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»**

**Тема**: «Засоби оптимізації роботи СУБД PostgreSQL»

Виконав:   
студент групи КВ-81

Путієнко Сергій  
Перевірив:  
Петрашенко А.В.

Київ 2020

*Метою роботи* є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL  
  
**Завдання**

*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:М, М:М 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*

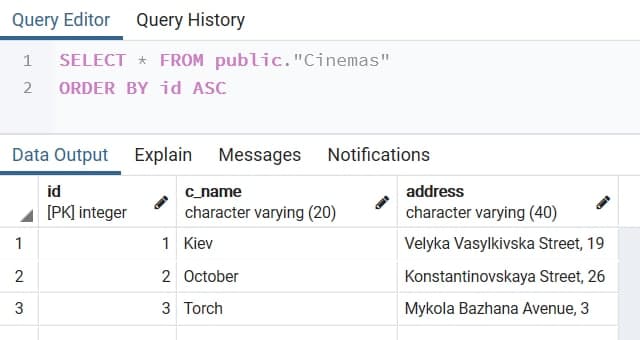
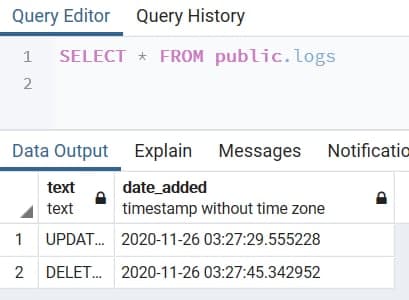
Відповідно до варіанту індексування продемонструвати на прикладах запитів SQL SELECT підвищення швидкодії їх виконання з використанням індексів, а також пояснити чому для деяких випадків індексування використовувати недоцільно. При цьому для наочного представлення слід використати функцію генерування 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).

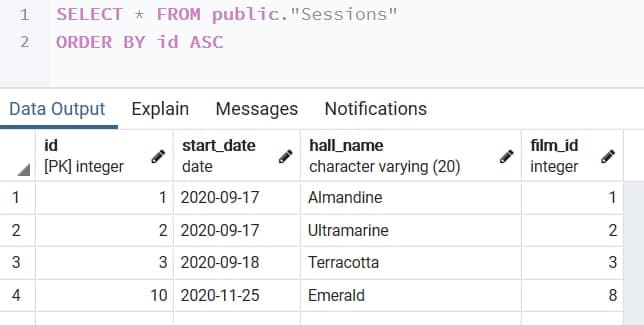
*Вимоги до пункту завдання №3*

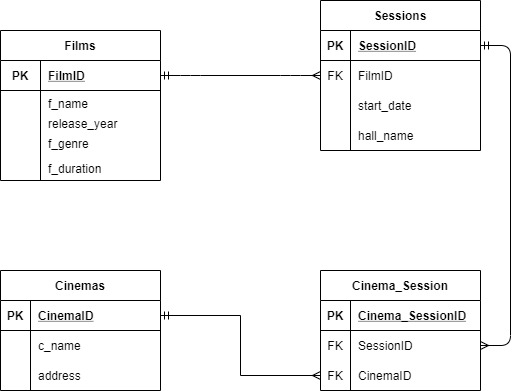
Створити тригер бази даних PostgreSQL відповідно до варіанта. Тригерна функція має включати обробку запису, що модифікується (вставляється або вилучається), умовні оператори, курсорні цикли та обробку виключних ситуацій. Виконати відлагодження тригера при різних вхідних даних, навівши 2-3 приклади його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

**Варіант 17**

|  |  |  |
| --- | --- | --- |
| *17* | *GIN, BRIN* | *before update, delete* |

**Завдання 1  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**





Зміст файлу base.py

from sqlalchemy import create\_engine  
from sqlalchemy.ext.declarative import declarative\_base  
from sqlalchemy.orm import sessionmaker  
  
engine = create\_engine('postgresql://postgres:qwerty@localhost:5432/tickets')  
Session = sessionmaker(bind=engine)  
  
Base = declarative\_base()

Перетворений модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM). Функції, що виконують вставку, вилучення, модифікації та отримання необхідних даних.

***Деякі основні функції з файлу orm\_model.py***

# Створити запис  
def create\_item(self, table\_name, columns, item):  
  
 obj = self.\_tables[table\_name]()  
 for i in range(len(columns)):  
 obj.\_\_dict\_\_[columns[i]] = item[i]  
 self.\_session.add(obj)  
 self.\_session.commit()  
  
# Створити декілька записів  
def create\_items(self, table\_name, columns, items):  
  
 for j in range(len(items)):  
 obj = self.\_tables[table\_name]()  
 for i in range(len(columns)):  
 obj.\_\_dict\_\_[columns[i]] = items[j][i]  
 self.\_session.add(obj)  
 self.\_session.commit()  
  
# Взяти дані про запис з бази  
def read\_item(self, table\_name, columns, item\_id):  
 col\_names = []  
 tbl\_entity = self.\_tables[table\_name]  
 for i in range(len(columns)):  
 col\_names.append(tbl\_entity.\_\_dict\_\_[columns[i]])  
  
 query = self.\_session.query(\*col\_names).filter(tbl\_entity.id == item\_id)  
 return query.all()  
  
# Прочитати дані таблиці з бази  
def read\_items(self, table\_name, columns):  
 tbl\_entity = self.\_tables[table\_name]  
 if columns is not None:  
 col\_names = []  
 for i in range(len(columns)):  
 col\_names.append(tbl\_entity.\_\_dict\_\_[columns[i]])  
 query = self.\_session.query(\*col\_names)  
 else:  
 query = self.\_session.query(tbl\_entity)  
 return query.all()  
  
# Оновити запис  
def update\_item(self, table\_name, columns, item, item\_id):  
  
 tbl\_entity = self.\_tables[table\_name]  
  
 update\_values = dict(zip(columns, item))  
 self.\_session.query(tbl\_entity) \  
 .filter(tbl\_entity.id == item\_id) \  
 .update(update\_values)  
  
 self.\_session.commit()  
  
# Видалити запис за ключем  
def delete\_item(self, table\_name, item\_id):  
 tbl\_entity = self.\_tables[table\_name]  
 self.\_session.query(tbl\_entity).filter(tbl\_entity.id == item\_id).delete()  
 self.\_session.commit()

Класи-сутності для об’єктів-сутностей, представлених відповідними таблицями БД.

***Файл cinema.py***

from sqlalchemy import Column, String, Integer  
from base import Base  
class Cinema(Base):  
 \_\_tablename\_\_ = 'Cinemas'  
  
 id = Column(Integer, primary\_key=True)  
 c\_name = Column(String(20))  
 address = Column(String(40))

def \_\_repr\_\_(self):  
 return "<Cinemas('%s', '%s')>" % (self.c\_name, self.address)

***Файл film.py***

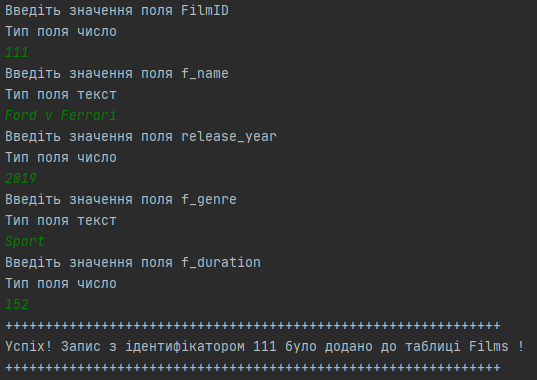
from sqlalchemy import Column, String, Integer, Date  
  
from base import Base  
  
class Film(Base):  
 \_\_tablename\_\_ = 'Films'  
 id = Column(Integer, primary\_key=True)  
 f\_name = Column(String(20))  
 release\_year = Column(Integer)  
 f\_genre = Column(String(10))  
 f\_duration = Column(Integer)  
 def \_\_repr\_\_(self):  
 return "<Films('%s', '%s', '%s', '%s')>" % (self.f\_name, self.release\_year, self.f\_genre, self.f\_duration)

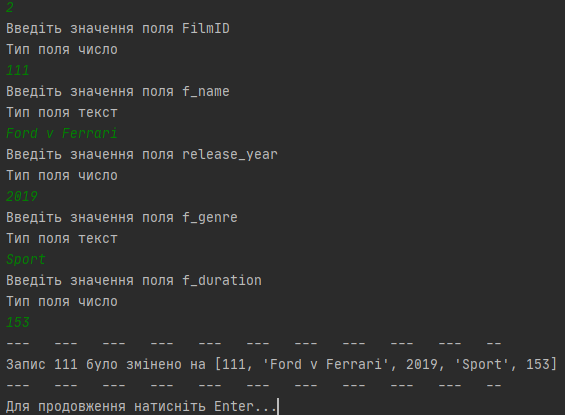
***Файл session.py***

from sqlalchemy import Column, String, Integer, Date, ForeignKey, Table  
from sqlalchemy.orm import relationship  
  
from base import Base  
  
cinema\_session\_association = Table(  
 'Cinema\_Session', Base.metadata,  
 Column('id', Integer, primary\_key=True),  
 Column('session\_id', Integer, ForeignKey('Sessions.id')),  
 Column('cinema\_id', Integer, ForeignKey('Cinemas.id'))  
)  
  
class Session(Base):  
 \_\_tablename\_\_ = 'Sessions'  
  
 id = Column(Integer, primary\_key=True)  
 start\_date = Column(Date)  
 hall\_name = Column(String(20))  
 film\_id = Column(Integer, ForeignKey('Films.id'))  
 film = relationship("Film", uselist=False)  
 cinemas = relationship("Cinema", secondary=cinema\_session\_association)

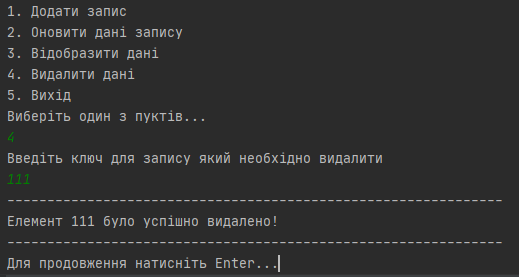
def \_\_repr\_\_(self):  
 return "<Sessions('%s', '%s', '%s')>" % (self.start\_date, self.hall\_name, self.film.s\_name)

Програма читає дані типів з бази даних і пропонує користувачу ввести відповідні дані. Приклад введення нових даних запису до таблиці Films.

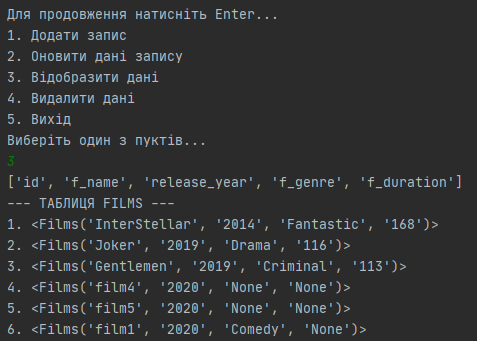


Оновлення даних таблиці Films:

Вилучення запису з таблиці Films:



Перегляд записів:



Запити з минулої роботи на мові SQL було замінено на запити засобами SQLAlchemy по роботі з об’єктами.

**Запит 1.**

SELECT f\_name, f\_genre, start\_date, hall\_name ' \  
 'FROM "Films", "Sessions", "Cinema\_Session", "Cinemas" ' \  
 'WHERE "Films"."FilmID" = "Sessions"."FilmID"' \  
 'AND "Sessions"."SessionID" = "Cinema\_Session"."SessionID"' \  
 'AND "Cinema\_Session"."CinemaID" = "Cinemas"."CinemaID"' \  
 'AND "Cinemas".c\_name = \'{}\' AND "Sessions".start\_date > \'{}\' '

def search\_query1(self, after\_date, cinema):  
 date\_obj = datetime.datetime.strptime(after\_date, '%Y-%m-%d').date()  
 query = self.\_session.query(Film.f\_name, Film.f\_genre, Session.start\_date, Session.hall\_name) \  
 .join(Session.film).join(Session.cinemas) \  
 .filter(Cinema.c\_name == cinema) \  
 .filter(Session.start\_date > date\_obj)  
 return query.all()

**Запит 2.**

'SELECT f\_name, f\_duration, start\_date, c\_name ' \  
 'FROM "Films", "Sessions", "Cinema\_Session", "Cinemas" ' \  
 'WHERE "Films"."FilmID" = "Sessions"."FilmID" ' \  
 'AND "Sessions"."SessionID" = "Cinema\_Session"."SessionID" ' \  
 'AND "Cinema\_Session"."CinemaID" = "Cinemas"."CinemaID" ' \  
 'AND "Sessions"."start\_date" > \'{}\' ' \  
 'AND "Films".f\_duration BETWEEN {} AND {}'

def search\_query2(self, after\_date, min\_duration, max\_duration):  
 date\_obj = datetime.datetime.strptime(after\_date, '%Y-%m-%d').date()  
 query = self.\_session.query(Film.f\_name, Film.f\_duration, Cinema.c\_name, Session.start\_date) \  
 .join(Session.film).join(Session.cinemas) \  
 .filter(Film.f\_duration.between(min\_duration, max\_duration)) \  
 .filter(Session.start\_date > date\_obj)  
 return query.all()

**Запит 3.**

'SELECT f\_name, f\_genre, f\_duration, release\_year ' \  
 'FROM "Films"' \  
 'WHERE ' \  
 '"Films".f\_name LIKE \'%{}%\' ' \  
 'AND "Films".f\_genre LIKE \'%{}%\' '

def search\_query3(self, str\_seq, str\_genre):  
  
 query = self.\_session.query(Film.f\_name, Film.f\_genre, Film.f\_duration, Film.release\_year)\  
 .filter(and\_(Film.f\_name.ilike('%'+str\_seq+'%'),  
 Film.f\_genre.ilike('%'+str\_genre+'%')))  
 return query.all()

**Завдання 2**

Для аналізу різних типів індексів у PostgreSQL було створено таблицю з двома полями.

CREATE TABLE test (

num integer,

txt character varying

);

Таблицю було заповнено 1000000 записів. Для заповнення таблиці було використано операції:

TRUNCATE test;

INSERT INTO test (num, txt)

SELECT

floor(random() \* 100 + 1)::int AS num,

md5(random()::character varying)

FROM generate\_series(1,1000000) ORDER BY num;

Для створення індексів GIN, BRIN по текстовому полю використовувалися такі операції:

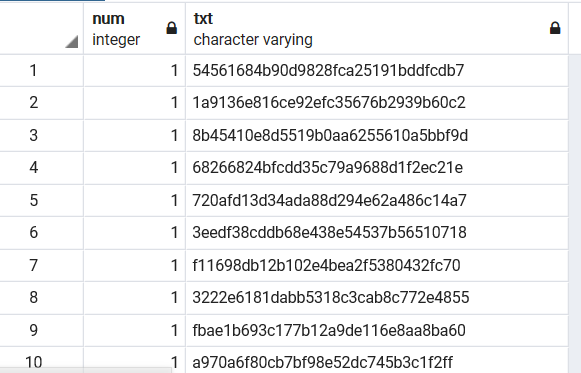
CREATE INDEX txt\_idx ON test USING gin (txt gin\_trgm\_ops);

CREATE INDEX txt\_idx ON test USING brin (txt);

Для створення індексів по числовому полю для BRIN (для індексу GIN не розглядалося):

CREATE INDEX num\_idx ON test USING brin (num);

Результат запиту SELECT \* FROM test.



Було розглянуто таки запити:

1. SELECT count(\*) FROM test WHERE txt ILIKE '%abc%';

2. SELECT count(\*) FROM test WHERE txt = '34c6b4e97f3f5a72d8e1e7df9dbf0367';

3. SELECT txt FROM test ORDER BY txt

4. SELECT num FROM test WHERE num > 33

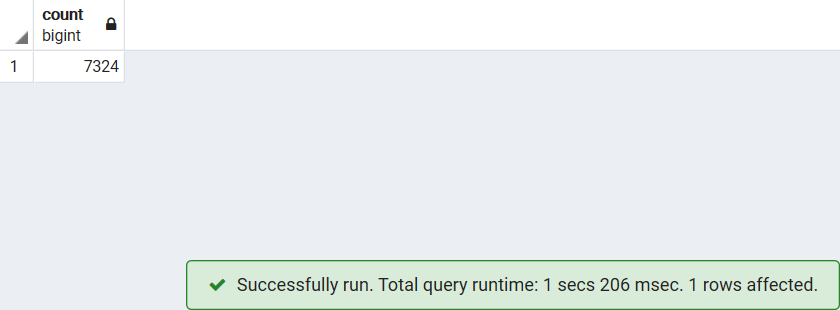
5. SELECT num, count(num) FROM test WHERE num > 70 GROUP BY num

Кожен запит виконувався декілька разів (5 разів) для визначення більш точного результату. У таблиці наведено середній час виконання запиту.

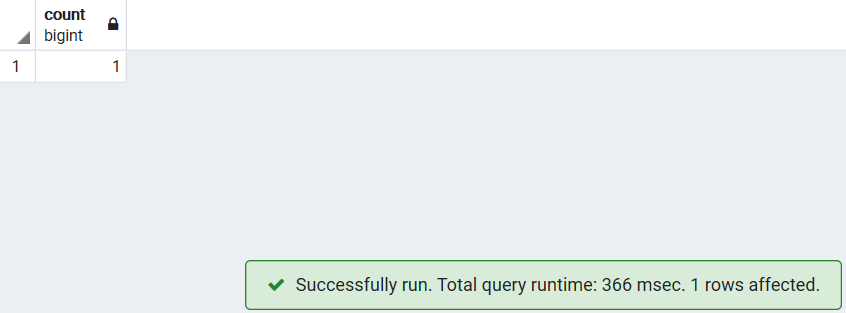
|  |  |  |  |
| --- | --- | --- | --- |
| № запиту | Без індексування | GIN | BRIN |
| 1 | 1171,8 | 193 | 1131,4 |
| 2 | 306,4 | 273,6 | 286,4 |
| 3 | 4562,2 | 4239,6 | 4231,4 |
| 4 | 353,4 |  | 325,4 |
| 5 | 322,8 |  | 282 |

Виведення результуючих даних для варіанту без індексів:

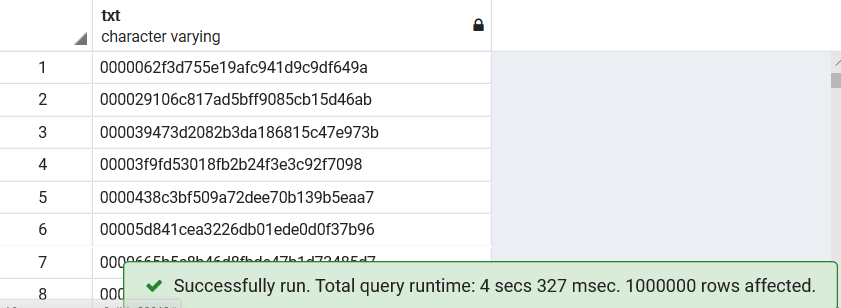
1. SELECT count(\*) FROM test WHERE txt ILIKE '%abc%';



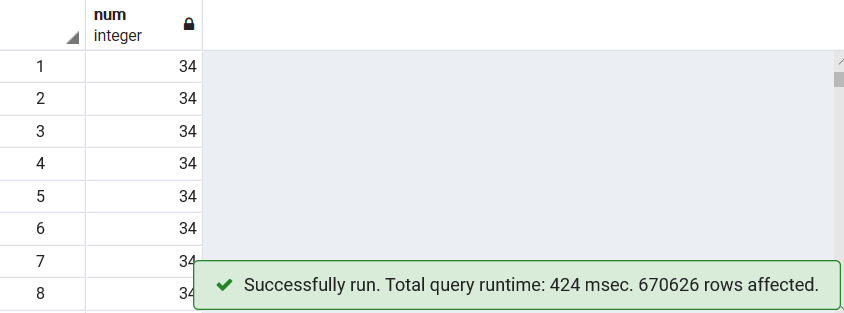
2. SELECT count(\*) FROM test WHERE txt = '34c6b4e97f3f5a72d8e1e7df9dbf0367';



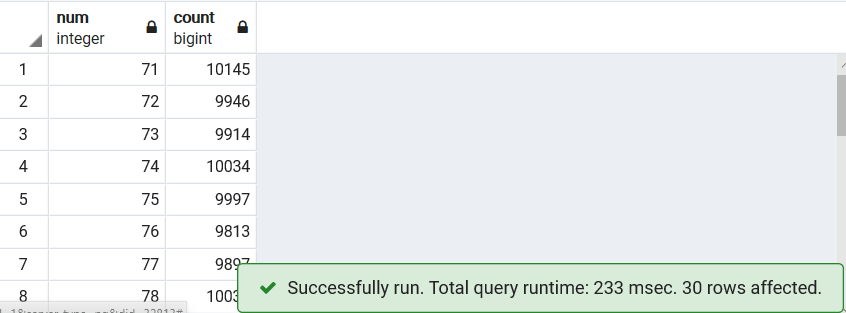
3. SELECT txt FROM test ORDER BY txt



4. SELECT num FROM test WHERE num > 33

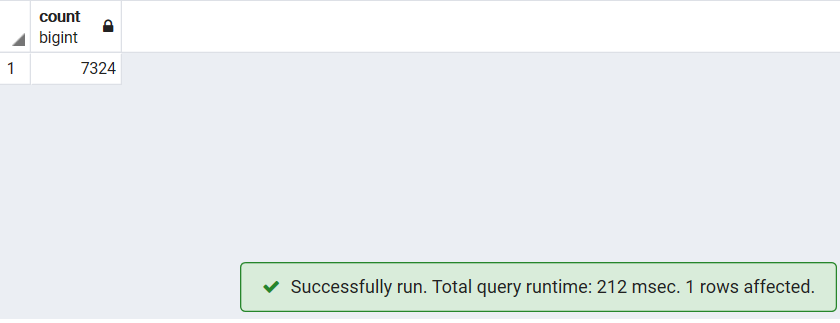


5. SELECT num, count(num) FROM test WHERE num > 70 GROUP BY

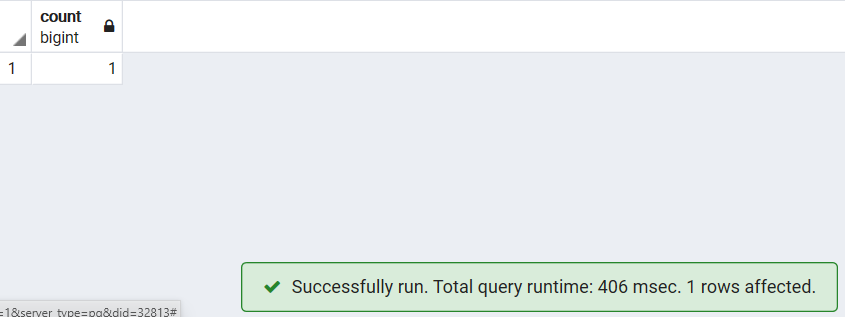


Варіант з індексом GIN:

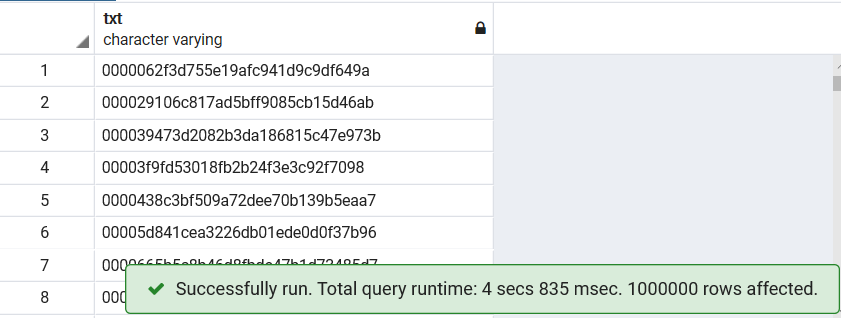
1. SELECT count(\*) FROM test WHERE txt ILIKE '%abc%';



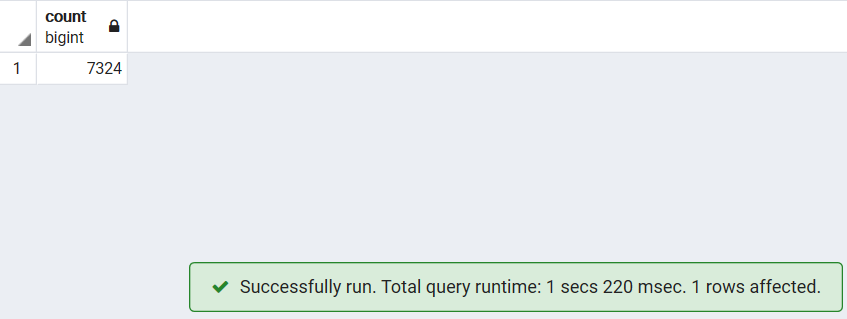
2. SELECT count(\*) FROM test WHERE txt = '34c6b4e97f3f5a72d8e1e7df9dbf0367';



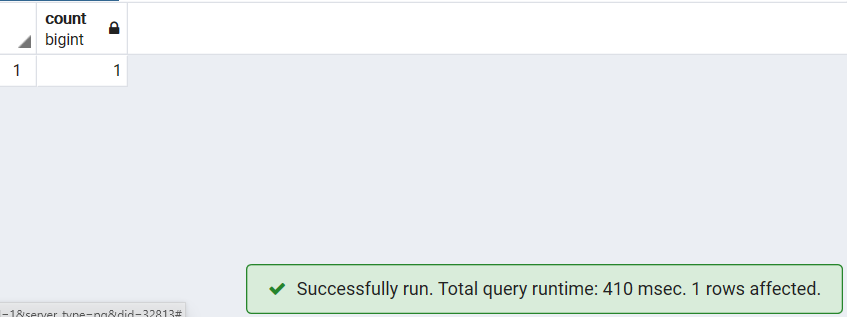
3. SELECT txt FROM test ORDER BY txt



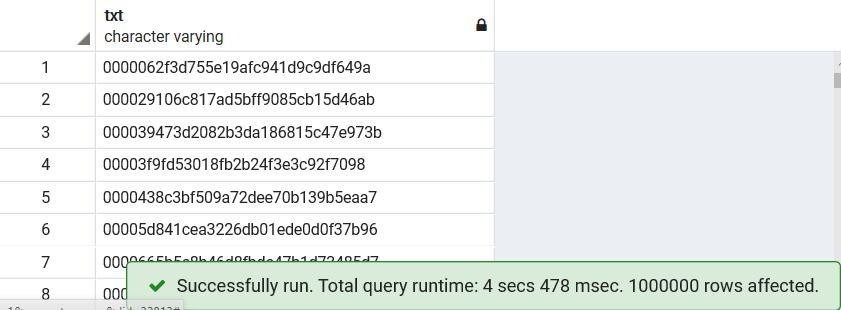
Варіант з індексом BRIN:

1. SELECT count(\*) FROM test WHERE txt ILIKE '%abc%';

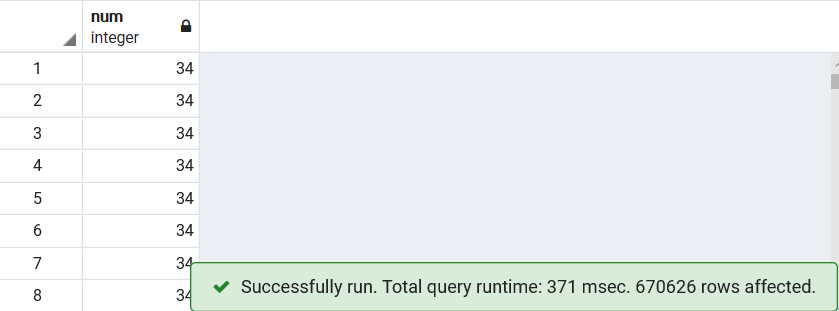
2. SELECT count(\*) FROM test WHERE txt = '34c6b4e97f3f5a72d8e1e7df9dbf0367';



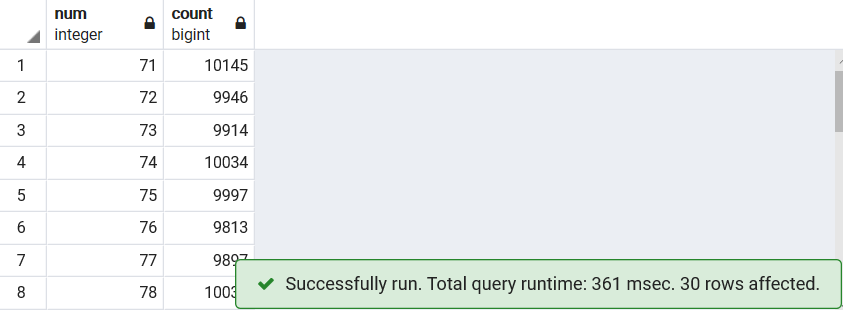
3. SELECT txt FROM test ORDER BY txt



4. SELECT num FROM test WHERE num > 33



5. SELECT num, count(num) FROM test WHERE num > 70 GROUP BY num



Як бачимо індекс GIN проявив себе досить добре при пошуку рядку символів у тексті. Але на GIN можливо розраховувати лише при повнотекстовому пошуку, або при використанні операцій LIKE, ILIKE. Та пошуку змісту полів типу JSON. В інших випадках його не слід застосовувати, а також він не може бути створений для чисельних типів стовпців. Також недоліком GIN досить довге створення індексу.

Індекс BRIN показав не досить значне підвищення швидкодії, але його було розраховано на застосування до дуже великих таблиць і 100000 записів не так вже і багато. Також загальна ідея BRIN це не збільшення швидкодії, а уникнення непотрібних рядків.

BRIN-індекс має сенс застосовувати для таблиць, в яких частина даних вже за своєю природою якось відсортована. Наприклад, це характерно для логів або для історії замовлень.

BRIN прискорює оператори порівняння, але не впливає на like-запити. BRIN - не є унікальним індексом, тому не може використовуватися в якості індексу первинного ключа.

**Завдання 3**

***Створення тригерної функції.***

CREATE FUNCTION check\_errors() RETURNS trigger

LANGUAGE plpgsql

AS $$

DECLARE

cur\_films cursor for select \* from "Films";

is\_found boolean;

sumstr varchar;

BEGIN

sumstr = '';

is\_found = false;

IF (TG\_OP = 'UPDATE') THEN

IF NEW.start\_date < CURRENT\_DATE OR NEW.start\_date is NULL THEN

sumstr := sumstr || ' start\_date';

RAISE NOTICE 'Start date % is not correct!', NEW.start\_date;

END IF;

IF NEW.hall\_name = '' OR NEW.hall\_name is NULL THEN

sumstr := sumstr || ' hall\_name';

RAISE NOTICE 'Hall name must be not empty!';

END IF;

FOR record IN cur\_films LOOP

IF NEW.film\_id = record.id THEN

is\_found = true;

EXIT;

END IF;

END LOOP;

IF is\_found = false THEN

sumstr := sumstr || ' film\_id';

RAISE NOTICE 'Film must be present in database!';

END IF;

IF sumstr <> '' THEN

sumstr := TG\_OP||' Sessions'||sumstr;

INSERT INTO logs(text,date\_added) values (sumstr,NOW());

--RAISE EXCEPTION 'Row % cannot updated!', NEW;

END IF;

RETURN NEW;

ELSIF TG\_OP = 'DELETE' THEN

sumstr := TG\_OP||' Sessions '|| OLD;

INSERT INTO logs(text,date\_added) values (sumstr,NOW());

RETURN OLD;

END IF;

RETURN NEW;

END;

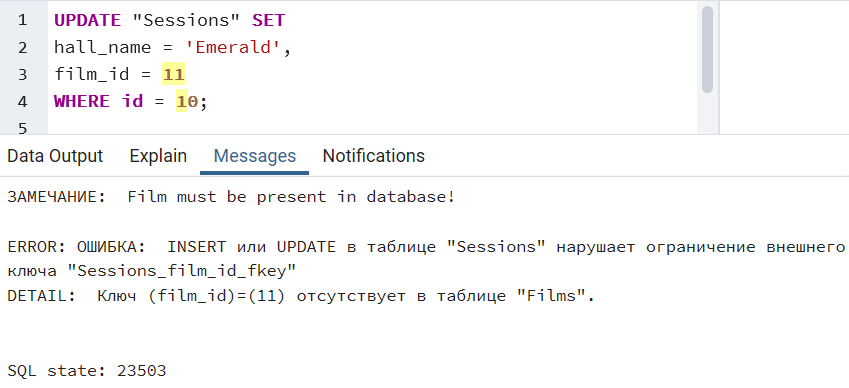
$$;

CREATE TRIGGER check\_errors BEFORE DELETE OR UPDATE ON "Sessions"

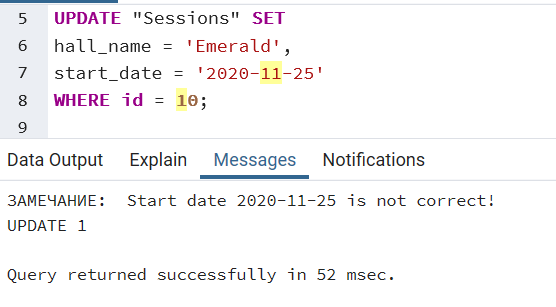
FOR EACH ROW EXECUTE FUNCTION check\_errors();

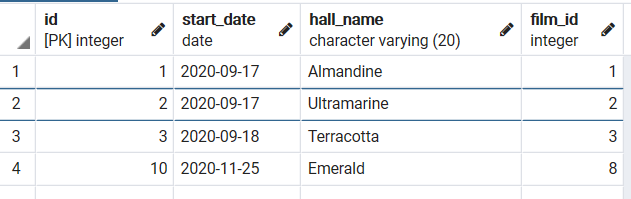
Даний тригер записує до таблиці logs всі видалення з таблиці «Sessions», спроби оновлення цієї таблиці з помилковими даними та сповіщає про помилки введення даних.

Перший приклад:

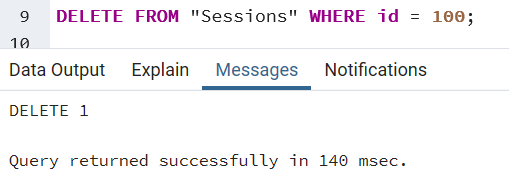


Другий приклад:

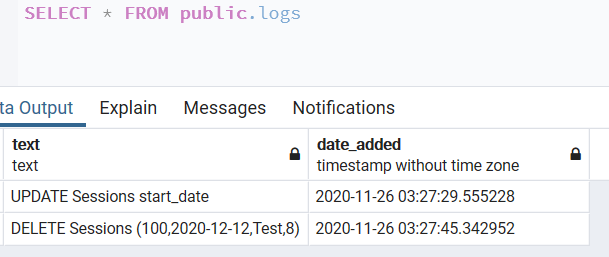




Третій приклад:



Зміст таблиці «logs» :



Код програми знаходиться у репозиторію github за посиланням:  
<https://github.com/CrispBS/Databases-and-controls.git>  
  
