

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО”

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп’ютерних систем

**Лабораторна робота № 3**

з дисципліни «Бази даних і засоби управління»

«**Засоби оптимізації роботи СУБД PostgreSQL**»

Виконав: Піскун Андрій

Студент групи КВ-93

Перевірив: Павловський В.І.

**Київ 2021**

**Лабораторна робота № 3.**

**Засоби оптимізації роботи СУБД PostgreSQL**

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.  
   

**Логічна модель учбової предметної області «Кінотеатр»**

Зображення, що містить текст, у приміщенні

Автоматично згенерований описРисунок 1 – логічна модель учбової предметної області «Кінотеатр»

**Середовище розробки**

Середовище розробки бази даних - PostgreSQL

Середовище розробки програми – PyCharm. Мова програмування - Python 3.8.

Використані бібліотеки: sqlAlchemy, psycopg2 (для зв’язку з СУБД), datetime (для роботи з датою і передачею її у запити до БД), time (для виміру часу запиту пошуку для завдання 3), sys (для реалізації консольного інтерфейсу).

**Класи ORM у реалізованому модулі Model**

class Cinema(Orders):  
 \_\_tablename\_\_ = 'cinema'  
 cinema\_id = Column(Integer, primary\_key=True)  
 name = Column(String)  
 adress = Column(String)  
 halls = relationship("Hall")  
 movies = relationship("Movie")  
  
 def \_\_init\_\_(self, cinema\_id, name, adress):  
 self.cinema\_id = cinema\_id  
 self.name = name  
 self.adress = adress  
  
 def \_\_repr\_\_(self):  
 return "{:>10}{:>15}{:>35}" \  
 .format(self.cinema\_id, self.name, self.adress)

class Hall(Orders):  
 \_\_tablename\_\_ = 'hall'  
 number = Column(Integer, primary\_key=True)  
 cinema\_id = Column(Integer, ForeignKey('cinema.cinema\_id'))  
 screen\_size = Column(String)  
 number\_of\_seats = Column(Integer)  
 sessions = relationship("Sessions")  
 seats = relationship("Seat")  
  
 def \_\_init\_\_(self, number, cinema\_id, screen\_size, number\_of\_seats):  
 self.number = number  
 self.cinema\_id = cinema\_id  
 self.screen\_size = screen\_size  
 self.number\_of\_seats = number\_of\_seats  
  
 def \_\_repr\_\_(self):  
 return "{:>10}{:>10}{:>15}{:>10}" \  
 .format(self.number, self.cinema\_id, self.screen\_size, self.number\_of\_seats)

class Movie(Orders):  
 \_\_tablename\_\_ = 'movie'  
 movie\_id = Column(Integer, primary\_key=True)  
 cinema\_id = Column(Integer, ForeignKey('cinema.cinema\_id'))  
 title = Column(String)  
 rating = Column(Integer)  
 sessions = relationship("Sessions")  
  
 def \_\_init\_\_(self, movie\_id, cinema\_id, title, rating):  
 self.movie\_id = movie\_id  
 self.cinema\_id = cinema\_id  
 self.title = title  
 self.rating = rating  
  
 def \_\_repr\_\_(self):  
 return "{:>10}{:>10}{:>25}{:>15}" \  
 .format(self.movie\_id, self.cinema\_id, self.title, self.rating)

class Sessions(Orders):  
 \_\_tablename\_\_ = 'session'  
 session\_id = Column(Integer, primary\_key=True)  
 movie\_id = Column(Integer, ForeignKey('movie.movie\_id'))  
 number = Column(Integer, ForeignKey('hall.number'))  
 time = Column(Date)  
 cost = Column(Float)  
  
  
 def \_\_init\_\_(self, session\_id, movie\_id, number, time, cost):  
 self.session\_id = session\_id  
 self.movie\_id = movie\_id  
 self.number = number  
 self.time = time  
 self.cost = cost  
  
 def \_\_repr\_\_(self):  
 return "{:>10}{:>30}{:>10}\t\t{}{:>10}" \  
 .format(self.session\_id, self.movie\_id, self.number, self.time, self.cost)

class Seat(Orders):  
 \_\_tablename\_\_ = 'seat'  
 seat\_id = Column(Integer, primary\_key=True)  
 number = Column(Integer, ForeignKey('hall.number'))  
 row = Column(Integer)  
 seat\_number = Column(Integer)  
 is\_occupied = Column(Integer)  
  
 def \_\_init\_\_(self, seat\_id, number, row, seat\_number, is\_occupied):  
 self.seat\_id = seat\_id  
 self.number = number  
 self.row = row  
 self.seat\_number = seat\_number  
 self.is\_occupied = is\_occupied  
  
 def \_\_repr\_\_(self):  
 return "{:>10}{:>10}{:>10}{:>10}{:>10}" \  
 .format(self.seat\_id, self.number, self.row, self.seat\_number, self.is\_occupied)

Вилучення, вставка, редагування даних на прикладі таблиці movie

def delete\_data\_movie(self, movie\_id) -> None:  
 self.session.query(Movie).filter\_by(movie\_id=movie\_id).delete()  
 self.session.commit()

def insert\_data\_movie(self, movie\_id: int, cinema\_id: int, title: str, rating: float) -> None:  
 movie = Movie(movie\_id=movie\_id, cinema\_id=cinema\_id, title=title, rating=rating)  
 self.session.add(movie)  
 self.session.commit()

def update\_data\_movie(self, movie\_id: int, cinema\_id: int, title: str, rating: int) -> None:  
 self.session.query(Movie).filter\_by(movie\_id=movie\_id) \  
 .update({Movie.cinema\_id: cinema\_id, Movie.title: title, Movie.rating: rating})  
 self.session.commit()

**Результати виконання базових операцій з базою даних засобами ORM**

Зображення, що містить текст

Автоматично згенерований описРисунок 2 – Видалення даних з таблиці movie за допомогою ORMЗображення, що містить текст

Автоматично згенерований описРисунок 3 – Вставка даних в таблиці movie за допомогою ORM

Зображення, що містить текст

Автоматично згенерований опис Рисунок 4 – Редагування даних в таблиці movie за допомогою ORM

**Завдання №2**

Для тестування індексів було створено таблиці з 1000000 записів.

Hash

Хеш-індекси в PostgreSQL використовують форму структури даних хештаблиці (використовують хеш-функцію). Хеш-коди поділені на обмежену кількість комірок. Коли до індексу додається нове значення, PostgreSQL застосовує хешфункцію до значення і поміщає хеш-код і вказівник на кортеж у відповідну комірку. Коли відбувається запит за допомогою індексу хешування, PostgreSQL бере значення індексу і застосовує хеш-функцію, щоб визначити, яка комірка може містити потрібні дані.

Зображення, що містить текст

Автоматично згенерований описРисунок 5 - Аналіз виконання запиту 1 без індексу

Зображення, що містить текст

Автоматично згенерований описРисунок 6 - Аналіз виконання запиту 1 з індексом

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 7 - Аналіз виконання запиту 2 без індексу

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 8 - Аналіз виконання запиту 2 з індексом

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 9 - Аналіз виконання запиту 3 без індексу

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 10 - Аналіз виконання запиту 3 з індексом

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 11 - Аналіз виконання запиту 4 без індексу

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 12 - Аналіз виконання запиту 4 з індексом

Індексування за допомогою hash не значно пришвидшує пошук даних у табиці, а іноді навіть показує гірші результати, ніж запити без ідексування. Це випливає з того, що це один із найпримітивніших методів індексування і для пошуку потрібних даних алгоритм все одно проходить через усі записи у таблиці.

GIN

GIN призначений для обробки випадків, коли елементи, що підлягають індексації, є складеними значеннями (наприклад - реченнями), а запити, які обробляються індексом, мають шукати значення елементів, які з'являються в складених елементах (повторювані частини слів або речень). Індекс GIN зберігає набір пар (ключ, список появи ключа), де список появи — це набір ідентифікаторів рядків, у яких міститься ключ. Один і той самий ідентифікатор рядка може знаходитись у кількох списках, оскільки елемент може містити більше одного ключа. Кожне значення ключа зберігається лише один раз, тому індекс GIN дуже швидкий для випадків, коли один і той же ключ з’являється багато разів. Цей індекс може взаємодіяти тільки з полем типу tsvector.

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 13 – Аналіз виконання запиту 1 з індексом

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 14 – Аналіз виконання запиту 1 з індексом

Зображення, що містить текст

Автоматично згенерований описРисунок 15 – Аналіз виконання запиту 2 без індексу

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 16 – Аналіз виконання запиту 2 з індексом

Зображення, що містить текст

Автоматично згенерований описРисунок 17 – Аналіз виконання запиту 3 без індексу

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 18 – Аналіз виконання запиту 3 з індексом

Зображення, що містить текст

Автоматично згенерований описРисунок 19 – Аналіз виконання запиту 4 без індексу

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 20 – Аналіз виконання запиту 4 з індексом

В усіх заданих випадках пошук з індексацією відбувається значно швидше, ніж пошук без індексації (окрім першого, оскільки на перший запит дана індексація не впливає). Це відбувається завдяки головній особливості індексування GIN: кожне значення шуканого ключа зберігається один раз і запит іде не по всій таблиці, а лише по тим даним, що містяться у списку появи цього ключа.

**Завдання № 3**

Таблиці для тестування

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

);

Тригер

CREATE OR REPLACE FUNCTION before\_update\_delete\_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 multiple of 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;

END IF;

END;

$trigger$ LANGUAGE plpgsql;

CREATE TRIGGER "before\_update\_delete\_trigger"

BEFORE UPDATE OR DELETE ON "trigger\_test"

FOR EACH ROW

EXECUTE procedure before\_update\_delete\_func();

Початкові дані у таблицях

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');

Зображення, що містить стіл

Автоматично згенерований опис

Рисунок 21 – Початкові дані які занесені до таблиці trigger\_test

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 22 – початкова таблиця trigger\_test\_log

Виконаєм запит на оновлення  
UPDATE "trigger\_test" SET "trigger\_testName" = "trigger\_testName" || '\_log' WHERE "trigger\_testID" % 2 = 0;

Зображення, що містить стіл

Автоматично згенерований опис

Рисунок 23 – Таблиця trigger\_test після запиту на оновлення

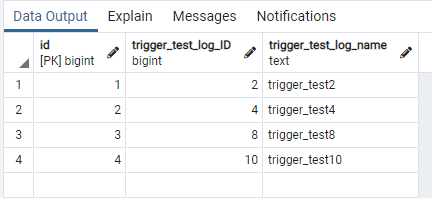


Рисунок 24 – Таблиця trigger\_test\_log після запиту на оновлення

Виконалась та гілка алгоритму тригера, що відповідає за парні рядки (оскільки є умова для парних), а для 6 рядка він також виконався, але пішов іншою (вкладеною) гілкою алгоритму та повернув старий стан (OLD). При запиті на оновлення потрібно повертати новий стан, а при запиті на видалення - старий.

Виконаєм запит на видалення  
DELETE FROM "trigger\_test" WHERE "trigger\_testID" % 3 = 0;

Зображення, що містить стіл

Автоматично згенерований опис

Рисунок 25 – Таблиця trigger\_test після запиту на видалення

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 26 – Таблиця trigger\_test\_log після запиту на видалення

Якщо виконувати ці запити окремо одне від одного, то у таблиці trigger\_test видаляються кратні трьом рядки, але таблиця trigger\_test\_log виявляється пустою. Так відбувається тому, що у гілці алгоритму для чисел кратних трьом у trigger\_test\_log лише модифікуються існуючі записи, але нові не додаються. Оскільки до цього не було виконано оновлення, ця таблиця пуста і модифікувати нема чого.

Виконаємо обидва запити підряд

**Зображення, що містить стіл

Автоматично згенерований опис**

Рисунок 27 – Таблиця trigger\_test після обох запитів підряд

Зображення, що містить текст

Автоматично згенерований опис

Рисунок 28 – Таблиця trigger\_test\_log після обох запитів підряд

Бачимо, що записи кратні трьом видалились з trigger\_test, а до текстових полів цих записів у кінці додалось "\_log". До текстових полів trigger\_test\_log на початку додались два вимволи "\_", а в кінці три "\_log". Один "\_log" в кінці додався завдяки виконанню запиту update для всіх парних рядків. А інші два "\_log" та два символи "\_" на початку додались тому, що запит на видалення для записів 3 та 9 виконались за тією самою гілкою алгоритму (кратні трьом), а запит на видалення запису 6 виконався за іншою гілкою (кратність 2 та 3).