Національний технічний університет України

“Київський політехнічний інститут ім. Ігоря Сікорського”

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп’ютерних систем

ЛАБОРАТОРНА РОБОТА №2

з дисципліни

«Бази даних та засоби управління»

ТЕМА: «Засоби оптимізації роботи СУБД PostgreSQL»

Група: КВ-31

Виконала: Галактіонова А.

Оцінка:

Київ – 2025

Звіт

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’єктнореляційної 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 №4
Види індексів: GIN, BRIN Умови для тригера: after delete, insert.

Пункт №1 :

«Платформа для зберігання історій хвороб пацієнтів»

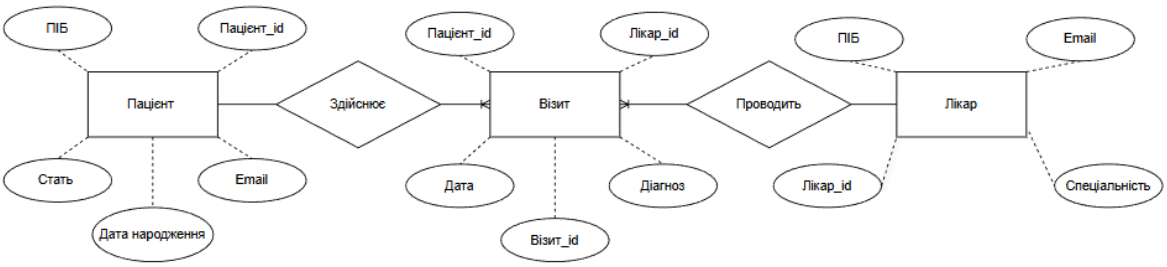
Перелік сутностей і опис їх призначення:
Patient (Пацієнт), сутність призначена для збереження даних про пацієнтів клініки – ПІБ, дата народження, стать та контактна інформація (email).

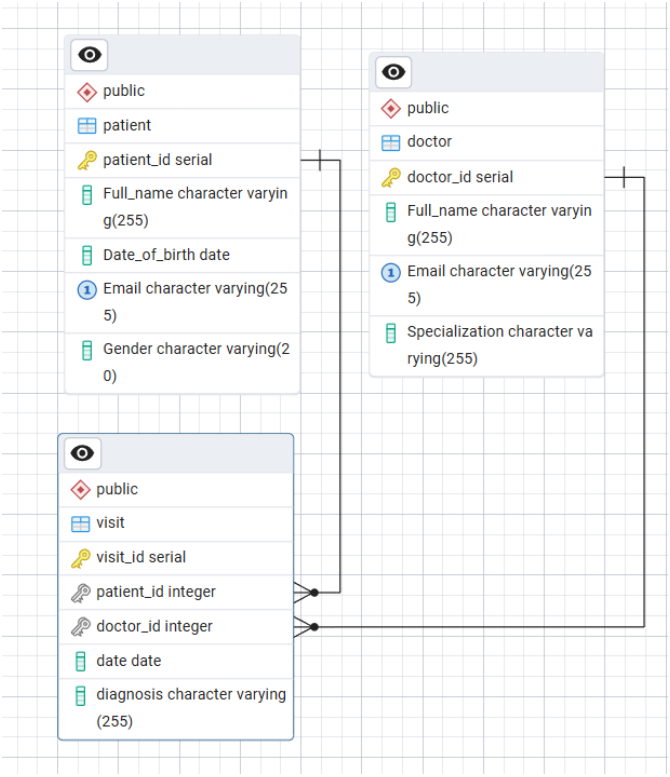
Doctor (Лікар), сутність призначена для збереження даних про лікарів, які працюють у клініці – ПІБ, спеціалізація та контактна інформація (email).

Visit (Візит), сутність призначена для збереження даних про факти візитів пацієнтів до лікарів, включаючи дату візиту та встановлений діагноз.

Опис зв'язків:
Один пацієнт може мати довільну кількість візитів до різних лікарів. Один лікар, у свою чергу, приймає багато пацієнтів та проводить багато візитів. Кожен візит однозначно пов'язує одного конкретного пацієнта з одним конкретним лікарем у певний день та фіксує діагноз.

Платформа для зберігання історій хвороб пацієнтів.
-пацієнт (ПІБ, пошта, дата народження, стать)
-лікар (ПІБ, пошта, кваліфікація)
-візиті (дата візиту, назва хвороби)





Розроблені класи-сутності для об’єктів-сутностей:

```
class Patient(Base):
    __tablename__ = 'patient'

    patient_id = Column(Integer, primary_key=True)
    full_name = Column(String(255), nullable=False)
    date_of_birth = Column(Date, nullable=False)
    email = Column(String(255), unique=True, nullable=False)
    gender = Column(String(20), nullable=False)

    visits = relationship("Visit", back_populates="patient", cascade="all,
delete")

class Doctor(Base):
    __tablename__ = 'doctor'

    doctor_id = Column(Integer, primary_key=True)
    full_name = Column(String(255), nullable=False)
    email = Column(String(255), unique=True, nullable=False)
    specialization = Column(String(255), nullable=False)

    visits = relationship("Visit", back_populates="doctor")

class Visit(Base):
    __tablename__ = 'visit'

    visit_id = Column(Integer, primary_key=True)
```

```
    patient_id = Column(Integer, ForeignKey('patient.patient_id'),
nullable=False)
    doctor_id = Column(Integer, ForeignKey('doctor.doctor_id'), nullable=False)
    date = Column(Date, nullable=False)
    diagnosis = Column(String(255), nullable=False)
    symptoms = Column(ARRAY(String), nullable=True)

    patient = relationship("Patient", back_populates="visits")
    doctor = relationship("Doctor", back_populates="visits")
```

Для реалізації об'єктно-реляційної проекції (ORM) було використано бібліотеку SQLAlchemy. Розроблено класи-сутності (Patient, Doctor, Visit), які відображають структуру таблиць бази даних та зв'язки між ними (One-to-Many).

Інтерфейси функцій модуля «Модель» залишено без змін, що забезпечує сумісність із попередньою версією програми (РГР). Зміни торкнулися лише внутрішньої реалізації, де прямі SQL-запити замінено на методи ORM. Повний код реалізації наведено у файлі model.py у репозиторії.

Пункт 2.

Аналіз та оптимізація запитів за допомогою індексів

Для дослідження ефективності індексування було згенеровано тестовий набір даних обсягом 1 000 000 записів у таблиці `visit`. Метою дослідження є порівняння часу виконання запитів `SELECT` до та після створення індексів типів GIN та BRIN.

```
2   TRUNCATE TABLE visit CASCADE;
3
4   INSERT INTO visit (patient_id, doctor_id, date, diagnosis, symptoms)
5   SELECT
6       1, 1,
7       '2020-01-01'::date + (i % 3000 * interval '1 day'),
8       'Diagnosis #' || i,
9       CASE
10          WHEN i % 100 = 0 THEN ARRAY['critical', 'surgery']
11          WHEN i % 2 = 0 THEN ARRAY['cough', 'fever']
12          ELSE ARRAY['pain', 'fracture']
13      END
14  FROM generate_series(1, 1000000) AS i;
```

Data Output | Messages | Notifications

```
INSERT 0 1000000

Query returned successfully in 1 min 21 secs.INSERT 0 1000000

Query returned successfully in 1 min 22 secs.
```

2.1. Використання GIN індексу

Було протестовано запит на пошук записів, що містять певний елемент у масиві `symptoms`.

Запит без використання індексу:

Query | Query History

```
1  DROP INDEX IF EXISTS idx_gin_symptoms;
2
3  EXPLAIN ANALYZE SELECT * FROM visit WHERE symptoms @> ARRAY['surgery']::varchar[];
```

Data Output | Messages | Notifications

Showing rows:

	QUERY PLAN text
1	Gather (cost=1000.00..20634.63 rows=10933 width=76) (actual time=6.574..284.487 rows=10000 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on visit (cost=0.00..18541.33 rows=4555 width=76) (actual time=1.285..166.663 rows=3333 loop…
5	Filter: (symptoms @> '{surgery}'::character varying[])
6	Rows Removed by Filter: 330000
7	Planning Time: 1.658 ms
8	Execution Time: 285.150 ms

Як видно зі скріншоту, пошук без індексу виконувався за допомогою повного сканування таблиці (`Parallel Seq Scan`) і зайняв значний час (близько 285,150 мс), що є неприйнятним для високонавантажених систем.

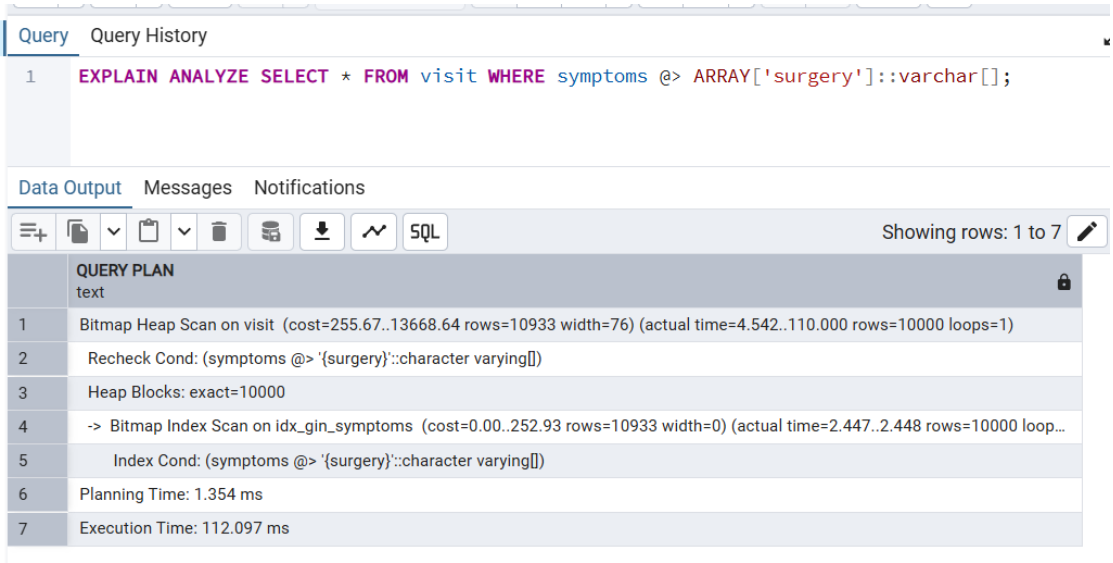
Query | Query History

```
1  CREATE INDEX idx_gin_symptoms ON visit USING GIN (symptoms);
```

Data Output | Messages | Notifications

```
CREATE INDEX

Query returned successfully in 3 secs 278 msec.
```

Запит з використанням GIN індексу: Було створено GIN індекс командою: CREATE INDEX idx_gin_symptoms ON visit USING GIN (symptoms);



```
EXPLAIN ANALYZE SELECT * FROM visit WHERE symptoms @> ARRAY['surgery']::varchar[];
```

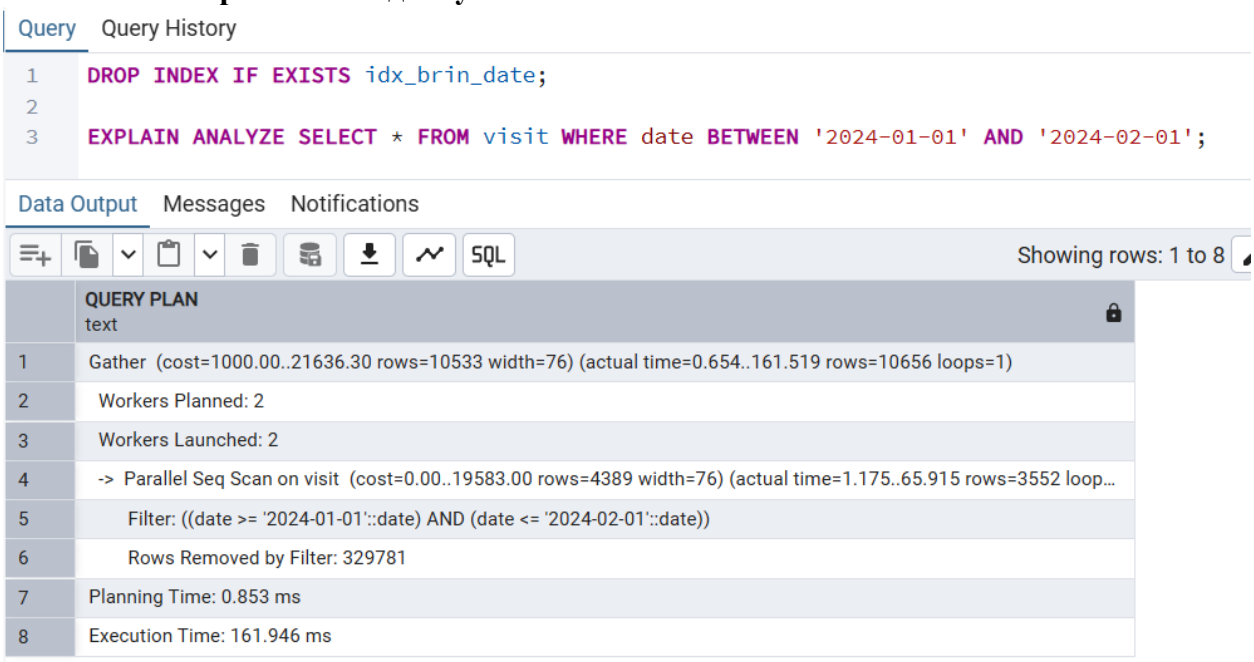
	QUERY PLAN text
1	Bitmap Heap Scan on visit (cost=255.67..13668.64 rows=10933 width=76) (actual time=4.542..110.000 rows=10000 loops=1)
2	Recheck Cond: (symptoms @> '{surgery}'::character varying[])
3	Heap Blocks: exact=10000
4	-> Bitmap Index Scan on idx_gin_symptoms (cost=0.00..252.93 rows=10933 width=0) (actual time=2.447..2.448 rows=10000 loop...
5	Index Cond: (symptoms @> '{surgery}'::character varying[])
6	Planning Time: 1.354 ms
7	Execution Time: 112.097 ms

GIN індекс зменшив час виконання запиту до 112,097 мс (прискорення у 2,5 рази). Цей тип індексу добре оптимізований для складених типів даних (масиви, JSONB). Він індексує окремі елементи масиву, що дозволяє миттєво знаходити потрібні записи. Проте варто враховувати, що GIN індекси мають вищу вартість оновлення при вставці нових даних порівняно зі звичайним B-Tree.

Використання BRIN індексу

Було протестовано вибірку даних за діапазоном дат.

Запит без використання індексу:



```
DROP INDEX IF EXISTS idx_brin_date;

EXPLAIN ANALYZE SELECT * FROM visit WHERE date BETWEEN '2024-01-01' AND '2024-02-01';
```

	QUERY PLAN text
1	Gather (cost=1000.00..21636.30 rows=10533 width=76) (actual time=0.654..161.519 rows=10656 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on visit (cost=0.00..19583.00 rows=4389 width=76) (actual time=1.175..65.915 rows=3552 loop...
5	Filter: ((date >= '2024-01-01'::date) AND (date <= '2024-02-01'::date))
6	Rows Removed by Filter: 329781
7	Planning Time: 0.853 ms
8	Execution Time: 161.946 ms

Час виконання склав **161,946** мс при повному переборі таблиці.

Query Query History

```
1  CREATE INDEX idx_brin_date ON visit USING BRIN (date);
```

Data Output Messages Notifications

```
CREATE INDEX

Query returned successfully in 268 msec.
```

Query Query History

```
1  EXPLAIN ANALYZE SELECT * FROM visit WHERE date BETWEEN '2024-01-01' AND '2024-02-01';
```

Data Output Messages Notifications

Showing rows: 1 to 8

	QUERY PLAN text
1	Gather (cost=1000.00..21636.30 rows=10533 width=76) (actual time=0.559..158.864 rows=10656 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on visit (cost=0.00..19583.00 rows=4389 width=76) (actual time=0.474..60.512 rows=3552 loop...
5	Filter: ((date >= '2024-01-01'::date) AND (date <= '2024-02-01'::date))
6	Rows Removed by Filter: 329781
7	Planning Time: 0.125 ms
8	Execution Time: 159.330 ms

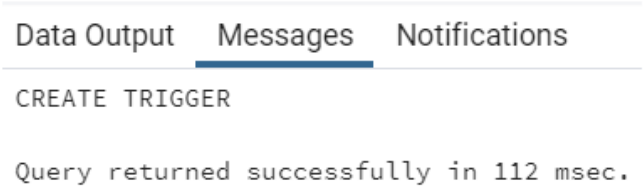
З використанням BRIN індексу час виконання зменшився до 159,330 мс. BRIN-індекси призначені для дуже великих таблиць, де дані мають природну кореляцію за фізичним розташуванням, як у нашому випадку з датами. BRIN зберігає лише мінімальне та максимальне значення для блоків сторінок, що дозволяє відсіювати великі шматки даних, які не потрапляють у діапазон пошуку. Цей індекс займає надзвичайно мало місця на диску порівняно з B-Tree, хоча і є дещо повільнішим за нього при пошуку точних значень.

Пункт 3.

Запит для створення тригеру:

```
CREATE TABLE IF NOT EXISTS audit_log (
   log_id SERIAL PRIMARY KEY,
   operation_type  VARCHAR(20),
   record_id INT,
   log_time TIMESTAMP DEFAULT NOW(),
   details TEXT
);

CREATE OR REPLACE FUNCTION audit_trigger_func()
RETURNS TRIGGER AS $$
DECLARE
   rec_check RECORD;
   dummy_counter  INT := 0;
   cur_audit CURSOR FOR SELECT log_id FROM audit_log  LIMIT 5;
```

```
BEGIN
  BEGIN

    IF (TG_OP = 'INSERT') THEN
      INSERT INTO audit_log(operation_type, record_id, details)
      VALUES ('INSERT', NEW.visit_id, 'Added visit with diagnosis: ' || NEW.diagnosis);

    ELSIF (TG_OP = 'DELETE') THEN
      INSERT INTO audit_log(operation_type, record_id, details)
      VALUES ('DELETE', OLD.visit_id, 'Deleted visit with diagnosis: ' || OLD.diagnosis);

      OPEN cur_audit;
      LOOP
        FETCH cur_audit INTO rec_check;
        EXIT WHEN NOT FOUND;
        dummy_counter := dummy_counter + 1;
      END LOOP;
      CLOSE cur_audit;
    END IF;

  EXCEPTION
    WHEN OTHERS THEN
      RAISE NOTICE 'Error in trigger execution: %', SQLERRM;
  END;

  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_audit_change
AFTER INSERT OR DELETE ON visit
FOR EACH ROW
EXECUTE FUNCTION audit_trigger_func();
```



Для демонстрації роботи тригера було виконано наступні SQL-запити:

1. **Додавання запису (INSERT):**

Query | Query History

```
1  INSERT INTO visit (patient_id, doctor_id, date, diagnosis, symptoms)
2  VALUES (1, 1, '2025-12-31', 'TEST TRIGGER', ARRAY['test']);
```

Data Output | Messages | Notifications

```
INSERT 0 1

Query returned successfully in 99 msec.
```

Для перевірки працездатності тригера було виконано наступні тестові запити: додавання нового візиту та його подальше видалення.

Query | Query History

```
1  INSERT INTO visit (patient_id, doctor_id, date, diagnosis, symptoms)
2  VALUES (1, 1, '2025-12-31', 'TRIGGER TEST', ARRAY['none']);
3
4  DELETE FROM visit WHERE diagnosis = 'TRIGGER TEST';
5
6  SELECT * FROM audit_log ORDER BY log_id DESC LIMIT 5;
```

Data Output | Messages | Notifications

Showing rows: 1 to 5

	log_id [PK] integer	operation_type character varying (20)	record_id integer	log_time timestamp without time zone	details text
1	12	DELETE	2500005	2025-12-11 00:42:20.72582	Deleted diagnosis: TRIGGER TE...
2	11	DELETE	2500005	2025-12-11 00:42:20.72582	Deleted diagnosis: TRIGGER TE...
3	10	INSERT	2500005	2025-12-11 00:42:20.72582	New diagnosis: TRIGGER TEST
4	9	INSERT	2500005	2025-12-11 00:42:20.72582	New diagnosis: TRIGGER TEST
5	8	DELETE	2500004	2025-12-11 00:38:54.070614	Deleted diagnosis: TEST TRIGGER

Пункт 4.

Аналіз рівнів ізоляції транзакцій

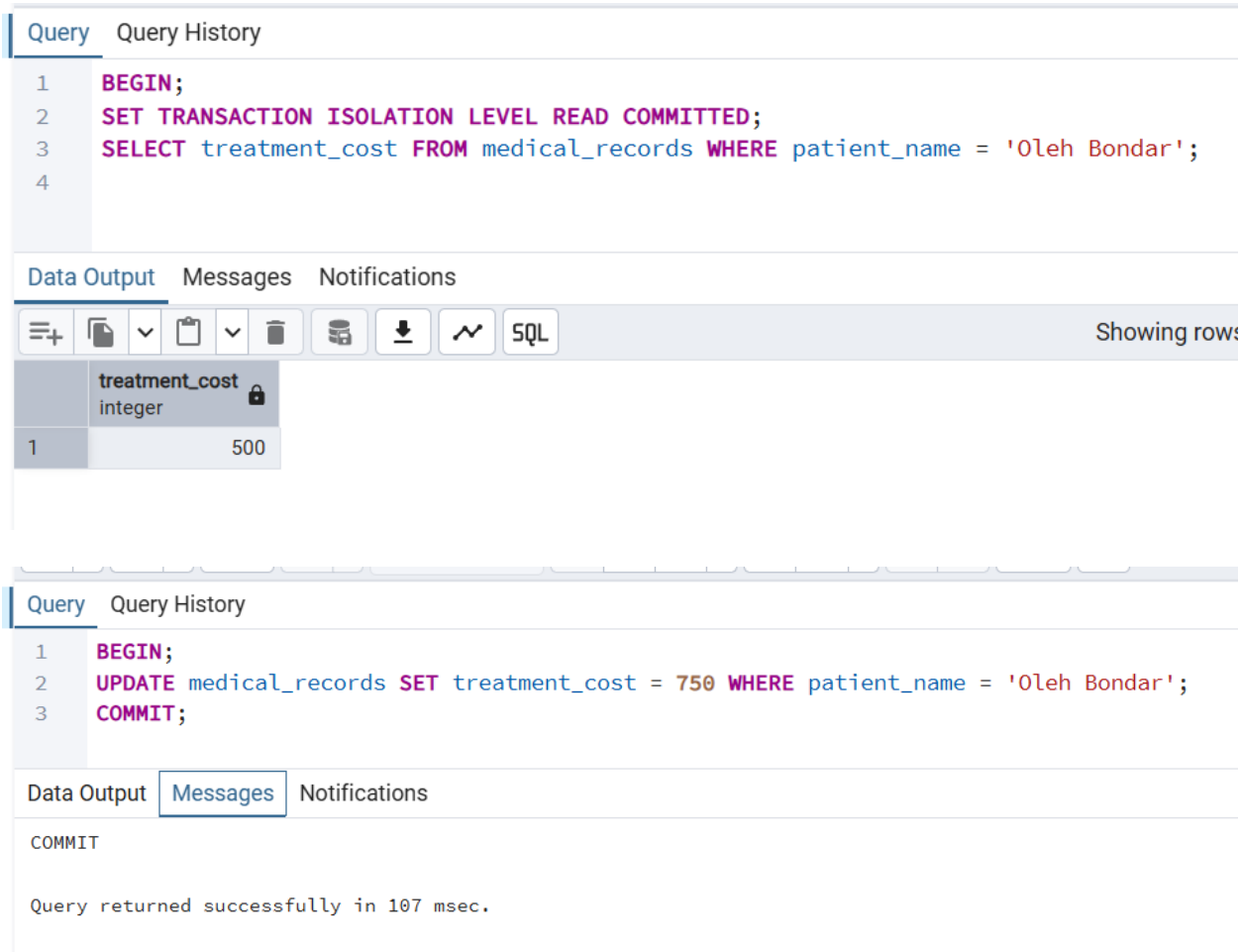
Коли кілька транзакцій працюють з базою одночасно, можуть виникати такі конфлікти (феномени):

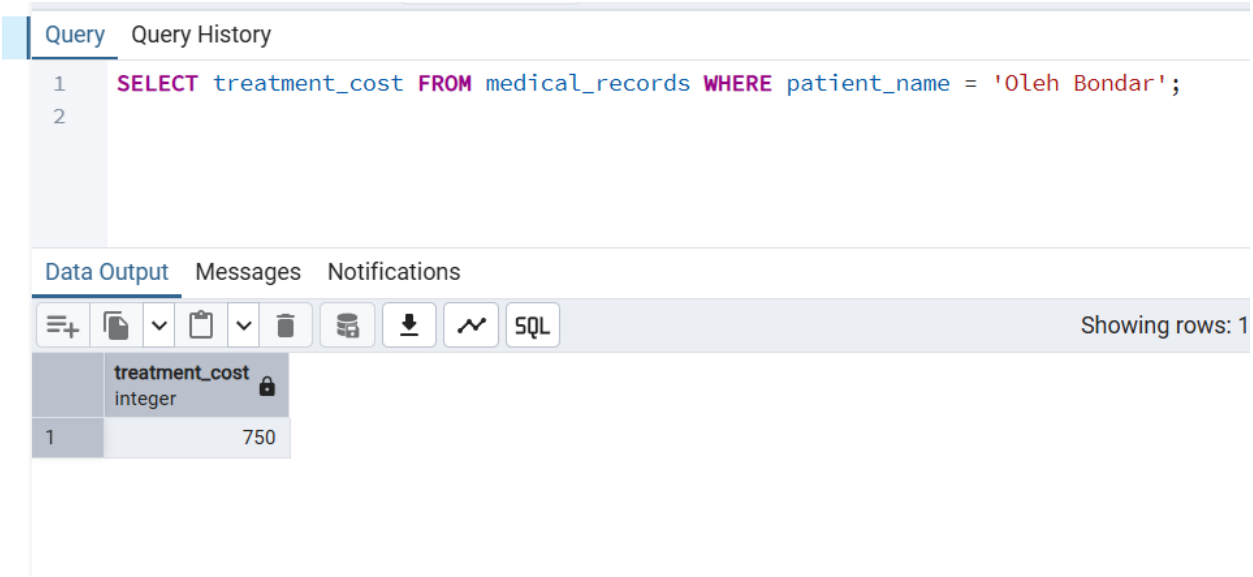
- **Втрачене оновлення (Lost Update):** Це коли двоє користувачів одночасно змінюють одні й ті самі дані, і зміни першого просто стираються, бо другий зберіг свою версію пізніше.
- **«Брудне» читання (Dirty Read):** Коли ми бачимо дані, які інша транзакція змінила, але ще не зберегла (не зробила commit). Якщо вона потім скасує зміни, вийде, що ми працювали з неіснуючими даними.
- **Неповторюване читання (Non-repeatable Read):** Це коли я читаю рядок, бачу одне значення, а через хвилину читаю той самий рядок (у тій же транзакції) — і значення вже інше, бо хтось встиг його змінити.

- **Фантомне читання (Phantom Read):** Схоже на попереднє, але тут змінюються не самі дані, а їх кількість. Наприклад, я роблю вибірку — було 10 записів, роблю ще раз — а їх вже 11, бо хтось додав новий.

Щоб цього уникнути, ми розглянули такі рівні ізоляції:

1. **Read Committed:** Дозволяє бачити тільки ті зміни, які вже точно збережені. Але якщо під час нашої роботи хтось змінить дані, ми це побачимо.
2. **Repeatable Read:** Гарантує стабільність. Навіть якщо хтось змінить дані поки ми працюємо, ми будемо бачити стару версію — ту, що була на момент початку нашої транзакції.
3. **Serializable:** Найсуворіший рівень. Система вибудовує транзакції так, ніби вони йдуть чітко по черзі. Це повністю захищає від будь-яких конфліктів.





Демонстрація рівня READ COMMITTED. У першому вікні було відкрито транзакцію і прочитано вартість лікування (500). У другому вікні паралельна транзакція змінила вартість на 750 і зафіксувала зміни. Повторний запит у першому вікні показав нове значення (750), що демонструє феномен неповторюваного читання.

У пункті 4 представлено аналіз лише рівня *Read Committed*. Решту сценаріїв (Repeatable Read, Serializable) не вдалося коректно відтворити через технічні нюанси роботи з транзакціями у pgAdmin. У зв'язку з цим, а також через брак часу, прошу оцінити роботу за результатами виконаних пунктів.