

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 та засоби управління»

**ТЕМА: «ЗАСОБИ ОПТИМІЗАЦІЇ РОБОТИ СУБД
POSTGRESQL»**

Виконав: студент III курсу

ФПМ групи КВ-04

Оніщук А.О.

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.

Вимоги до пункту завдання №1

Для перетворення функцій, що реалізують запити до об’єктної бази даних, необхідно встановити бібліотеку sqlalchemy, налаштувати програму на роботу з ORM, розробити класи-сутності для об’єктів-сутностей, представлених відповідними таблицями БД та пов’язаних зв’язками 1:M, M:M та 1:1 виконати опис схеми бази даних. Особливу увагу приділити контролю зовнішніх зв’язків між таблицями засобами ORM.

Замінити виклики запитів мовою SQL на відповідні запити засобами SQLAlchemy по роботі з об’єктами. Обов’язковим є реалізація вставки, вилучення та редагування екземплярів класів-сутностей. Розробка запитів на генерацію даних та пошук екземплярів класів-сутностей вітається, але не є обов’язковою.

Інтерфейси функцій (вхідні та вихідні аргументи функцій модуля “Модель”) мають залишитись без змін.

Вимоги до пункту завдання №2

Відповідно до варіанту індексування продемонструвати на прикладах запитів SQL SELECT підвищення швидкодії їх виконання з використанням індексів, а також пояснити чому для деяких випадків індексування використовувати недоцільно. При цьому для наочного представлення слід використати функцію генерування рандомізованих даних з лабораторної роботи №2, створивши необхідну кількість тестових даних. Навести 4-5 прикладів запитів SELECT (із виведенням результуючих даних), що містять фільтрацію, агрегатні функції, групування та сортування (у необхідних комбінаціях).

Вимоги до пункту завдання №3

Створити тригер бази даних PostgreSQL відповідно до варіанта. Тригерна функція має включати обробку запису, що модифікується (вставляється або вилучається), умовні оператори, курсорні цикли та обробку виключних ситуацій. Виконати відлагодження тригера при різних вхідних даних, навівши 2-3 приклади його використання.

Вимоги до пункту завдання №4

Проаналізувати на прикладах використання рівнів ізоляції транзакцій READ COMMITTED, REPEATABLE READ та SERIALIZABLE, продемонструвавши феномени, які виникають, і способ їх уникнення завдяки встановленню відповідного рівня ізоляції транзакцій. Для виконання завдання необхідно відкрити дві транзакції у різних вікнах pgAdmin4 і виконати послідовність запитів INSERT, UPDATE або DELETE у обох транзакціях, що доводять наявність або відсутність певних феноменів.

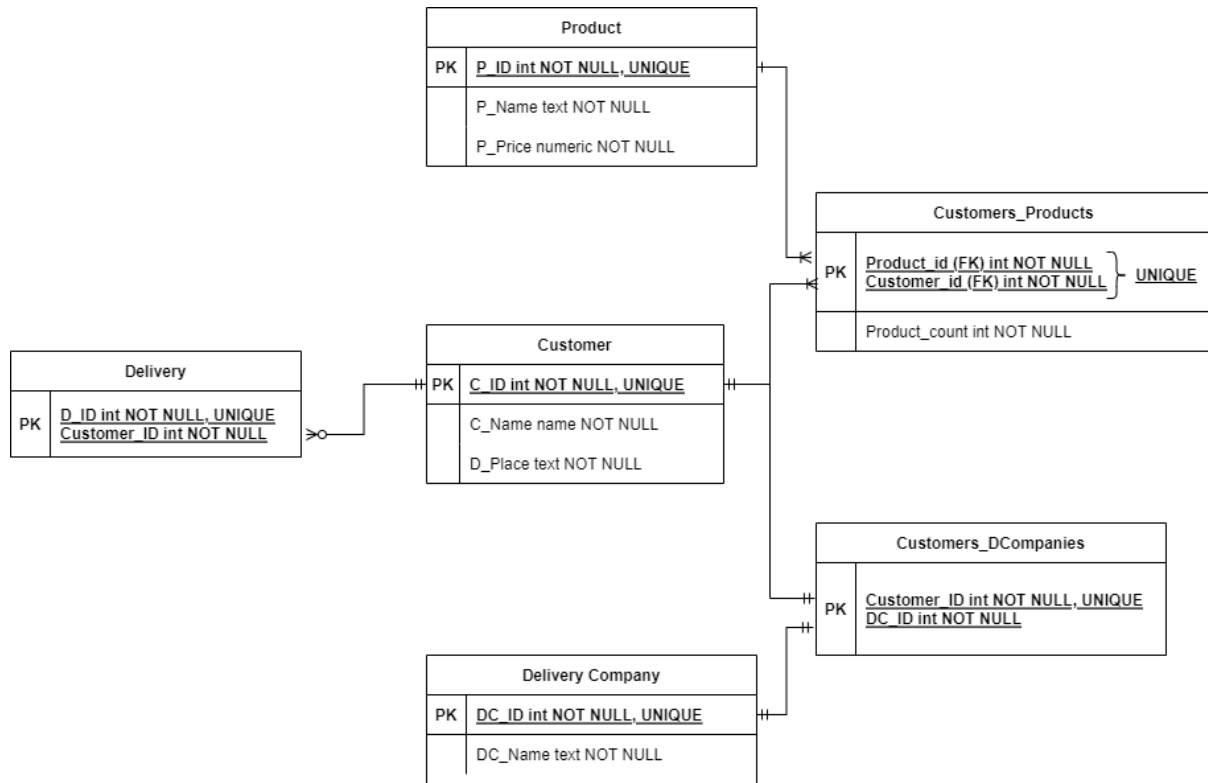
Варіант 17

17	GIN, BRIN	<u>before update, delete</u>
----	-----------	------------------------------

Бібліотека для реалізації ORM - SQLAlchemy

Git-репозиторій: <https://github.com/ViDjet-Git/Data-Base-University>

Схема бази даних



Завдання №1

Реалізація класів ORM

Delivery:

```
class delivery(Base):
    __tablename__ = 'Delivery'
    D_ID = Column('D_ID', Integer, primary_key=True)
    Customer_ID = Column('Customer_ID', Text,
        ForeignKey('Customer.C_ID'), primary_key=True)

    def __repr__(self):
        return "<Delivery(Customer_ID='{}')>" \
            .format(self.Customer_ID)
```

Product:

```
class product(Base):
    __tablename__ = 'Product'
    P_ID = Column('P_ID', Integer, primary_key=True)
    P_Name = Column('P_Name', Text)
    P_Price = Column('P_Price', Numeric)

    def __repr__(self):
        return "<Product(P_Name='{}', P_Price='{}')>" \
            .format(self.P_Name, self.P_Price)
```

Customer:

```
class customer(Base):
    __tablename__ = 'Customer'
    C_ID = Column('C_ID', Integer, primary_key=True)
    C_Name = Column('C_Name', Text)
    D_Place = Column('D_Place', Text)

    def __repr__(self):
        return "<Customer(C_Name='{}', D_Place='{}')>" \
            .format(self.C_Name, self.D_Place)
```

Delivery_Company:

```
class delivery_company(Base):
    __tablename__ = 'Delivery Company'
    DC_ID = Column('DC_ID', Integer, primary_key=True)
    DC_Name = Column('DC_Name', Text)

    def __repr__(self):
        return "<Delivery Company(DC_Name='{}')>" \
            .format(self.DC_Name)
```

Customers_Products:

```
class customers_products(Base):
    __tablename__ = 'Customers_Products'
    Product_id = Column('Product_id', Integer,
        ForeignKey('Product.P_ID'), primary_key=True)
    Customer_id = Column('Customer_id', Integer,
        ForeignKey('Customer.C_ID'), primary_key=True)
    Product_count = Column('Product_count', Integer)

    def __repr__(self):
        return "<Customers_Products(Product_id='{}',
        Customer_id='{}', Product_count='{}')>" \
            .format(self.Product_id, self.Customer_id,
            self.Product_count)
```

Delivery:

```
class customers_DCompanies(Base):
    __tablename__ = 'Customers_DCompanies'
    Customer_ID = Column('Customer_ID', Integer,
        ForeignKey('Customer.C_ID'), primary_key=True)
    DC_ID = Column('DC_ID', Integer, ForeignKey('Delivery
        Company.DC_ID'), primary_key=True)

    def __repr__(self):
        return "<Customers_DCompanies(Customer_ID='{}',
        DC_ID='{}')>" \
            .format(self.Customer_ID, self.DC_ID)
```

Реалізація функцій

Insert:

```
def add_row(table_name, files):
    try:
        if(table_name == 'Customer'):
            temp = customer(
                C_Name=list(files.values())[0],
                D_Place=list(files.values())[1]
            )
        elif(table_name == 'Product'):
            temp = product(
                P_Name=list(files.values())[0],
                P_Price=list(files.values())[1]
            )
        elif(table_name == 'Delivery Company'):
            temp = delivery_company(
                DC_Name=list(files.values())[0]
            )
        elif(table_name == 'Delivery'):
            temp = delivery(
                Customer_ID=list(files.values())[0]
            )
        elif(table_name == 'Customers_Products'):
            temp = customers_products(
                Product_id=list(files.values())[0],
                Customer_id=list(files.values())[1],
                Product_count=list(files.values())[2]
            )
        elif(table_name == 'Customers_DCompanies'):
            temp = customers_DCompanies(
                Customer_ID=list(files.values())[0],
                DC_ID=list(files.values())[1]
            )
        s.add(temp)
        s.commit()
    except Exception as error:
        print(error)
        s.rollback()
        return False
```

Delete:

```
def del_row(table_name, key, value):
    try:
        if type(value) is str:
            filter_txt = "{}" = \'{ }\'.format(key, value)
        else:
            filter_txt = "{}" = { }\'.format(key, value)

        if (table_name == 'Customer'):

s.query(customer).filter(text(filter_txt)).delete(synchronize_session=False)
        elif (table_name == 'Product'):
```

```

s.query(product).filter(text(filter_txt)).delete(synchronize_session=False)
    elif (table_name == 'Delivery Company'):

s.query(delivery_company).filter(text(filter_txt)).delete(synchronize_session=False)
    elif (table_name == 'Delivery'):

s.query(delivery).filter(text(filter_txt)).delete(synchronize_session=False)
    elif (table_name == 'Customers_Products'):

s.query(customers_products).filter(text(filter_txt)).delete(synchronize_session=False)
    elif (table_name == 'Customers_DCompanies'):

s.query(customers_DCompanies).filter(text(filter_txt)).delete(synchronize_session=False)

    s.commit()
except Exception as error:
    print(error)
    s.rollback()
    return False

```

Update:

```

def edit_value(table_name, key, key_change, new_val, key_val):
    try:
        if (table_name == 'Customer'):
            table_class = customer
        elif (table_name == 'Product'):
            table_class = product
        elif (table_name == 'Delivery Company'):
            table_class = delivery_company
        elif (table_name == 'Delivery'):
            table_class = delivery
        elif (table_name == 'Customers_Products'):
            table_class = customers_products
        elif (table_name == 'Customers_DCompanies'):
            table_class = customers_DCompanies

        if key_val is str:
            filter_txt = "\"{}\" = '{}'.format(key, key_val)
        else:
            filter_txt = "\"{}\" = {}".format(key, key_val)









        update_values = {key_change: new_val}
        s.query(table_class) \
            .filter(text(filter_txt)) \
            .update(update_values, synchronize_session=False)
        s.commit()
    except Exception as error:
        print(error)
        s.rollback()
        return False

```

Приклади виконання запитів









Вставка

```
add_row("Customers_DCompanies", {'Customer_ID': 1, 'DC_ID': 1})
```

Data output			Messages	Notifications
       				
	Customer_ID [PK] integer	DC_ID [PK] integer		
1	1	1		









Видалення

```
del_row('Customers_DCompanies', 'DC_ID', '1')
```

Data output			Messages	Notifications
       				
	Customer_ID [PK] integer	DC_ID [PK] integer		

Редагування

```
edit_value("Delivery Company", "DC_ID", "DC_Name", "UkrPoshta", 2)
```

Data output			Messages	Notifications
       				
	DC_ID [PK] integer	DC_Name text		
1	2	UkrPoshta		

Завдання №2

В даному завданні розглянемо використання індексів GIN, BRIN.
Для тестування була створена таблиця з типом даних tsvector, щоб можна було працювати з індексом GIN:

```
CREATE TABLE public.test
(
    ID serial PRIMARY KEY,
    vect tsvector,
    txt text COLLATE pg_catalog."default",
);
```

Далі було згенеровано випадкові дані в таблицю:

```
INSERT INTO test(txt, vect)
(
    SELECT
        chr(trunc(60 + random() * 25)::int) || chr(trunc(60 + random() *
50)::int) || chr(trunc(60 + random() * 50)::int),
        to_tsvector(chr(trunc(60 + random() * 25)::int) || chr(trunc(60 +
random() * 50)::int) || chr(trunc(60 + random() * 50)::int))
    FROM generate_series(1,50)
);
```

Посортуємо дані без використання індексу:
explain analyze select * from test order by txt

1	explain analyze select * from test order by txt
Data output Messages Notifications	
	QUERY PLAN text
1	Sort (cost=2.91..3.04 rows=50 width=24) (actual time=0.093..0.096 rows=50 loops=1)
2	Sort Key: txt
3	Sort Method: quicksort Memory: 28kB
4	-> Seq Scan on test (cost=0.00..1.50 rows=50 width=24) (actual time=0.015..0.017 rows=50 loops=1)
5	Planning Time: 6.598 ms
6	Execution Time: 0.110 ms

Отже час сортування 110 мс, далі виконаємо сортування використовуючи індекс BRIN:

create index on test using BRIN (txt);

explain analyze select * from test order by vect;

1	CREATE INDEX ON test using BRIN (txt);
2	explain analyze select * from test order by txt

Data output	Messages	Notifications
QUERY PLAN		
text		
1	Sort (cost=2.91..3.04 rows=50 width=24) (actual time=0.080..0.081 rows=50 loops=1)	
2	Sort Key: txt	
3	Sort Method: quicksort Memory: 28kB	
4	-> Seq Scan on test (cost=0.00..1.50 rows=50 width=24) (actual time=0.013..0.015 rows=50 loops=...)	
5	Planning Time: 1.605 ms	
6	Execution Time: 0.092 ms	

І сортування з індексом GIN:

create index on test using GIN (vect);

explain analyze select * from test order by vect;

1	CREATE INDEX ON test using GIN (vect);
2	explain analyze select * from test order by vect

Data output	Messages	Notifications
QUERY PLAN		
text		
1	Sort (cost=2.91..3.04 rows=50 width=24) (actual time=0.035..0.037 rows=50 loops=1)	
2	Sort Key: vect	
3	Sort Method: quicksort Memory: 28kB	
4	-> Seq Scan on test (cost=0.00..1.50 rows=50 width=24) (actual time=0.012..0.014 rows=50 loops=...)	
5	Planning Time: 2.192 ms	
6	Execution Time: 0.048 ms	

Як видно з тестів, індекс GIN зменшив час сортування майже в 2,5 рази, що є дуже хорошим результатом. Даний індекс створений для повнотекстового пошуку, тому дуже добре оптимізований. Індекс BRIN теж зменшив час сортування, але на 18 мс. Оскільки індекс BRIN призначений для обробки дуже великих таблиць, то для такої таблиці як в нас, він теж показав гарний результат.

Завдання №3


Для тестування тригера було створено дві таблиці:

```
DROP TABLE IF EXISTS "trigger";
CREATE TABLE "trigger"(
    "ID" bigserial PRIMARY KEY,
    "Name" text
);
```

```
DROP TABLE IF EXISTS "triggerLog";
CREATE TABLE "triggerLog"(
    "LogID" bigserial PRIMARY KEY,
    "TriggerID" bigint,
    "TriggerName" text
);
```

І в таблицю "trigger" занесено значення:

```
insert into public."trigger" ("Name")
VALUES ('Name_1'), ('Name_2'), ('Name_3'),
('Name_4'), ('Name_5')
```

Data output			Messages	Notifications
				
	ID [PK] bigint	Name text		
1	1	Name_1		
2	2	Name_2		
3	3	Name_3		
4	4	Name_4		
5	5	Name_5		

Код тригера:

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

BEGIN

IF new."Name" = 'ERROR' THEN
    RAISE EXCEPTION 'NOT CORRECT NAME';
ELSE
    IF old."ID" % 2 = 0 THEN
        INSERT INTO "triggerLog"("TriggerID", "TriggerName")
        VALUES (old."ID", old."Name");
        RETURN NEW;
    ELSE
        INSERT INTO "triggerLog"("TriggerID", "TriggerName")
        VALUES (old."ID", old."Name" || '_odd');
        RETURN NEW;
    END IF;
END IF;
END;

$$ language plpgsql;
```

Ініціалізація тригера:

```
drop trigger if exists test_trigger ON public."trigger";
create trigger test_trigger before update or delete
on public."trigger" for each row
execute procedure update_delete_func();
```

Перевірка тригера

- 1) Якщо при зміні значення в таблиці "trigger", нове значення містить заборонене слово, то програма видає помилку "NOT CORRECT NAME"

The screenshot shows a database query editor with a 'Query' tab selected. The query is:

```
1 UPDATE public."trigger" SET "Name" = 'ERROR'
2 WHERE "ID" = 3
```

Below the query, there are tabs for 'Data output', 'Messages', and 'Notifications'. The 'Messages' tab is active, displaying the following error message:

```
ERROR: ОШИБКА: NOT CORRECT NAME
CONTEXT:  функция PL/pgSQL update_delete_func(), строка 6, оператор RAISE
```

At the bottom, the 'SQL state' is shown as 'P0001'.

- 2) При зміні або видаленні значення в "trigger", попереднє значення записується в таблицю "triggerLog". Якщо значення знаходиться під непарним "ID", то до значення в таблиці "triggerLog" додається '_odd'

Query Query History

```
1 -- UPDATE public."trigger" SET "Name" = 'Andrew'
2 -- WHERE "ID" = 3
3
4 select * from "triggerLog"
```

Data output Messages Notifications

	LogID [PK] bigint	TriggerID bigint	TriggerName text
1	1	3	Name_3_odd

Query Query History

```
1 -- UPDATE public."trigger" SET "Name" = 'Andrew'
2 -- WHERE "ID" = 3
3
4 select * from "trigger"
```

Data output Messages Notifications

	ID [PK] bigint	Name text
1	1	Name_1
2	2	Name_2
3	4	Name_4
4	5	Name_5
5	3	Andrew

На цих знімках видно, що при зміні значення під непарним "ID", значення в таблиці "triggerLog" записано з припискою '_odd'

3)

Query

Query History

1

-- DELETE FROM public."trigger" WHERE "ID" = 2;

2

SELECT * FROM public."triggerLog"

3

Data output

Messages

Notifications

≡+

	LogID [PK] bigint	TriggerID bigint	TriggerName text
1	1	3	Name_3_odd
2	2	2	Name_2

А на цьому знімку можна побачити, що при видаленні рядка під парним "ID", значення в таблиці "triggerLog" записано в початковому вигляді.