НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування та спеціалізованих комп’ютерних**

**систем**

**Лабораторна робота №2**

з дисципліни

**«Бази даних і засоби управління»**

Тема: «Засоби оптимізації роботи СУБД PostgreSQL»

Виконав: студент--ка III курсу

ФПМ групи КВ-11

Шевчук Я.О.

Перевірив:

Київ – 2023

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

*Завдання* роботи полягає у наступному:

1. Перетворити модуль “Модель” з шаблону MVC РГР 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.

*Варіант 26*

*У другому завданні проаналізувати індекси BTree, BRIN.*

*Умова для тригера – before update, delete.*

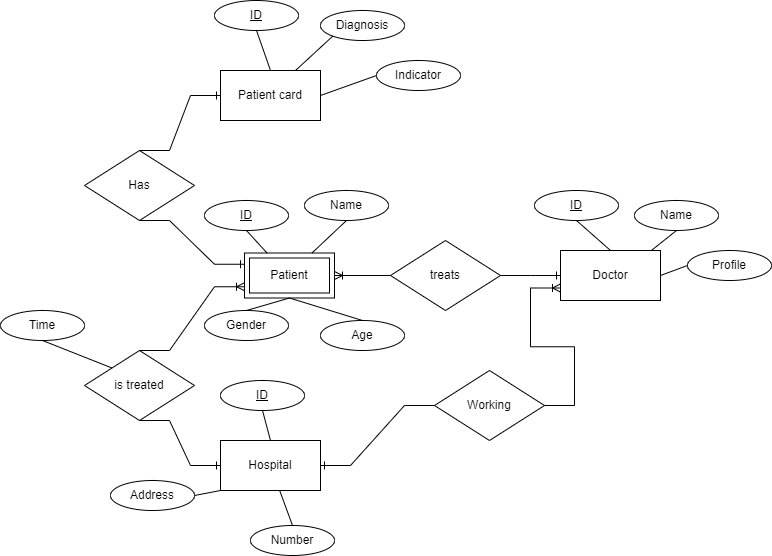
Telegram: @minnie\_mouse\_19

GitHub: https://github.com/YaroslavaKV-11/database/upload/Lab2

**Завдання 1**

**Інформація про базу даних**

Розробка моделі «сутність-зв’язок» предметної галузі для проектування бази даних «Patient health monitoring system». Предметна галузь - «Система відстеження стану здоров’я пацієнтів».



Малюнок 1. ER-діаграма побудована за нотацією «Crow`s foot»

**Сутності з описом призначення:**

Предметна галузь «Patient health monitoring system» включає в себе 4 сутності, кожна сутність містить декілька атрибутів:

1. Patient (Id, name, gender, age).
2. Patient card (Id, diagnosis, indicator).
3. Hospital (Id, number, address).
4. Doctor (id, name, profile).

Сутність Patient описує пацієнта, який відвідує лікарню. Кожен пацієнт має свій ідентифікатор Id, а також має ім’я, стать та вік.

Сутність Patient card описує стан здоров’я пацієнта. Кожна карта пацієнта має свій ідентифікатор, діагноз пацієнта та показник про його здоров’я(у відсотках).

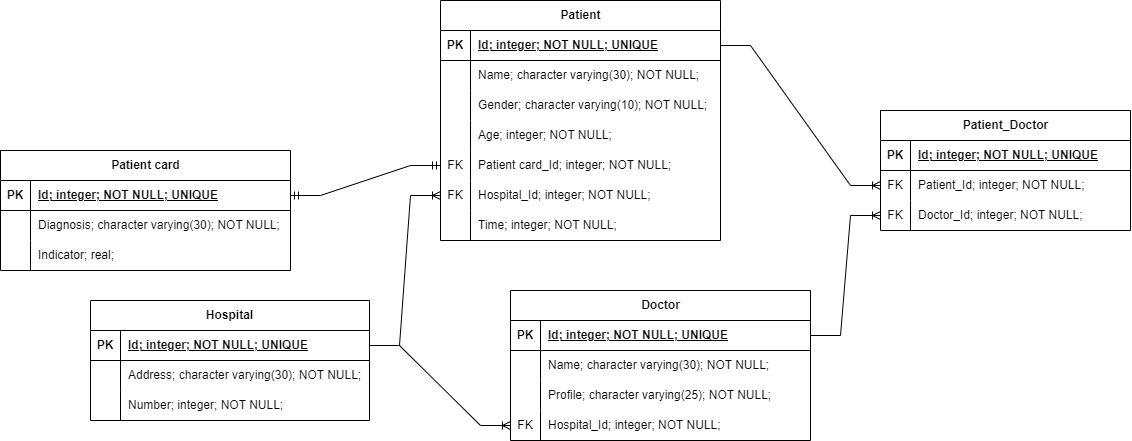
Сутність Hospital описує лікарню, яку відвідує пацієнт. Кожна лікарня має свій ідентифікатор, адресу і кількість пацієнтів, які її відвідують.

Сутність Doctor описує лікаря до якого ходить пацієнт. Кожен лікар має свій ідентифікатор, ім’я та спеціалізацію(профіль).

**Зв’язки між сутностями:**

Зв’язок між Patient та Patient card:  
Кожен пацієнт має свою картку пацієнта, де міститься інформація про його стан здоров’я. Оскільки, пацієнт має тільки одну картку і одна карта відповідає лише одному пацієнту, то зв’язок 1:1.  
Зв’язок між Patient та Hospital:  
Кожен пацієнт для лікування повинен відвідувати лікарню до якої приписаний. Оскільки, в одній лікарні лікується багато пацієнтів, а один пацієнт відвідує одну лікарню, то зв’язок 1:N.  
Зв’язок між Patient та Doctor:  
Кожен пацієнт лікується у свого сімейного лікаря. Оскільки в сімейного лікаря своїх пацієнтів багато, але в одного пацієнта лише один сімейний лікар, то зв’язок 1:N.  
Зв’язок між Doctor та Hospital:  
Кожен лікар працює у лікарні. Оскільки, в лікарні багато лікарів, але один лікар працює лише в одній лікарні, то зв’язок 1:N.

**Схема бази даних PostgreSQL**



Малюнок 2. Схема бази даних у графічному вигляді

**Функціональні залежності:**

**Patient (*Id*, name, gender, age).**

*Id* → name

*Id* → gender

*Id* → age

**Patient card (*Id*, diagnosis, indicator).**

*Id* → diagnosis  
*Id* → indicator

**Hospital (*Id*, number, address).**

*Id* → number

*Id* → address

**Doctor (*Id*, name, profile).**

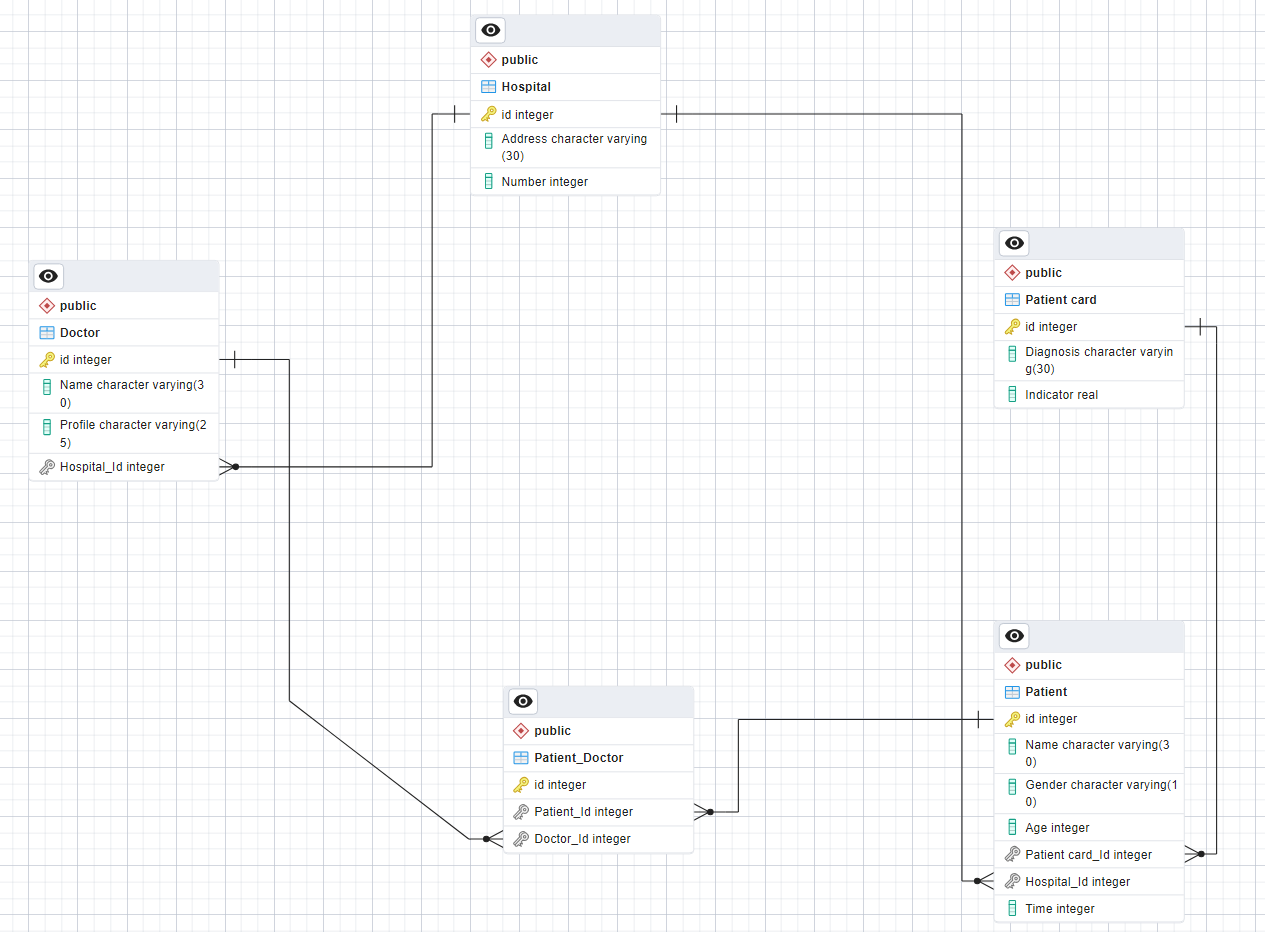
*Id* → name

*Id* → profile

Схема бази даних відповідає 1НФ, тому що значення в кожній комірці таблиці є атомарними, кожне поле таблиці є неподільним, кожен рядок є унікальним, немає повторень рядків.

Схема бази даних відповідає 2НФ, бо вона відповідає 1НФ та кожен неключовий атрибут залежить від первинного повного ключа, отже первинний ключ одразу визначає запис та не є надмірним.

Схема бази даних відповідає 3НФ, тому що вона відповідає 2НФ та кожен неключовий атрибут не є транзитивно залежним від кожного кандидатного ключа. В таблицях нема не ключового поля, яке залежить від значення іншого не ключового поля.



Малюнок 3. Схема бази даних у pgAdmin4

Для перетворення модулів моделей програми, створених в розрахунковій роботі, у вигляд об’єктно-реляційної моделі було використано Entity Framework Core.

public class TDoctor

{

public int Id { get; set; }

public string Name { get; set; }

public string Profile { get; set; }

public int Hospital\_Id { get; set; }

}

public class THospital

{

public int Id { get; set; }

public string Address { get; set; }

public int Number { get; set; }

}

public class TPatient

{

public int Id { get; set; }

public string Name { get; set; }

public string Gender { get; set; }

public int Age { get; set; }

public int Card\_Id { get; set; }

public int Hospital\_Id { get; set; }

public int Time { get; set; }

}

public class TPatientCard

{

public int Id { get; set; }

public string Diagnosis { get; set; }

public double Indication { get; set; }

}

public class TPatientDoctor

{

public int Id { get; set; }

public int Patient\_Id { get; set; }

public int Doctor\_Id { get; set; }

}

public class HospitalContext : DbContext

{

public DbSet<TDoctor> Doctor { get; set; }

public DbSet<THospital> Hospital { get; set; }

public DbSet<TPatient> Patient { get; set; }

public DbSet<TPatientCard> Patient\_Card { get; set; }

public DbSet<TPatientDoctor> Patient\_Doctor { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseNpgsql(

"Host=localhost;Port=5432;Database=Hospital;Username=postgres;Password=011427223");

}

}

class ModelClass

{

HospitalContext context { get; set; } //Об'єкт класу HospitalContext, DbContext (для доступу до бази даних)

public ModelClass()

{

context = new HospitalContext();

}

static string GenerateRandomString(int length)

{

const string chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

Random random = new Random();

char[] randomArray = new char[length];

for (int i = 0; i < length; i++)

{

randomArray[i] = chars[random.Next(chars.Length)];

}

string randomString = new string(randomArray);

return randomString;

}

/////// Методи для таблиці Doctor

public List<TDoctor> AllDoctors()

{

List<TDoctor> docs = context.Doctor.ToList();

return docs;

}// Повертає всі записи таблиці Doctor

public int InputDoctor(TDoctor doc)

{

try

{

var docs = context.Doctor.ToList();

List<int> ids = new List<int>();

List<int> hosps = new List<int>();

foreach (TDoctor d in docs)

{

ids.Add(d.Id);

}

int Id = ids.Max(id => id) + 1;

doc.Id = Id;

context.Doctor.Add(doc);

context.SaveChanges();

return 1;

}

catch

{

return 0;

}

}// Метод вставки нового рядку в таблицю Doctor

public int DeleteDoctor(int id)

{

try

{

TDoctor doctor = context.Doctor.Find(id);

if (doctor == null)

{

return 0;

}

context.Doctor.Remove(doctor);

context.SaveChanges();

return 1;

}

catch (Exception ex)

{

return 0;

}

}//Метод видалення рядка з таблиці Doctor

public int UpdateDoctor(TDoctor doc, int id\_)

{

var updatedDoctor = context.Doctor.Find(id\_);

try

{

updatedDoctor.Name = doc.Name;

updatedDoctor.Profile = doc.Profile;

updatedDoctor.Hospital\_Id = doc.Hospital\_Id;

return 1;

}

catch (Exception ex)

{

return 0;

}

}// Метод модифікації даних запису таблиці Doctor

public List<TDoctor> SearchDoctorName(string nm)

{

List<TDoctor> docs = context.Doctor

.Where(e => e.Name.Contains(nm))

.ToList();

return docs;

}// Метод пошуку за іменем доктора

public List<TDoctor> SearchDoctorProfile(string pf)

{

List<TDoctor> docs = context.Doctor

.Where(e => e.Profile.Contains(pf))

.ToList();

return docs;

}// Метод пошуку за профілем або спеціальністю доктора

public List<TDoctor> SearchDoctorHospital(int hp)

{

List<TDoctor> docs = context.Doctor

.Where(e => e.Hospital\_Id == hp)

.ToList();

return docs;

}// Метод пошуку за номером лікарні, в якій працює доктор

public void GenerateDoctors(int n)

{

for (int i = 0; i < n; i++)

{

var docs = context.Doctor.ToList();

List<int> ids = new List<int>();

List<int> hosps = new List<int>();

foreach (TDoctor d in docs)

{

ids.Add(d.Id);

}

int Id = ids.Max(id => id) + 1;

var hospitals = context.Hospital.ToList();

foreach (THospital h in hospitals)

{

hosps.Add(h.Id);

}

Random rnd = new Random();

int hosp\_id = hosps[rnd.Next(0, hosps.Count)];

TDoctor doctor = new TDoctor();

doctor.Id = Id;

doctor.Hospital\_Id = hosp\_id;

doctor.Name = GenerateRandomString(10);

doctor.Profile = GenerateRandomString(10);

context.Doctor.Add(doctor);

context.SaveChanges();

}

} // Метод генерації нових даних в таблицю Doctor

/////// Методи для таблиці Hospital

public List<THospital> AllHospitals()

{

var hsp = context.Hospital.ToList();

return hsp.ToList();

}//Повертає всі записи таблиці Hospital

public int InputHospital(THospital hsp)

{

var hospitals = context.Hospital.ToList();

List<int> ids = new List<int>();

foreach (THospital h in hospitals)

{

ids.Add(h.Id);

if (hsp.Address == h.Address)

return 0;

}

hsp.Id = ids.Max(id => id) + 1;

context.Hospital.Add(hsp);

context.SaveChanges();

return 1;

}// Метод вставки нового рядку в таблицю Hospital

public int DeleteHospital(int id)

{

THospital hospital = context.Hospital.Find(id);

if (hospital != null)

{

try

{

context.Hospital.Remove(hospital);

context.SaveChanges();

return 1;

}

catch

{

return 0;

}

}

else return 0;

}//Метод видалення рядка з таблиці Hospital

public int UpdateHospital(THospital hsp, int id\_)

{

THospital hospital = context.Hospital.Find(id\_);

if (hospital != null)

{

hospital.Address = hsp.Address;

hospital.Number = hsp.Number;

context.SaveChangesAsync();

context = new HospitalContext();

return 1;

}

else return 0;

}// Метод модифікації даних запису таблиці Hospital

public List<THospital> SearchAddress(string ad)

{

List<THospital> hospitals = context.Hospital

.Where(e => e.Address.Contains(ad))

.ToList();

return hospitals;

}// Метод пошуку за адресою лікарні

public List<THospital> SearchNumber(int min, int max)

{

List<THospital> hospitals = context.Hospital

.Where(e => e.Number >= min && e.Number <= max)

.ToList();

return hospitals;

}// Метод пошуку за кількістю місць в лікарні

public void GenerateHospitals(int n)

{

for (int i = 0; i < n; i++)

{

var hospitals = context.Hospital.ToList();

List<int> ids = new List<int>();

foreach (THospital h in hospitals)

{

ids.Add(h.Id);

}

int Id = ids.Max(id => id) + 1;

Random rnd = new Random();

THospital hospital = new THospital();

hospital.Id = Id;

hospital.Address = GenerateRandomString(10);

hospital.Number = rnd.Next(30, 1000);

context.Hospital.Add(hospital);

context.SaveChanges();

}

}// Метод генерації нових даних в таблицю Hospital

/////// Методи для таблиці Patient

public List<TPatient> AllPatients()

{

var pts = context.Patient.ToList();

return pts.ToList();

}//Повертає всі записи таблиці Patient

public int InputPatient(TPatient pat)

{

int cid = -1;

int hid = -1;

var pats = context.Patient\_Card.ToList();

foreach (TPatientCard tp in pats)

{

if (tp.Id == pat.Card\_Id)

cid = tp.Id;

}

var hosps = context.Hospital.ToList();

foreach (THospital h in hosps)

{

if (h.Id == pat.Hospital\_Id)

hid = h.Id;

}

if (cid == -1)

return -1;

if (hid == -1)

return 0;

else

{

var pts = context.Patient.ToList();

List<int> ids = new List<int>();

foreach (TPatient p in pts)

{

ids.Add(p.Id);

}

pat.Id = ids.Max(id => id) + 1;

context.Patient.Add(pat);

context.SaveChanges();

return 1;

}

}// Метод вставки нового рядку в таблицю Patient

public int DeletePatient(int id)

{

TPatient patient = context.Patient.Find(id);

if (patient == null)

return 0;

context.Patient.Remove(patient);

context.SaveChanges();

return 1;

}//Метод видалення рядка з таблиці Patient

public int UpdatePatient(int id, TPatient pat)

{

int cid = -1;

int hid = -1;

var pats = context.Patient\_Card.ToList();

foreach (TPatientCard tp in pats)

{

if (tp.Id == pat.Card\_Id)

cid = tp.Id;

}

var hosps = context.Hospital.ToList();

foreach (THospital h in hosps)

{

if (h.Id == pat.Hospital\_Id)

hid = h.Id;

}

if (cid == -1)

return -1;

if (hid == -1)

return 0;

else

{

var hospitals = context.Patient.ToList();

List<int> ids = new List<int>();

foreach (TPatient p in hospitals)

{

ids.Add(p.Id);

}

if (ids.Contains(id))

{

try

{

TPatient patient = context.Patient.Find(id);

patient.Name = pat.Name;

patient.Gender = pat.Gender;

patient.Age = pat.Age;

patient.Card\_Id = pat.Card\_Id;

patient.Hospital\_Id = pat.Hospital\_Id;

patient.Time = pat.Time;

context.SaveChangesAsync();

context = new HospitalContext();

return 1;

}

catch

{

return -3;

}

}

else return -2;

}

}// Метод модифікації даних запису таблиці Patient

public List<TPatient> SearchPatientName (string name)

{

List<TPatient> patients = context.Patient

.Where(e => e.Name.Contains(name))

.ToList();

return patients;

}// Метод пошуку за іменем пацієнта

public List<TPatient> SearchPatientGender(string gender)

{

List<TPatient> patients = context.Patient

.Where(e => e.Gender.Contains(gender))

.ToList();

return patients;

}// Метод пошуку за статтю пацієнта

public List<TPatient> SearchAge(int min, int max)

{

List<TPatient> patients = context.Patient

.Where(e => e.Age >= min && e.Age <= max)

.ToList();

return patients;

}// Метод пошуку за віком пацієнта

public List<TPatient> SearchPatientCard(int id)

{

List<TPatient> patients = context.Patient

.Where(e => e.Card\_Id == id)

.ToList();

return patients;

}// Метод пошуку за номером картки

public List<TPatient> SearchPatientHospital(int id)

{

List<TPatient> patients = context.Patient

.Where(e => e.Hospital\_Id == id)

.ToList();

return patients;

}// Метод пошуку за номером лікарні пацієнта

public List<TPatient> SearchPatientTime(int min, int max)

{

List<TPatient> patients = context.Patient

.Where(e => e.Time >= min && e.Age <= max)

.ToList();

return patients;

}// Метод пошуку за кількістю годин, проведених у лікарні

public void GeneratePatients(int n)

{

for (int i = 0; i < n; i++)

{

var pats = context.Patient.ToList();

List<int> ids = new List<int>();

foreach (TPatient tp in pats)

{

ids.Add(tp.Id);

}

int Id = ids.Max(id => id) + 1;

List<int> cards = new List<int>();

List<int> hosps = new List<int>();

var cds = context.Patient\_Card.ToList();

foreach (TPatientCard tc in cds)

{

cards.Add(tc.Id);

}

var hsp = context.Hospital.ToList();

foreach (THospital tc in hsp)

{

hosps.Add(tc.Id);

}

Random rnd = new Random();

int card\_id = cards[rnd.Next(0, cards.Count)];

int hosp\_id = hosps[rnd.Next(0, hosps.Count)];

List<string> genders = new List<string> { "Male", "Female", "Other" };

string gender = genders[rnd.Next(0, genders.Count)];

TPatient patient = new TPatient();

patient.Id = Id;

patient.Name = GenerateRandomString(10);

patient.Gender = gender;

patient.Age = rnd.Next(0, 111);

patient.Card\_Id = card\_id;

patient.Hospital\_Id = hosp\_id;

patient.Time = rnd.Next(0, 200);

context.Patient.Add(patient);

context.SaveChanges();

}

} // Метод генерації нових даних в таблицю Patient

/////// Методи для таблиці PatientCard

public List<TPatientCard> AllPatientCards()

{

List<TPatientCard> ptntcd = context.Patient\_Card.ToList();

return ptntcd.ToList();

}//Повертає всі записи таблиці PatientCard

public int InputPatientCard(TPatientCard card)

{

var cards = context.Patient\_Card.ToList();

List<int> ids = new List<int>();

foreach (TPatientCard h in cards)

{

ids.Add(h.Id);

if (card.Diagnosis == h.Diagnosis)

return 0;

}

card.Id = ids.Max(id => id) + 1;

context.Add(card);

context.SaveChanges();

return 1;

}// Метод вставки нового рядку в таблицю PatientCard

public int DeletePatientCard(int id)

{

var cards = context.Patient\_Card.ToList();

List<int> ids = new List<int>();

foreach (TPatientCard c in cards)

{

ids.Add(c.Id);

}

if (ids.Contains(id))

{

try

{

TPatientCard pc = context.Patient\_Card.Find(id);

context.Patient\_Card.Remove(pc);

context.SaveChanges();

return 1;

}

catch

{

return 0;

}

}

else return 0;

}//Метод видалення рядка з таблиці PatientCard

public int UpdatePatientCard(TPatientCard hsp, int id\_)

{

var cards = context.Patient\_Card.ToList();

List<int> ids = new List<int>();

foreach (TPatientCard c in cards)

{

ids.Add(c.Id);

}

if (ids.Contains(id\_))

{

TPatientCard patientCard = context.Patient\_Card.Find(id\_);

patientCard.Diagnosis = hsp.Diagnosis;

patientCard.Indication = hsp.Indication;

context.SaveChangesAsync();

context = new HospitalContext();

return 1;

}

else return 0;

}// Метод модифікації даних запису таблиці PatientCard

public List<TPatientCard> SearchDiagnosis(string ad)

{

List<TPatientCard> patients = context.Patient\_Card

.Where(e => e.Diagnosis.Contains(ad))

.ToList();

return patients;

}// Метод пошуку за діагнозом

public List<TPatientCard> SearchIndication(int min, int max)

{

List<TPatientCard> patients = context.Patient\_Card

.Where(e => e.Indication >= min && e.Indication <= max)

.ToList();

return patients;

}// Метод пошуку за індікатором

public void GeneratePatientCard(int n)

{

for (int i = 0; i < n; i++)

{

var hospitals = context.Patient\_Card.ToList();

List<int> ids = new List<int>();

foreach (TPatientCard h in hospitals)

{

ids.Add(h.Id);

}

int Id = ids.Max(id => id) + 1;

Random rnd = new Random();

TPatientCard patientCard = new TPatientCard();

patientCard.Id = Id;

patientCard.Diagnosis = GenerateRandomString(15);

patientCard.Indication = Convert.ToDouble(rnd.Next(0, 100));

context.Patient\_Card.Add(patientCard);

context.SaveChanges();

}

}// Метод генерації нових даних в таблицю PatientCard

/////// Методи для таблиці PatientDoctor

public List<TPatientDoctor> AllPatientDoctors()

{

var pd = context.Patient\_Doctor.ToList();

return pd.ToList();

}//Повертає всі записи таблиці PatientDoctor

public int InputPatientDoctor(TPatientDoctor pd)

{

int patid = -1;

int docid = -1;

List<int> ids = new List<int>();

var pt = context.Patient.ToList();

foreach (TPatient p in pt)

{

if (p.Id == pd.Patient\_Id)

patid = p.Id;

}

var dc = context.Doctor.ToList();

foreach (TDoctor d in dc)

{

if (d.Id == pd.Doctor\_Id)

docid = d.Id;

}

if (patid == -1)

return -1;

if (docid == -1)

return 0;

var pds = context.Patient\_Doctor.ToList();

foreach (TPatientDoctor tpd in pds)

{

ids.Add(tpd.Id);

}

pd.Id = ids.Max(id => id) + 1;

context.Patient\_Doctor.Add(pd);

context.SaveChanges();

return 1;

}// Метод вставки нового рядку в таблицю PatientDoctor

public int DeletePatientDoctor(int id)

{

var pds = context.Patient\_Doctor.ToList();

List<int> ids = new List<int>();

foreach (TPatientDoctor pd in pds)

{

ids.Add(pd.Id);

}

if (ids.Contains(id))

{

try

{

TPatientDoctor? patientDoctor = context.Patient\_Doctor.Find(id);

context.Patient\_Doctor.Remove(patientDoctor);

context.SaveChanges();

return 1;

}

catch

{

return 0;

}

}

else return 0;

}//Метод видалення рядка з таблиці PatientDoctor

public int UpdatePatientDoctor(int id, TPatientDoctor pd)

{

var pds = context.Patient\_Doctor.ToList();

List<int> ids = new List<int>();

foreach (TPatientDoctor pd\_ in pds)

{

ids.Add(pd\_.Id);

}

if (ids.Contains(id))

{

int patid = -1;

int docid = -1;

var pt = context.Patient.ToList();

foreach (TPatient p in pt)

{

if (p.Id == pd.Patient\_Id)

patid = p.Id;

}

var dc = context.Doctor.ToList();

foreach (TDoctor d in dc)

{

if (d.Id == pd.Doctor\_Id)

docid = d.Id;

}

if (patid == -1)

return -2;

if (docid == -1)

return -1;

TPatientDoctor? patientDoctor = context.Patient\_Doctor.Find(id);

patientDoctor.Doctor\_Id = pd.Doctor\_Id;

patientDoctor.Patient\_Id = pd.Patient\_Id;

context.SaveChangesAsync();

context = new HospitalContext();

return 1;

}

else return 0;

}// Метод модифікації даних запису таблиці PatientDoctor

public List<TPatientDoctor> SearchPDPatient(string name)

{

List<TPatient> patients = context.Patient.ToList();

List<int> ids = new List<int>();

foreach (TPatient p in patients)

{

if (p.Name.Contains(name))

{

ids.Add(p.Id);

}

}

List<TPatientDoctor> patientDoctor = context.Patient\_Doctor

.Where(e => ids.Contains(e.Patient\_Id))

.ToList();

return patientDoctor;

}// Метод пошуку за пацієнтом

public List<TPatientDoctor> SearchPDDoctor(string name)

{

List<TDoctor> doctors = context.Doctor.ToList();

List<int> ids = new List<int>();

foreach (TDoctor p in doctors)

{

if (p.Name.Contains(name))

{

ids.Add(p.Id);

}

}

List<TPatientDoctor> patientDoctor = context.Patient\_Doctor

.Where(e => ids.Contains(e.Doctor\_Id))

.ToList();

return patientDoctor;

}// Метод пошуку за лікарем

public void GeneratePatientDoctor(int n)

{

for (int i = 0; i < n; i++)

{

List<int> pats = new List<int>();

List<int> docs = new List<int>();

List<int> ids = new List<int>();

var pt = context.Patient.ToList();

foreach (TPatient p in pt)

{

pats.Add(p.Id);

}

var dc = context.Doctor.ToList();

foreach (TDoctor d in dc)

{

docs.Add(d.Id);

}

Random rnd = new Random();

int pid = pats[rnd.Next(0, pats.Count())];

int did = docs[rnd.Next(0, docs.Count())];

var pds = context.Patient\_Doctor.ToList();

foreach (TPatientDoctor tpd in pds)

{

ids.Add(tpd.Id);

}

int id = ids.Max(id => id) + 1;

TPatientDoctor patientDoctor = new TPatientDoctor();

patientDoctor.Id = id;

patientDoctor.Doctor\_Id = did;

patientDoctor.Patient\_Id = pid;

context.Patient\_Doctor.Add(patientDoctor);

context.SaveChanges(); }

}// Метод генерації нових даних в таблицю PatientDoctor

}

**Завдання 2**

Створення таблиці:

create table text1(

body text

);

Вставка та генерація даних:

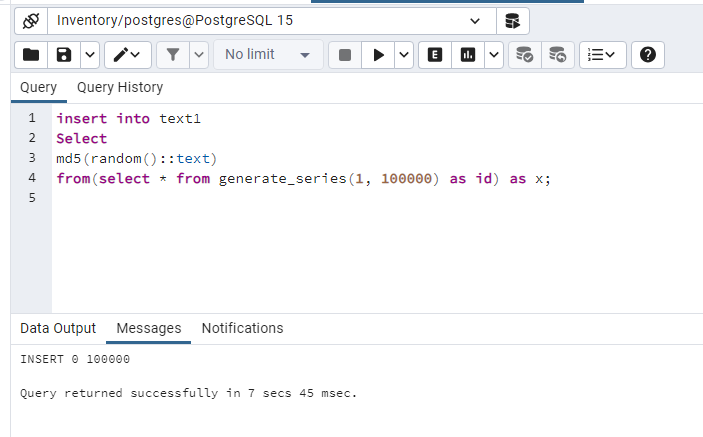
insert into text1

Select

md5(random()::text)

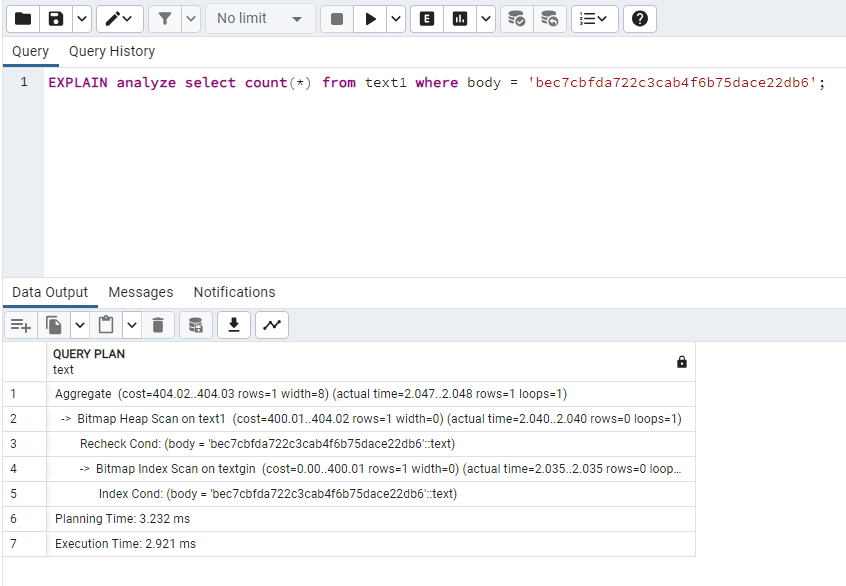
from(select \* from generate\_series(1, 100000) as id) as x;



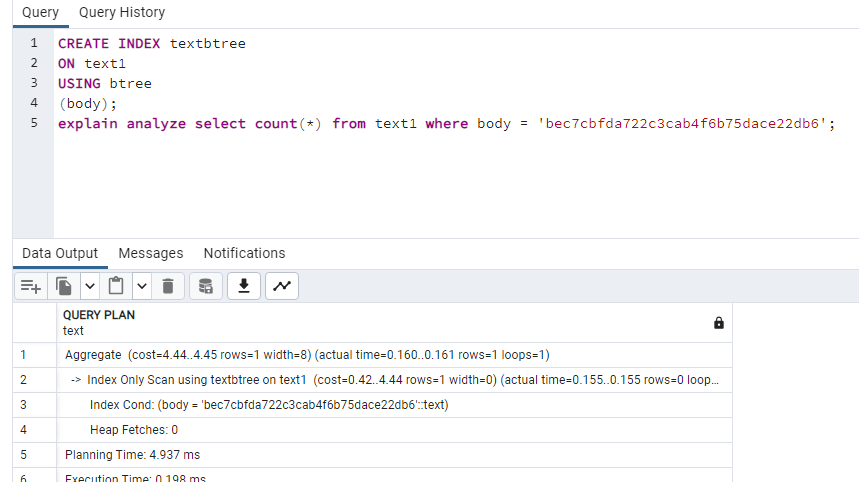


BTREE

Час виконання без індексу:



Час виконання з використанням індексу:



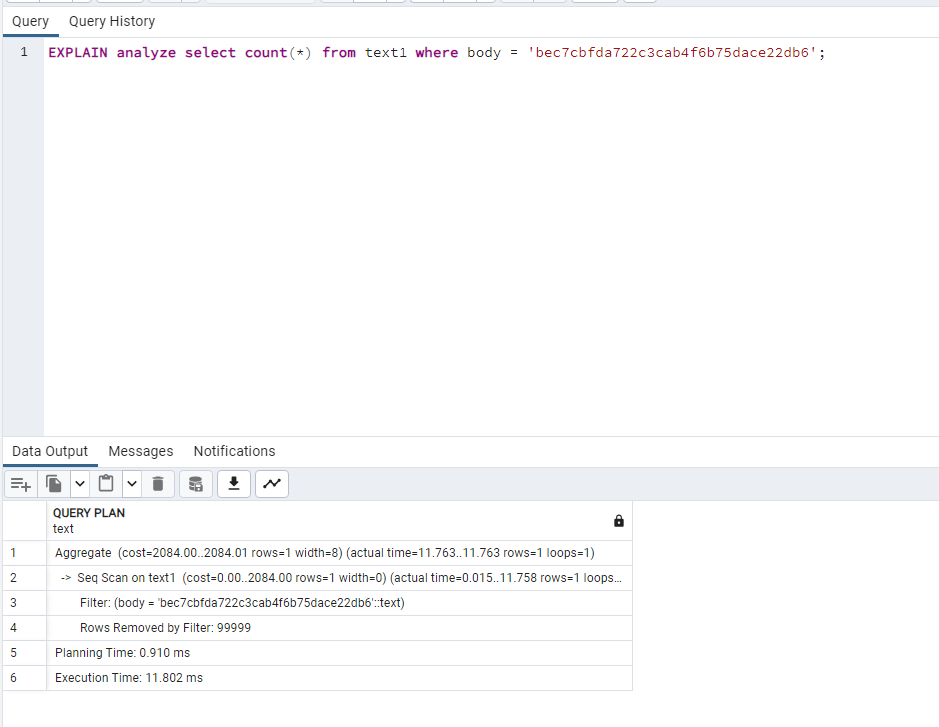
BTREE (Binary Tree) - це особливий тип індексу в PostgreSQL, спроектований для ефективного управління та пошуку даних. В основі його функціонування лежить структура бінарного дерева, де кожен вузол містить ключ та вказівники на лівого та правого нащадка.

Дані в таблиці організовані в бінарне дерево, де кожен вузол представляє собою ключ і вказівники на два піддерева. В результаті операцій вставки, оновлення та видалення дані розташовуються у відсортованому порядку, що дозволяє ефективно використовувати бінарне дерево для швидкого пошуку конкретних значень.

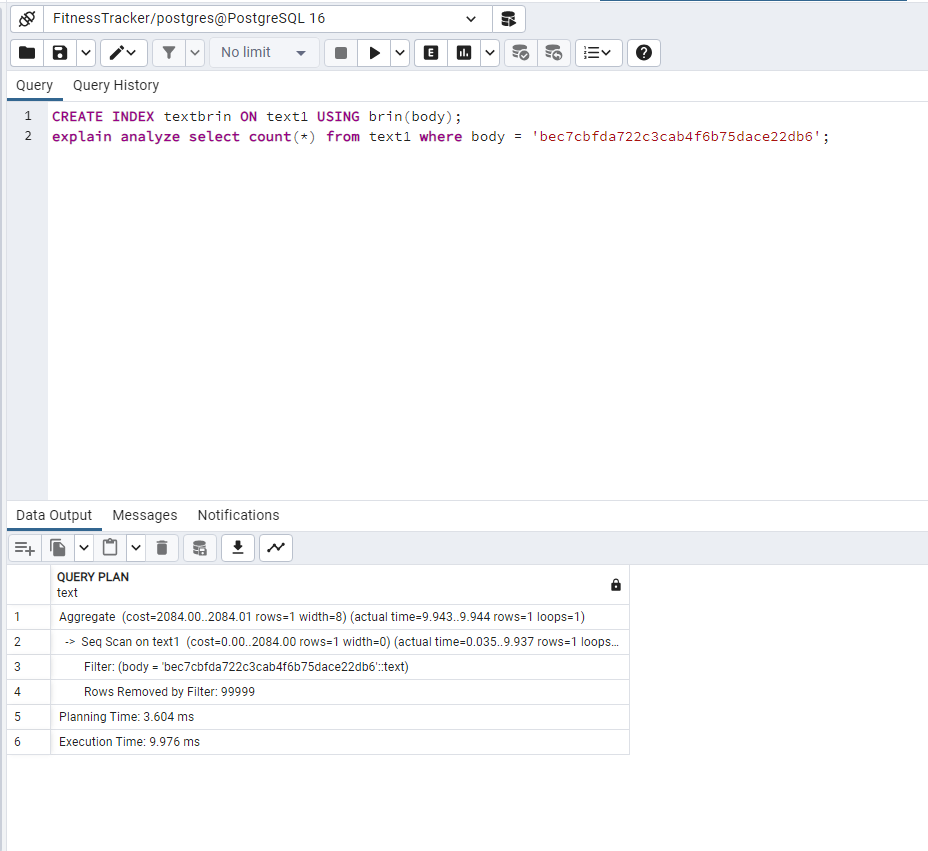
Кожен вузол бінарного дерева має властивість бути меншим за всі ключі в його правому піддереві та більшим за всі ключі в його лівому піддереві, забезпечуючи швидкий доступ до конкретних значень через порівняння ключів.

BRIN

Час виконання без індексу:



Час виконання з використанням індексу:

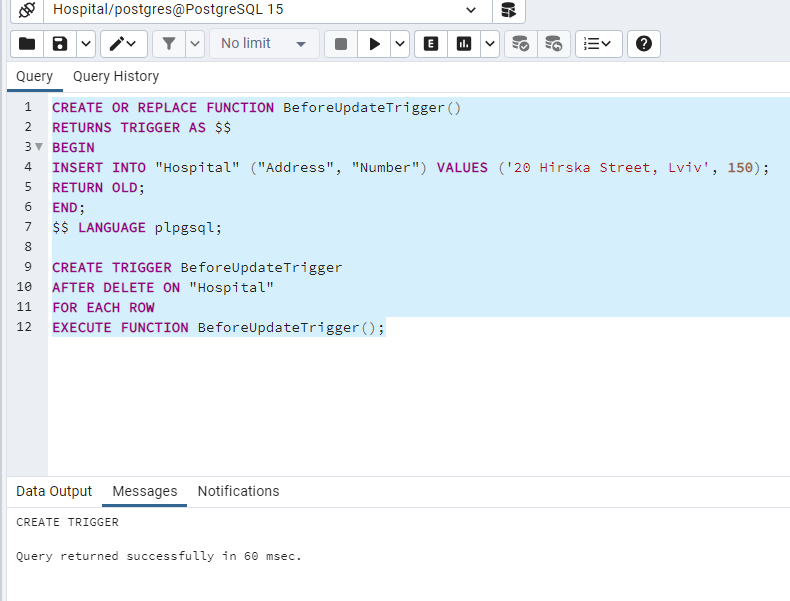


BRIN (Block Range INdex) - це особливий тип індексу в PostgreSQL, спроектований для ефективного роботи з діапазонами даних. В основі його дії лежить ідея того, що дані в таблиці групуються в блоки (частини) за певними діапазонами значень. Кожен блок утримує метадані про максимальне та мінімальне значення у цьому блоку.

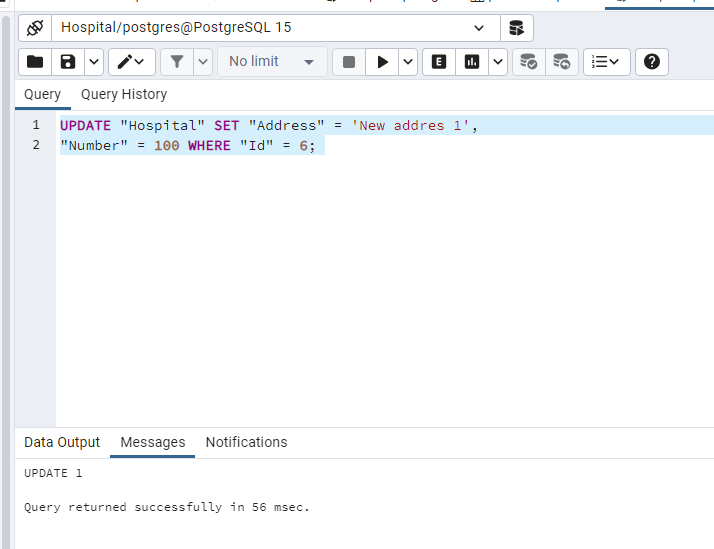
**Завдання 3**

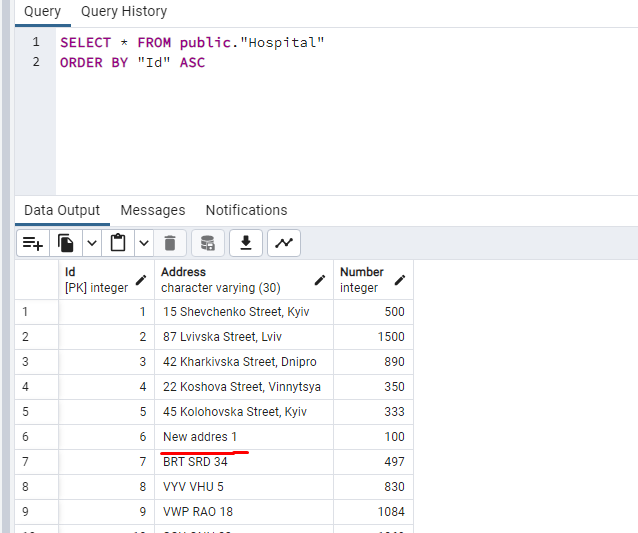
***before update***

**Код тригеру:**



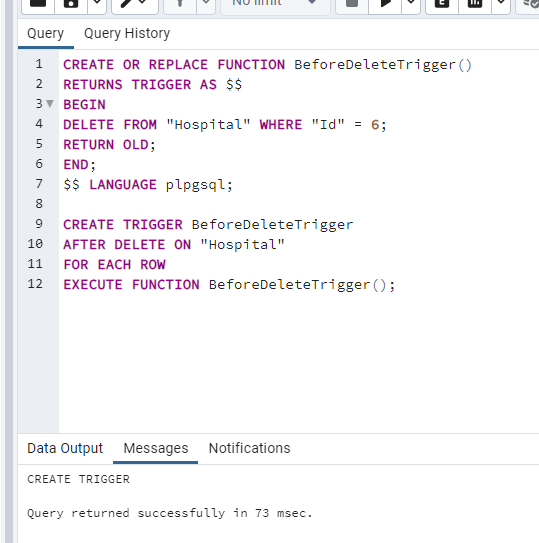
**Тестування роботи тригеру:**



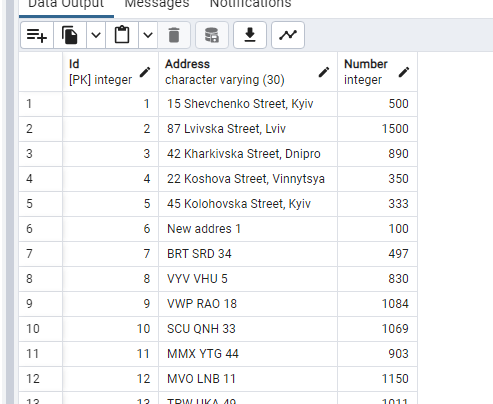


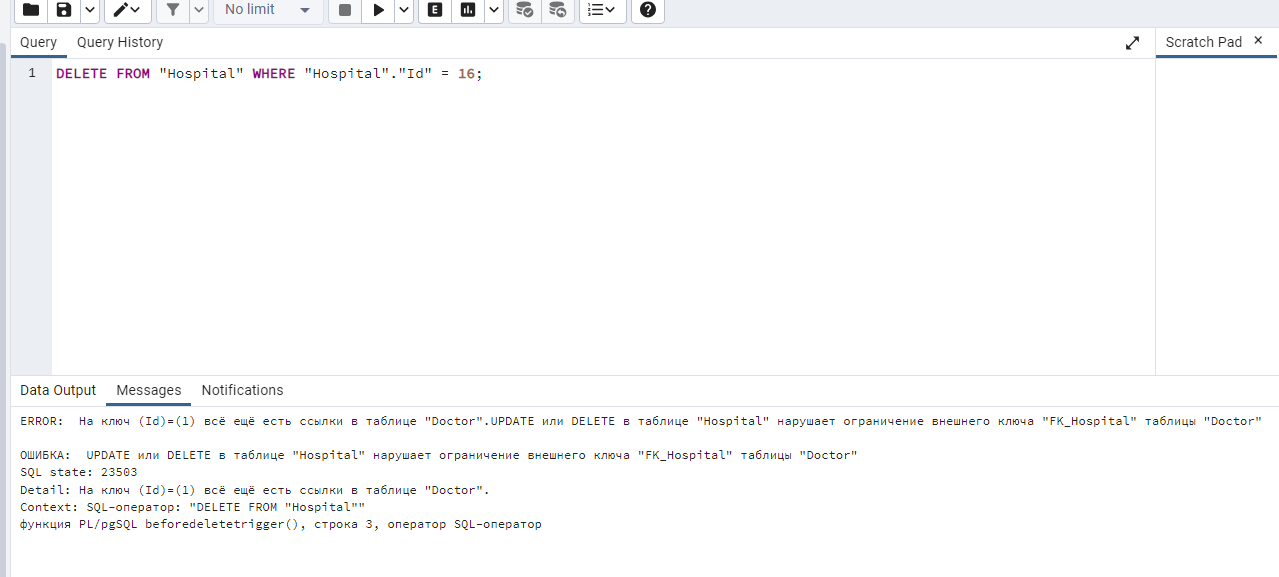
***before delete***

**Код тригеру:**

****

**Тестування роботи тригеру:**

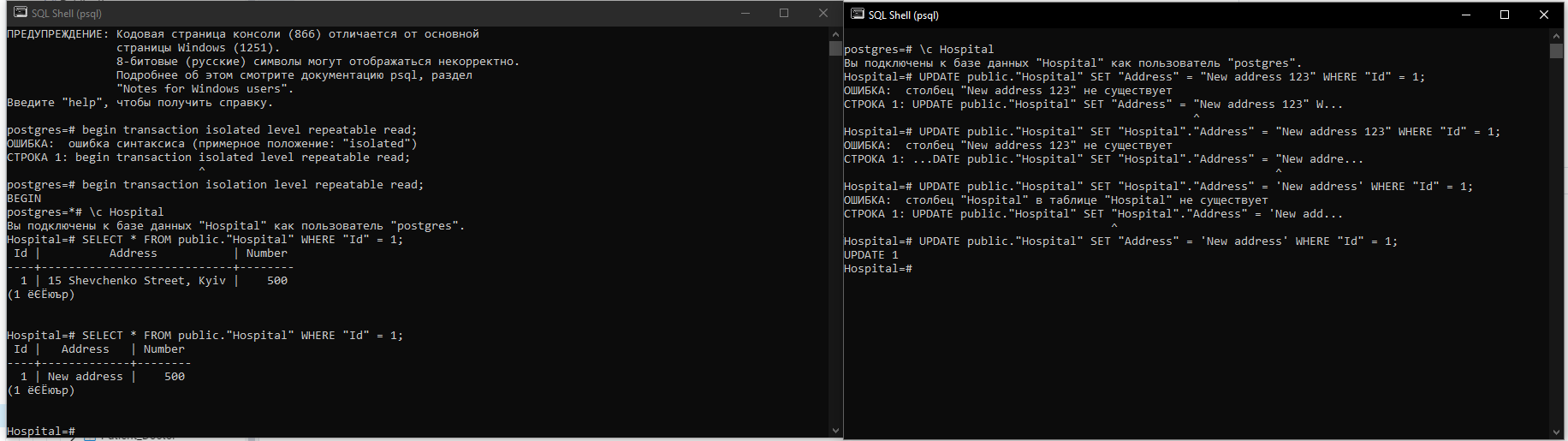




**Завдання 4**

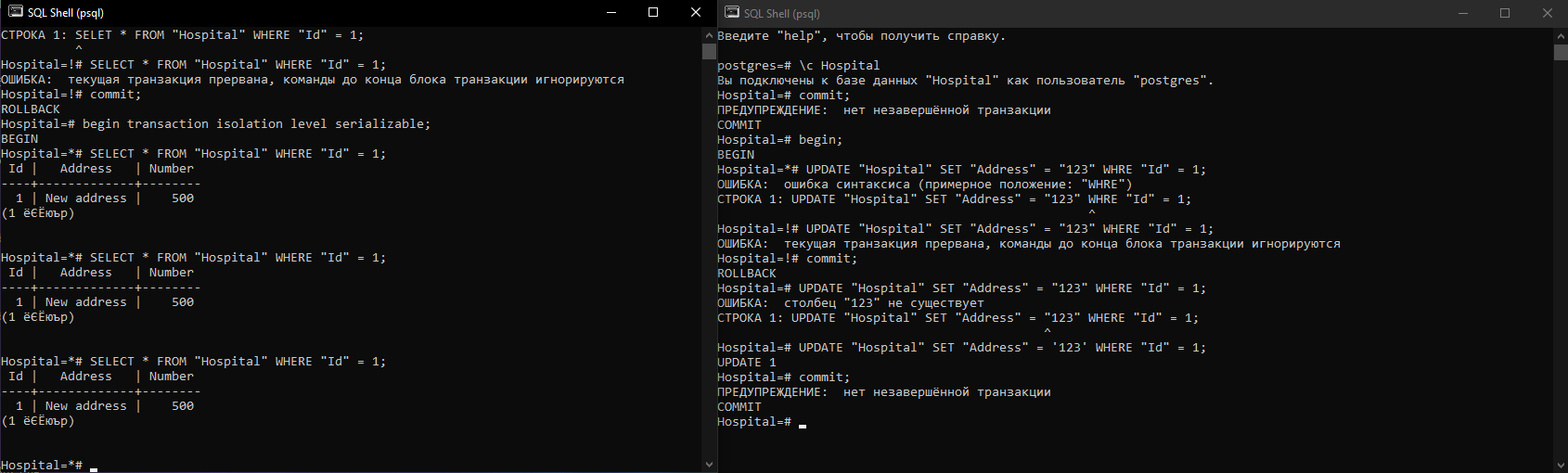
REPEATABLE READ

REPEATABLE READ - це рівень ізоляції транзакцій у базах даних, який забезпечує найвищий рівень стійкості до "фантомних читань" і "фантомних вставок", але не запобігає "фантомним оновленням". Цей рівень ізоляції гарантує, що якщо транзакція читає дані, вона побачить той самий набір даних при подальших читаннях в рамках цієї транзакції. Основні характеристики рівня ізоляції REPEATABLE READ: Читання набору даних, заблокованого для запису: Транзакція, яка працює на рівні REPEATABLE READ, блокує рядки даних, з якими вона взаємодіє для читання. Це означає, що інші транзакції не можуть змінювати ці рядки до завершення транзакції, що запобігає "брудному читанню". Запобігання фантомним читанням: Транзакція з рівнем REPEATABLE READ також блокує діапазони даних, щоб запобігти "фантомним читанням". Це означає, що якщо в транзакції був виконаний запит, що повертає набір даних, інші транзакції не можуть вставляти нові рядки в цей набір даних до завершення першої транзакції. Неблокуючі оновлення: Хоча REPEATABLE READ забезпечує високий рівень ізоляції, він не запобігає "фантомним оновленням". Це означає, що інші транзакції можуть вставляти нові рядки, які відповідають умовам запиту поточної транзакції, навіть якщо ці рядки не існували на момент першого читання.

****

SERIALIZABLE

SERIALIZABLE - це найвищий рівень ізоляції транзакцій в базах даних. Цей рівень забезпечує повну ізоляцію транзакцій, включаючи запобігання "фантомним читанням", "фантомним вставкам" і "фантомним оновленням". Основна ідея - гарантувати, що кожна транзакція бачить базу даних в однаковому стані, якщо порівняти стан бази даних перед і після її виконання. Основні характеристики рівня ізоляції SERIALIZABLE: Повна блокада для читання і запису: Транзакція, яка працює на рівні SERIALIZABLE, блокує всі рядки даних, з якими вона взаємодіє, для читання і запису. Це забезпечує повну ізоляцію від інших транзакцій, уникнення "грязного читання" і будь-яких інших видів конфліктів. Запобігання фантомним читанням і вставкам: SERIALIZABLE гарантує, що транзакції не побачать нових рядків, які були вставлені іншими транзакціями, і не зможуть вставити нові рядки в той самий набір даних після читання. Це включає в себе запобігання "фантомним читанням" і "фантомним вставкам". Недопущення фантомних оновлень: SERIALIZABLE також запобігає "фантомним оновленням", гарантуючи, що транзакції не зможуть оновлювати дані, які були змінені іншими транзакціями після початкового читання. Важливо відзначити, що SERIALIZABLE може призвести до значного блокування ресурсів і може впливати на продуктивність системи, тому вибір рівня ізоляції повинен бути обдуманим і залежати від конкретних потреб та вимог додатка.

****

READ COMMITED

READ COMMITTED - це рівень ізоляції транзакцій в базах даних, який дозволяє читати тільки ті дані, які вже були фіксовані (збережені) в базі даних і виводиться з факту того, що інші транзакції можуть змінювати дані в момент читання. Основні характеристики рівня ізоляції READ COMMITTED: Блокування для запису, але не для читання: Транзакція, яка працює на рівні READ COMMITTED, блокує рядки даних для запису, коли вони зберігаються в рамках цієї транзакції. Це дозволяє іншим транзакціям читати ці рядки, але вони не зможуть їх змінити до завершення першої транзакції, що запобігає "грязному читанню". Недопущення фантомних читань: READ COMMITTED запобігає "фантомним читанням" - вставці нових рядків в момент читання. Однак цей рівень ізоляції не гарантує, що унікальність результату SELECT залишиться незмінною для всієї транзакції. Допуск фантомних вставок і оновлень: Транзакція READ COMMITTED допускає "фантомні вставки" і "фантомні оновлення", тобто вставку чи оновлення даних, які відповідають умовам запиту, під час виконання транзакції. Рівень READ COMMITTED зазвичай є компромісом між високою ізоляцією та продуктивністю, оскільки він дозволяє одночасні читання та записи. Він підходить для більшості сценаріїв, але слід ретельно розглядати його використання в залежності від конкретних потреб вашого додатка.

