger_test5
5	7	trigger_test7
6	8	trigger_test8

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text

Коли виконуються ці запити окремо, таблиця **trigger_test** втрачає рядки, які є кратними трьом, але таблиця **trigger_test_log** залишається пустою. Це відбувається через те, що у гілці алгоритму для чисел кратних трьом у таблиці **trigger_test_log** лише змінюються існуючі записи, нові ж не додаються. Оскільки перед цим не відбулося оновлення, ця таблиця залишається порожньою і її нема за що модифікувати.

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	5	trigger_test5
3	7	trigger_test7
4	2	trigger_test2_log
5	4	trigger_test4_log
6	8	trigger_test8_log

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text
1	1	2	___trigger_test2_log_log_log
2	2	4	___trigger_test4_log_log_log
3	3	8	___trigger_test8_log_log_log

У результаті послідовного виконання цих запитів відбулися наступні зміни: рядки, що були кратні трьом у таблиці **trigger_test**, були видалені. При цьому до текстових полів цих видалених рядків було додано суфікс "*log". У таблиці* ***trigger_test_log*** *також сталися зміни: текстовим полям додалися два символи "*" в початок і три "_log" в кінець. Один "_log" в кінці був доданий через виконання запиту на оновлення для всіх парних рядків. Два "*log" та два символи "*" на початку додались через те, що запити на видалення для записів 3 та 9 виконались через одну гілку алгоритму (кратні трьом), а запит на видалення запису 6 виконався через іншу гілку (згідно кратності 2 та 3).

Завдання №4

Транзакція в базі даних - це сукупність одного чи більше запитів, які виконуються як один блок. Ці запити можуть бути виконані або взагалі не виконані, або ж виконані разом.

Рівень ізоляції транзакцій визначає ступінь, до якої одна транзакція впливає на інші, якщо вони виконуються паралельно. Вибір рівня ізоляції полягає у знаходженні балансу між узгодженістю даних між транзакціями та швидкістю їх виконання.

Різні рівні ізоляції мають різні властивості. Наприклад, рівень **READ UNCOMMITTED** може забезпечити високу швидкість виконання, але при цьому допускає "брудне" читання та інші аномалії. У той же час, рівень **SERIALIZABLE** має найвищу узгодженість, але може викликати затримки через свою строгість у виконанні транзакцій.

Паралельне виконання транзакцій може призвести до різних проблем, таких як втрачене оновлення, "брудне" читання, неповторюване читання та фантомне читання. Ці проблеми виникають через взаємодію паралельно виконуваних транзакцій та зміни даних.

Стандарт SQL-92 визначає різні рівні ізоляції, які визначають ступінь узгодженості даних між паралельно виконуваними транзакціями. Наприклад, рівень **Serializable** гарантує найвищу ізоляцію, де результати виконання транзакцій майже ідентичні послідовному виконанню тих же транзакцій, незалежно від порядку виконання.

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.

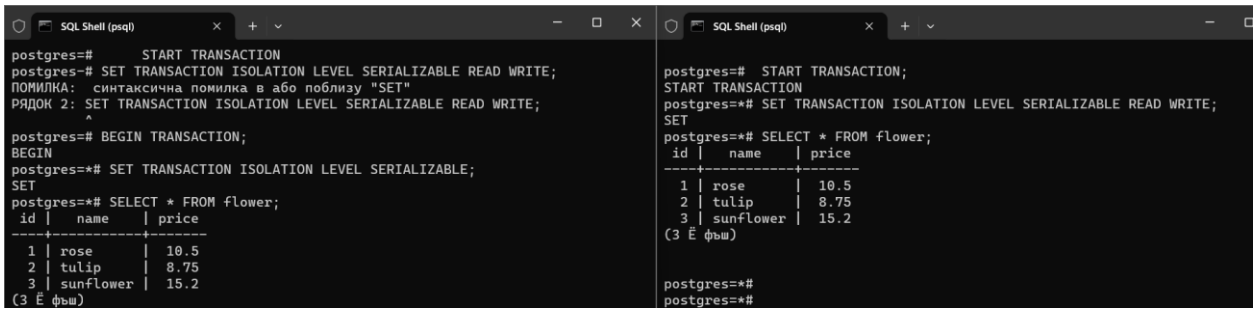
Стандарт SQL-92 визначає такі рівні ізоляції:

- Рівень ізоляції **Serializable** надає найвищий рівень узгодженості даних між транзакціями. На цьому рівні транзакції повністю ізолюються одна

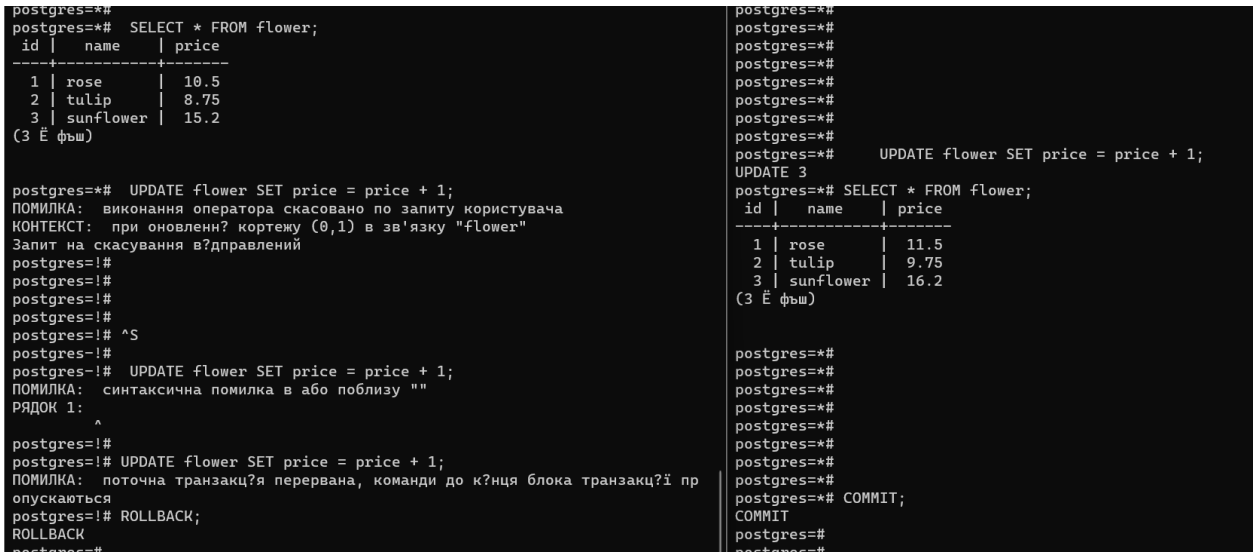
від одної. В більшості випадків результати паралельного виконання транзакцій на цьому рівні можна розглядати як результати послідовного виконання тих самих транзакцій, але в будь-якому порядку. Отже, дані в транзакціях ізольовані одна від одної, що дозволяє виконання транзакцій без втручання або впливу однієї транзакції на іншу, забезпечуючи високий рівень надійності та узгодженості даних.

Для тестування створено таблицю та внесено дані:

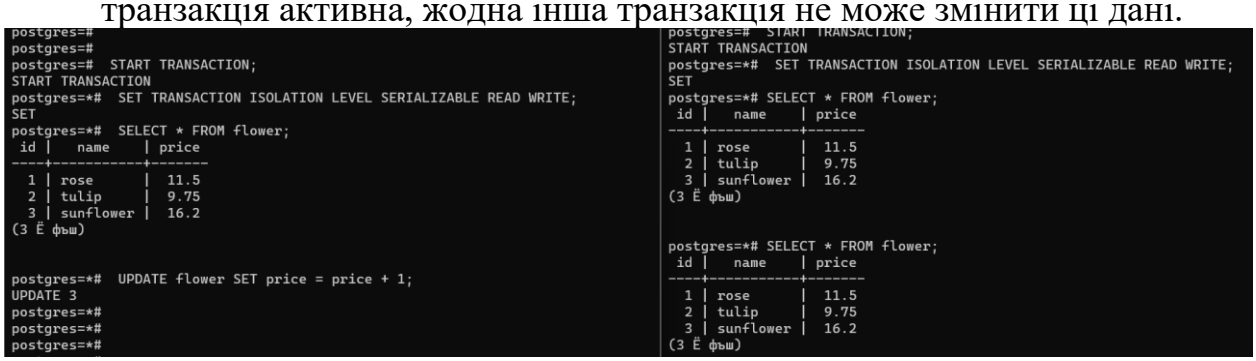
```
CREATE TABLE "flower" (
    "id" bigserial PRIMARY KEY,
    "name" text,
    "price" numeric
);

INSERT INTO "flower"("name", "price")
VALUES ('rose', 10.5), ('tulip', 8.75), ('sunflower', 15.2);
```



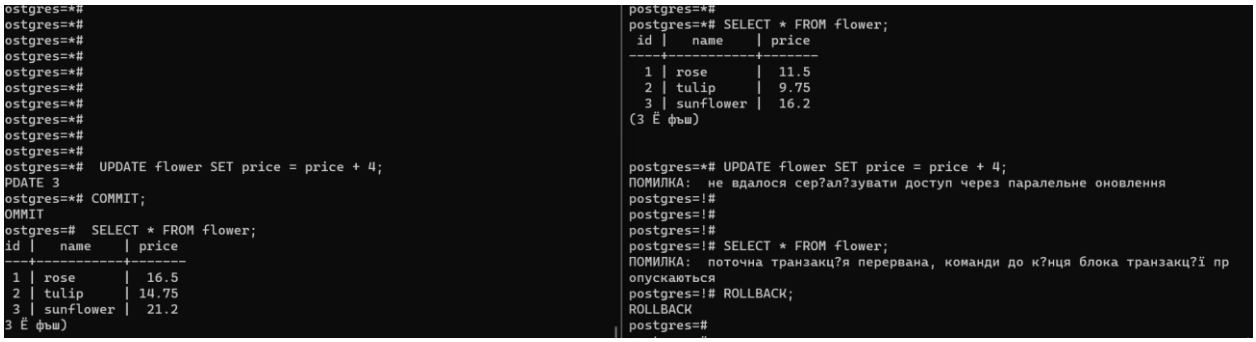
Зправа блокується під час оновлення, поки транзакція *зліва* не зафіксує або не скасує свої зміни.



- На рівні **Repeatable Read** читання одного або кількох рядків в рамках транзакції завжди повертає той самий результат. Це означає, що поки транзакція активна, жодна інша транзакція не може змінити ці дані.

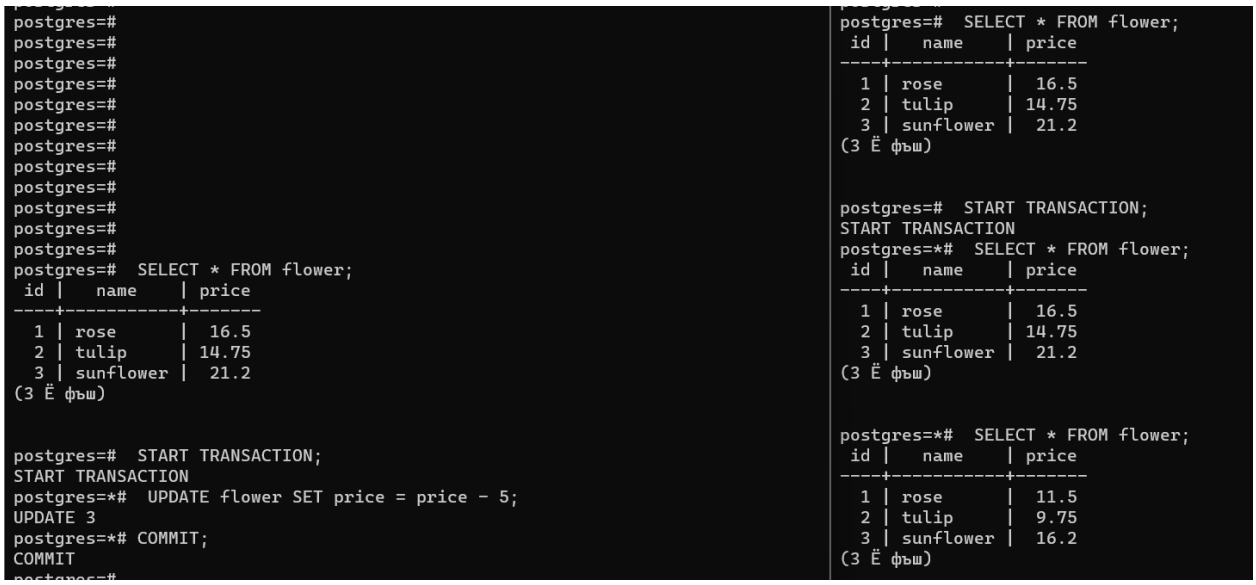


Транзакція *справа* буде призупинена або заблокована, поки транзакція *зліва* не закінчить свої зміни чи не скасує їх.



Repeatable Read унеможливлює зміну даних, якщо ці дані вже були модифіковані у незавершеній транзакції. Тому цей рівень ізоляції рекомендується використовувати переважно для читання даних, а не для модифікацій.

- На рівні **Read Committed** відбувається читання даних, які вже були зафіксовані в базі даних після коміту. "Брудне" читання (чи є дані, які ще не були зафіксовані) відсутнє, але під час роботи однієї транзакції інша транзакція може завершитись успішно, зберігши свої зміни. Це може призвести до ситуації, коли перша транзакція працює з набором даних, який був змінений іншою транзакцією, що створює проблему неповторюваного читання.



- Рівень **Read Uncommitted** є найнижчим рівнем ізоляції в базі даних, відповідає рівню 0. Цей рівень гарантує лише відсутність втрачених оновлень. Якщо кілька транзакцій одночасно намагаються змінити один і той же рядок, то в кінцевому результаті рядок матиме значення, встановлене останньою успішно виконаною транзакцією. У PostgreSQL, READ UNCOMMITTED розглядається як READ COMMITTED, що вказує на те, що читання даних нефіксованих в БД допускається тільки після їх зафіксування.