

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.

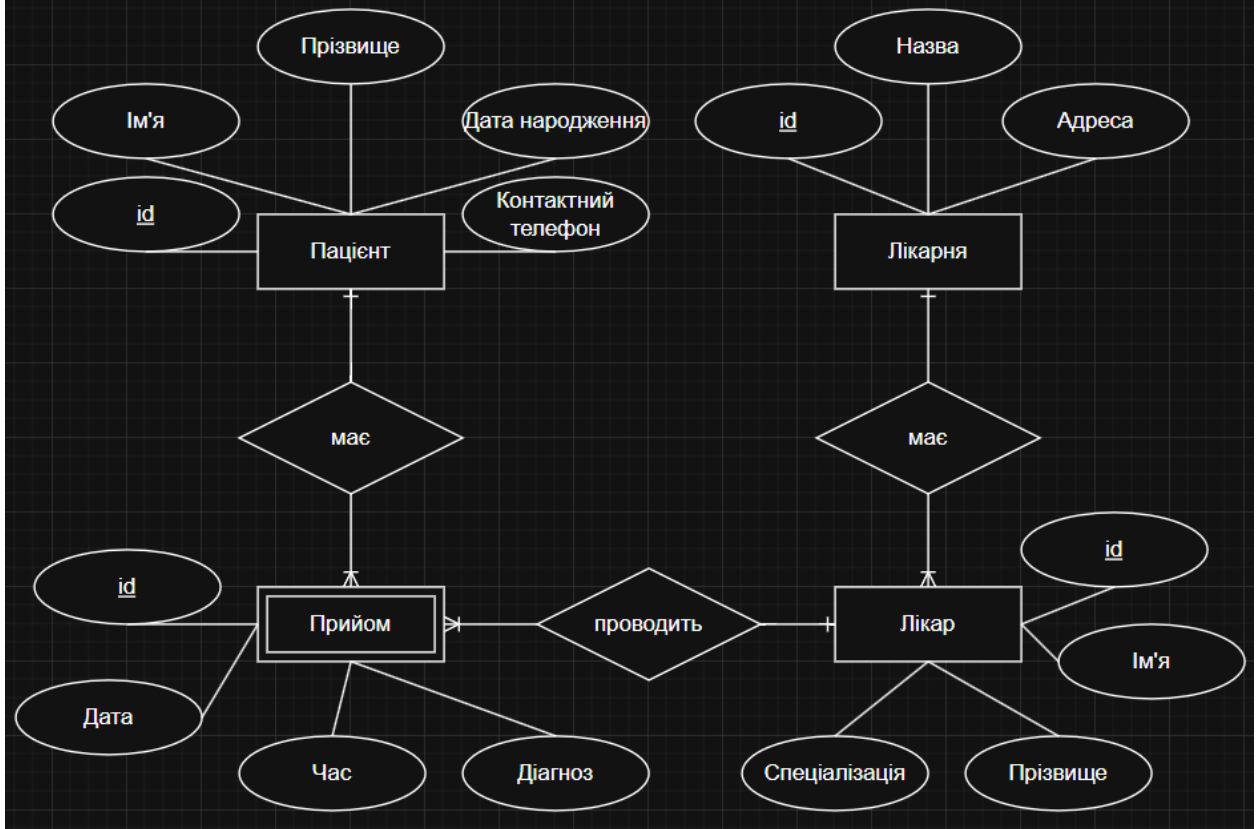
#### Таблиці та атрибути:

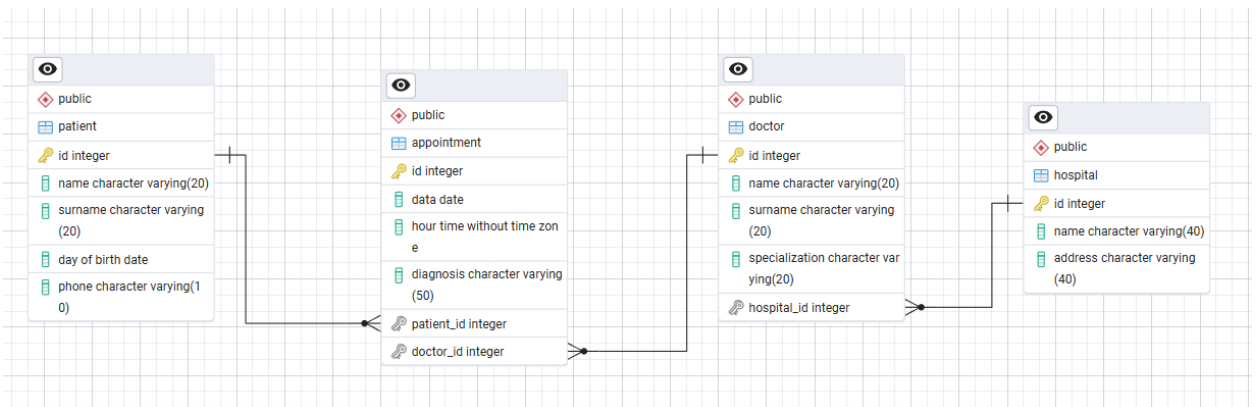
- **hospital** (Лікарні): id (PK), name, address.
- **doctor** (Лікари): id (PK), name, surname, specialization, hospital\_id (FK).
- **patient** (Пацієнти): id (PK), name, surname, day of birth, phone.
- **appointment** (Прийоми): id (PK), data, hour, diagnosis, patient\_id (FK), doctor\_id (FK).

#### 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).

### Медична система для збереження даних пацієнтів





## 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:M 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 М:1 (Належить лікарні)
[ForeignKey("HospitalId")]
public virtual Hospital Hospital { get; set; }

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

    // Зв'язок М: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 ApplicationDbContext();
    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 ApplicationDbContext();
    // 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 ApplicationDbContext();
    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 ApplicationDbContext();
    // 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()
};
```

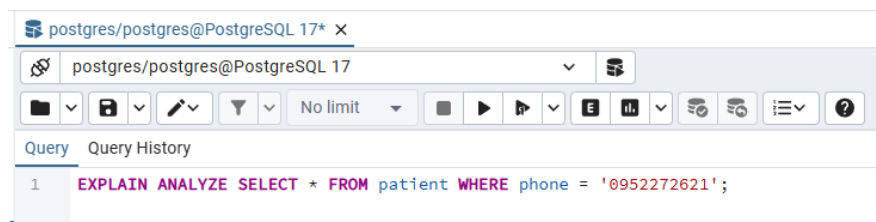
## Завдання №2. Засоби прискорення роботи (Індекси)

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.

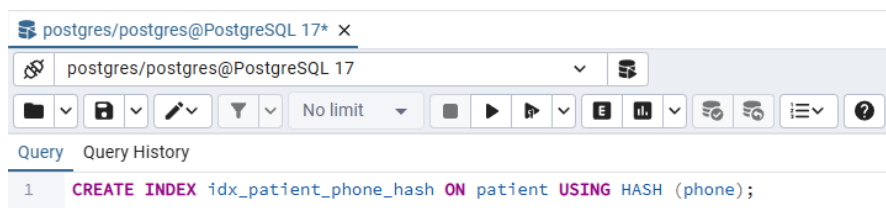
**SQL-запит:**



### Результати виконання:

Data Output	Messages	Notifications
<div> <div>QUERY PLAN</div> <div>text</div> <div> 1 Seq Scan on patient (cost=0.00..224.00 rows=1 width=47) (actual time=0.926..0.926 rows=1 loops=1)  2 Filter: ((phone).text = '0952272621'::text)  3 Rows Removed by Filter: 9999  4 Planning Time: 0.774 ms  5 Execution Time: 0.934 ms </div> </div>		
Total rows: 5	Query complete 00:00:00.126	

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



Data Output	Messages	Notifications
<div> <div> <div>SQL</div> </div> </div>		
<div> <div>QUERY PLAN</div> <div>text</div> <div> <div>1</div> <div>Index Scan using idx_patient_phone_hash on patient (cost=0.00..8.02 rows=1 width=47) (actual time=0.011..0.011 rows=1 loops=...</div> </div> <div> <div>2</div> <div>Index Cond: ((phone)::text = '0952272621'::text)</div> </div> <div> <div>3</div> <div>Planning Time: 0.951 ms</div> </div> <div> <div>4</div> <div>Execution Time: 0.022 ms</div> </div> </div>		
Total rows: 4    Query complete 00:00:00.035		

- **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:**

postgres/postgres@PostgreSQL 17* x	
postgres/postgres@PostgreSQL 17	
<div> <div>Query</div> <div>Query History</div> </div>	
1	EXPLAIN ANALYZE SELECT * FROM patient WHERE phone > '0500000000';

**Результати виконання:**

Data Output	Messages	Notifications
<div> <div>SQL</div> </div>		
<div> <div>QUERY PLAN</div> <div>text</div> <div> <div>1</div> <div>Seq Scan on patient (cost=0.00..224.00 rows=9999 width=47) (actual time=0.009..2.576 rows=10000 loops=...</div> </div> <div> <div>2</div> <div>Filter: ((phone)::text &gt; '0500000000'::text)</div> </div> <div> <div>3</div> <div>Planning Time: 0.081 ms</div> </div> <div> <div>4</div> <div>Execution Time: 2.835 ms</div> </div> </div>		
Total rows: 4    Query complete 00:00:00.050		

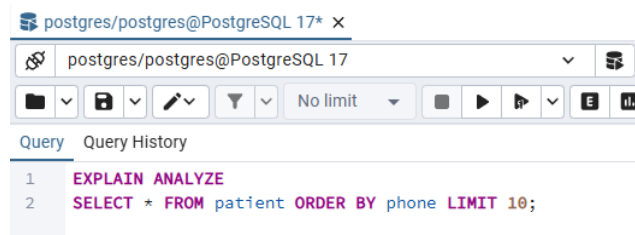
- Планувальник запитів **проігнорував** створений Hash-індекс `idx_patient_phone_hash`.
- Було виконано Seq Scan (повне сканування таблиці).
- Час виконання: **2.835 ms**.

**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:**

Data Output	
Messages	
Notifications	
QUERY PLAN	
1	Limit (cost=415.10..415.12 rows=10 width=47) (actual time=2.984..2.985 rows=10 loops=1)
2	→ Sort (cost=415.10..440.10 rows=10000 width=47) (actual time=2.983..2.984 rows=10 loops=1)
3	Sort Key: phone
4	Sort Method: top-N heapsort Memory: 27kB
5	→ Seq Scan on patient (cost=0.00..199.00 rows=10000 width=47) (actual time=0.010..0.389 rows=10000 loops=1)
6	Planning Time: 0.059 ms
7	Execution Time: 2.997 ms

- 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

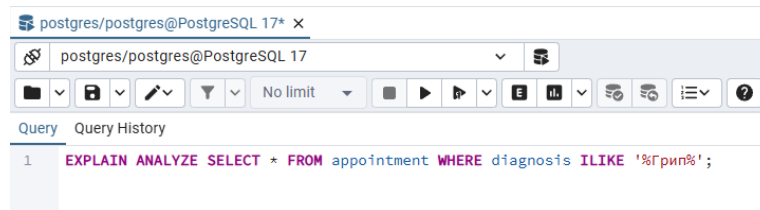


прискорення операцій сортування (ORDER BY) або пошуку діапазонів (>, <, BETWEEN). У таких випадках доцільніше використовувати B-Tree індекс.

#### 4. GIN Index (Generalized Inverted Index)

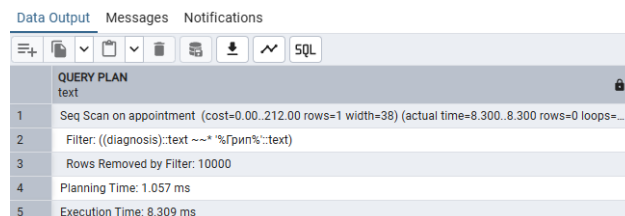
**Сценарій:** Пошук записів, де діагноз містить певний підрядок (наприклад, "Грип"). Стандартні індекси (B-Tree) неефективні для запитів з LIKE '%...%'.

**SQL-запит:**



```
postgres/postgres@PostgreSQL 17* x
postgres/postgres@PostgreSQL 17
Query History
1 EXPLAIN ANALYZE SELECT * FROM appointment WHERE diagnosis ILIKE '%Грип%';
```

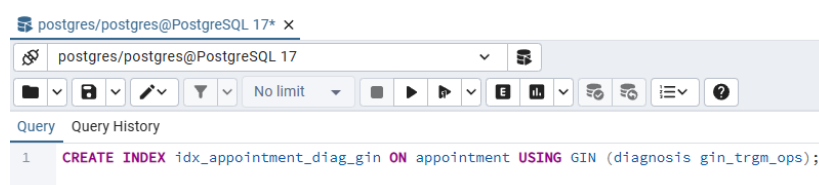
**Результати виконання:**



	QUERY PLAN
1	Seq Scan on appointment (cost=0.00..212.00 rows=1 width=38) (actual time=8.300..8.300 rows=0 loops=...
2	Filter: ((diagnosis)::text ~* '%Грип%':text)
3	Rows Removed by Filter: 10000
4	Planning Time: 1.057 ms
5	Execution Time: 8.309 ms

Total rows: 5    Query complete 00:00:00.045

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



```
postgres/postgres@PostgreSQL 17* x
postgres/postgres@PostgreSQL 17
Query History
1 CREATE INDEX idx_appointment_diag_gin ON appointment USING GIN (diagnosis gin_trgm_ops);
```

**SQL**

CREATE INDEX idx\_appointment\_diag\_gin ON appointment USING GIN (diagnosis gin\_trgm\_ops);

Data Output		Messages	Notifications
<div> <div>SQL</div> </div>			
<div> <div>QUERY PLAN</div> <div>text</div> </div>			
1	Bitmap Heap Scan on appointment (cost=17.32..21.33 rows=1 width=38) (actual time=0.023..0.023 rows=0 loops=1)		
2	Recheck Cond: ((diagnosis):text ~* "%грин%":text)		
3	-> Bitmap Index Scan on idx_appointment_diag_gin (cost=0.00..17.32 rows=1 width=0) (actual time=0.012..0.012 rows=0 loop...)		
4	Index Cond: ((diagnosis):text ~* "%грин%":text)		
5	Planning Time: 1.152 ms		
6	Execution Time: 0.050 ms		

Total rows: 6 Query complete 00:00:00.033

- **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:**

postgres/postgres@PostgreSQL 17* X	
postgres/postgres@PostgreSQL 17	
<div> <div>No limit</div> </div>	
<div> <div>Query</div> <div>Query History</div> </div>	
1	EXPLAIN ANALYZE
2	SELECT diagnosis, COUNT(*)
3	FROM appointment
4	WHERE diagnosis ILIKE '%грин%'
5	GROUP BY diagnosis;

**Результати виконання:**

Data Output		Messages	Notifications
<div> <div>SQL</div> </div>			
<div> <div>QUERY PLAN</div> <div>text</div> </div>			
1	GroupAggregate (cost=21.34..21.36 rows=1 width=22) (actual time=0.030..0.030 rows=0 loops=1)		
2	Group Key: diagnosis		
3	-> Sort (cost=21.34..21.34 rows=1 width=14) (actual time=0.029..0.030 rows=0 loops=1)		
4	Sort Key: diagnosis		
5	Sort Method: quicksort Memory: 25kB		
6	-> Bitmap Heap Scan on appointment (cost=17.32..21.33 rows=1 width=14) (actual time=0.015..0.015 rows=0 loops=1)		
7	Recheck Cond: ((diagnosis):text ~* "%грин%":text)		
8	-> Bitmap Index Scan on idx_appointment_diag_gin (cost=0.00..17.32 rows=1 width=0) (actual time=0.009..0.009 rows=0 loops=1)		
9	Index Cond: ((diagnosis):text ~* "%грин%":text)		
10	Planning Time: 0.126 ms		
11	Execution Time: 0.052 ms		

Total rows: 11 Query complete 00:00:00.043

- 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();

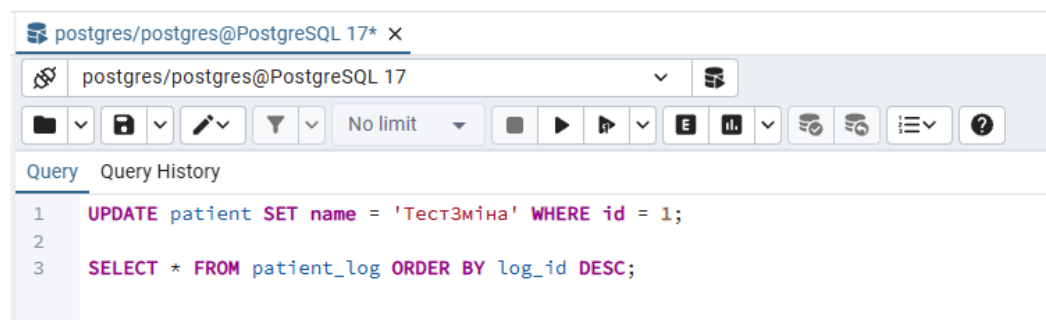
```

## 3. Тестування та результати

### Сценарій 1: Перевірка оновлення даних (UPDATE)

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

*Команда:*



The screenshot shows a PostgreSQL client window titled 'postgres/postgres@PostgreSQL 17\*'. The query editor contains the following SQL commands:

```

1 UPDATE patient SET name = 'ТестЗміна' WHERE id = 1;
2
3 SELECT * FROM patient_log ORDER BY log_id DESC;

```

The results pane shows the output of the second query, displaying a single row from the patient\_log table.

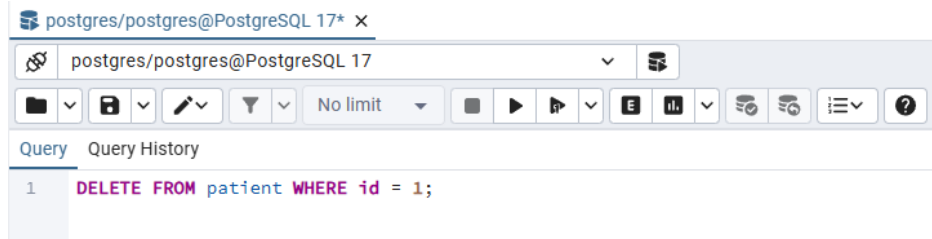
Результат (зміни в таблиці patient\_log):

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	UPDATE	2025-11-24 11:29:41.674251	postgres	1	Дмитро	Бондаренко	0956055196

## Сценарій 2: Перевірка обробки виключних ситуацій (Exception)

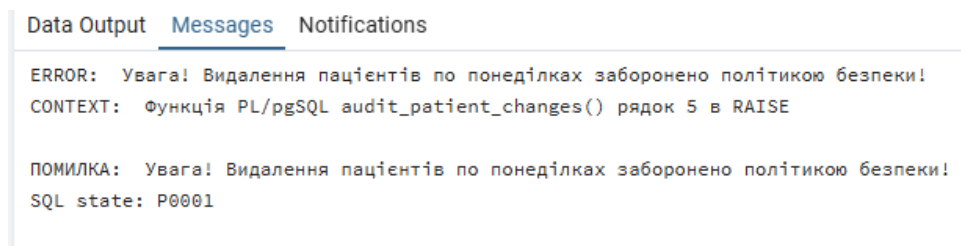
Спроба видалення запису в заборонений день (понеділок). Тригер перериває транзакцію повідомленням про помилку.

*Команда:*



The screenshot shows a PostgreSQL client window with the title 'postgres/postgres@PostgreSQL 17\* x'. The address bar shows 'postgres/postgres@PostgreSQL 17'. The query editor contains the command: `1 DELETE FROM patient WHERE id = 1;`

Результат (повідомлення про помилку):



The screenshot shows the 'Messages' tab of the PostgreSQL client. It displays an error message: `ERROR: Увага! Видалення пацієнтів по понеділках заборонено політикою безпеки!`. The context is: `CONTEXT: Функція PL/pgSQL audit_patient_changes() рядок 5 в RAISE`. Below the error, it says: `ПОМИЛКА: Увага! Видалення пацієнтів по понеділках заборонено політикою безпеки!` and `SQL state: P0001`.

## Сценарій 3: Успішне видалення (DELETE)

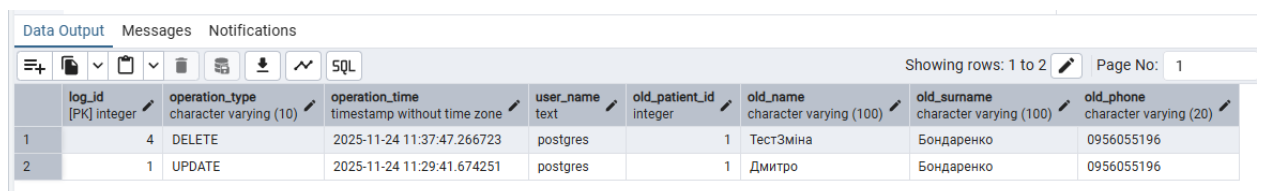
Після зняття часового обмеження виконується видалення пацієнта. Попередньо видаляються пов'язані записи в таблиці appointment, щоб уникнути помилки FK (Foreign Key Constraint).

*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):



The screenshot shows the 'Data Output' tab of the PostgreSQL client. It displays the results of a query on the 'patient\_log' table. The table has 9 columns: log\_id, operation\_type, operation\_time, user\_name, old\_patient\_id, old\_name, old\_surname, and old\_phone. There are two rows of data.

log_id	operation_type	operation_time	user_name	old_patient_id	old_name	old_surname	old_phone
1	DELETE	2025-11-24 11:37:47.266723	postgres	1	ТестЗміна	Бондаренко	0956055196
2	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).

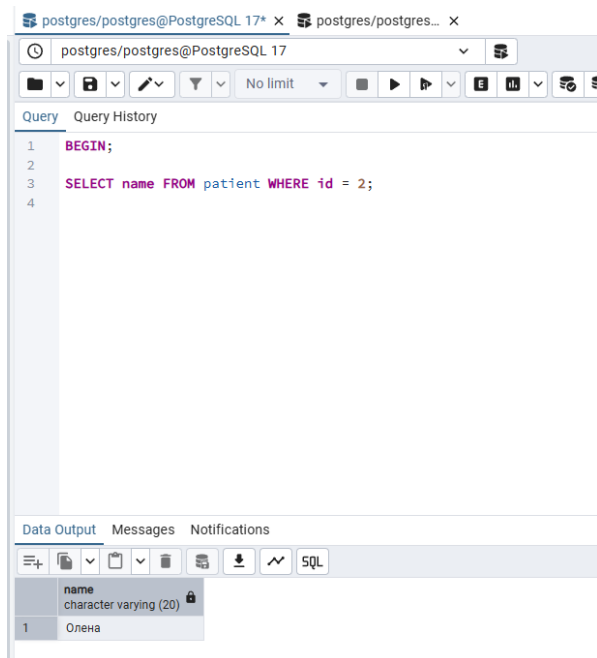
## 1. Рівень ізоляції READ COMMITTED (За замовчуванням)

Феномен: Неповторюване читання (Non-repeatable Read).

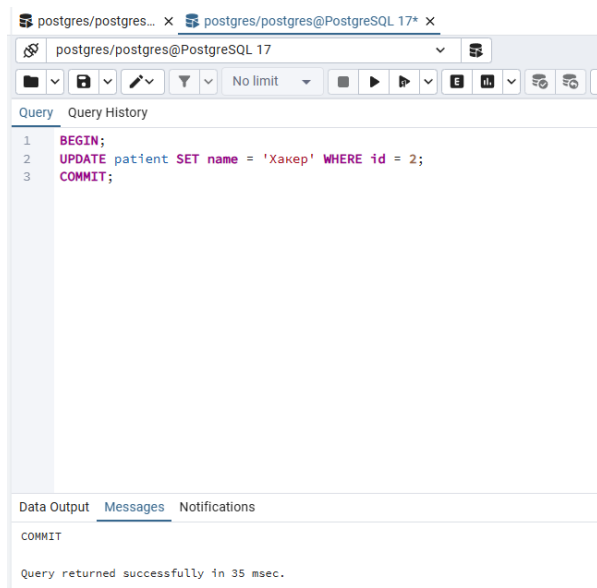
Опис: Транзакція бачить зміни, зафіксовані іншими транзакціями, безпосередньо під час свого виконання. Це означає, що повторний запит до того ж рядка в межах однієї транзакції може повернути інші дані.

### Хід виконання:

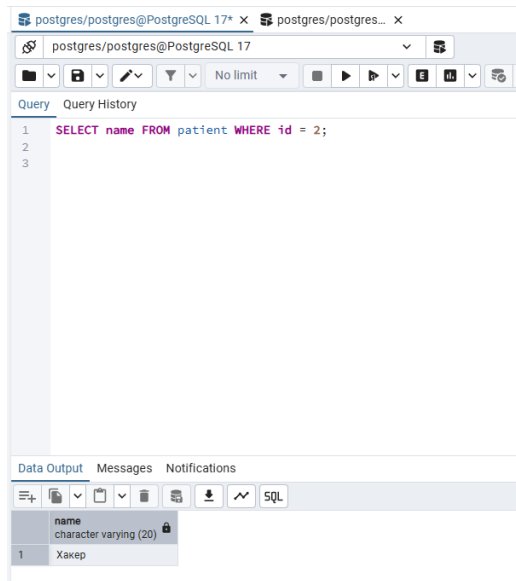
*Транзакція А (Вікно 1):*



*Транзакція Б (Вікно 2):*



## Транзакція А (Вікно 1):



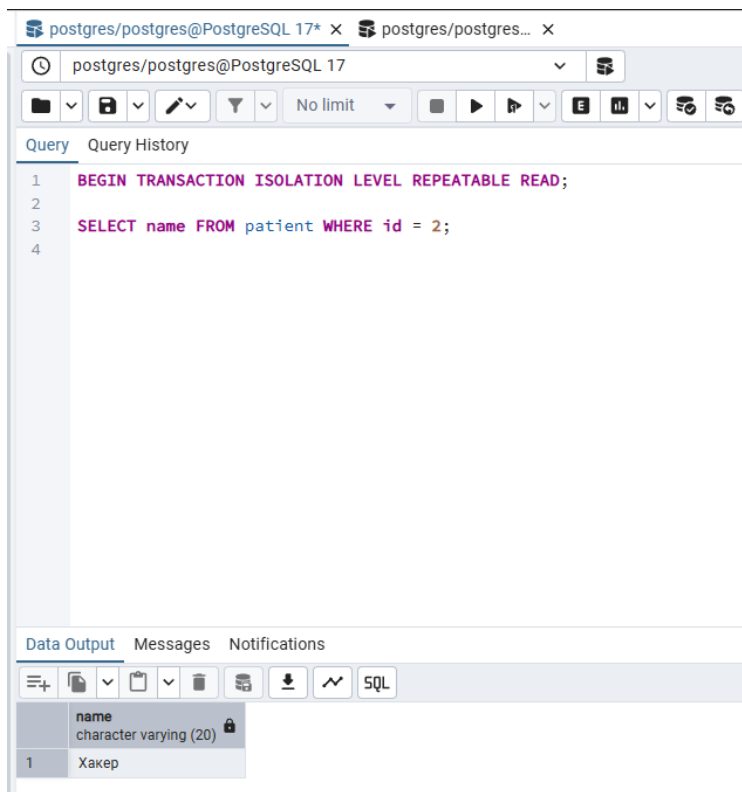
## 2. Рівень ізоляції REPEATABLE READ

Мета: Уникнення феномену неповторюваного читання.

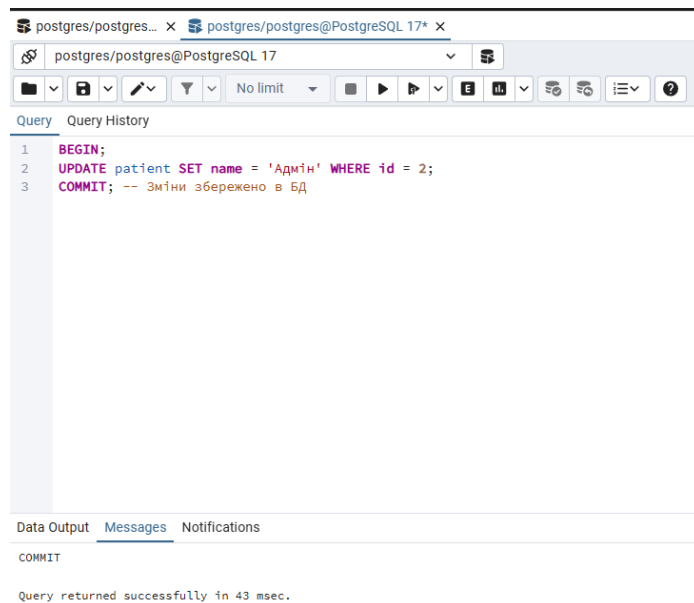
Опис: На цьому рівні транзакція "бачить" стан бази даних (snapshot) на момент свого початку. Будь-які зміни, зафіксовані іншими транзакціями після цього моменту, ігноруються.

### Хід виконання:

## Транзакція А (Вікно 1):



## Транзакція Б (Вікно 2):



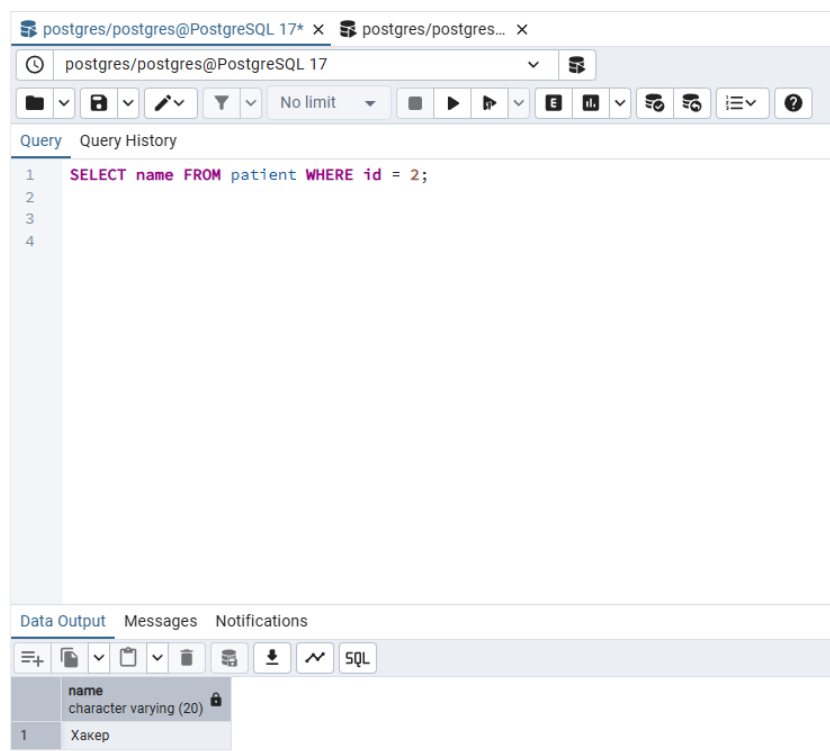
The screenshot shows a PostgreSQL client interface with a query editor and a messages pane. The query editor contains a transaction commit sequence. The messages pane shows the successful execution of the commit.

```
1 BEGIN;  
2 UPDATE patient SET name = 'Адмін' WHERE id = 2;  
3 COMMIT; -- зміни збережено в БД
```

COMMIT

Query returned successfully in 43 msec.

## Транзакція А (Вікно 1):



The screenshot shows a PostgreSQL client interface with a query editor and a data output pane. The query editor contains a SELECT statement. The data output pane shows the result of the query.

```
1 SELECT name FROM patient WHERE id = 2;  
2  
3  
4
```

name
Хакер

## 3. Рівень ізоляції SERIALIZABLE

Мета: Запобігання аномаліям серіалізації.

Опис: Найсуворіший рівень ізоляції. СУБД емує 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):*

The screenshot shows the pgAdmin interface. At the top, there are two browser tabs: 'postgres/postgres@PostgreSQL 17\*' and 'postgres/postgres...'. Below the tabs is a search bar containing 'postgres/postgres@PostgreSQL 17'. A toolbar with various icons is visible. The main area is titled 'Query' and 'Query History'. The SQL editor contains the following query:

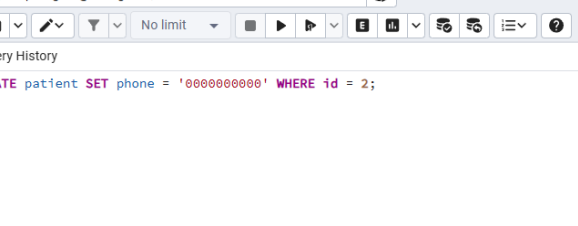
```
1 BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
2
3 SELECT * FROM patient WHERE id = 2;
```

At the bottom, the 'Data Output' section shows a table with 5 columns: 'id [PK] integer', 'name character varying (20)', 'surname character varying (20)', 'day of birth date', and 'phone character varying (10)'. The table contains one row of data:

id [PK] integer	name character varying (20)	surname character varying (20)	day of birth date	phone character varying (10)
1	Админ	Иванов	1995-09-04	0959401113

*Транзакція Б (Вікно 2):*

*Транзакція А (Вікно 1):*



The screenshot shows the pgAdmin interface with the following components:

- Top Bar:** Displays the connection name "postgres/postgres@PostgreSQL 17\*" and the server name "postgres/postgres@PostgreSQL 17".
- Toolbar:** Contains icons for file operations (new, open, save, delete), query execution (run, refresh, stop), and other utilities (search, help).
- Query Editor:** The main area where the SQL query is entered. The query is:
 

```
1 UPDATE patient SET phone = '0000000000' WHERE id = 2;
```
- Bottom Panel:** Shows the execution results. It includes tabs for "Data Output", "Messages", and "Notifications". The "Messages" tab is active, displaying an error message:
 

```
ERROR: не вдалося серіалізувати доступ через паралельне оновлення
```

 Below the error, it also shows the SQL state:
 

```
SQL state: 40001
```

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.