S

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 ІII курсу

групи КВ-91

Бітлян А. В.

Перевірив: Павловський В. І.

Київ – 2021

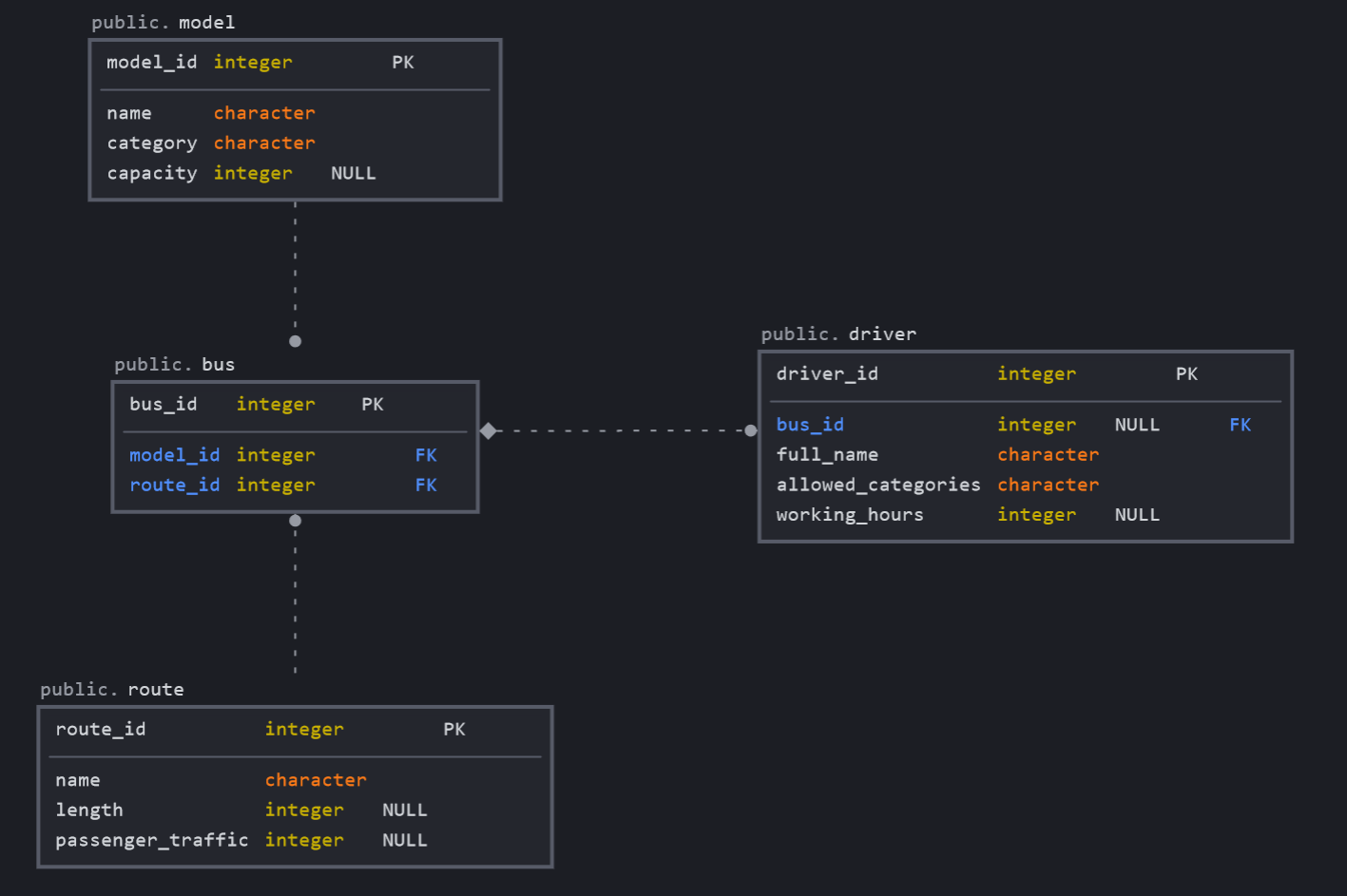
**Варіант 2**

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 бази даних**

Нижче (Рис. 1) наведено логічну модель бази даних:



Логічна модель бази даних

Зміни у порівнянні з першою лабораторною роботою відсутні.

**Класи ORM**

BusEntity.java

import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Table;  
import java.io.Serializable;  
import java.sql.Time;  
  
  
@Entity  
@Table (name = "bus")  
public class BusEntity implements Serializable{  
  
 @Id  
 @Column  
 private int bus\_id;  
  
 @Column  
 private int model\_id;  
  
 @Column  
 private int route\_id;  
  
 public int getBus\_id()  
 {  
 return bus\_id;  
 }  
  
 public void setBus\_id(int bus\_id)  
 {  
 this.bus\_id = bus\_id;  
 }  
  
 public int getModel\_id()  
 {  
 return model\_id;  
 }  
  
 public void setModel\_id(int model\_id)  
 {  
 this.model\_id = model\_id;  
 }  
  
 public int getRoute\_id()  
 {  
 return route\_id;  
 }  
  
 public void setRoute\_id(int route\_id)  
 {  
 this.route\_id = route\_id;  
 }  
  
  
 @Override  
 public String toString(){  
 return "Bus {" +  
 " bus\_id = " + bus\_id +  
 " model\_id = " + model\_id +  
 " route\_id = " + route\_id +  
 " }";  
 }  
}

ModelEntity.java

import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Table;  
import java.io.Serializable;  
import java.sql.Time;  
  
  
@Entity  
@Table (name = "model")  
public class ModelEntity implements Serializable{  
  
 @Id  
 @Column  
 private int model\_id;  
  
 @Column  
 private String name;  
  
 @Column  
 private String category;  
  
 @Column  
 private int capacity;  
  
 public int getModel\_id()  
 {  
 return model\_id;  
 }  
  
 public void setModel\_id(int model\_id)  
 {  
 this.model\_id = model\_id;  
 }  
  
 public String getName()  
 {  
 return name;  
 }  
  
 public void setName(String name)  
 {  
 this.name = name;  
 }  
  
 public String getCategory()  
 {  
 return category;  
 }  
  
 public void setCategory(String category)  
 {  
 this.category = category;  
 }  
  
 public int getCapacity()  
 {  
 return capacity;  
 }  
  
 public void setCapacity(int capacity)  
 {  
 this.capacity = capacity;  
 }  
  
  
 @Override  
 public String toString(){  
 return "Model {" +  
 " model\_id = " + model\_id +  
 " name = " + name +  
 " category = " + category +  
 " capacity = " + capacity +  
 " }";  
 }  
  
}

RouteEntity.java

import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Table;  
import java.io.Serializable;  
  
@Entity  
@Table (name = "route")  
public class RouteEntity implements Serializable{  
  
 @Id  
 @Column  
 private int route\_id;  
  
 @Column  
 private String name;  
  
 @Column  
 private int length;  
  
 @Column  
 private int passenger\_traffic;  
  
 public int getRoute\_id()  
 {  
 return route\_id;  
 }  
  
 public void setRoute\_id(int ticket\_id)  
 {  
 this.route\_id = ticket\_id;  
 }  
  
 public String getName()  
 {  
 return name;  
 }  
  
 public void setName(String name)  
 {  
 this.name = name;  
 }  
  
 public int getLength()  
 {  
 return length;  
 }  
  
 public void setLength(int length)  
 {  
 this.length = length;  
 }  
  
 public int getPassenger\_traffic()  
 {  
 return passenger\_traffic;  
 }  
  
 public void setPassenger\_traffic(int passenger\_traffic)  
 {  
 this.passenger\_traffic = passenger\_traffic;  
 }  
  
  
 @Override  
 public String toString(){  
 return "Route {" +  
 " route\_id = " + route\_id +  
 " name = " + name +  
 " length = " + length +  
 " passenger\_traffic = " + passenger\_traffic +  
 " }";  
 }  
}

DriverEntity.java

import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Table;  
import java.io.Serializable;  
  
@Entity  
@Table (name = "driver")  
public class DriverEntity implements Serializable{  
  
 @Id  
 @Column  
 private int driver\_id;  
  
 @Column  
 private int bus\_id;  
  
 @Column  
 private String full\_name;  
  
 @Column  
 private int allowed\_categories;  
  
 @Column  
 private int working\_hours;  
  
 public int getDriver\_id()  
 {  
 return driver\_id;  
 }  
  
 public void setDriver\_id(int ticket\_id)  
 {  
 this.driver\_id = ticket\_id;  
 }  
  
 public int getBus\_id() {  
 return bus\_id;  
 }  
  
 public void setBus\_id(int bus\_id) {  
 this.bus\_id = bus\_id;  
 }  
  
 public String getFull\_name()  
 {  
 return full\_name;  
 }  
  
 public void setFull\_name(String full\_name)  
 {  
 this.full\_name = full\_name;  
 }  
  
 public int getAllowed\_categories()  
 {  
 return allowed\_categories;  
 }  
  
 public void setAllowed\_categories(int passenger\_traffic)  
 {  
 this.allowed\_categories = passenger\_traffic;  
 }  
  
 public int getWorking\_hours() {  
 return working\_hours;  
 }  
  
 public void setWorking\_hours(int working\_hours) {  
 this.working\_hours = working\_hours;  
 }  
  
  
 @Override  
 public String toString(){  
 return "Driver {" +  
 " driver\_id = " + driver\_id +  
 " bus\_id = " + bus\_id +  
 " full\_name = " + full\_name +  
 " allowed\_categories = " + allowed\_categories +  
 " working\_hours = " + working\_hours +  
 " }";  
 }  
}

**Приклад запитів за допомогою ORM:**

* Model model = bus.get(Bus.class, Integer.*parseInt*(bus\_id));
* bus.delete(route);
* bus.save(driver);
* bus.update(model);

**HASH**

Для дослідження індексу була створена таблиця, яка має дві колонки:

числову і текстову. Вони проіндексовані за допомогою HASH. У таблицю занесено 1000000 записів.

**Створення таблиці та її заповнення:**

DROP TABLE IF EXISTS "test\_hash";

CREATE TABLE "test\_hash"(

"id" bigserial PRIMARY KEY,

"test\_text" varchar(255)

);

INSERT INTO "test\_hash"("test\_text")

SELECT

substr(characters, (random() \* length(characters) + 1)::integer, 10)

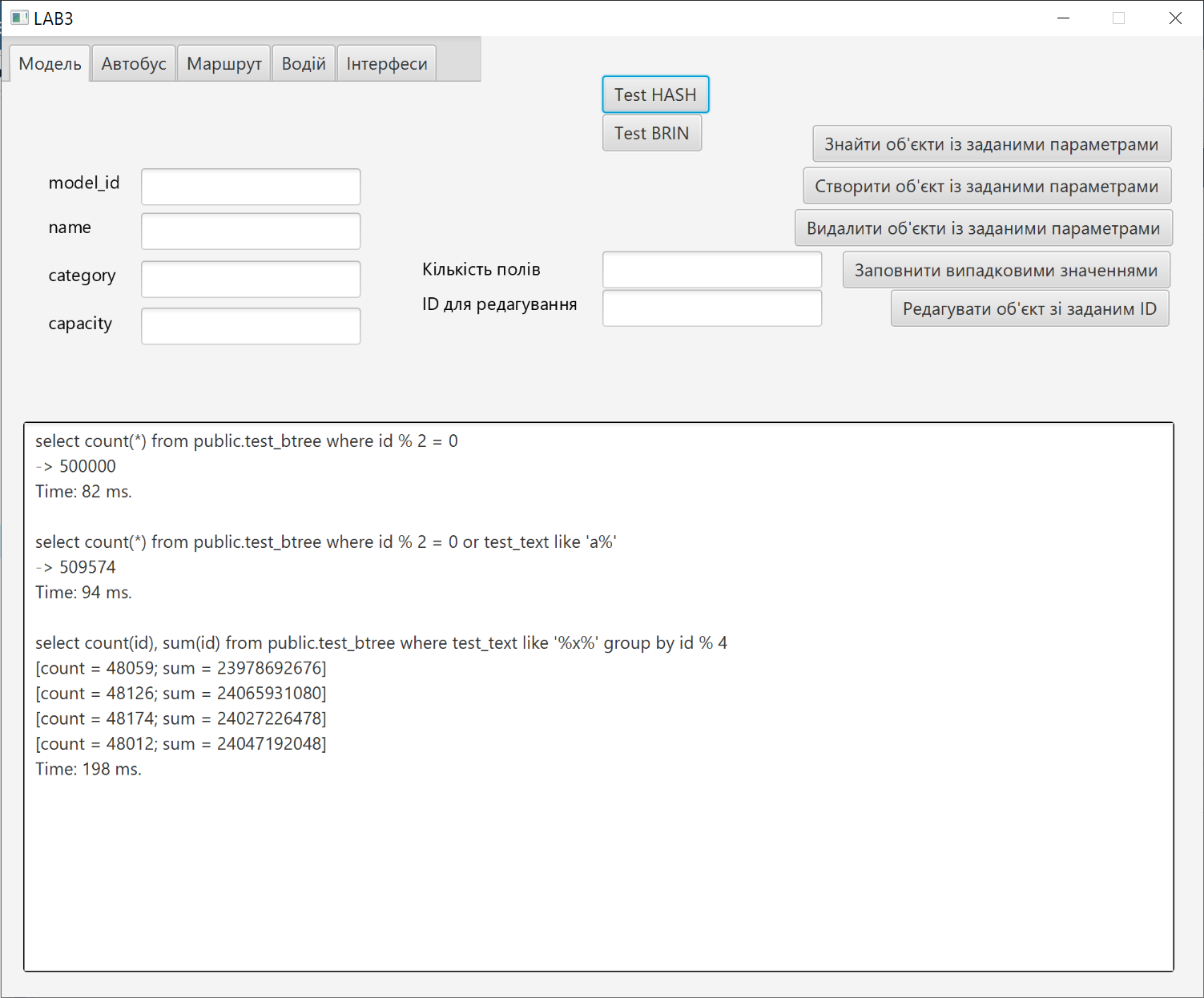
FROM

(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM'))

as symbols(characters),

generate\_series(1, 1000000) as q;

**Вибір даних без індексу**

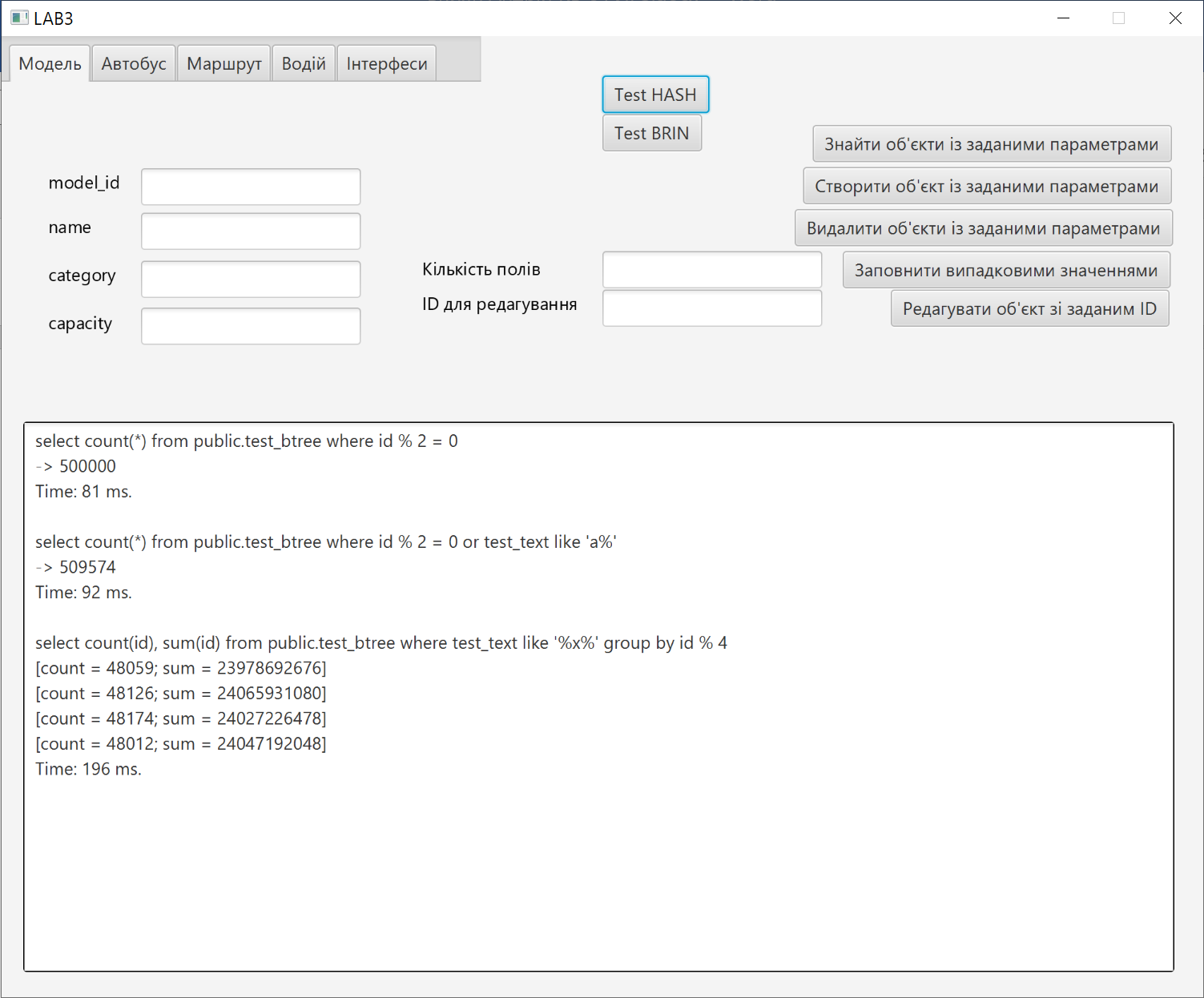
****

**Створюємо індекс**

DROP INDEX IF EXISTS "test\_hash\_test\_text\_index";

CREATE INDEX "test\_hash\_test\_text\_index" ON "test\_hash" USING hash ("test\_text");

**Вибір даних з створеним індексом**

****

Отже, не було отримано помітного пришвидшення пошуку.

**BRIN**

Для дослідження індексу була створена таблиця, яка має дві колонки:

числову і текстову. Вони проіндексовані за допомогою BRIN. У таблицю занесено 1000000 записів.

**Створення таблиці та її заповнення**

DROP TABLE IF EXISTS "test\_brin";

CREATE TABLE "test\_brin"(

"id" bigserial PRIMARY KEY,

"test\_text" varchar(255)

);

INSERT INTO "test\_brin"("test\_text")

SELECT

substr(characters, (random() \* length(characters) + 1)::integer, 10)

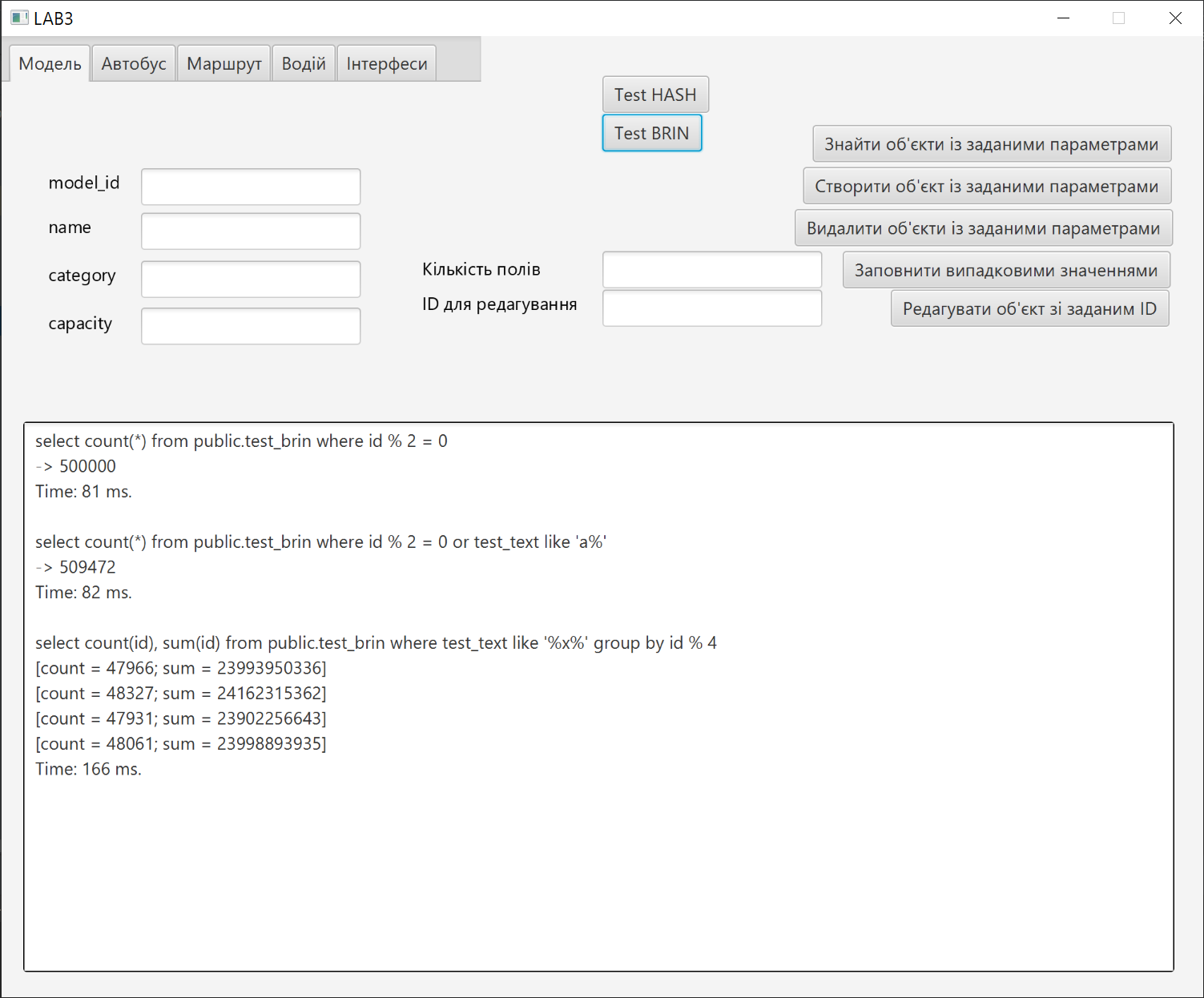
FROM

(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM'))

as symbols(characters),

generate\_series(1, 1000000) as q;

**Вибір даних без індексу**

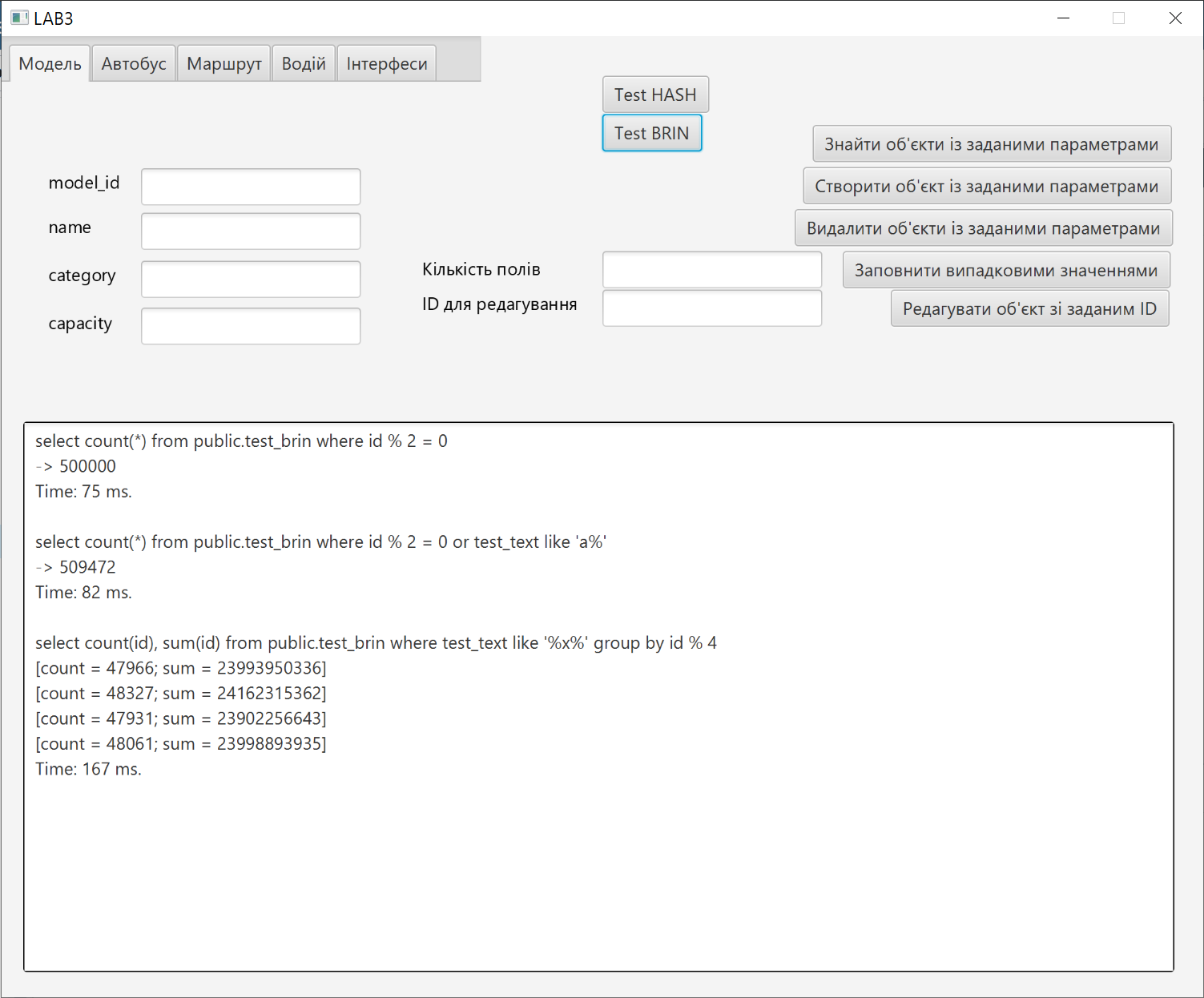
****

**Створюємо індекс**

DROP INDEX IF EXISTS "test\_brin\_test\_text\_index";

CREATE INDEX "test\_brin\_test\_text\_index" ON "test\_brin" USING brin ("test\_text");

**Вибір даних з створеним індексом**

****

Отже, не було отримано помітного пришвидшення пошуку.

**Тригер бази даних PostgreSQL. Умова для тригера – after update, insert**

**Тригер:**

CREATE TRIGGER insertModel

AFTER INSERT

ON model

FOR EACH ROW

EXECUTE PROCEDURE insertIntoModel();

**Функція після спрацювання тригера:**

CREATE OR REPLACE FUNCTION insertIntoModel()

RETURNS TRIGGER

LANGUAGE PLPGSQL

AS

$$

BEGIN

UPDATE bus

SET model\_id = NEW.model\_id

WHERE bus\_id = 1;

RETURN NEW;

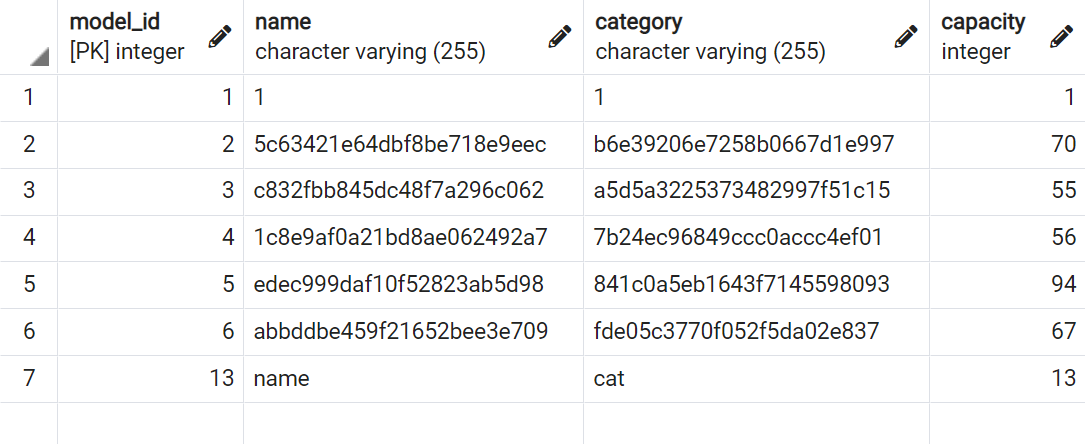
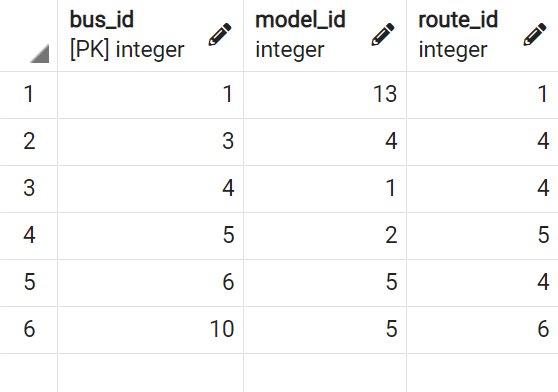
END;

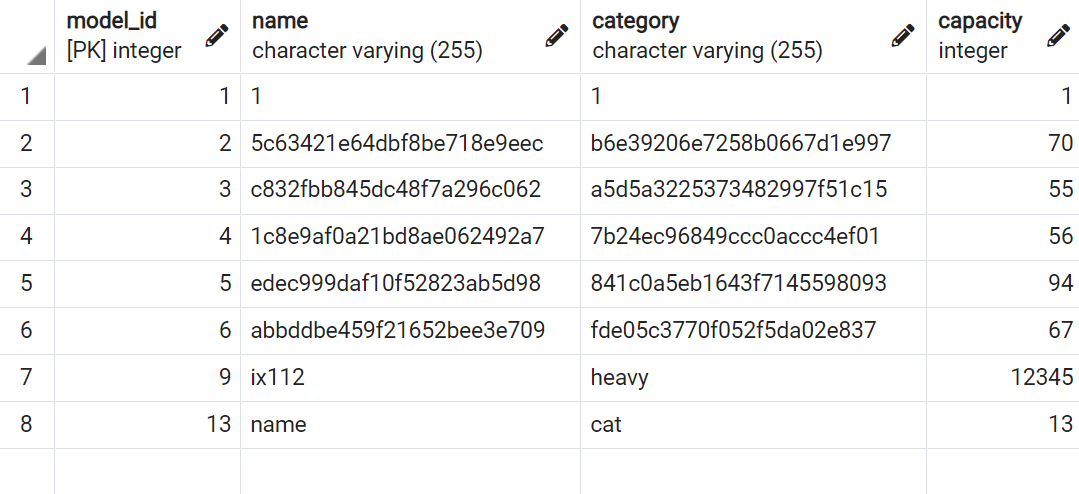
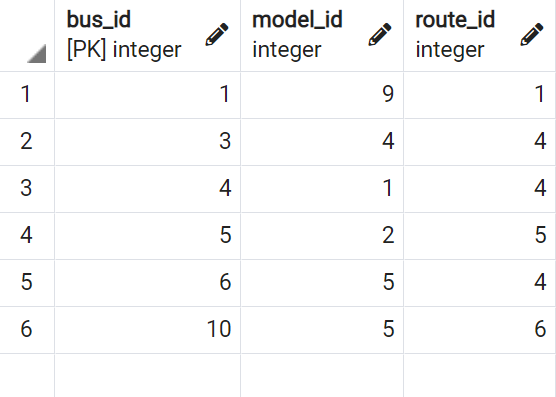
$$

**Принцип роботи**

Після додавання нового рядка в таблицю model, спрацьовує тригер insertModel, який викликає функцію insertIntoModel() – і ця функція модифікує інформацію в першому рядку таблиці bus в полі model\_id на model\_id з щойно доданого рядка в таблицю model.

***Приклад***

Таблиці до змін:

Додали запис в таблицю model:

**Рівні ізоляції транзакцій у PostgreSQL**

Транзакція — це N (N≥1) запитів до БД, які успішно виконуються всі разом або зовсім не виконуються. Ізольованість транзакції показує те, наскільки сильно вони впливають одне на одного паралельно виконуються транзакції.

Вибираючи рівень транзакції, ми намагаємося дійти консенсусу у

виборі між високою узгодженістю даних між транзакціями та швидкістю

виконання цих транзакцій.

Варто зазначити, що найвищу швидкість виконання та найнижчу

узгодженість має рівень read uncommitted. Найнижчу швидкість виконання та

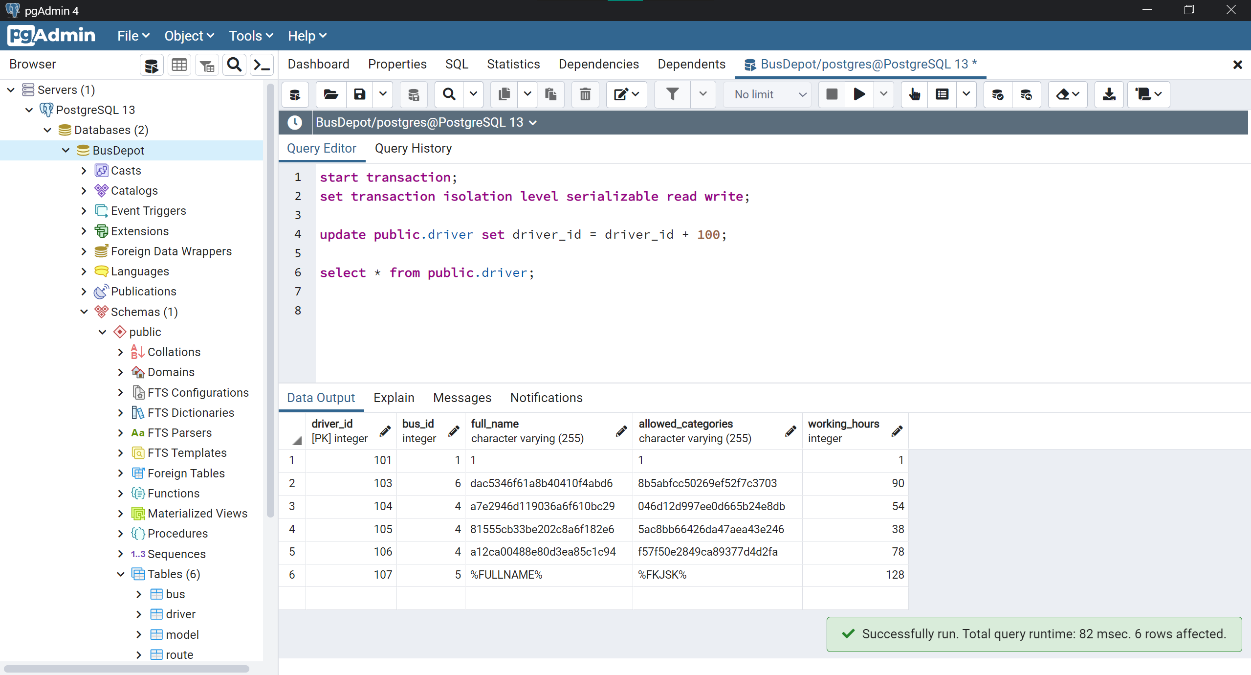
найвищу узгодженість — serializable.

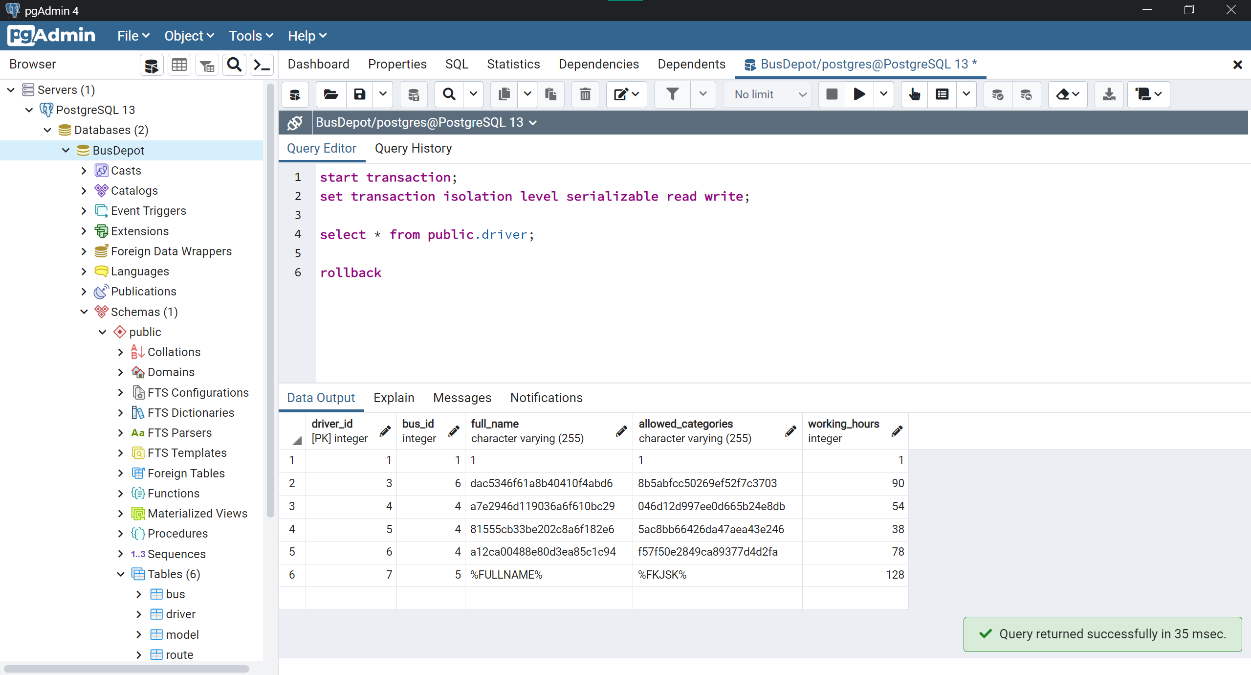
При паралельному виконанні транзакцій можливі виникненя таких проблем:

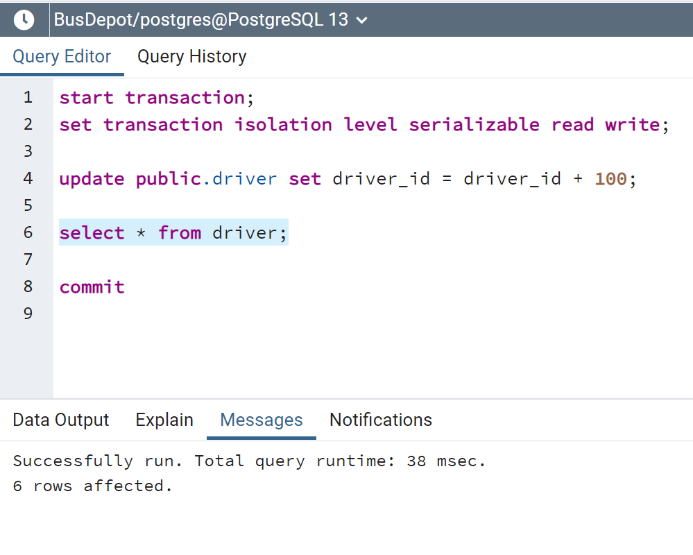
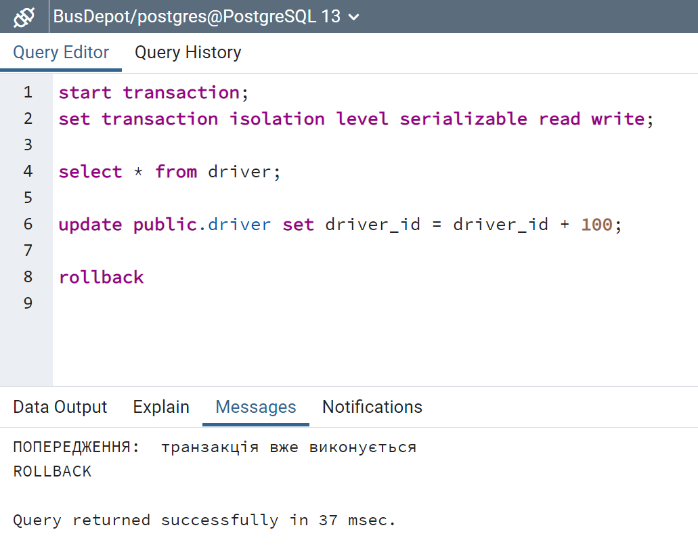
1. Втрачене оновлення - ситуація, коли при одночасній зміні одного блоку даних різними транзакціями, одна зі змін втрачається.
2. «Брудне» читання - читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (відкотиться).
3. Неповторюване читання - ситуація, коли при повторному читанні в рамках однієї транзакції, раніше прочитані дані виявляються зміненими.
4. Фантомне читання - ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків.

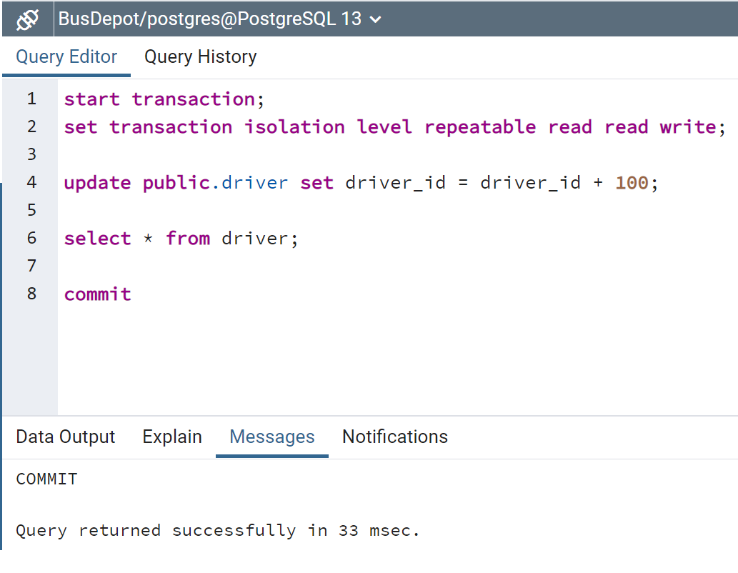
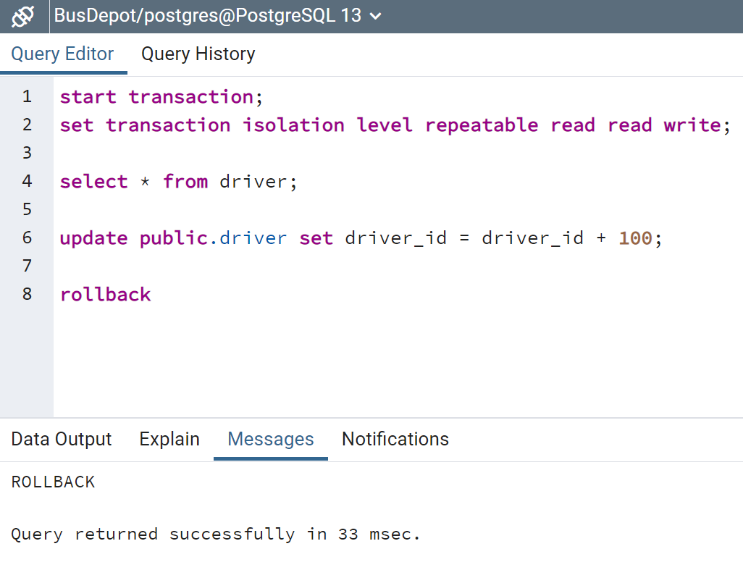
**Стандарт SQL-92**

1. **Serializable (впорядкованість)**

****

****

Як бачимо, дані у транзакціях ізольовано.

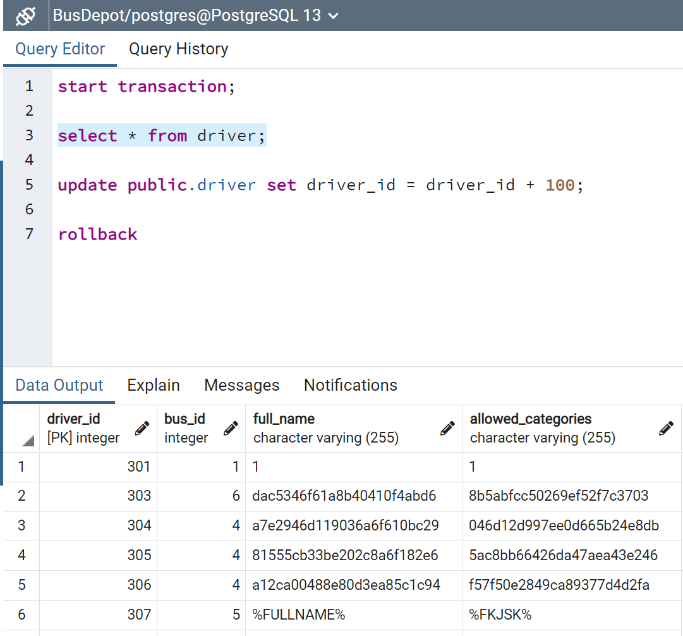
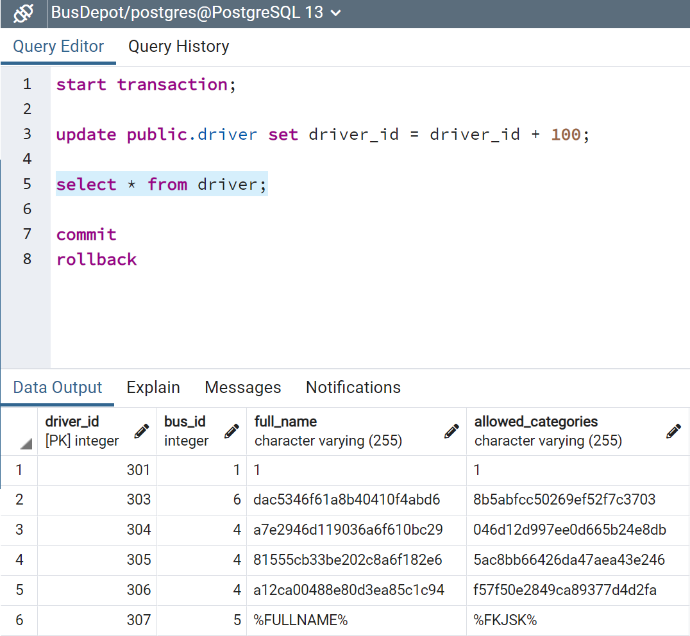
1. **Repeatable read (повторюваність читання)**

Тепер транзакція T2(cправа) буде чекати поки T1 не зафіксує зміни

або не відмінить їх.

Як бачимо, Repeatable Read не дозволяє виконувати операції зміни даних, якщо дані вже було модифіковано у іншій незавершеній транзакції. Тому використання Repeatable Read рекомендоване тільки для режиму читання.

1. **Read committed (читання фіксованих даних)**

Прийнятий за замовчуванням рівень для PostgreSQL. Закінчене читання, при якому відсутнє «брудне» читання (тобто, читання одним користувачем даних, що не були зафіксовані в БД командою COMMIT). Проте, в процесі роботи однієї транзакції інша може бути успішно закінчена, і зроблені нею зміни зафіксовані. В підсумку, перша транзакція буде працювати з іншим набором даних. Це проблема неповторюваного читання

1. **Read uncommitted (читання незафіксованих даних)**

Найнижчий рівень ізоляції, який відповідає рівню 0. Він гарантує тільки відсутність втрачених оновлень. Якщо декілька транзакцій одночасно намагались змінювати один і той же рядок, то в кінцевому варіанті рядок буде мати значення, визначений останньою успішно виконаною транзакцією. У PostgreSQL READ UNCOMMITTED розглядається як READ COMMITTED.