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 3 курсу
ФПМ групи КВ-82
Бікерей Олексій Ігорович
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.

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.

3	*GIN, Hash*	*before delete, update*

Меню для навігації

Оглавление

Модель бази даних

На Рисунку 1 наведено структуру бази даних яка використовується в даній роботі. Змін, в порівнянні з 1 та 2 лабораторною роботами не відбулося.

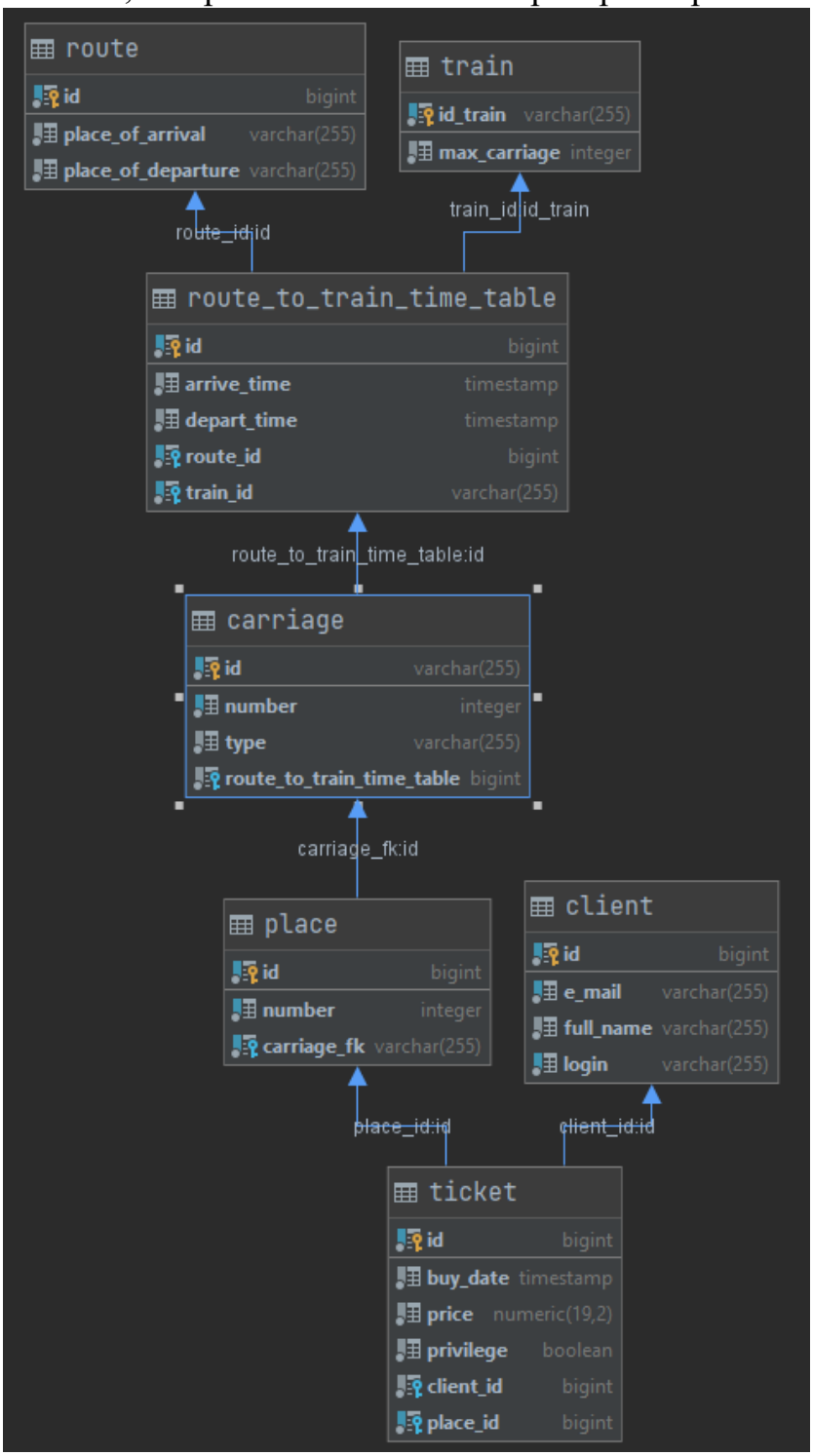


Рисунок 1 – Логічна модель (Структура) БД “ Сервіс продажу залізничних квитків”

Завдання 1

Середовище розробки та налаштування підключення до бази даних

Для виконання лабораторної роботи використовувалася мова програмування Java фреймворк Spring Data який включає в себе бібліотеку Hibernate , яка є найпопулярнішою реалізацією специфікації JPA, створеної для вирішення задач об'єктно-реляційного відображення.

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>

   <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <scope>runtime</scope>
   </dependency>
   <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
   </dependency>
</dependencies>
```

Рисунок 2 – Залежності які використовуються в проекті

Доступ до бази даних

Доступ відбувається за допомогою файлу конфігурації в якому зберігаються всі необхідні параметри для доступу.

```
spring.jpa:
  database: POSTGRESQL
  hibernate.ddl-auto: create-drop
  show-sql: true

spring.datasource:
  platform: postgres
  driverClassName: org.postgresql.Driver
  url: jdbc:postgresql://localhost:4814/trainTicket
  username: postgres
  password: 4814
```

Рисунок 3 – Вміст файлу конфігурації

Структура програми

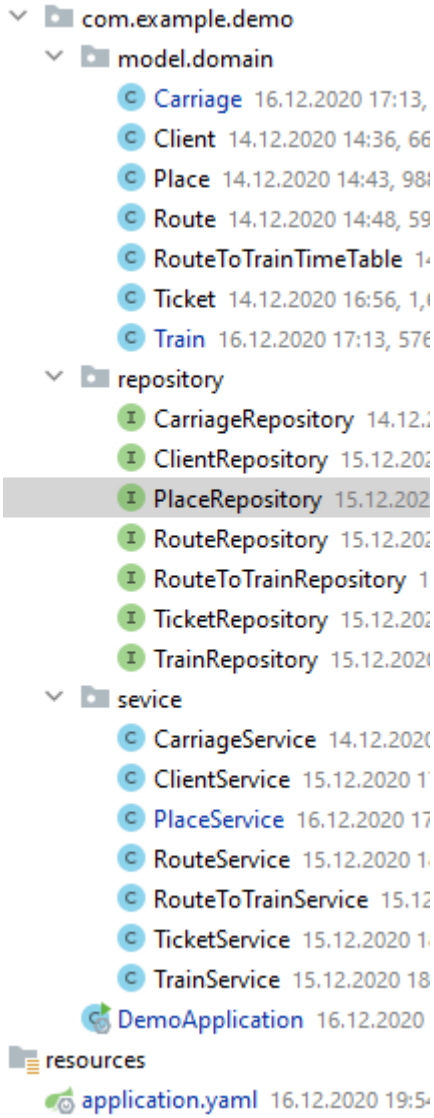


Рисунок 4 – Структура програми

Опис структури програми:

В пакеті model.domain описуються класи які описують всі таблиці у БД та їх зв’язки.

В пакеті repository описуються інтерфейси JPA Entity забезпечують основні операції для пошуку зберігання та видалення. Але для реалізації оновлювання даних використовується методи запитів @Query

В пакеті service описуються класи які реалізують надання кінцевих даних для інтерфейсу користувача.

Продемонструємо код лише для одної класу Carriage

Сам клас Carriage, де через анотації описується зв’язок класу із таблицею із бази даних.

```
package com.example.demo.model.domain;

import com.example.demo.repository.RouteToTrainRepository;
import com.example.demo.sevice.RouteToTrainService;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;

@Entity
@Data
@NoArgsConstructor
@Table(uniqueConstraints = {@UniqueConstraint(columnNames = {"number",
"id"})})
public class Carriage {

    public Carriage(String id, String type, String number, String
routeToTrainTimeTable,
                    RouteToTrainRepository routeToTrainRepository) throws
Exception {
        this.id = id;
        this.type = type;
        this.number = Integer.parseInt(number);
        final RouteToTrainService routeToTrainService = new
RouteToTrainService(routeToTrainRepository);
        this.routeToTrainTimeTable =
routeToTrainService.findById(Long.parseLong(routeToTrainTimeTable));
    }

    @Id
    private String id;

    @Column(nullable = false)
    private String type;

    @Column(nullable = false)
    private Integer number;

    @ManyToOne
    @JoinColumn(name = "route_to_train_time_table", nullable = false)
    private RouteToTrainTimeTable routeToTrainTimeTable;
}
```

Інтерфейс CarriageRepository

```
package com.example.demo.repository;

import com.example.demo.model.domain.Carriage;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

@Repository
public interface CarriageRepository extends JpaRepository<Carriage, String> {

    @Query("update Carriage c set c.type = :type where c.id = :carriageId")
    void setCarriageType(@Param("carriageId") String id, @Param("type")
String type);
}
```

Клас CarriageService

```
package com.example.demo.sevice;

import com.example.demo.model.domain.Carriage;
import com.example.demo.repository.CarriageRepository;
import org.springframework.stereotype.Service;

@Service
public class CarriageService {

    private final CarriageRepository carriageRepository;

    public CarriageService(CarriageRepository carriageRepository) {
        this.carriageRepository = carriageRepository;
    }

    public void add(Carriage carriage) {
        carriageRepository.saveAndFlush(carriage);
    }

    public Carriage findById(String id) throws Exception {
        return carriageRepository.findById(id).orElseThrow(() -> new
Exception("Не существует записи с данным id - " + id));
    }

    public void deleteById(String id) {
        carriageRepository.deleteById(id);
    }

    public void setCarriageType(String id, String newType) {
        carriageRepository.setCarriageType(id, newType);
    }

}
```

Інші класи та функції реалізовані по аналогії.

Завдання 2
Створення та аналіз індекса GIN

GIN - це Generalized Inverted Index, або обернений індекс. Його основною задачею є прискорення повнотекстового пошуку і тому вивчати даний індекс будемо на цьому прикладі.

Додамо до таблиці Client колонку ts_vec с типом tsvector та заповнимо його за допомогою функції to_tsvector, який буде приймати як аргумент значення колонки full_name . Саме на колонці ts_vec будемо створювати індекс.

	full_name character varying (40)	ts_vec tsvector
1	ns	'ns':1
2	al	'al':1
3	um	'um':1
4	bn	'bn':1
5	ga	'ga':1
6	fj	'fj':1
7	ob	'ob':1
8	qs	'qs':1
9	pw	'pw':1
10	rc	'rc':1
11	iq	'iq':1
12	ue	'ue':1
13	jg	'jg':1

Рисунок 5 – Згенерована таблиця

Для початку дізнаємось, які лексеми найчастіше зустрічаються, а які ні.

```
1  select word, ndoc from ts_stat('select ts_vec from "client"') order by ndoc desc limit 5
```

Data Output | Explain | Messages | Notifications

	word text	ndoc integer
1	gf	752
2	hp	747
3	rb	747
4	hs	744
5	vr	743

Рисунок 6 – Найчастіше вживані лексеми

Бачимо, що найчастіше зустрічається “gf” та “hp”.

```
1  select word, ndoc from ts_stat('select ts_vec from "client"') order by ndoc asc limit 5
```

Data Output | Explain | Messages | Notifications

	word text	ndoc integer
1	yyuvukwv	1
2	yyubuyk	1
3	yytknfni	1
4	yyrolmxc	1
5	yyxvxdgp	1

Рисунок 6 – Найменш вживані лексеми

А рідше всього зустрічається “ yyuvukwv”.

Тепер виконаємо пошук по цих словах без індексу

```
1  select * from client where ts_vec @@ to_tsquery('gf')
```

✔ Successfully run. Total query runtime: 341 msec. 752 rows affected.

✔ Successfully run. Total query runtime: 340 msec. 752 rows affected.

✔ Successfully run. Total query runtime: 343 msec. 752 rows affected.

Середній час = 341 msec

```
1  select * from client where ts_vec @@ to_tsquery('hp')
```

✔ Successfully run. Total query runtime: 350 msec. 747 rows affected.

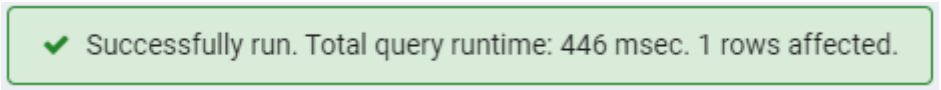
✔ Successfully run. Total query runtime: 356 msec. 747 rows affected.

✔ Successfully run. Total query runtime: 355 msec. 747 rows affected.

Середній час = 354 msec

```
1  select * from client where ts_vec @@ to_tsquery('yyuvukwv')
```

✔ Successfully run. Total query runtime: 456 msec. 1 rows affected.



Середній час = 444 msec

Тепер створимо GIN індекс:

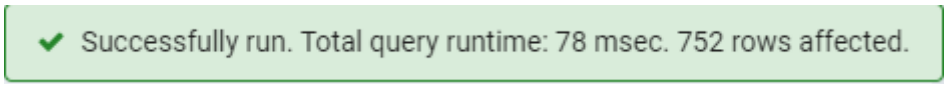
```
1  create index on client using gin(ts_vec)
```

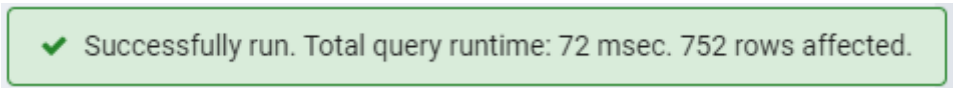
Data Output | Explain | Messages | Notifications

```
CREATE INDEX

Query returned successfully in 372 msec.
```

Виконаємо ті ж самі запити

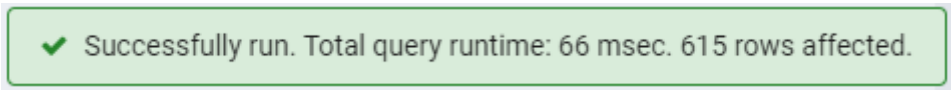
```
1  select * from client where ts_vec @@ to_tsquery('gf')
```

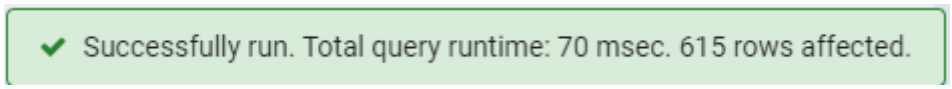


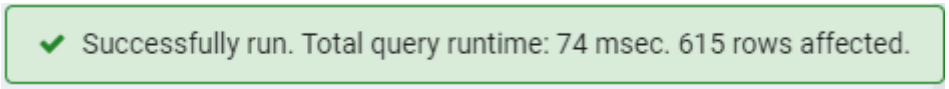


Середній час = 71 msec

```
1  select * from client where ts_vec @@ to_tsquery('hp')
```







Середній час = 70 msec

```
1  select * from client where ts_vec @@ to_tsquery('yyuvukwv')
```

Successfully run. Total query runtime: 57 msec. 1 rows affected.

Successfully run. Total query runtime: 59 msec. 1 rows affected.

Successfully run. Total query runtime: 59 msec. 1 rows affected.

Середній час = 58 msec

Тепер бачимо, що після створення індекса, пошук та відображення даних для найчастіше зустрічаємих лексем змінився у меншу сторону, але для рідко зустрічаємих лексем видно покращення результату ще більше. Тобто бачимо, що застосування індексу GIN покращило роботу повнотекстового пошуку.

GIN добре підходить для даних, які не часто оновлюються. Якщо поміркувати то для таблиці, де зберігаються часто-змінні дані не рекомендовано використовувати індекс GIN, бо переіндексація може займати багато часу.

Створення та аналіз індекса HASH

HASH – це режим індексу, який автоматично застосовує хеш-функцію, до даних індексу. Хоча хешування і займає додатковий час, при великої кількості даних, це може значно прискорити виконання запитів, через те, що час доступу хеш-таблиці менший ніж у звичайних колекцій.

Виконаємо тестовий запит на пошук записів за значення поля id_client.

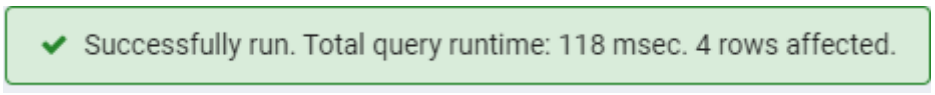
```
1  select * from client where id_client = 242274 or id_client = 400000 or id_client = 600000 or id_client = 500000
```

Data Output | Explain | Messages | Notifications

	id_client [PK] integer	email character varying (50)	login character varying (100)	full_name character varying (40)
1	600000	wxmqepdex@gmail.com	gsqrilb	yf
2	242274	uhuibiqqi@gmail.com	kcndhcj	hl
3	400000	urpiqhqwj@gmail.com	tvbouav	ty
4	500000	iyqosbwmb@gmail.com	upsfqhn	ee

Successfully run. Total query runtime: 102 msec. 4 rows affected.

Successfully run. Total query runtime: 106 msec. 4 rows affected.



Середній час = 109 msec

Створимо індекс на полі id_client, використовуючи хеш-функцію для індексації даних:

```
1  create index on client using hash(id_client)
```

Data Output | Explain | Messages | Notifications

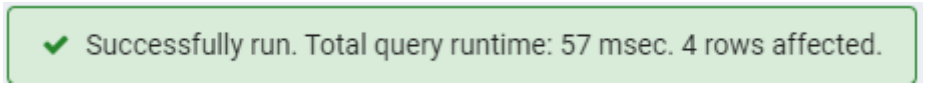
```
CREATE INDEX

Query returned successfully in 1 secs 283 msec.
```

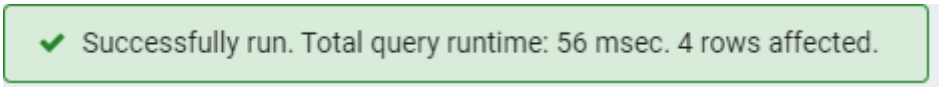
Виконуємо попередній запит ще раз, та порівняємо затрачений час:

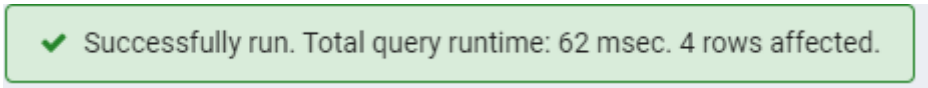
```
1  select * from client where id_client = 242274 or id_client = 400000 or id_client = 600000 or id_client = 500000
```

Data Output | Explain | Messages | Notifications

	id_client [PK] integer	email character varying (50)	login character varying (100)	full_name character varying (40)
1	600000	wxmqepdex@gmail.com	gsqrilb	yf
2	242274	uhuibiqqi@gmail.com	kcndhcj	hl
3	400000	urpiqhqwj@gmail.com	tvbouav	ty
4	500000	iyqosbwmb@gmail.com	upsfqhn	ee







Середній час = 58 msec

Отриманий результат показує, що у даному випадку застосування хешування пришвидшило виконання запиту приблизно у два рази. Все через те, що пошук – найсильніша сторона цього методу індексування, через метод роботи хештаблиці.

Наведемо ще декілька прикладів.

Запит, з використанням предикату над індексом:

```
1  select * from client where id_client > 240000 and id_client < 500000 and login like '%repo%'
```

Data Output | Explain | Messages | Notifications

	id_client [PK] integer	email character varying (50)	login character varying (100)	full_name character varying (40)
1	323200	uuptgvgqy@gmail.com	repoiyc	bg
2	416484	lcheojcku@gmail.com	repolii	ql
3	443836	nsxpujlfj@gmail.com	krepotj	qt
4	477095	somemnikn@gmail.com	pvdrepo	hi

Час виконання без хешування:

✔ Successfully run. Total query runtime: 163 msec. 4 rows affected.

✔ Successfully run. Total query runtime: 163 msec. 4 rows affected.

✔ Successfully run. Total query runtime: 165 msec. 4 rows affected.

Середній час = 163 msec

З використанням хешування:

✔ Successfully run. Total query runtime: 156 msec. 4 rows affected.

✔ Successfully run. Total query runtime: 165 msec. 4 rows affected.

✔ Successfully run. Total query runtime: 171 msec. 4 rows affected.

Середній час = 164 msec

Значного прискорення у цьому випадку немає.

Запит, з використанням сортування, яке потребує багато порівнянь об’єктів:

```
1  select * from client order by id_client desc
```

Data Output | Explain | Messages | Notifications

	id_client [PK] integer	email character varying (50)	login character varying (100)	full_name character varying (40)
1	624967	mwolsvupy@gmail.com	fvhhtvh	cw
2	624966	kxgjyiudn@gmail.com	mmdhbpj	qk
3	624965	kffelgiyp@gmail.com	vwwepux	pb
4	624964	hptdfrrcl@gmail.com	enjrxtr	lj
5	624963	pnkxyrcaw@gmail.com	waiqhkv	ua

Час виконання без хешування:

✔ Successfully run. Total query runtime: 337 msec. 489694 rows affected.

✔ Successfully run. Total query runtime: 342 msec. 489694 rows affected.

✔ Successfully run. Total query runtime: 330 msec. 489694 rows affected.

Середній час = 336 msec

З використанням хешування:

✔ Successfully run. Total query runtime: 313 msec. 489694 rows affected.

✔ Successfully run. Total query runtime: 323 msec. 489694 rows affected.

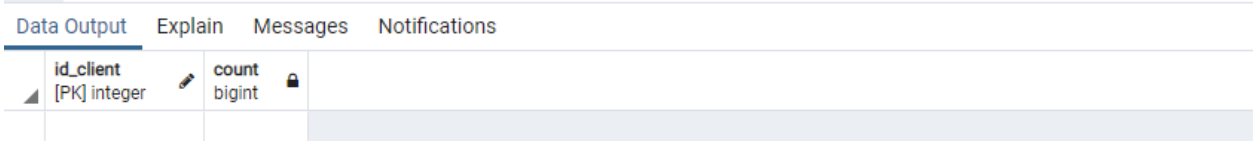
✔ Successfully run. Total query runtime: 333 msec. 489694 rows affected.

Середній час = 323 msec

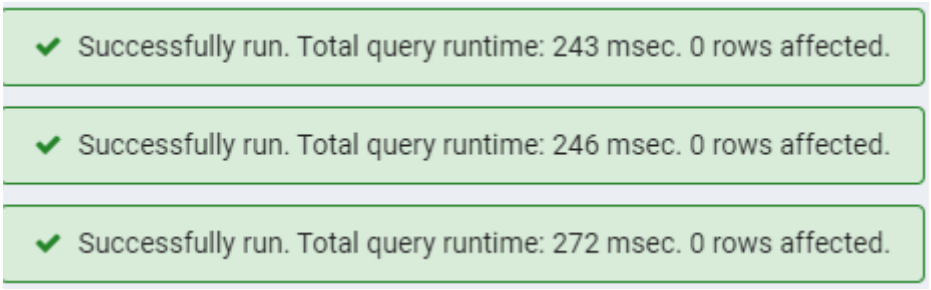
Бачимо, що результат однаковий у межах похибки. Через те, що порівняння хешів, виконується однаково швидко, з порівнянням чисел індексів. Якби полем був рядок, ми б мали незначний приріст швидкодії.

Запит пошуку користувачів з однаковим полем client_id:

```
1  select id_client, count(full_name) from client group by id_client having count(full_name) > 1
```

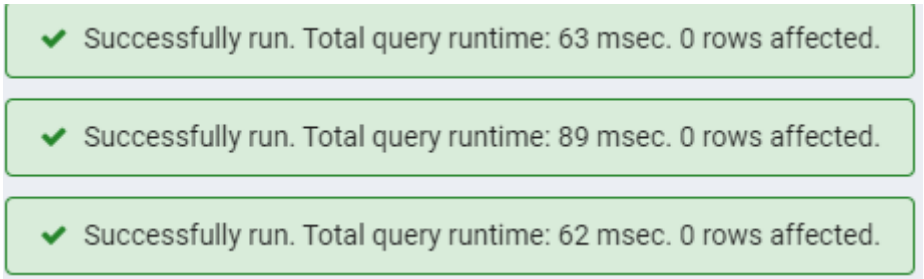


Час виконання без хешування:



Середній час = 254 msec

З використанням хешування:



Середній час = 71 msec

Бачимо приріст більше ніж у два рази, через те, що тут виконується операція схожа на пошук. Тому хеш-таблиця дозволяє значно пришвидшити процес групування даних.

Наведені приклади кажуть, що хешування індексу ефективне, для виконання сортування, групування або пошуку даних (випадки, де використовується багато порівнянь поля), проте воно неефективне, для прямої роботи з даними.

Також, хешування займає значний час, тому цей спосіб буде ефективний, лише при великій кількості даних, та за умови, що індекси оновлюються не часто.

Завдання 3

Для тестування тригерів створимо ще одну таблицю, для зберігання повідомлень від тригеру.

```
-- Table: public.logs

-- DROP TABLE public.logs;

CREATE TABLE public.logs
(
    text character varying COLLATE pg_catalog."default",
    "time" date,
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1 START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),
    CONSTRAINT logs_pkey PRIMARY KEY (id)
)

TABLESPACE pg_default;

ALTER TABLE public.logs
    OWNER to postgres;
```

Команда створення самого тригеру:

```
CREATE OR REPLACE FUNCTION my_trigger_func() RETURNS trigger AS $$
DECLARE
  curs CURSOR FOR SELECT * FROM client;
  ROW client%ROWTYPE;
BEGIN
  IF (TG_OP = 'UPDATE') THEN
    FOR row IN curs LOOP
        IF new.email LIKE row.email THEN
        RAISE NOTICE 'Email should be unique';
        RETURN NULL;
        END IF;
    END LOOP;
    INSERT INTO logs (text, time) VALUES('Update operation from client table', NOW());
    RAISE NOTICE 'Successful Update';
    RETURN NEW;
ELSEIF(TG_OP = 'DELETE') THEN
    INSERT INTO logs (text, time) VALUES ('Delete operation from client table', NOW());
    RAISE NOTICE 'Successful Delete';
    RETURN OLD;
    END IF;
END;
$$ language plpgsql;

DROP TRIGGER IF EXISTS my_trigger on client;

CREATE TRIGGER my_trigger
BEFORE DELETE OR UPDATE
ON client
FOR EACH ROW EXECUTE PROCEDURE my_trigger_func();
```

Data Output | Explain | Messages | Notifications

```
ПОВІДОМЛЕННЯ:  тригеру "my_trigger" для відношення "client" не існує, пропускаємо
CREATE TRIGGER

Query returned successfully in 83 msec.
```

Виконаємо запит видалення з таблиці:

```
1  delete from client where email = 'vtseuepmi@gmail.com' or email = 'txnxunsib@gmail.com'
```

Data Output | Explain | Messages | Notifications

```
ПОВІДОМЛЕННЯ:  Successful Delete
ПОВІДОМЛЕННЯ:  Successful Delete
DELETE 2

Query returned successfully in 57 msec.
```

Бачимо повідомлення від тригеру. Зміст таблиці Logs також був змінений:

```
1  SELECT * FROM public.logs
2  ORDER BY id ASC
```

Data Output | Explain | Messages | Notifications

	text character varying	time date	id [PK] integer
1	Delete operation from client table	2020-12-17	3
2	Delete operation from client table	2020-12-17	4

Виконаємо запит оновлення даних:

```
1  Update client SET full_name = 'update' where email = 'syjxjyiyv@gmail.com'
```

Data Output | Explain | Messages | Notifications

```
ПОВІДОМЛЕННЯ:  Email should be unique
UPDATE 0

Query returned successfully in 208 msec.
```

Тригер повернув «Помилку» через те, що ми не виконали його умову оновлення даних. Виконаємо запит, з збереженням унікальності поля email:

```
1  Update client SET email = 'asihfaohsf', full_name = 'update' where email = 'syjxjyiyv@gmail.com'
```

Data Output Explain Messages Notifications

```
ПОВІДОМЛЕННЯ:  Successful Update
UPDATE 1

Query returned successfully in 430 msec.
```

Бачимо повідомлення від тригеру. Зміст таблиці Logs також був змінений:

```
1  SELECT * FROM public.logs
2  ORDER BY id ASC
```

Data Output Explain Messages Notifications

	text character varying	time date	id [PK] integer
1	Delete operation from client table	2020-12-17	3
2	Delete operation from client table	2020-12-17	4
3	Update operation from client table	2020-12-17	5