Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут» ім. Ігоря Сікорського

Лабораторна робота №2

з дисципліни «Бази Даних»

«Засоби оптимізації роботи СУБД PostgreSQL»

Виконав студент групи: КВ-32

Косарук Захар

Варіант: Медична система для збереження даних пацієнтів

Варіант в групі: 11

Репозиторій на GitHub: https://github.com/zZaKko96/Medicine-Database.git

Telegram: https://t.me/zZaKko

Київ 2025

Завдання №1. Об'єктно-реляційна проекція (ORM)

Для реалізації ORM було використано бібліотеку **Entity Framework Core** з провайдером Npgsql.EntityFrameworkCore.PostgreSQL.

1. Схема бази даних

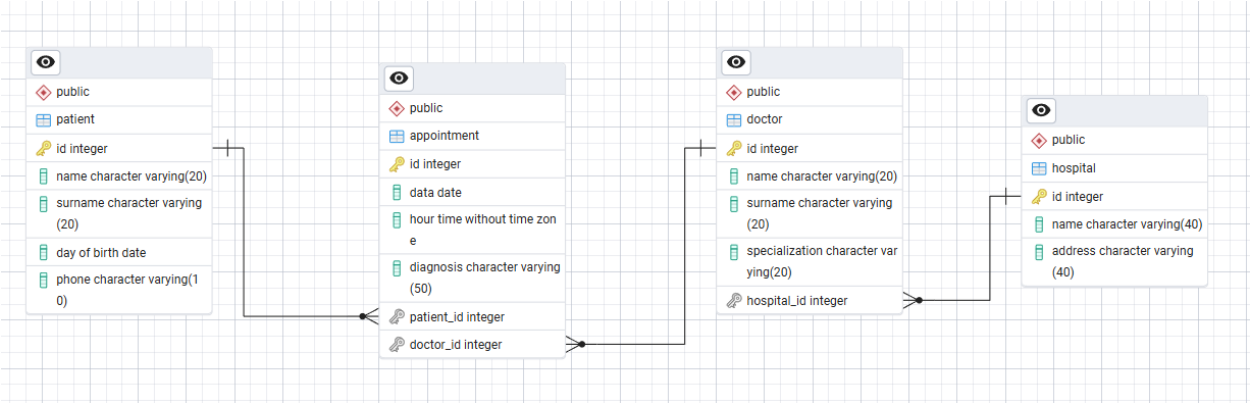
База даних postgres складається з чотирьох пов'язаних таблиць.

Таблиці та атрибути:

- **hospital** (Лікарні): id (PK), name, address.
- **doctor** (Лікарі): id (PK), name, surname, specialization, hospital_id (FK).
- **patient** (Пацієнти): id (PK), name, surname, day of birth, phone.
- **appointment** (Прийоми): id (PK), data, hour, diagnosis, patient_id (FK), doctor_id (FK).

Зв'язки:

- **1:M (Один до багатьох):** hospital -> doctor (Одна лікарня має багато лікарів).
- **1:M (Один до багатьох):** doctor -> appointment (Один лікар проводить багато прийомів).
- **1:M (Один до багатьох):** patient -> appointment (Один пацієнт може мати багато прийомів).



2. Класи ORM та зв'язки

Для відображення таблиць у коді створено класи-сутності з анотаціями даних ([Key], [Column], [ForeignKey]). Зв'язки реалізовані через **навігаційні властивості** (virtual).

Файл DataModels.cs:

```
C#
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

// Сутність "Лікарня"
public class Hospital
{
    [Key]
    [Column("id")]
    public int Id { get; set; }

    [Column("name")]
    public string Name { get; set; }

    [Column("address")]
    public string Address { get; set; }

    // Зв'язок 1:М з лікарями
    public virtual ICollection<Doctor> Doctors { get; set; } = new List<Doctor>();
}

// Сутність "Лікар"
public class Doctor
{
    [Key]
    [Column("id")]
    public int Id { get; set; }

    [Column("name")]
    public string Name { get; set; }

    [Column("surname")]
    public string Surname { get; set; }

    [Column("specialization")]
    public string Specialization { get; set; }

    [Column("hospital_id")]
    public int HospitalId { get; set; }
```

```
    // Зв'язок M:1 (Належить лікарні)
    [ForeignKey("HospitalId")]
    public virtual Hospital Hospital { get; set; }

    // Зв'язок 1:M з прийомами
    public virtual ICollection<Appointment> Appointments { get; set; } = new
List<Appointment>();
}

// Сутність "Пацієнт"
public class Patient
{
    [Key]
    [Column("id")]
    public int Id { get; set; }

    [Column("name")]
    public string Name { get; set; }

    [Column("surname")]
    public string Surname { get; set; }

    [Column("day of birth")]
    public DateOnly DayOfBirth { get; set; }

    [Column("phone")]
    public string Phone { get; set; }

    // Зв'язок 1:M з прийомами
    public virtual ICollection<Appointment> Appointments { get; set; } = new
List<Appointment>();
}

// Сутність "Прийом"
public class Appointment
{
    [Key]
    [Column("id")]
    public int Id { get; set; }

    [Column("data")]
    public DateOnly Data { get; set; }

    [Column("hour")]
    public TimeOnly Hour { get; set; }

    [Column("diagnosis")]
    public string Diagnosis { get; set; }

    [Column("patient_id")]
    public int PatientId { get; set; }

    [Column("doctor_id")]
    public int DoctorId { get; set; }

    // Зв'язок M:1 (Належить пацієнту)
    [ForeignKey("PatientId")]
    public virtual Patient Patient { get; set; }

    // Зв'язок M:1 (Належить лікарю)
    [ForeignKey("DoctorId")]
```

```
    public virtual Doctor Doctor { get; set; }
}
```

3. Приклади запитів ORM

У класі DatabaseModel замість прямих SQL-запитів використано методи Entity Framework.

Вставка (INSERT):

```
C#
public async Task AddPatientAsync(Patient patient)
{
    using var db = new ApplicationContext();
    db.Patients.Add(patient); // Додавання об'єкта в контекст
    await db.SaveChangesAsync(); // Генерація INSERT запиту
}
```

Вибірка (SELECT):

```
C#
public async Task<List<Patient>> GetAllPatientsAsync()
{
    using var db = new ApplicationContext();
    // Отримання списку всіх пацієнтів із сортуванням
    return await db.Patients.OrderBy(p => p.Id).ToListAsync();
}
```

Оновлення (UPDATE):

```
C#
public async Task<string> UpdatePatientAsync(Patient patient)
{
    using var db = new ApplicationContext();
    db.Patients.Update(patient); // Позначення об'єкта як зміненого
    int rows = await db.SaveChangesAsync(); // Генерація UPDATE запиту
    return rows > 0 ? "Дані оновлено." : "Помилка.";
}
```

Видалення (DELETE):

```
C#
public async Task<string> DeletePatientAsync(int id)
{
    using var db = new ApplicationContext();
    // Ефективне видалення без завантаження об'єкта в пам'ять
    int rows = await db.Patients.Where(p => p.Id == id).ExecuteDeleteAsync();
    return rows > 0 ? "Видалено." : "Не знайдено.";
}
```

Складний пошуковий запит (JOIN, FILTER, GROUP BY):

Реалізація пошуку статистики пацієнтів за допомогою LINQ.

```
C#
var query = from p in db.Patients
            join a in db.Appointments on p.Id equals a.PatientId
            join d in db.Doctors on a.DoctorId equals d.Id
            where EF.Functions.ILike(d.Surname, $"%{surnamePattern}%")
                  && a.Data >= startDate && a.Data <= endDate
            group new { p, d } by new { p.Id, p.Name, p.Surname, p.Phone, DoctorSurname
= d.Surname } into g
            orderby g.Count() descending
```

```
select new
{
    g.Key.Name,
    g.Key.Surname,
    g.Key.Phone,
    g.Key.DoctorSurname,
    Count = g.Count()
};
```

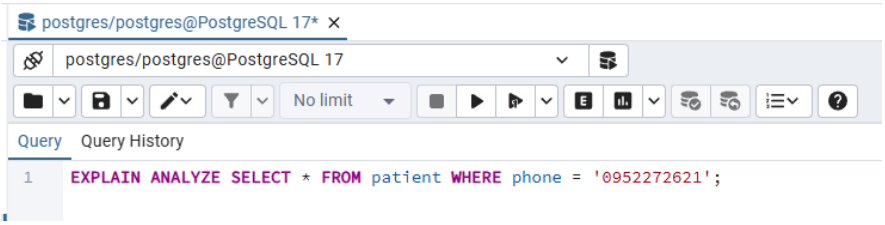
Завдання №2. Засоби прискорення роботи (Індекси)

Згідно з варіантом №11, було досліджено роботу індексів типів **Hash** та **GIN**. Для тестування використано набір згенерованих рандомізованих даних (10000 записів).

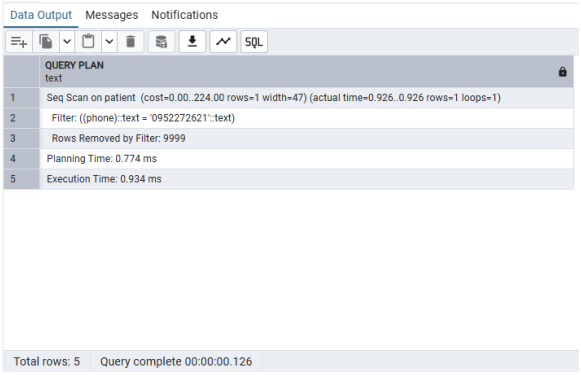
1. Hash Index (Хеш-індекс)

Сценарій: Пошук пацієнта за точним номером телефону.

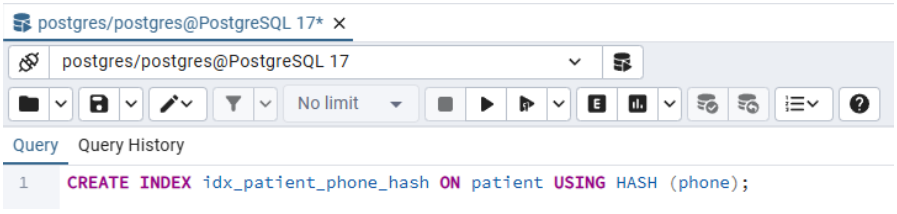
SQL-запит:

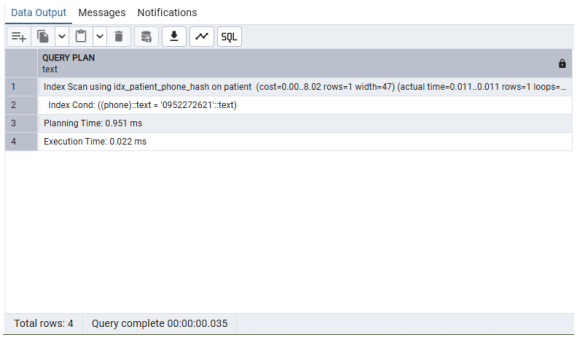


Результати виконання:



- **Без індексу:** Виконувався Seq Scan (повний перебір таблиці). Час виконання: **~0.934 ms**.
- **Команда створення індексу:**





- **З індексом:** Виконується Index Scan. Час виконання: **~0.022 ms**.

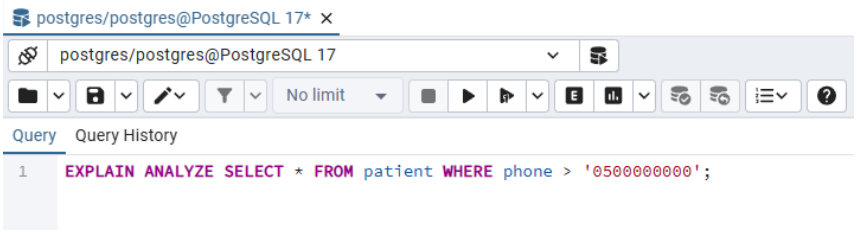
Пояснення:

Швидкість виконання запиту зросла у ~42 рази. Hash-індекс працює за принципом хеш-таблиці: обчислює хеш від значення ключа (phone) і миттєво звертається до відповідного "відра" (bucket) з даними. Це забезпечує складність пошуку O(1) для операцій точної рівності (=).

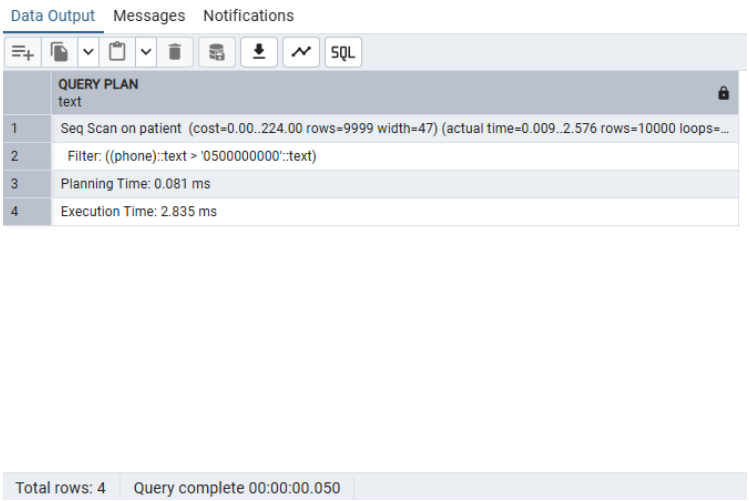
2. Обмеження Hash-індексу (Пошук діапазону)

Сценарій: Пошук пацієнтів, у яких номер телефону більше заданого значення (оператор >).

SQL-запит:



Результати виконання:



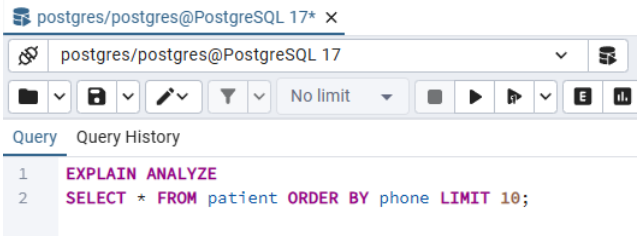
- Планувальник запитів **проігнорував** створений Hash-індекс idx_patient_phone_hash.
- Було виконано Seq Scan (повне сканування таблиці).
- Час виконання: **2.835 ms**.

Пояснення: Це демонструє, що Hash-індекси **не підтримують операції діапазону** (>, <, >=, <=, BETWEEN). Хеш-функція перетворює ключ у довільне число, яке не корелює з порядком самих ключів (наприклад, хеш від "A" може бути більшим за хеш від "Z"). Тому СУБД не може використати індекс для знаходження "всіх значень, більших за X", і змушена перевіряти кожен рядок таблиці вручну.

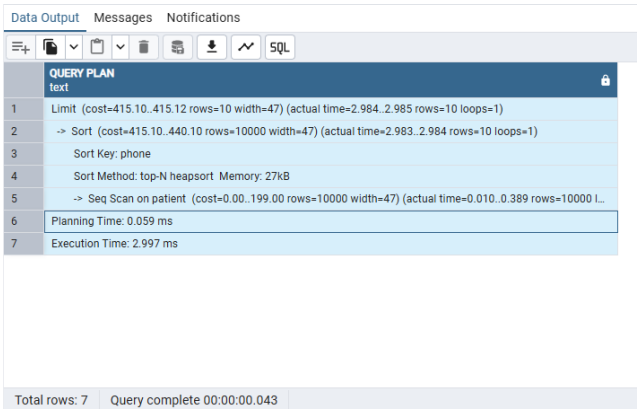
3. Обмеження Hash-індексу

Сценарій: Сортування даних за номером телефону.

SQL-запит:



Результати виконання:



- Планувальник запитів **проігнорував** Hash-індекс і використав Seq Scan та сортування в пам'яті (Sort Method: top-N heapsort).
- Час виконання: **~2.997 ms**.

Пояснення (чому індекс не прискорив):

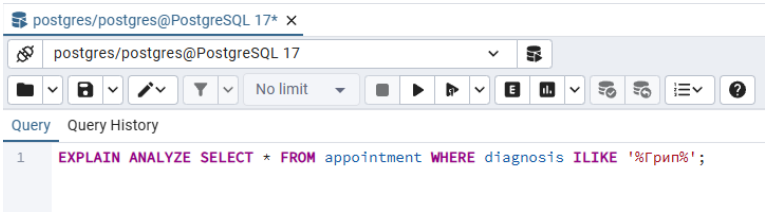
Hash-індекси зберігають дані хаотично (за значенням хеш-функції) і не підтримують впорядкування. Тому вони не можуть використовуватися для

прискорення операцій сортування (ORDER BY) або пошуку діапазонів (>, <, BETWEEN). У таких випадках доцільніше використовувати B-Tree індекс.

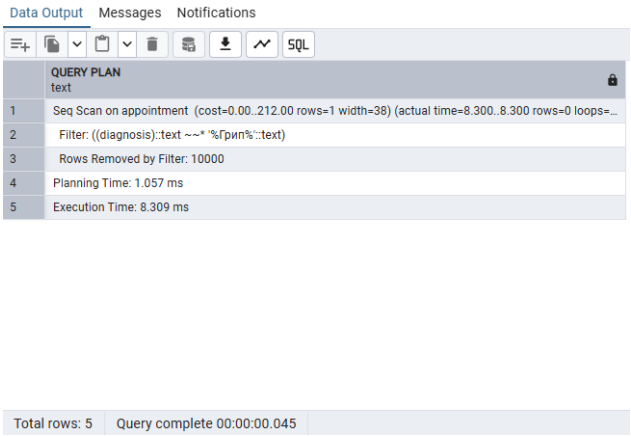
4. GIN Index (Generalized Inverted Index)

Сценарій: Пошук записів, де діагноз містить певний підрядок (наприклад, "Грип"). Стандартні індекси (B-Tree) неефективні для запитів з LIKE '%...%'.

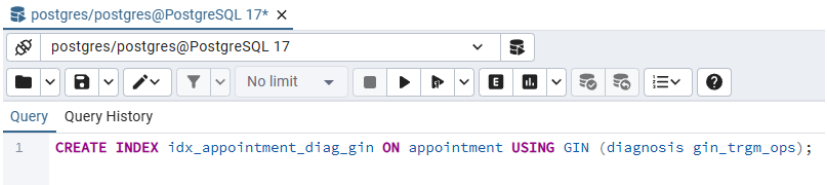
SQL-запит:



Результати виконання:

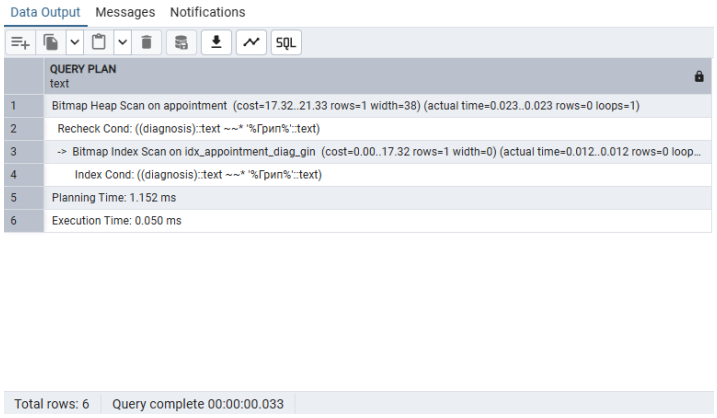


- **Без індексу:** Виконувався Seq Scan. Час виконання: **~8.309 ms**.
- **Команда створення індексу:**



SQL

CREATE INDEX idx_appointment_diag_gin ON appointment USING GIN (diagnosis gin_trgm_ops);



- **З індексом:** Виконується Bitmap Heap Scan. Час виконання: **~0.050 ms**.

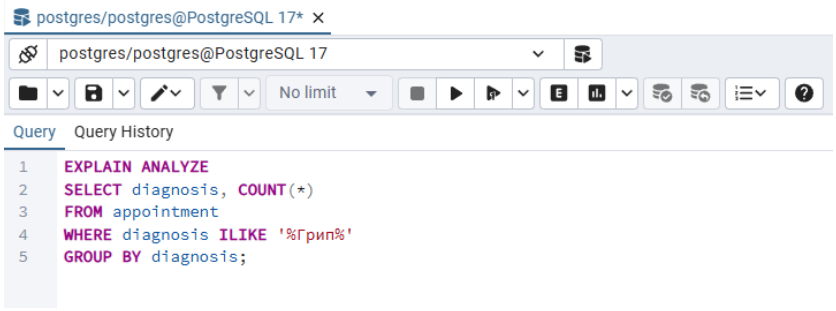
Пояснення:

Швидкість зросла у ~166 разів. GIN-індекс з класом операторів gin_trgm_ops розбиває текст на триграми (групи з 3 символів) і будує інвертований індекс. Це дозволяє ефективно шукати входження підрядка в будь-якій частині тексту, що неможливо зі звичайними індексами.

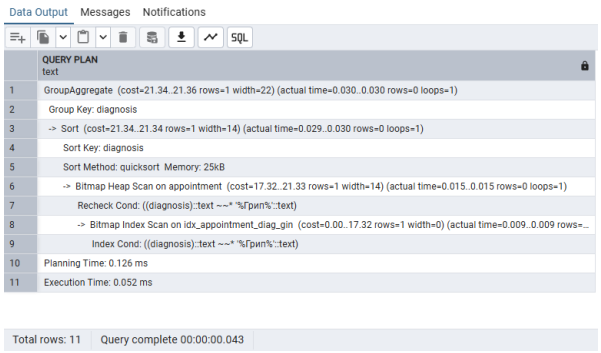
5. Використання індексу в аналітичних запитах

Сценарій: Підрахунок кількості діагнозів певної групи.

SQL-запит:



Результати виконання:



- Використано Bitmap Index Scan on idx_appointment_diag_gin.
- Час виконання: **0.052 ms**.

Пояснення:

Індекс дозволив СУБД відфільтрувати необхідні рядки до етапу групування та агрегації. Замість сканування та сортування всієї таблиці, операції GROUP BY та COUNT виконувалися лише над невеликою вибіркою знайдених записів, що забезпечило миттєвий результат.

Завдання №3. Розробка тригера (Trigger)

Відповідно до варіанту №11, було розроблено тригер з умовою спрацювання **BEFORE UPDATE, DELETE**.

1. Призначення та логіка роботи

Розроблений тригер trg_patient_audit_before виконує функцію аудиту та контролю цілісності даних:

1. **Логування (Audit):** Перед будь-якою зміною (UPDATE) або видаленням (DELETE) запису в таблиці patient, поточний стан даних (старі значення) автоматично зберігається в архівну таблицю patient_log із зазначенням типу операції та часу.
2. **Обмеження (Exception Handling):** Реалізовано бізнес-правило, яке забороняє видалення пацієнтів у певні дні тижня (для демонстрації обробки виключних ситуацій).

2. Текст SQL-скриптів

Створення таблиці для логування:

```
SQL
CREATE TABLE patient_log (
    log_id SERIAL PRIMARY KEY,
    operation_type VARCHAR(10), -- 'UPDATE' або 'DELETE'
    operation_time TIMESTAMP DEFAULT NOW(),
    user_name TEXT DEFAULT current_user,
    old_patient_id INT,
    old_name VARCHAR(100),
    old_surname VARCHAR(100),
    old_phone VARCHAR(20)
);
```

Текст тригерної функції:

```
SQL
CREATE OR REPLACE FUNCTION audit_patient_changes()
RETURNS TRIGGER AS $$
BEGIN
    -- 1. БЛОК ПЕРЕВІРКИ (Business Logic & Exceptions)
    -- Заборона видалення у певний день тижня (наприклад, Понеділок = 1)
    IF (TG_OP = 'DELETE') THEN
        IF (EXTRACT(ISODOW FROM NOW()) = 1) THEN
```

```
            RAISE EXCEPTION 'Увага! Видалення пацієнтів по понеділках заборонено політикою безпеки!';
        END IF;
    END IF;

    -- 2. БЛОК ЛОГУВАННЯ (Archiving)
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO patient_log (operation_type, old_patient_id, old_name, old_surname, old_phone)
        VALUES ('DELETE', OLD.id, OLD.name, OLD.surname, OLD.phone);
        RETURN OLD; -- Повертаємо старий рядок для продовження видалення
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO patient_log (operation_type, old_patient_id, old_name, old_surname, old_phone)
        VALUES ('UPDATE', OLD.id, OLD.name, OLD.surname, OLD.phone);
        RETURN NEW; -- Повертаємо новий рядок для оновлення
    END IF;

    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Створення тригера:

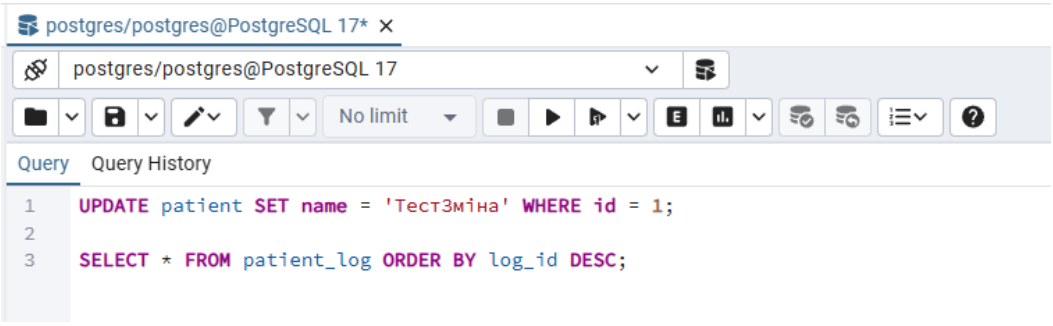
```
SQL
CREATE TRIGGER trg_patient_audit_before
BEFORE UPDATE OR DELETE ON patient
FOR EACH ROW
EXECUTE FUNCTION audit_patient_changes();
```

3. Тестування та результати

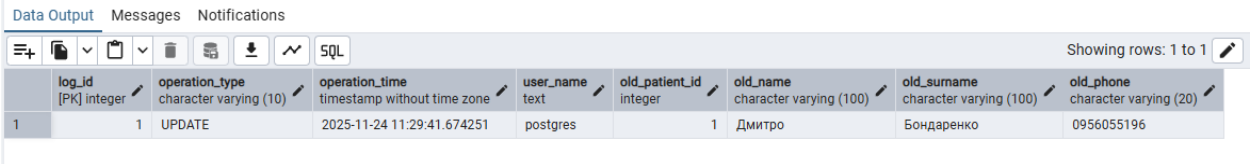
Сценарій 1: Перевірка оновлення даних (UPDATE)

Виконується команда зміни імені пацієнта. Тригер має зберегти попереднє ім'я в лог.

Команда:



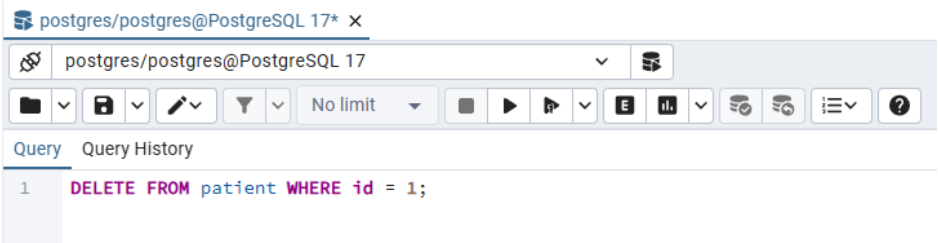
Результат (зміни в таблиці patient_log):



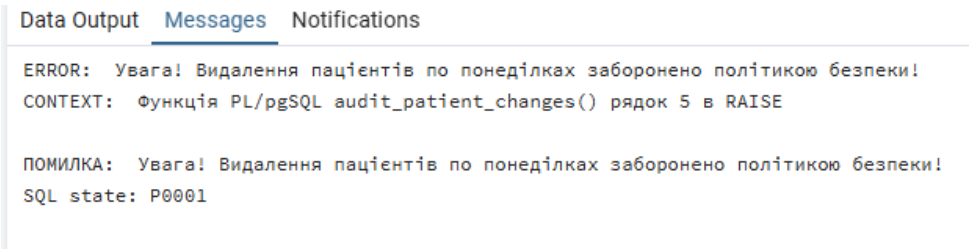
Сценарій 2: Перевірка обробки виключних ситуацій (Exception)

Спроба видалення запису в заборонений день (понеділок). Тригер перериває транзакцію повідомленням про помилку.

Команда:



Результат (повідомлення про помилку):



Сценарій 3: Успішне видалення (DELETE)

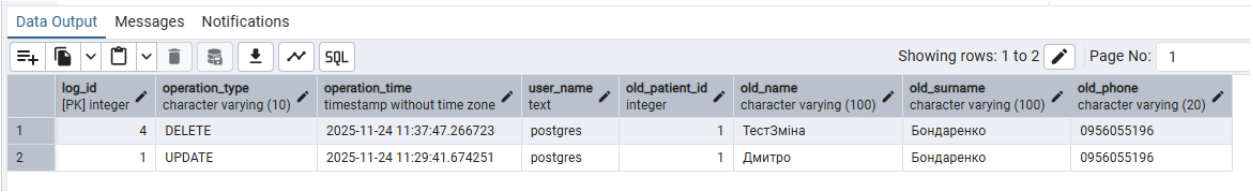
Після зняття часового обмеження виконується видалення пацієнта. Попередньо видаляються пов'язані записи в таблиці appointment, щоб уникнути помилки FK (Foreign Key Constraint).

Команди:

```
SQL
-- Спочатку видаляємо залежні записи (прийоми)
DELETE FROM appointment WHERE patient_id = 1;

-- Тепер видаляємо пацієнта (тригер спрацьовує тут)
DELETE FROM patient WHERE id = 1;
```

Результат (зміни в таблиці patient_log):



	log_id [PK] integer	operation_type character varying (10)	operation_time timestamp without time zone	user_name text	old_patient_id integer	old_name character varying (100)	old_surname character varying (100)	old_phone character varying (20)
1	4	DELETE	2025-11-24 11:37:47.266723	postgres	1	ТестЗміна	Бондаренко	0956055196
2	1	UPDATE	2025-11-24 11:29:41.674251	postgres	1	Дмитро	Бондаренко	0956055196

Завдання №4. Аналіз рівнів ізоляції транзакцій

Метою завдання було дослідження поведінки СУБД PostgreSQL при одночасній роботі декількох транзакцій та демонстрація механізмів

запобігання конфліктам даних. Експерименти проводилися у двох паралельних сесіях (вікнах запитів).

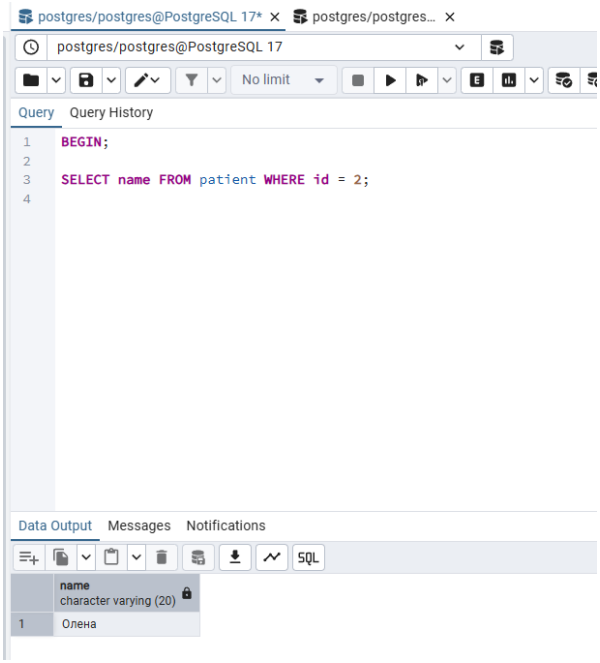
1. Рівень ізоляції READ COMMITTED (За замовчуванням)

Феномен: Неповторюване читання (Non-repeatable Read).

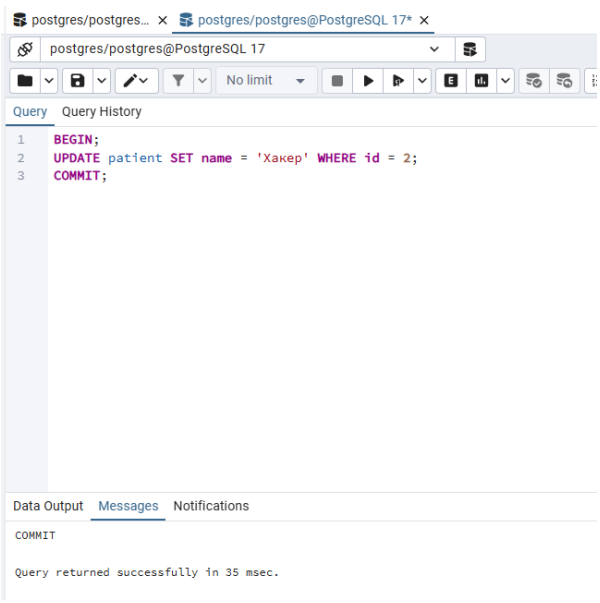
Опис: Транзакція бачить зміни, зафіксовані іншими транзакціями, безпосередньо під час свого виконання. Це означає, що повторний запит до того ж рядка в межах однієї транзакції може повернути інші дані.

Хід виконання:

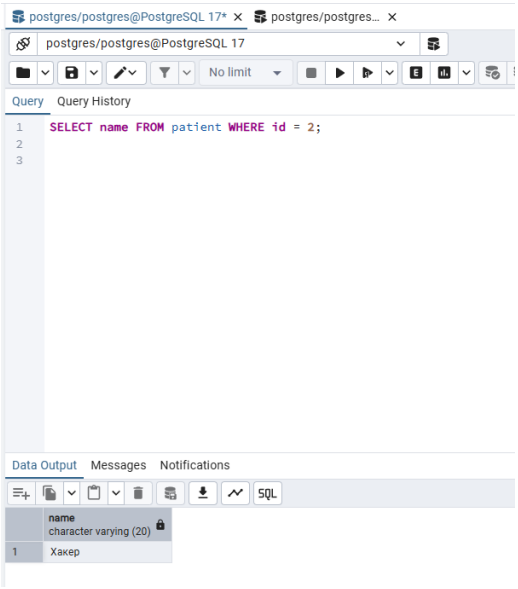
Транзакція А (Вікно 1):



Транзакція Б (Вікно 2):



Транзакція А (Вікно 1):



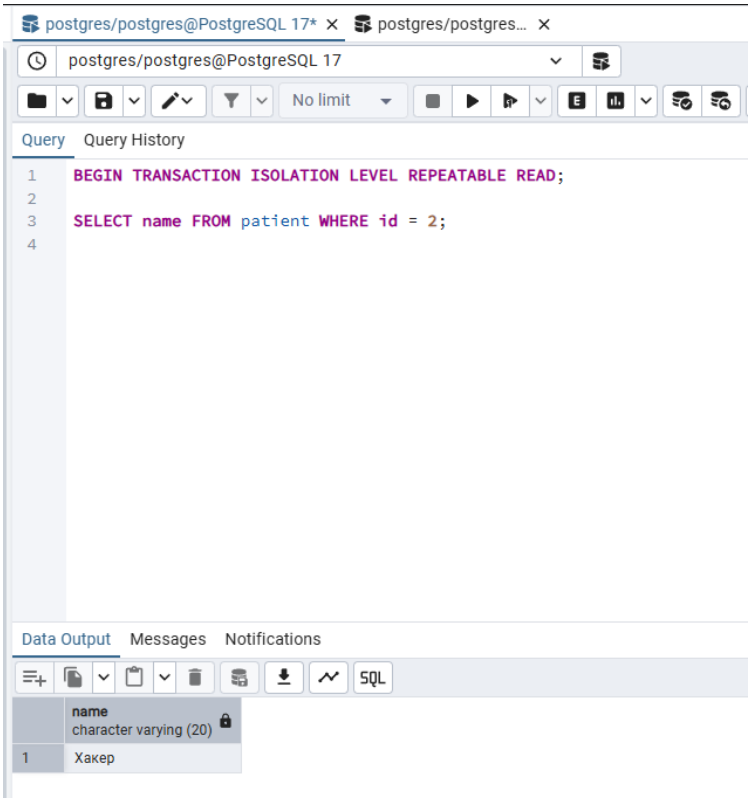
2. Рівень ізоляції REPEATABLE READ

Мета: Уникнення феномену неповторюваного читання.

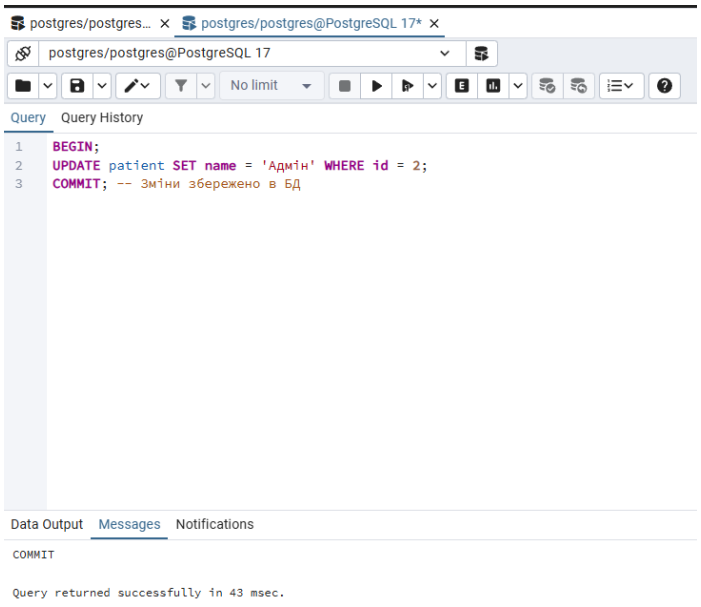
Опис: На цьому рівні транзакція "бачить" стан бази даних (snapshot) на момент свого початку. Будь-які зміни, зафіксовані іншими транзакціями після цього моменту, ігноруються.

Хід виконання:

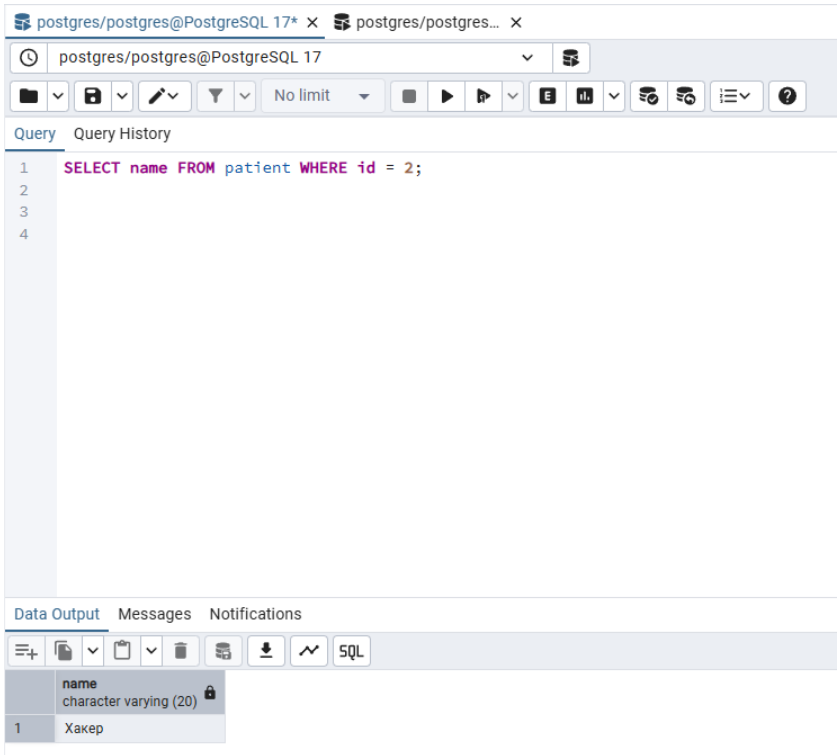
Транзакція А (Вікно 1):



Транзакція Б (Вікно 2):



Транзакція А (Вікно 1):



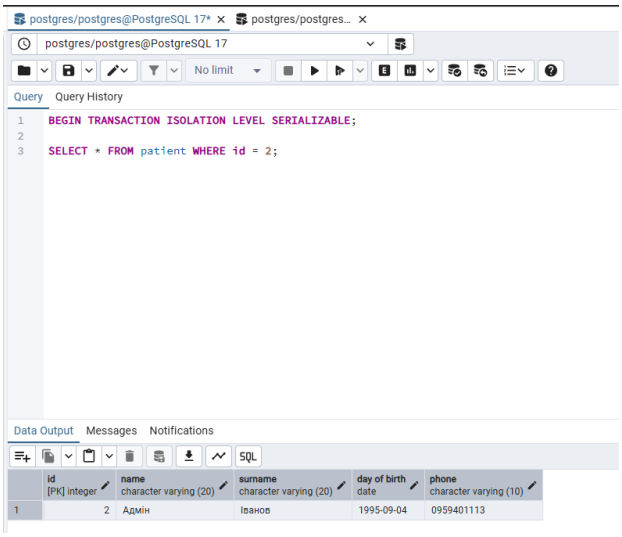
3. Рівень ізоляції SERIALIZABLE

Мета: Запобігання аномаліям серіалізації.

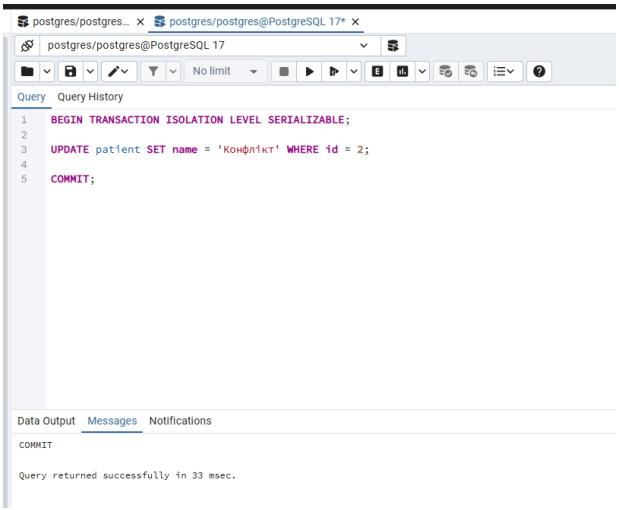
Опис: Найсуворіший рівень ізоляції. СУБД емулює послідовне виконання транзакцій. Якщо виявляється конфлікт (одночасна спроба зміни одних і тих самих даних), одна з транзакцій примусово завершується помилкою.

Хід виконання:

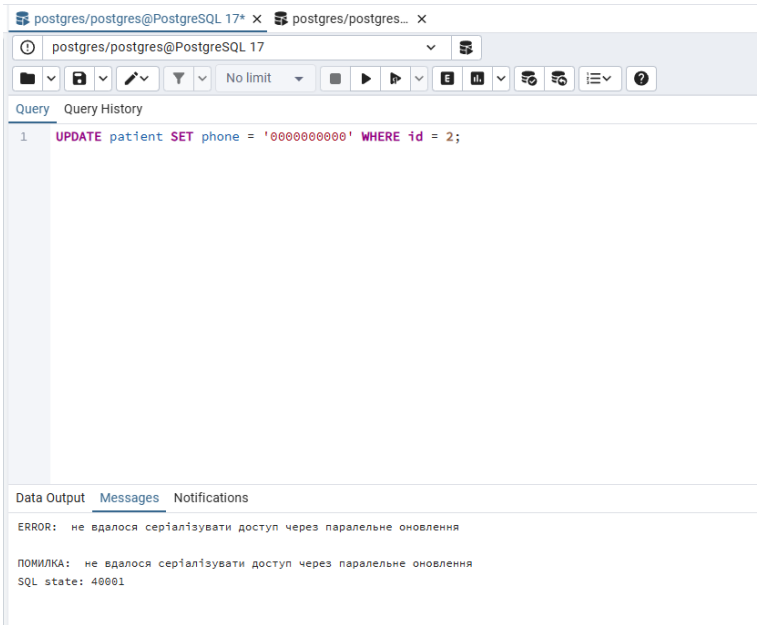
Транзакція А (Вікно 1):



Транзакція Б (Вікно 2):



Транзакція А (Вікно 1):



Результат:

Отримано помилку: СУБД заблокувала операцію для збереження цілісності даних.

Висновки

У ході виконання лабораторної роботи №2 було отримано практичні навички використання засобів оптимізації СУБД PostgreSQL та налаштування взаємодії з базою даних через ORM.

В результаті виконання завдань було досягнуто наступних результатів:

1. **Впровадження ORM:** Модифіковано архітектуру програмного додатку ("Медична система"), замінивши прямі SQL-запити на використання **Entity Framework Core**. Було розроблено класи-сутності (DataModels.cs) та налаштовано контекст даних (ApplicationContext), що дозволило реалізувати CRUD-операції та складні аналітичні вибірки за допомогою технології LINQ, забезпечивши типізацію даних та контроль зовнішніх зв'язків на рівні об'єктної моделі.
2. **Оптимізація запитів (Індекси):** Відповідно до варіанту №11, досліджено вплив індексів на швидкодію:
    - **Hash-індекс** продемонстрував високу ефективність для точного пошуку (=), зменшивши час виконання запиту з ~0.9 мс до ~0.02 мс, однак підтверджено його непридатність для операцій діапазону (>) та сортування.
    - **GIN-індекс** дозволив прискорити текстовий пошук (ILIKE) та агрегацію даних у понад 100 разів (з ~8.3 мс до ~0.05 мс), що робить його оптимальним для фільтрації за підрядком.
3. **Автоматизація та аудит (Тригери):** Розроблено тригерну функцію на мові PL/pgSQL з умовою BEFORE UPDATE/DELETE. Вона забезпечила автоматичне логування змін у таблицю patient_log та реалізацію бізнес-правил (заборона видалення записів у певні дні) з генерацією виключних ситуацій.
4. **Керування конкурентним доступом (Транзакції):** Проаналізовано роботу рівнів ізоляції транзакцій. Практично продемонстровано феномен "неповторюваного читання" на рівні READ COMMITTED, його усунення на рівні REPEATABLE READ, а також механізм блокування конфліктних транзакцій на найсуворішому рівні SERIALIZABLE.

Набуті знання дозволяють ефективно проєктувати високопродуктивні та надійні бази даних, враховуючи вимоги до швидкості пошуку, цілісності даних та безпеки транзакцій.