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»

На тему «Засоби оптимізації роботи СУБД PostgreSQL»

Виконав: студент III курсу
ФПМ групи КВ-82
Гришко Валерій Валерійович

Перевірив: Павловський В. І.

Київ – 2020

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.

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.

Корисні посилання: тут і тут.

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).

Корисні посилання: Hash, B-tree, GIN, BRIN.

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.

Корисні посилання: тут, тут.

Вимоги до інструментарію:

1. Бібліотека для реалізації ORM - SQLAlchemy для Python або інша з подібною функціональністю.
2. Середовище для відлагодження SQL-запитів до бази даних – pgAdmin 4.
3. СУБД - PostgreSQL 11-12.

Варіант:

9	*BTree, BRIN*	*before delete, update*

Меню для навігації

Завдання 1

На рисунку 1 наведено логічну модель бази даних «Порушення ПДР»

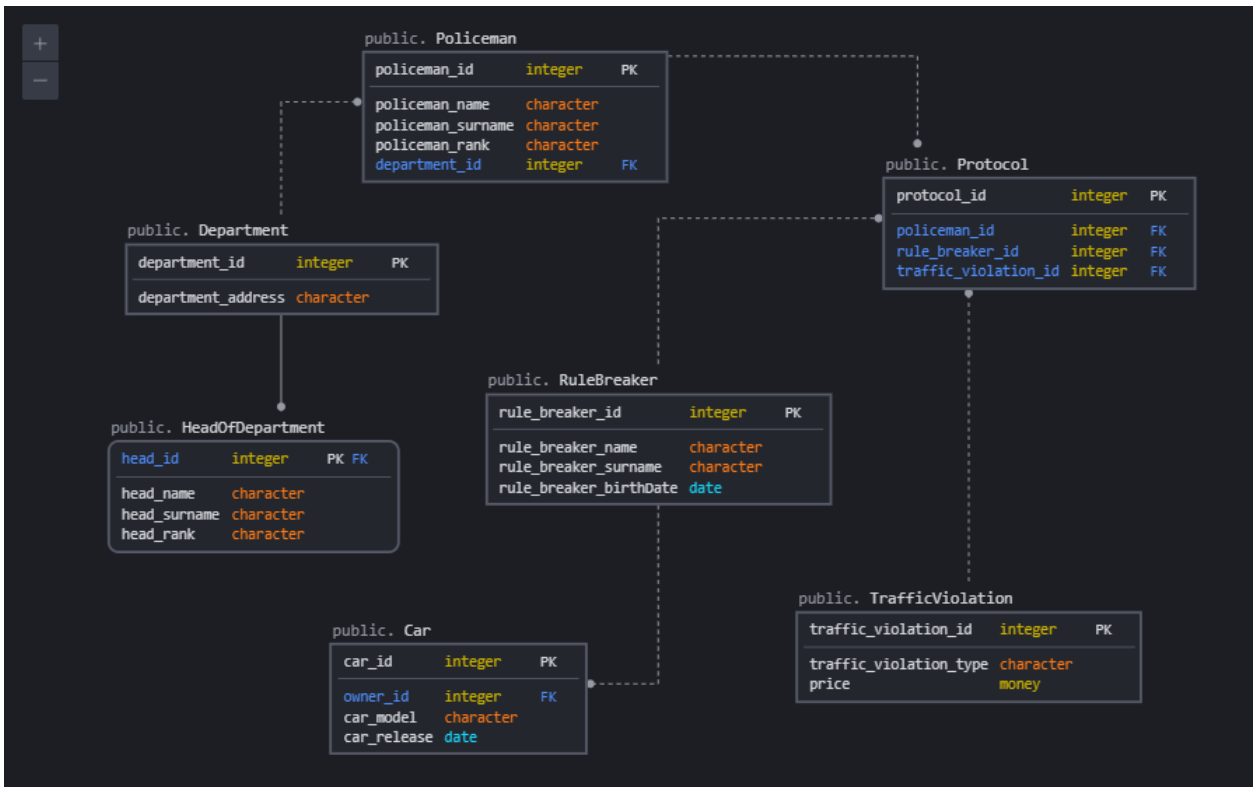


Рисунок 1 – Логічна модель бази даних «Порушення ПДР»

Для вирішення задачі перетворення моделі з MVC у вигляд об’єктно-реляційної проекції (ORM) використовується бібліотека SQLAlchemy для мови програмування Python, яка є однією з найпопулярніших для вирішення задач об’єктно-реляційного відображення.

Представлення таблиць у класах полягає у представленні полів класів як колон таблиці. Для цього потрібно врахувати відповідність типів даних між сервером PostgreSQL і відповідною мовою програмування. Наприклад, типам даних *date* та *time without time zone* PostgreSQL відповідає один тип даних – *DateTime*.

Для реалізації зовнішнього ключа у класі, який відповідає таблиці із зовнішнім ключем з відношенням до колонки таблиці A, потрібно додати поле типу класу, який відповідає таблиці A. Також потрібно написати відповідну команду для визначення цього поля як зовнішній ключ.

Програма створена за допомогою мови програмування Python в середовищі розробки PyCharm Community Edition 2020.2.3

Підключення SQLAlchemy до БД:

```python
from sqlalchemy import create_engine, Column, Integer, String, DateTime, ForeignKey
from sqlalchemy.engine.url import URL
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from View import View

DATABASE = {
    'drivername': 'postgres',
    'host': 'localhost',
    'port': '5432',
    'username': 'postgres',
    'password': 'dfkthfuhbirj',
    'database': 'Penalty for violation of traffic rules'
}

db = declarative_base()
```

```python
engine = create_engine(URL(**DATABASE))
db.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
session = Session()
```

Для прикладу реалізації перетворення моделі з MVC у вигляд об’єктно-реляційної проекції (ORM) розглянемо код для таблиці «RuleBreaker».

Клас для співвідношення даних класу з таблицею в БД:

```python
class RuleBreaker(db):
    __tablename__ = 'RuleBreaker'

    rule_breaker_id = Column(Integer, primary_key = True, nullable = False)
    rule_breaker_name = Column(String, nullable = False)
    rule_breaker_surname = Column(String, nullable = False)
    rule_breaker_birthDate = Column(DateTime, nullable = False)

    def __init__(self, id, name, surname, birthDate):
        self.rule_breaker_id = id
        self.rule_breaker_name = name
        self.rule_breaker_surname = surname
        self.rule_breaker_birthDate = birthDate
```

Метод додавання даних до таблиці «RuleBreaker»:

```python
@staticmethod
def Insert():
    new_rule_breaker_id = input('rule_breaker_id = ')
    new_rule_breaker_name = input('rule_breaker_name = ')
    new_rule_breaker_surname = input('rule_breaker_surname = ')
    new_rule_breaker_birthDate = input('rule_breaker_birthDate = ')
    new_rule_breaker = RuleBreaker(new_rule_breaker_id, new_rule_breaker_name, new_rule_breaker_surname, new_rule_breaker_birthDate)
    session.add(new_rule_breaker)
    session.commit()
```

Метод видалення даних за полем rule_breaker_id з таблиці «RuleBreaker»:

```
@staticmethod
def Delete():
    delete_id = input('rule_breaker_id delete = ')
    session.delete(session.query(RuleBreaker).filter(RuleBreaker.rule_breaker_id == delete_id).first())
    session.commit()
```

Метод редагування даних за будь-яким атрибутом таблиці «RuleBreaker»:

```
@staticmethod
def Update():
    num = View.attribute_list(1)
    value = input('Attribute value to update = ')
    value2 = input('New attribute value = ')
    if num == 1:
        session.query(RuleBreaker).filter(RuleBreaker.rule_breaker_id == value).update({RuleBreaker.rule_breaker_id: value2})
    elif num == 2:
        session.query(RuleBreaker).filter(RuleBreaker.rule_breaker_name == value).update({RuleBreaker.rule_breaker_name: value2})
    elif num == 3:
        session.query(RuleBreaker).filter(RuleBreaker.rule_breaker_surname == value).update({RuleBreaker.rule_breaker_surname: value2})
    elif num == 4:
        session.query(RuleBreaker).filter(RuleBreaker.rule_breaker_birthDate == value).update({RuleBreaker.rule_breaker_birthDate: value2})
    session.commit()
```

На прикладі таблиці «Car» розглянемо, як визначається зовнішній ключ (поле owner_id):

```
class Car(db):
    __tablename__ = 'Car'

    car_id = Column(Integer, primary_key = True, nullable = False)
    owner_id = Column(Integer, ForeignKey(RuleBreaker.rule_breaker_id), nullable = False)
    car_model = Column(String, nullable = False)
    car_release = Column(DateTime, nullable = False)

    def __init__(self, id, owner, model, release):
        self.car_id = id
        self.owner_id = owner
        self.car_model = model
        self.car_release = release
```

На рисунку 2 зображена структура класів програми, що відповідають таблицям БД

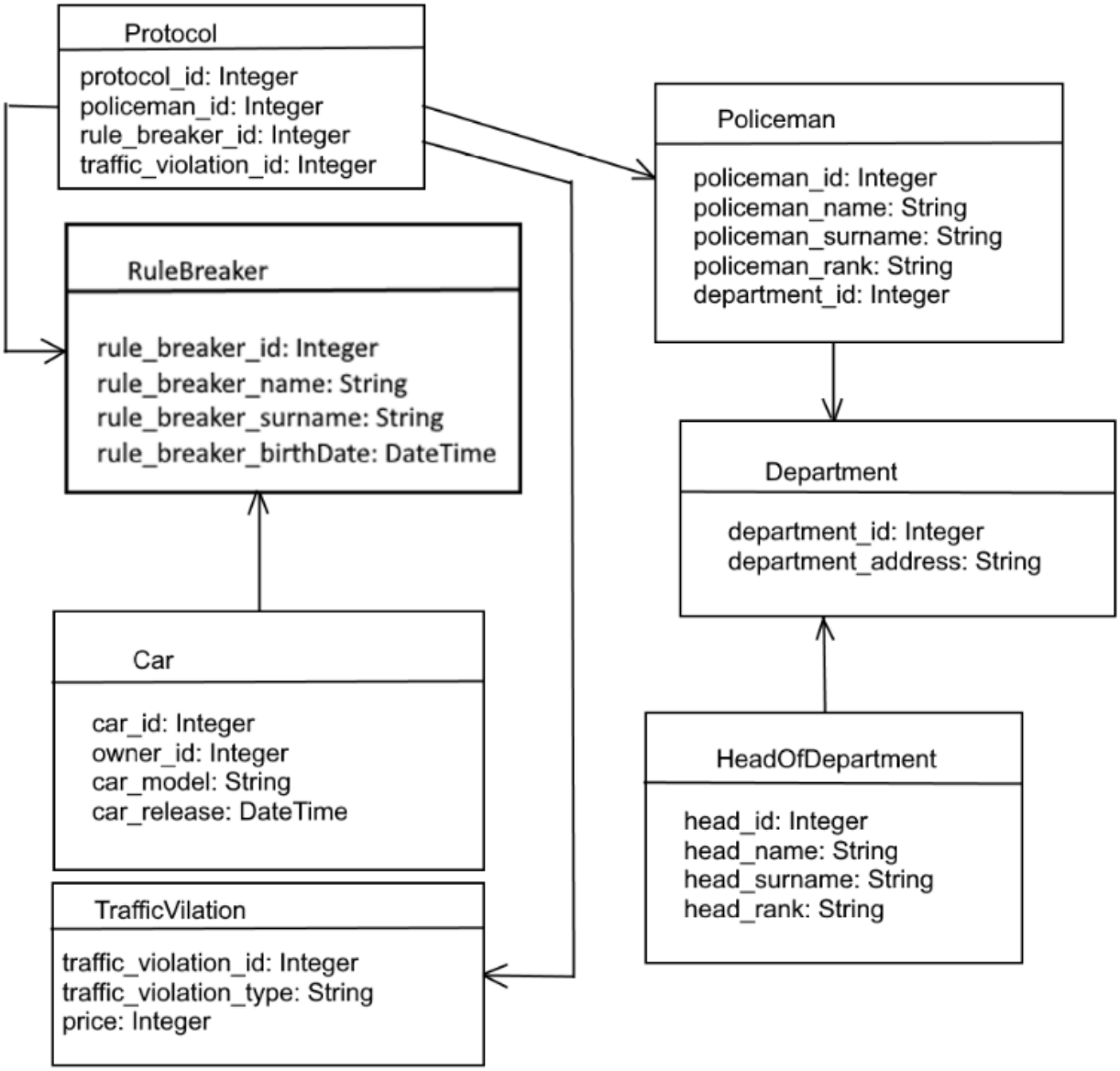


Рисунок 2 – Структура класів програми, що відповідають таблицям БД

Інші класи та функції реалізовано аналогічно.

Завдання 2

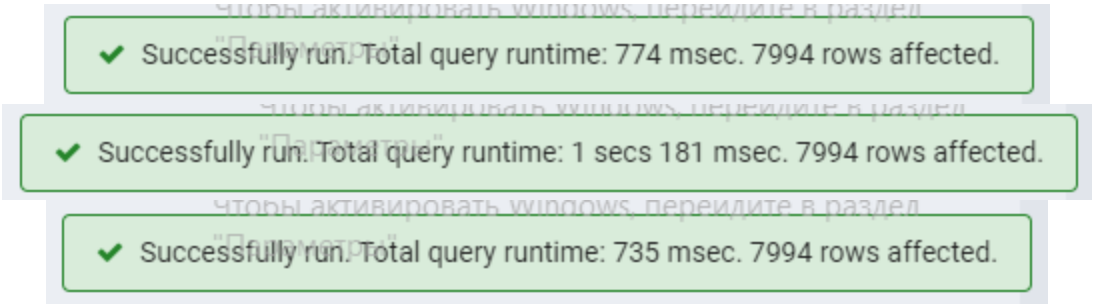
Створення та аналіз індексу BTree

Оскільки індекс BTree використовується для даних, які можна відсортувати, аналіз проводиться на числових даних. Для дослідження використаємо таблицю example з єдиною колонкою num типу integer. До таблиці було додано 10 000 рандомізованих даних:

```
1  insert into public.example("num ")
2      select trunc(random()*10000)::int from generate_series(1, 10000);
```

Виконаємо пошук даних для num > 2000.
Спочатку, без використання індексу:

```
1  select * from example where example."num " > 2000
```



Середній час виконання запиту: 896 msec.

Тепер створимо індекс BTree на колонці num та виконаємо запит:

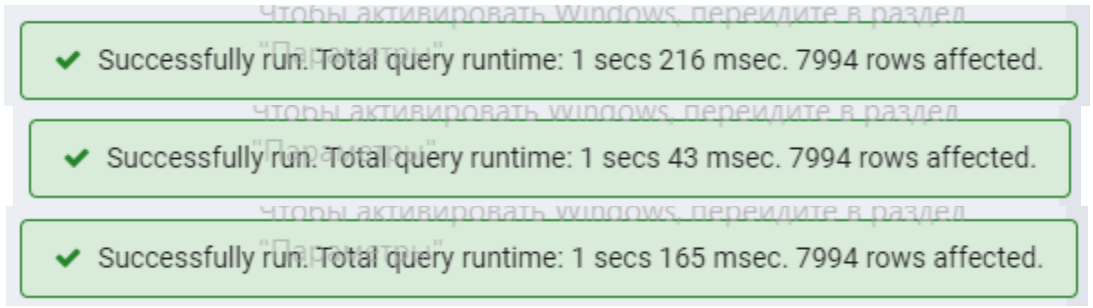
```
1  create index BTree_test on example using BTree("num ")
2
3
```

Data Output | Explain | Messages | Notifications

```
CREATE INDEX

Query returned successfully in 1 secs 214 msec.
```

```
1  select * from example where example."num " > 2000
```



Середній час виконання запиту: 1141 msec.

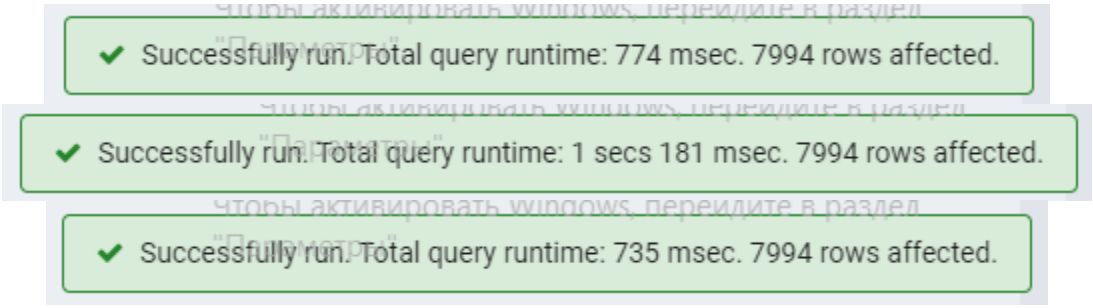
Як бачимо, середній час виконання запиту з використанням індексу трохи збільшився в даному випадку. Це свідчить про те, що даний індекс не ефективний при пошуку значної кількості даних.

Створення та аналіз індексу BRIN

Для дослідження індексу BRIN використаємо таблицю example з єдиною колонкою num типу integer, використану для дослідження індексу BTree та дані, отримані при дослідженні індексу BTree.

Виконаємо той самий запит, що і при дослідженні індексу BTree. Спочатку, без використання індексів:

```
1  select * from example where example."num " > 2000
```



Середній час виконання запиту: 896 msec.

Тепер створимо індекс BRIN на колонці num:

```
1  create index BRIN_test on example using BRIN("num ")
2
```

Data Output | Explain | Messages | Notifications

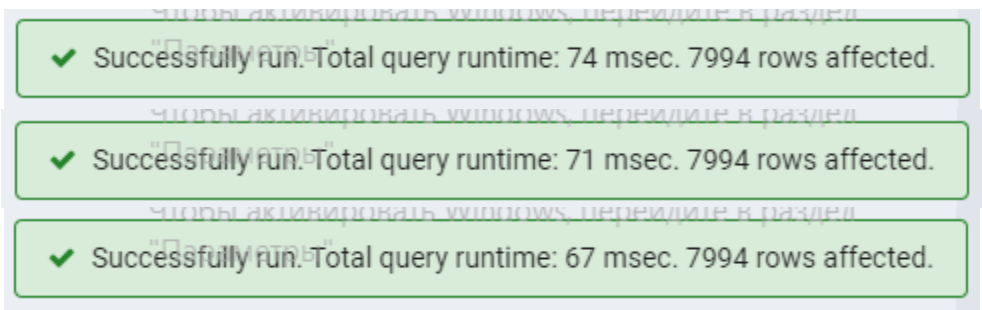
```
CREATE INDEX

Query returned successfully in 146 msec.
```

Виконаємо повторно запит:

```
1  select * from example where example."num " > 2000
```

Отримаємо результати:



Середній час виконання запиту: 71 msec.

Як бачимо, середній час виконання запиту з використанням індексу BRIN суттєво зменшився (більш ніж в 10 разів).

Отже, використання цього індексу значно пришвидшує виконання запиту та є доречним, якщо нам потрібно виконати пошук значної кількості даних за числовими атрибутами.

Завдання 3

Логіка тригера:

Тригер працює перед видаленням або оновленням таблиці «HeadOfDepartment».

Якщо була спроба видалити дані, то до нової таблиці «newTable», що має лише один атрибут operation, заноситься назва операції, тобто «DELETE». При цьому дані з таблиці «HeadOfDepartment» видалені не будуть.

Якщо була спроба оновити дані, то тут маємо два варіанти: якщо поле head_rank = ‘colonel’, то до таблиці «newTable» буде додана назва операції з «1» в кінці, тобто «UPDATE1». Дані в таблиці «HeadOfDepartment» при цьому оновлено не буде; якщо поле head_rank != ‘colonel’, то до всіх даних поля operation таблиці «newTable» буде додано «2» в кінець. Дані в таблиці «HeadOfDepartment» при цьому оновлено не буде.

Створення тригеру:

```
create or replace function Trigger_test() returns trigger as $$
declare
 crs cursor for select * from "newTable";
 row "newTable"%rowtype;
begin
 if (TG_OP = 'DELETE') then
  insert into "newTable"(operation) values ('DELETE');
  return NULL;
 elseif (TG_OP = 'UPDATE') then
  if (old.head_rank = 'colonel') then
   insert into "newTable"(operation) values ('UPDATE1');
   return NULL;
  elseif (old.head_rank != 'colonel') then
   for row in crs loop
    update "newTable" set operation=operation || '2' where current of crs;
   end loop;
   return NULL;
  end if;
 end if;
end;
$$ language plpgsql;
```

Data Output | Explain | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 1 secs 386 msec.
```

Підключення тригеру до таблиці «HeadOfDepartment»:

```
create trigger trigger_test
before delete or update
on "HeadOfDepartment"
for each row execute procedure Trigger_test()
```

Data Output | Explain | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 73 msec.
```

Вміст таблиці «HeadOfDepartment»:

```
1  select * from "HeadOfDepartment"
2
3
```

Data Output | Explain | Messages | Notifications

	head_id [PK] integer	head_name character varying (30)	head_surname character varying (30)	head_rank character varying (30)
1	1	vladyslav	petrenko	colonel
2	2	igor	maznichenko	lieutenant colonel

Спроба видалення даних з таблиці «HeadOfDepartment»:

```
1  delete from "HeadOfDepartment" where "HeadOfDepartment".head_id = 1
2
3
```

Data Output | Explain | Messages | Notifications

```
DELETE 0

Query returned successfully in 132 msec.
```

Вміст таблиці «newTable»:

```
1  select * from "newTable"
2
3
```

Data Output | Explain | Messages | Notifications

	operation character varying (10)
1	DELETE

Спроба оновити дані таблиці «HeadOfDepartment», де поле head_rank != ‘colonel’:

```
1  update "HeadOfDepartment" set head_rank = 'lieutenant' where head_rank = 'lieutenant colonel'
```

Data Output | Explain | Messages | Notifications

```
UPDATE 0

Query returned successfully in 1 secs 335 msec.
```

Вміст таблиці «newTable»:

```
1  select * from "newTable"
```

Data Output | Explain | Messages | Notifications

	operation character varying (10)
1	DELETE2

Спроба оновити дані таблиці «HeadOfDepartment», де поле head_rank = ‘colonel’ (в нашому випадку ми оновлюємо дані за head_id. Поле head_rank = ‘colonel’ при head_id = 1):

```
1  update "HeadOfDepartment" set head_rank = 'major' where head_id = 1
```

Data Output | Explain | Messages | Notifications

```
UPDATE 0

Query returned successfully in 1 secs 8 msec.
```

Вміст таблиці «newTable»:

```
1  select * from "newTable"
```

Data Output | Explain | Messages | Notifications

	operation character varying (10)
1	DELETE2
2	UPDATE1

Спробуємо ще раз оновити дані таблиці «HeadOfDepartment», де поле head_rank != ‘colonel’:

```
1  update "HeadOfDepartment" set head_rank = 'lieutenant' where head_rank = 'lieutenant colonel'
```

Data Output | Explain | Messages | Notifications

```
UPDATE 0

Query returned successfully in 840 msec.
```

Вміст таблиці «newTable»:

```
1  select * from "newTable"
```

Data Output | Explain | Messages | Notifications

	operation character varying (10)
1	DELETE22
2	UPDATE12

Вміст таблиці «newTable» є коректним після проведення дослідження. Отже, тригер працює відповідно до заданої логіки.