НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

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

**Лабораторна робота № 2**

з дисципліни

**«Бази даних та засоби управління»**

**«Засоби оптимізації роботи СУБД PostgreSQL»**

Виконав: студент 3 курсу

ФПМ групи КВ-23

Зленко Артем Андрійович

**Київ – 2024**

**Проектування бази даних та ознайомлення з базовими операціями СУБД PostgreSQL**

***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.

**Посилання на Github**: [https://github.com/ZlenkoArtem/BD\_Lab\_2](https://github.com/ZelenkoArtem/BD_Lab_2)

***Вимоги до пункту завдання №1***

Для перетворення функцій, що реалізують запити до об’єктної бази даних, необхідно встановити бібліотеку sqlAlchemy, налаштувати програму на роботу з ORM, розробити класи-сутності для об’єктів-сутностей, представлених відповідними таблицями БД та пов’язаних зв’язками 1:М, М:М та 1:1 виконати опис схеми бази даних. Особливу увагу приділити контролю зовнішніх зв’язків між таблицями засобами ORM.

Замінити виклики запитів мовою SQL на відповідні запити засобами SQLAlchemy по роботі з об’єктами. Обов’язковим є реалізація вставки, вилучення та редагування екземплярів класів-сутностей. Розробка запитів на генерацію даних та пошук екземплярів класів-сутностей вітається, але не є обов’язковою.

Інтерфейси функцій (вхідні та вихідні аргументи функцій модуля “Модель”) мають залишитись без змін.

***Вимоги до пункту завдання №2***

Відповідно до варіанту індексування продемонструвати на прикладах запитів SQL SELECT підвищення швидкодії їх виконання з використанням індексів, а також пояснити чому для деяких випадків індексування використовувати недоцільно. При цьому для наочного представлення слід використати функцію генерування рандомізованих даних з лабораторної роботи №2, створивши необхідну кількість тестових даних. Навести 4-5 прикладів запитів SELECT (із виведенням результуючих даних), що містять фільтрацію, агрегатні функції, групування та сортування (у необхідних комбінаціях).

***Вимоги до пункту завдання №3***

Створити тригер бази даних PostgreSQL відповідно до варіанта. Тригерна функція має включати обробку запису, що модифікується (вставляється або вилучається), умовні оператори, курсорні цикли та обробку виключних ситуацій. Виконати відлагодження тригера при різних вхідних даних, навівши 2-3 приклади його використання.

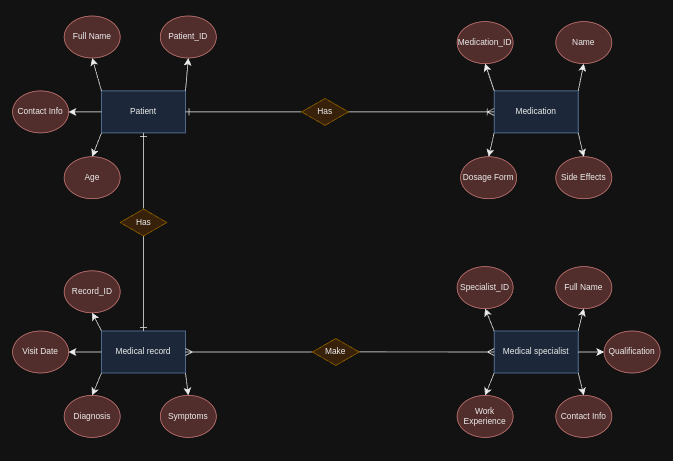
***Вимоги до пункту завдання №4***

Проаналізувати на прикладах використання рівнів ізоляції транзакцій READ COMMITTED, REPEATABLE READ та SERIALIZABLE, продемонструвавши феномени, які виникають, і спосіб їх уникнення завдяки встановленню відповідного рівня ізоляції транзакцій. Для виконання завдання необхідно відкрити дві транзакції у різних вікнах pgAdmin4 і виконати послідовність запитів INSERT, UPDATE або DELETE у обох транзакціях, що доводять наявність або відсутність певних феноменів.

***Варіант №8***

| *№ варіанта* | *Види індексів* | *Умови для тригера* |
| --- | --- | --- |
| *12* | *BTree, GIN* | *after update, insert* |

Графічне подання концептуальної моделі зображено на рисунку 1



*Рис. 1 ER-діаграма, побудована за нотацією Чена*

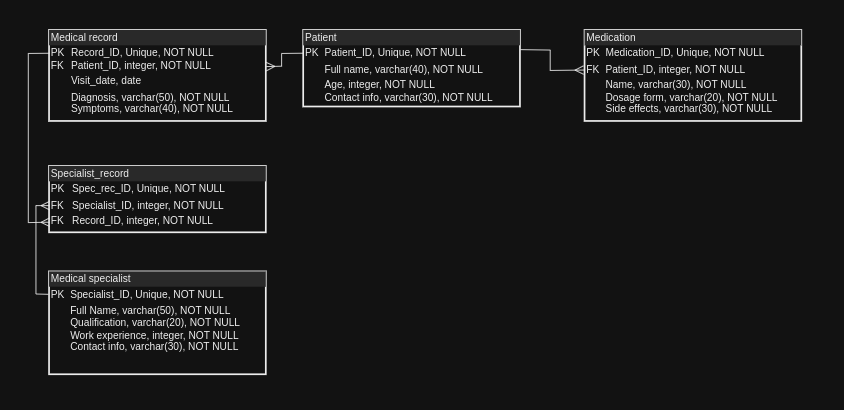
**Перетворення концептуальної моделі на логічну модель та схему бази даних**

1. Сутність Patient перетворено в таблицю Patient
2. Сутність Medical record перетворено в таблицю Medical record
3. Сутність Medical specialist перетворено в таблицю Medical specialist
4. Сутність Medication перетворено в таблицю Medication

В моделі неможливо представити зв’язок N:M але в концептуальній моделі він існує між сутностями Medical specialist i Medical record, для цього вводимо допоміжну таблицю Specialist\_Record

| Сутність | Атрибут | Тип атрибуту |
| --- | --- | --- |
| Patient – користувач, пацієнт | Patient\_ID – ID пацієнта  Full name – ПІБ пацієнта  Contact info – пошта пацієнта  Age – вік пацієнта | integer (числовий)  character varying (рядок)  character varying (рядок)  integer (числовий) |
| Medical record – запис пацієнта, створений медичним працівником | Record\_ID – ID запису  Patient\_ID – ID пацієнта  Visit date – дата візиту пацієнта  Diagnosis – діагноз  Symptoms – симптоми | integer (числовий)  integer (числовий)  date (дата)  character varying (рядок)  character varying (рядок) |
| Medical specialist – медичний працівник, який створює медичний запис | Specialist\_ID – ID спеціаліста  Full name – ПІБ лікаря  Qualification – рівень кваліфікації лікаря(категорія ступінь)  Work experience – стаж роботи  Contact info – пошта лікаря | integer (числовий)  character varying (рядок)  character varying (рядок)  integer (числовий)  character varying (рядок) |
| Medication – ліки, які можуть бути в пацієнта | Medication\_ID – ID ліків Patient\_ID – ID пацієнта  Name - назва ліків  Dosage form – форма випуску(капсули, таблетки, сироп)  Side effects - побічні ефекти | integer (числовий)  integer (числовий)  character varying (рядок)  character varying (рядок)  character varying (рядок) |
| Specialist\_Record – відповідність медичного працівника до медичного запису, який він зробив | Spec\_Rec\_ID – ID запису певного спеціаліста  Specialist\_ID – ID спеціаліста  Record\_ID – ID запису | integer (числовий)  integer (числовий)  integer (числовий) |

**Перетворення розробленої моделі «сутність-зв’язок» у схему бази даних PostgreSQL**



*Рис 2. Схема бази даних у графічному вигляді*

Сутність Patient було перетворено на таблицю Patient. Первинний ключ(ідентифікатор) Id, атрибути: Full name, Age, Contact info.

Сутність Medication було перетворено на таблицю Medication.

Первинний ключ (ідентифікатор) Id, атрибути: Name, Dosage form, Side effects.

Сутність Medical record було перетворено на таблицю Medical record. Первинний ключ (ідентифікатор) Id, атрибути: Visit date, Diagnosis, Symptom.

Сутність Medical specialist було перетворено на таблицю Medical specialist. Первинний ключ (ідентифікатор) Id, атрибути: Full Name, Qualification, Work experience, Contact info.

Було також створено таблицю Specialist\_record. Первинний ключ

(ідентифікатор) Id, атрибути: Specialist\_Id, Record\_Id. Таблиця Specialist\_record була створена для того, щоб утворювати пару

медичний спеціаліст - медичний запис. Можна було б додати в таблицю Specialist поле Record\_Id, яке теоретично б пов’язувало спеціаліста з записом в медичній картці. АЛЕ: якщо б 1 і той самий спеціаліст мав доступ до двох медичних записів, то довелося би мати 2 майже однакові записи в таблиці Specialist. Єдиною відмінністю було б поле Record\_Id. Отже, знову була б виділена пам’ять на всі інші поля (Qualification, Work experience, Contact info). Це вимагало б додаткової пам’яті. А так, завдяки такій таблиці Specialist\_record, реалізована можливість одному спеціалісту мати декілька записів в медичних картках. Це також реалізовує 1НФ, оскільки 1 нормальна форма вимагає відсутні груп полів, які

повторюються. В таблиці Specialist\_Record створено два зовнішніх ключа: FK\_Specialist та FK\_Record. Вони посилаються на поля Id в таблицях Medical specialist та Medical Record відповідно.

Кожен пацієнт має медичну картку і записи в ній. Для отримання цього зв’язку було створено зовнішній ключ FK\_Patient. В таблиці Medical Record створено поле, яке посилається на поле (первинний ключ) Id в таблиці Patient. (1:N)

Кожному пацієнту виписують ліки для лікування. Для отримання цього зв’язку було створено зовнішній ключ FK\_Patient. В таблицю Medication створено поле, яке посилається на поле (первинний ключ) Id в таблиці Patient. (1:N)

**Завдання №1**

У даній лабораторній роботі було реалізовано 5 класів відповідно до 5 існуючих таблиць у розробленій базі даних, а саме:

1. Patient
2. Medication
3. Medical\_record
4. Medical\_specialist
5. Specialist\_record

**Patient**

Цей клас представляє таблицю пацієнтів, з атрибутами, такими як ідентифікатор, ім’я, вік та контактна інформація. Пацієнт має медичний запис та медикаменти. За допомогою зовнішніх ключів можна прив’язати ліки та медичні записи до кожного пацієнта

Програмна реалізація класу

class Patient(Base):

\_\_tablename\_\_ = 'patient'

id = Column(Integer, primary\_key=True)

full\_name = Column(String, nullable=False)

age = Column(Integer, nullable=False)

contact\_info = Column(String, nullable=False)

medical\_records = relationship("MedicalRecord", back\_populates="patient", cascade="all, delete-orphan")

**Medication**

Ця таблиця зберігає дані про медикаменти, які призначаються пацієнтам. Вона має атрибути: ідентифікатор, назва, форма дозування та побічні ефекти. Зовнішній ключ на пацієнта, щоб вказати, який пацієнт отримав ці ліки

Програмна реалізація класу

class Medication(Base):

\_\_tablename\_\_ = 'medication'

id = Column(Integer, primary\_key=True)

id\_patient = Column(Integer, ForeignKey('patient.id'), nullable=False)

name = Column(String, nullable=False)

dosage\_form = Column(String, nullable=False)

side\_effect = Column(String)

**Medical\_record**

Цей клас описує медичні записи пацієнтів, що включають в себе дату візиту, діагноз, симптоми та зв’язок із пацієнтом через зовнішній ключ

Програмна реалізація класу

class MedicalRecord(Base):

\_\_tablename\_\_ = 'medical\_record'

id = Column(Integer, primary\_key=True)

id\_patient = Column(Integer, ForeignKey('patient.id'), nullable=False)

visit\_date = Column(Date, nullable=False)

diagnosis = Column(Text, nullable=False)

symptoms = Column(Text, nullable=True)

patient = relationship("Patient", back\_populates="medical\_records")

**Medical\_specialist**

Цей клас описує медичних спеціалістів, які мають такі атрибути: ім’я, кваліфікація, досвід роботи, контактна інформація

Програмна реалізація класу

class MedicalSpecialist(Base):

\_\_tablename\_\_ = 'medical\_specialist'

id = Column(Integer, primary\_key=True)

full\_name = Column(String, nullable=False)

qualification = Column(String, nullable=False)

work\_experience = Column(Integer, nullable=False)

contact\_info = Column(String, nullable=False)

specialist\_records = relationship("SpecialistRecord", back\_populates="specialist", cascade="all, delete-orphan")

**Specialist\_record**

Цей клас створює зв’язок між спеціалістами та медичними записами. Кожен спеціаліст може мати кілька записів, що дозволяє уникнути дублювання даних

Програмна реалізація класу

class SpecialistRecord(Base):

\_\_tablename\_\_ = 'specialist\_record'

id = Column(Integer, primary\_key=True)

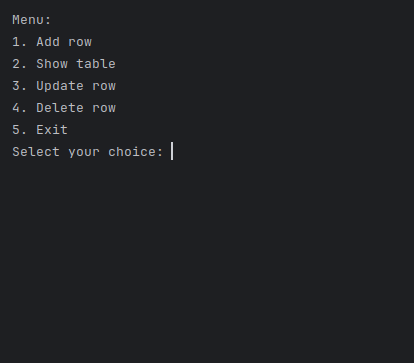
id\_specialist = Column(Integer, ForeignKey('medical\_specialist.id'), nullable=False)

id\_record = Column(Integer, ForeignKey('medical\_record.id'), nullable=False)

specialist = relationship("MedicalSpecialist", back\_populates="specialist\_records")

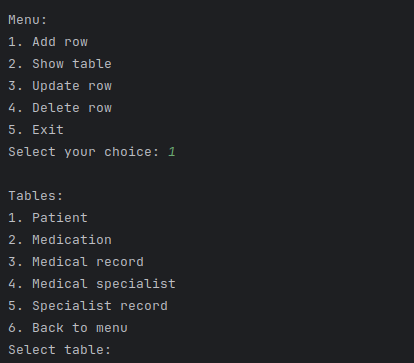
record = relationship("MedicalRecord")

*Меню складається із 5 пунктів, кожен з яких буде розглянуто далі*



1. **Add row (Додати рядок)**

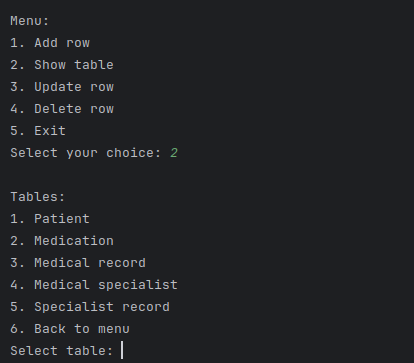
Цей пункт створений для додавання рядка у таблицю. Після його вибору, відкривається список всіх таблиць БД, де потрібно обрати таблицю, до якої хочемо додати рядок:



Після вибору таблиці, користувачу потрібно ввести всі необхідні дані для нового рядка.

1. **Show table (Показ таблиці)**

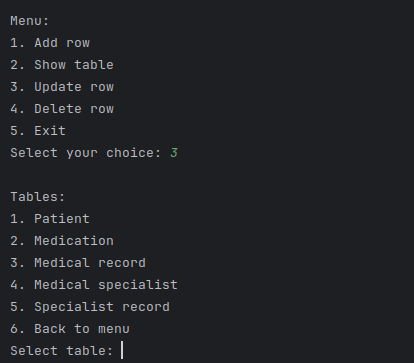
Цей пункт створений для показу таблиць. Після його вибору, відкривається список доступних таблиць БД, де потрібно вибрати таблицю, яку бажаємо побачити.



Після вибору таблиці, мають вивестися всі рядки і стовпці з обраної таблиці БД.

1. **Update row (Редагувати рядок)**

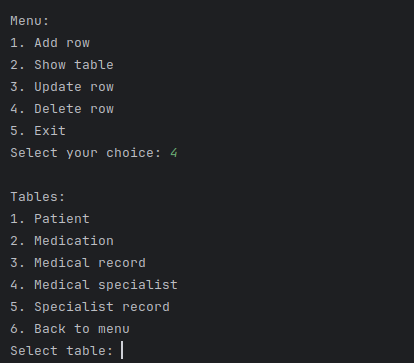
Цей пункт створений для редагування рядків у таблицях. Після вибору цього пункту, відкривається список доступних таблиць, де потрібно вибрати таблицю, в якій бажаємо зробити зміну.



Після вибору таблиці, користувачу потрібно ввести ідентифікатор існуючого рядка в таблиці. Потім записати нові дані для обраного рядка.

1. **Delete row (Видалити рядок)**

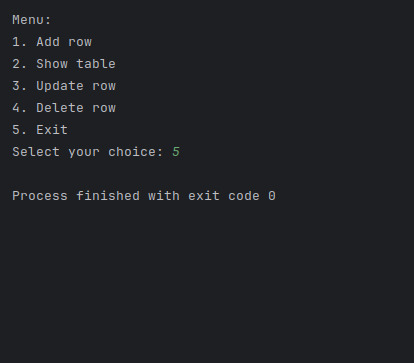
Цей пункт створений для видалення рядків у таблицях. Після вибору цього пункту, відкривається список доступних таблиць БД, де потрібно вибрати таблицю, в якій бажаємо видалити рядок.



Після вибору таблиці, користувачу потрібно ввести ідентифікатор існуючого рядка в таблиці для видалення.

1. **Exit (Вихід)**

Пункт виходу з програми: закривається з’єднання і програма завершується.

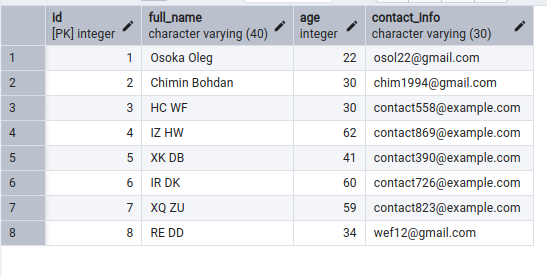


**Приклади запитів у вигляді ORM**

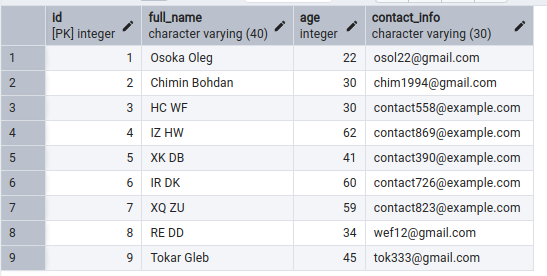
*Для демонстрації запитів виберемо по 1 таблиці до кожного*

***Запити вставки*** реалізовані за допомогою функцій add. Спочатку в меню користувач обирає опцію додавання, далі обирає таблицю, до якої хоче додати рядок і вводить необхідні дані

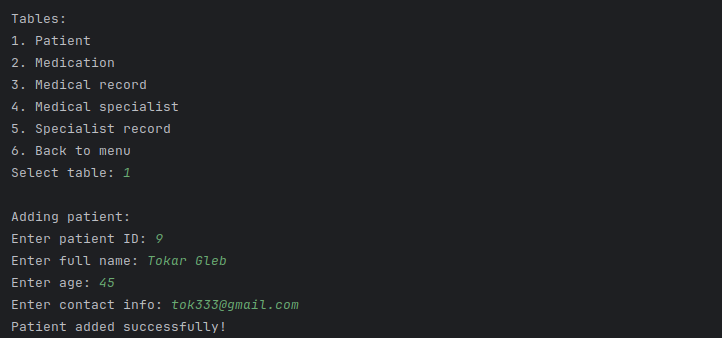
*Таблиця Patient до вставки:*



*Таблиця Patient після вставки:*



*Робота програми:*



Лістинг функцій add для кожної таблиці:

def add\_patient(self, id, full\_name, age, contact\_info):

new\_patient = Patient(id=id, full\_name=full\_name, age=age, contact\_info=contact\_info)

self.db.add(new\_patient)

self.db.commit()

def add\_medication(self, id, id\_patient, name, dosage\_form, side\_effect):

new\_medication = Medication(id=id, id\_patient=id\_patient, name=name, dosage\_form=dosage\_form, side\_effect=side\_effect)

self.db.add(new\_medication)

self.db.commit()

def add\_medical\_record(self, id, id\_patient, visit\_date, diagnosis, symptoms):

new\_record = MedicalRecord(id=id, id\_patient=id\_patient, visit\_date=visit\_date, diagnosis=diagnosis, symptoms=symptoms)

self.db.add(new\_record)

self.db.commit()

def add\_medical\_specialist(self, id, full\_name, qualification, work\_experience, contact\_info):

new\_specialist = MedicalSpecialist(

id=id, full\_name=full\_name, qualification=qualification,

work\_experience=work\_experience, contact\_info=contact\_info

)

self.db.add(new\_specialist)

self.db.commit()

def add\_specialist\_record(self, id, id\_specialist, id\_record):

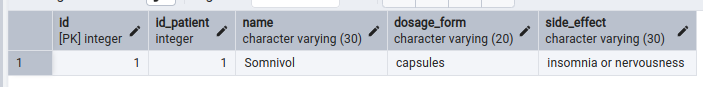
new\_specialist\_record = SpecialistRecord(id=id, id\_specialist=id\_specialist, id\_record=id\_record)

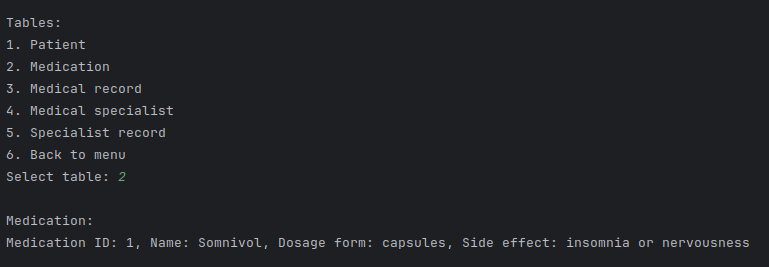
self.db.add(new\_specialist\_record)

self.db.commit()

***Запити показу*** реалізовані за допомогою функцій show. Спочатку в меню користувач обирає опцію показу, далі обирає таблицю, яку хоче побачити

*Показ таблиці Medication:*





*Лістинг функцій show для кожної таблиці:*

def show\_patient(self, patient):

print("\nPatient:")

for pat in patient:

print(f"Patient ID: {pat.id}, Full name: {pat.full\_name}, Age: {pat.age}, Contact info: {pat.contact\_info}")

def show\_medication(self, medication):

print("\nMedication:")

for med in medication:

print(f"Medication ID: {med.id}, Name: {med.name}, Dosage form: {med.dosage\_form}, Side effect: {med.side\_effect}")

def show\_medical\_record(self, medical\_record):

print("\nMedical record:")

for record in medical\_record:

print(f"Medical record ID: {record.id}, ID patient: {record.id\_patient}, Visit date: {record.visit\_date}, Diagnosis: {record.diagnosis}, Symptoms: {record.symptoms}")

def show\_medical\_specialist(self, medical\_specialist):

print("\nMedical specialist:")

for specialist in medical\_specialist:

print(f"Medical specialist ID: {specialist.id}, Full name: {specialist.full\_name}, Qualification: {specialist.qualification}, Work experience: {specialist.work\_experience}, Contact info: {specialist.contact\_info}")

def show\_specialist\_record(self, specialist\_record):

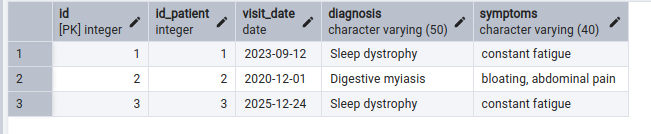
print("\nSpecialist record:")

for specialist in specialist\_record:

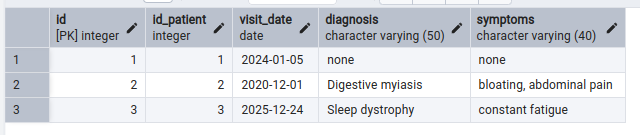
print(f"Specialist record ID: {specialist.id}, Specialist ID: {specialist.id\_specialist}, Record ID: {specialist.id\_record}")

***Запит редагування*** реалізовано за допомогою функції update. Спочатку користувач обирає, у якій таблиці потрібно змінити запис і за яким ідентифікатором. Потім треба ввести всі необхідні дані для редагування рядка.

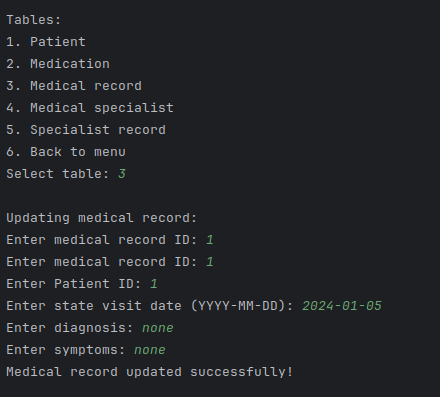
*Таблиця Medical record до редагування:*

**

*Таблиця Medical record після редагування:*

**

*Робота програми:*



*Лістинг функцій update для кожної таблиці:*

def update\_patient(self, id, full\_name, age, contact\_info):

patient = self.db.query(Patient).filter(Patient.id == id).first()

if patient:

patient.full\_name = full\_name

patient.age = age

patient.contact\_info = contact\_info

self.db.commit()

def update\_medication(self, id, id\_patient, name, dosage\_form, side\_effect):

medication = self.db.query(Medication).filter(Medication.id == id).first()

if medication:

medication.id\_patient = id\_patient

medication.name = name

medication.dosage\_form = dosage\_form

medication.side\_effect = side\_effect

self.db.commit()

def update\_medical\_record(self, id, id\_patient, visit\_date, diagnosis, symptoms):

record = self.db.query(MedicalRecord).filter(MedicalRecord.id == id).first()

if record:

record.id\_patient = id\_patient

record.visit\_date = visit\_date

record.diagnosis = diagnosis

record.symptoms = symptoms

self.db.commit()

def update\_medical\_specialist(self, id, full\_name, qualification, work\_experience, contact\_info):

specialist = self.db.query(MedicalSpecialist).filter(MedicalSpecialist.id == id).first()

if specialist:

specialist.full\_name = full\_name

specialist.qualification = qualification

specialist.work\_experience = work\_experience

specialist.contact\_info = contact\_info

self.db.commit()

def update\_specialist\_record(self, id, id\_specialist, id\_record):

specialist\_record = self.db.query(SpecialistRecord).filter(SpecialistRecord.id == id).first()

if specialist\_record:

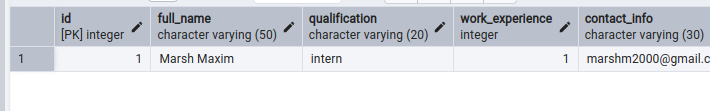
specialist\_record.id\_specialist = id\_specialist

specialist\_record.id\_record = id\_record

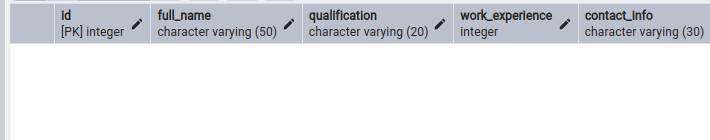
self.db.commit()

***Запити видалення*** реалізовані за допомогою функцій delete. Спочатку користувач обирає таблицю, з якої потрібно видалити дані. Потім потрібно ввести номер ідентифікатора рядка для видалення.

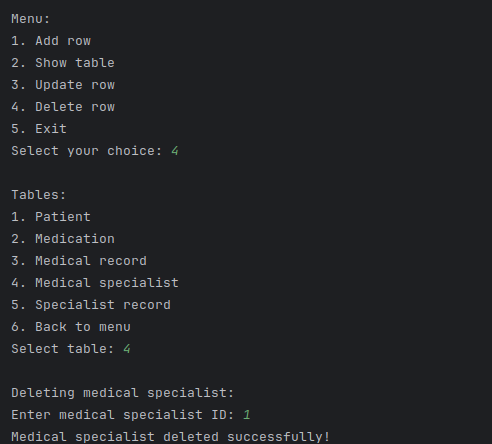
*Таблиця Medical specialist до видалення:*

**

*Таблиця Medical specialist після видалення:*

**

*Робота програми:*

**

*Лістинг функцій delete для кожної таблиці:*

def delete\_patient(self, id):

patient = self.db.query(Patient).filter(Patient.id == id).first()

if patient:

self.db.delete(patient)

self.db.commit()

def delete\_medication(self, id):

medication = self.db.query(Medication).filter(Medication.id == id).first()

if medication:

self.db.delete(medication)

self.db.commit()

def delete\_medical\_record(self, id):

record = self.db.query(MedicalRecord).filter(MedicalRecord.id == id).first()

if record:

self.db.delete(record)

self.db.commit()

def delete\_medical\_specialist(self, id):

specialist = self.db.query(MedicalSpecialist).filter(MedicalSpecialist.id == id).first()

if specialist:

self.db.delete(specialist)

self.db.commit()

def delete\_specialist\_record(self, id):

record = self.db.query(SpecialistRecord).filter(SpecialistRecord.id == id).first()

if record:

self.db.delete(record)

self.db.commit()

**Завдання №2**

Індекс – це спеціальна структура даних, яка зберігає групу ключових значень та покажчиків. Індекс використовується для управління даними. Для тестування індексів було створено окремі таблиці у базі даних test з 1000000 записами.

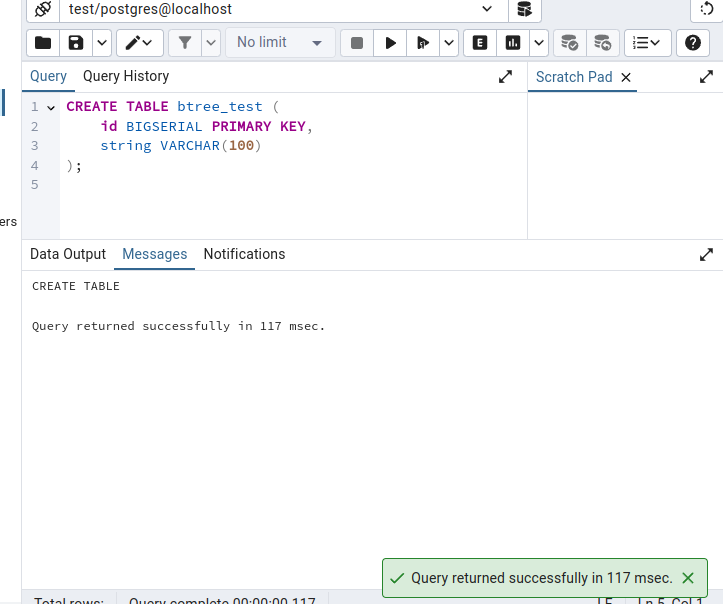
Індекс BTree, або B-дерево, є ефективним для даних, які можна відсортувати. Це означає, що для типу даних мають бути визначені оператори порівняння: «більше», «менше», «більше або дорівнює», «менше або дорівнює» та «дорівнює». Важливо зазначити, що одні й ті ж дані можна сортувати різними способами, що відображає концепцію сімейства операторів.

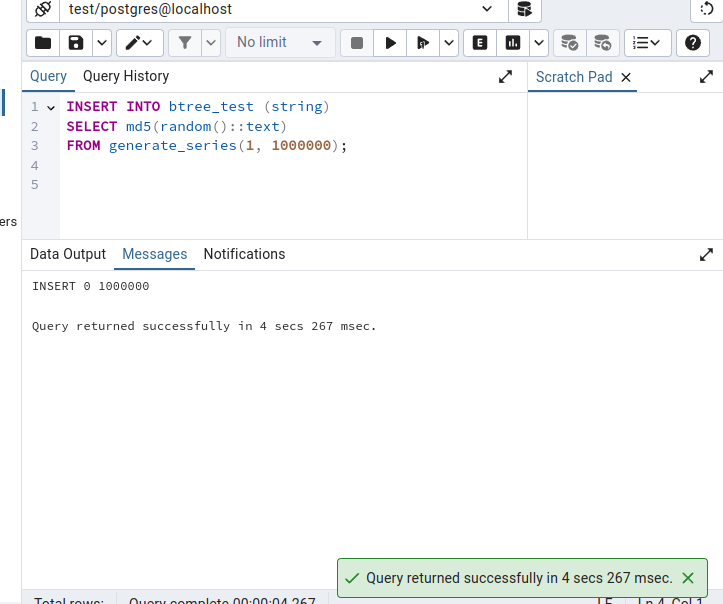
Записи індексу B-дерева організовані в сторінки. Листкові сторінки містять індексовані ключі та посилання на відповідні рядки таблиці (TID-ів), а в серединних сторінках кожен запис вказує на дочірню сторінку та містить мінімальне значення ключа цієї сторінки.

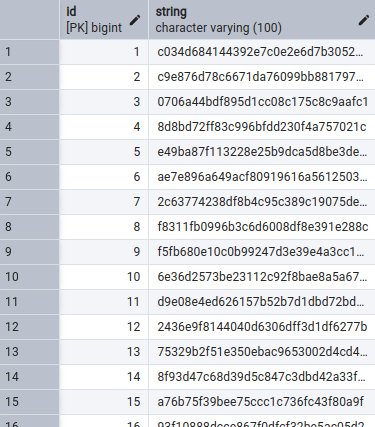
B-дерева мають кілька важливих характеристик:

1. Вони збалансовані, що означає, що всі листкові сторінки віддаляються від кореня на однакову кількість рівнів внутрішніх сторінок. Тому час на пошук будь-якого значення є сталим.
2. Вони сильно розгалужені, оскільки кожна сторінка (звичайно, розміром 8 КБ) містить багато (іноді сотні) TID-ів. Це дозволяє зробити глибину B-дерева дуже малою, навіть для великих таблиць, де вона зазвичай складає від 4 до 5 рівнів.
3. Дані в індексі впорядковані за принципом неспадання (як всередині кожної сторінки, так і між сторінками), і сторінки одного рівня з’єднані двостороннім списком. Завдяки цьому можна отримати впорядкований набір даних, просто проходячи по списку в одному або іншому напрямку, не повертаючись до кореня дерева.

Для дослідження індексу була створена таблиця btree\_test, яка має дві колонки: “id” та “string”:





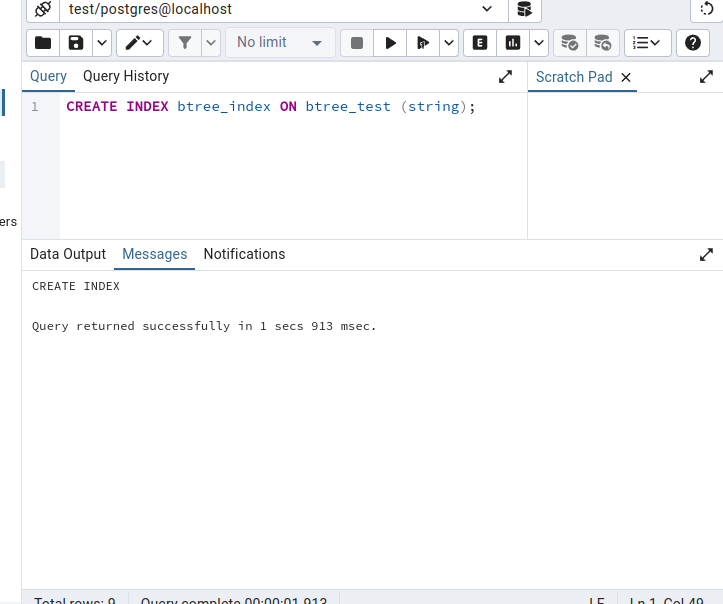
****

*Для тестування візьмемо 4 запити:*

1. EXPLAIN ANALYZE SELECT \* FROM btree\_test WHERE string LIKE 'abc%';
2. EXPLAIN ANALYZE SELECT COUNT(\*), MAX(id), MIN(id) FROM btree\_test;
3. EXPLAIN ANALYZE SELECT string, COUNT(\*) FROM btree\_test GROUP BY string ORDER BY string DESC;
4. EXPLAIN ANALYZE SELECT LEFT(string, 1) AS first\_letter, COUNT(\*), MAX(id) FROM btree\_test WHERE id < 500000 GROUP BY first\_letter ORDER BY first\_letter;

*Створення індексу:*

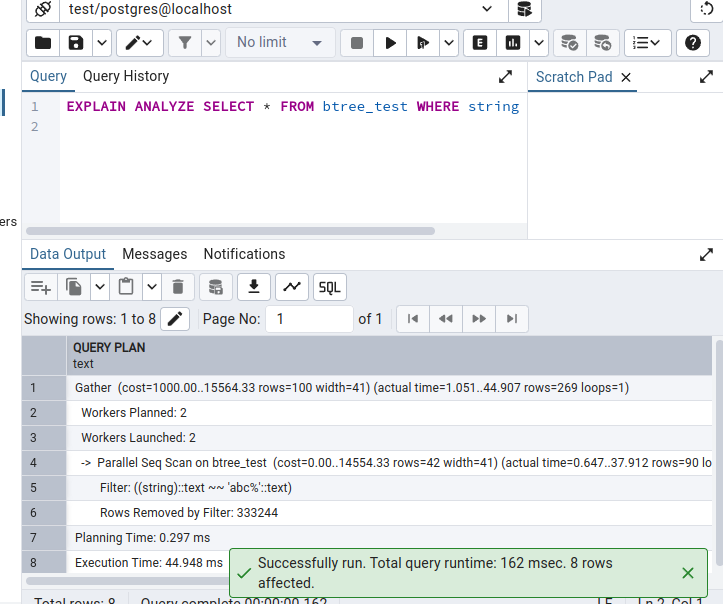
CREATE INDEX btree\_index ON btree\_test (string);



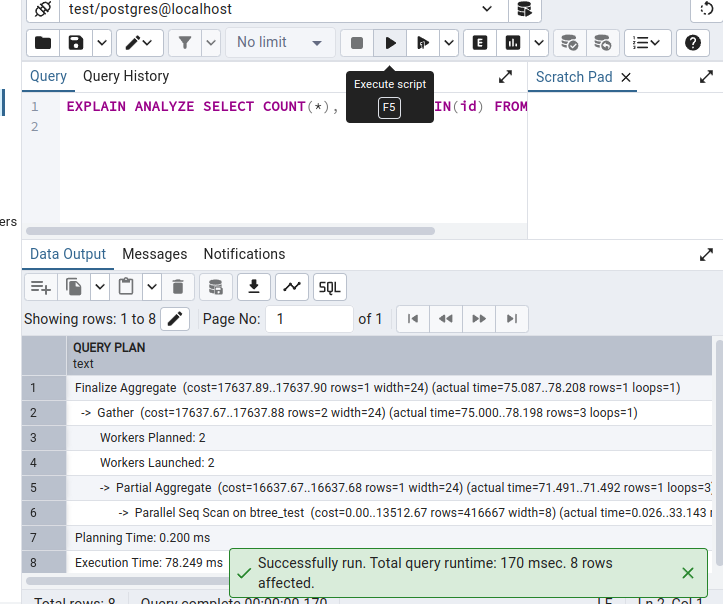
**Результати виконання запитів**

*Без індекса BTree*

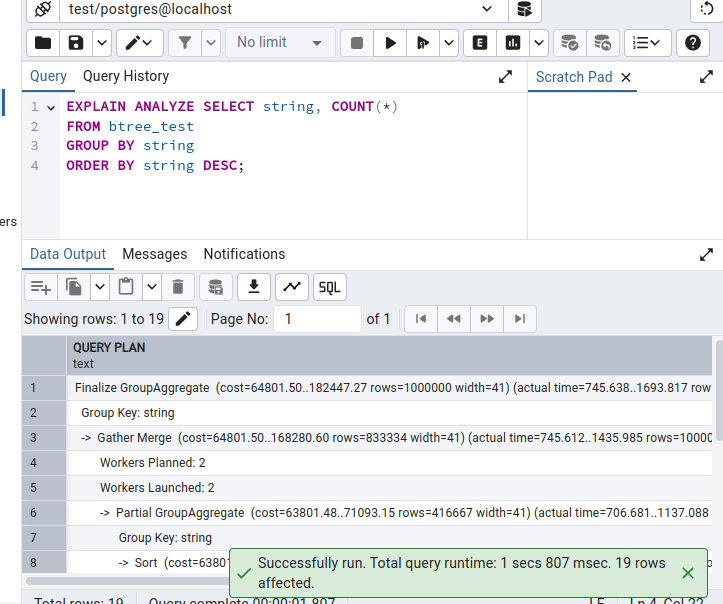
Запит №1



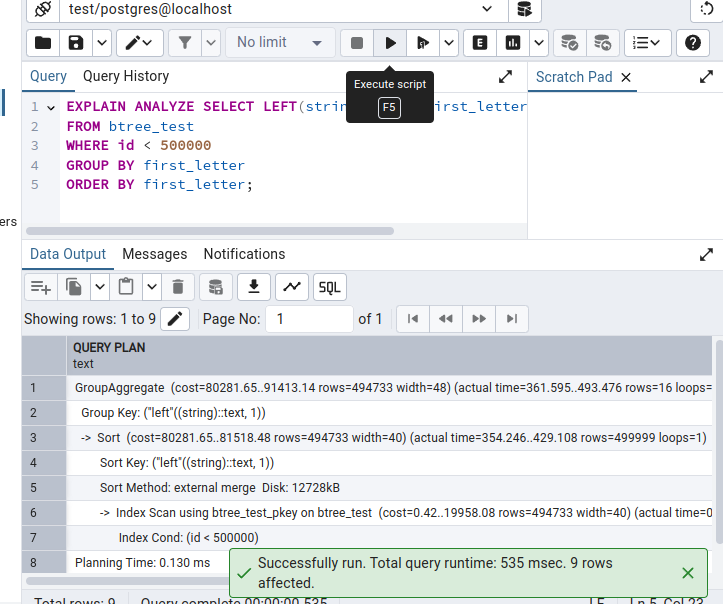
Запит №2



Запит №3

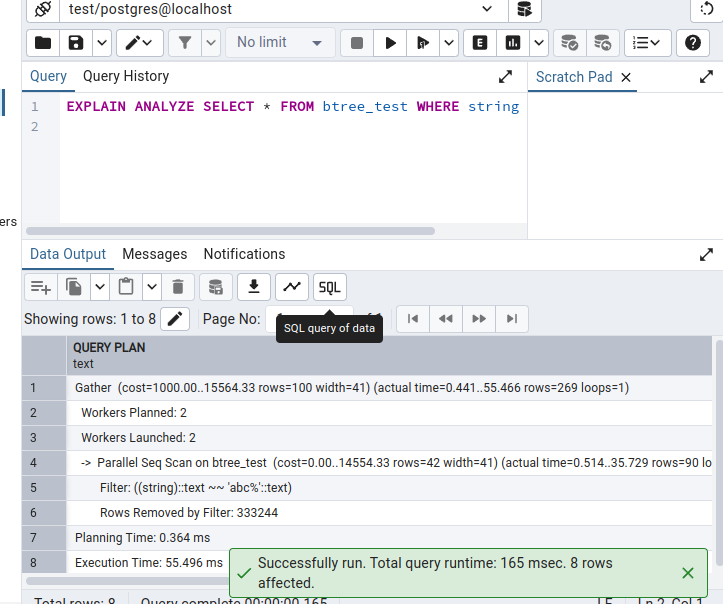


Запит №4

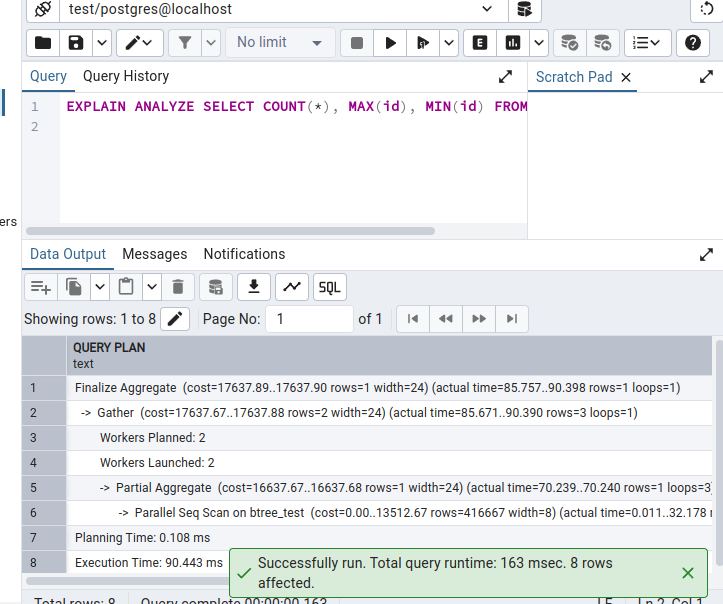


З індексом BTree

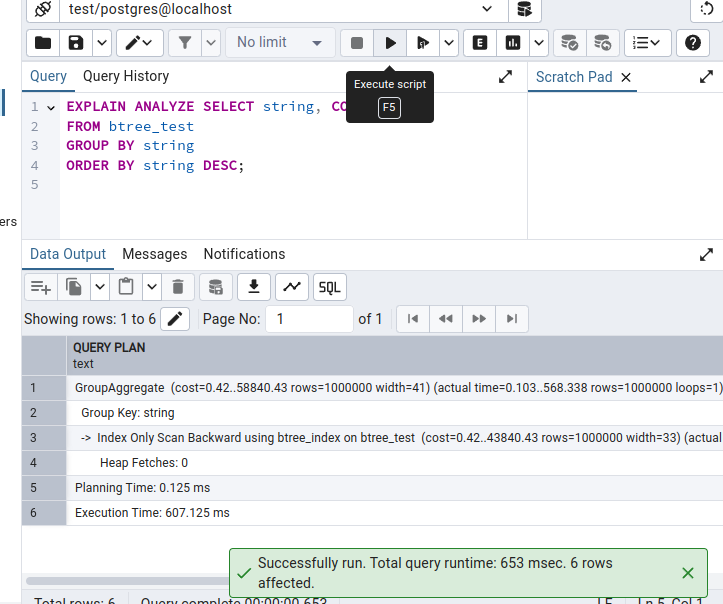
Запит №1



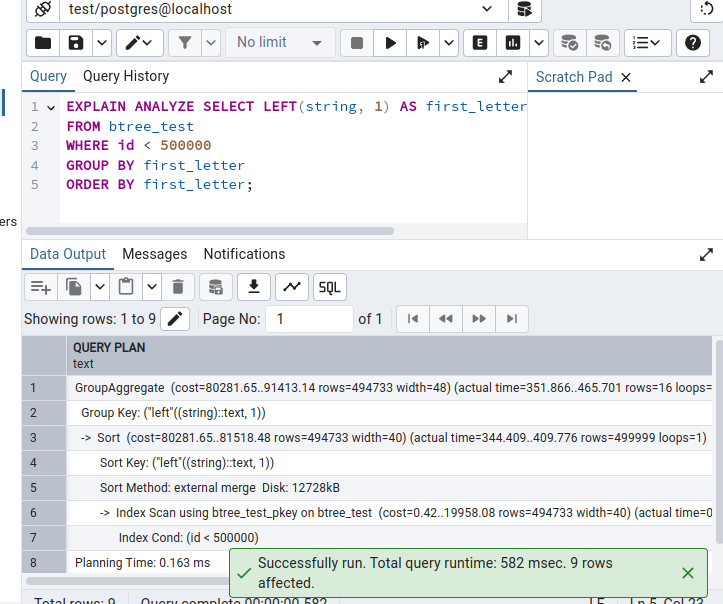
Запит №2



Запит №3



Запит №4



### *Висновок по результатам запитів з індексом BTree та без нього*

При порівнянні виконання запитів на таблиці з індексом BTree і без нього, можна зробити кілька важливих висновків:

1. Запити без індексу BTree:
2. Без індексу кожен запит вимушений сканувати всю таблицю, що може значно збільшити час виконання, особливо при великій кількості рядків.
3. Запити, які містять фільтрацію, агрегацію або сортування, можуть бути повільнішими, оскільки СУБД не може швидко знайти потрібні рядки без індексу.
4. Наприклад, запит із LIKE 'abc%' без індексу потребує повного сканування таблиці для пошуку всіх рядків, що починаються на 'abc'.
5. Запити з індексом BTree:
6. Індекс BTree значно покращує швидкість виконання запитів, оскільки дозволяє СУБД швидше знаходити відповідні рядки завдяки сортуванню даних та оптимізації пошуку через дерево.
7. Запити, що містять порівняння за операторами (наприклад, =, >, <, >=), набагато швидші при використанні BTree індексу. Індекс дає змогу уникнути сканування всіх рядків таблиці.
8. Для запиту з фільтрацією LIKE 'abc%', де шаблон має фіксований префікс (на початку рядка), BTree може оптимізувати пошук, оскільки індекс дозволяє швидко знаходити всі значення, що починаються з цього префікса.
9. Агрегатні функції (COUNT(), MAX(), MIN()) також працюють значно швидше при використанні індексу, оскільки індекс дає змогу обробляти лише частину таблиці, що відповідає умовам запиту.
10. Запити з групуванням та сортуванням:
11. Без індексу, групування та сортування можуть бути дуже повільними, оскільки СУБД повинна повністю обробити всі рядки таблиці.
12. З індексом BTree сортування та групування відбуваються швидше, оскільки індекс вже впорядковує дані за певним полем (наприклад, string), і СУБД може просто пройтися по вже відсортованим даним.
13. Запити з умовами, групуванням і сортуванням:
14. Запити, що включають складніші умови (наприклад, WHERE id < 500000 з групуванням та сортуванням), за відсутності індексу можуть бути дуже повільними.
15. З індексом BTree ці запити можуть значно пришвидшити виконання, оскільки індекс дозволяє фільтрувати рядки до того, як виконується групування або сортування.

### *Загальний висновок:*

Індекс BTree значно покращує продуктивність запитів, особливо для великих таблиць. Запити на фільтрацію, агрегацію, групування та сортування працюють швидше завдяки використанню індексу, оскільки індекс дозволяє швидше знаходити відповідні рядки та зменшує кількість рядків, які потрібно обробити. Однак для деяких запитів, таких як LIKE '%abc%', індекс BTree може бути менш ефективним порівняно з іншими типами індексів, такими як GIN, які оптимізують пошук за патернами.

Таким чином, індекс BTree є дуже корисним для запитів, що використовують оператори порівняння та забезпечують швидкий доступ до відсортованих даних, але його ефективність залежить від типу запиту та структури таблиці.

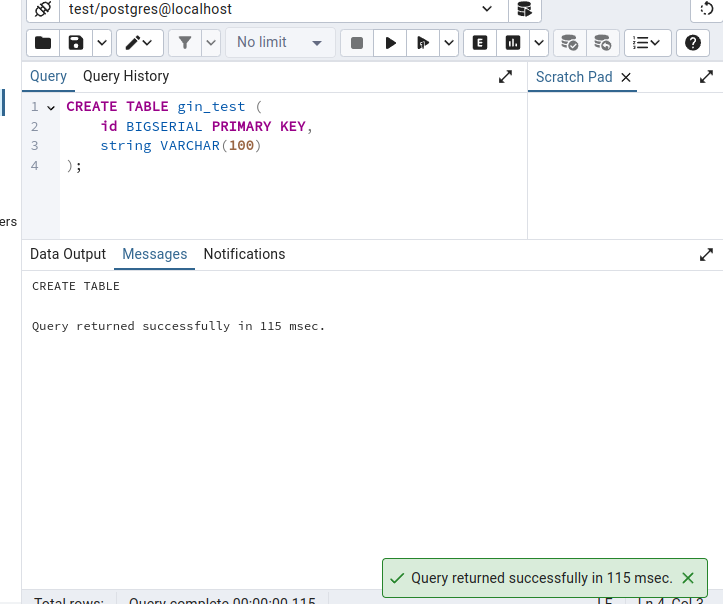
***GIN***

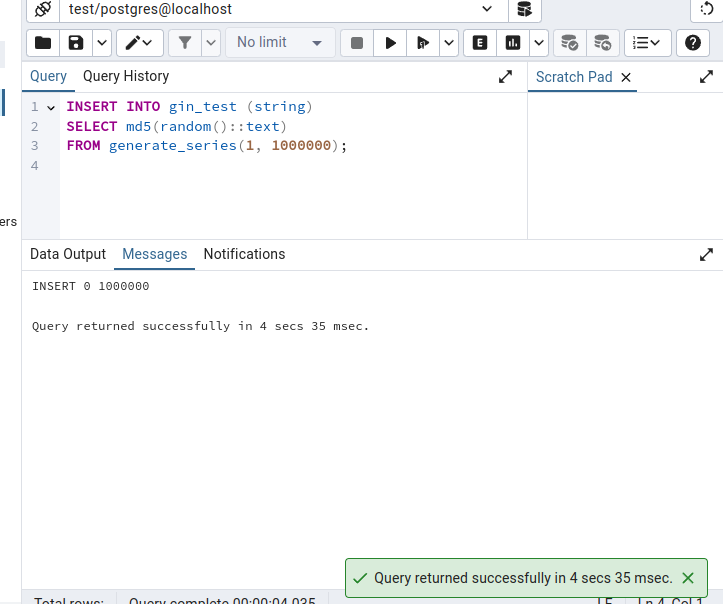
Індекс GIN, або Generalized Inverted Index, є типом зворотного індексу. Він використовується для даних, які складаються з декількох елементів, а не є простими атомарними значеннями. Замість індексації самих значень, індексуються окремі елементи; кожен елемент вказує на ті записи, в яких він з’являється.

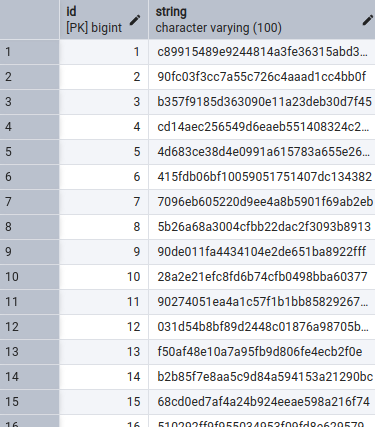
Можна порівняти цей метод з алфавітним покажчиком в книзі, де для кожного терміна наведені сторінки, на яких він згадується. Подібно до покажчика в книзі, GIN індекс забезпечує швидкий пошук елементів у таблиці. Для цього використовується структура даних, схожа на B-дерево, хоча її реалізація може бути спрощена. Кожному елементу прив’язано впорядковане посилання на рядки таблиці, які містять цей елемент. Порядок сортування елементів у самій вибірці не є критичним, але важливий для ефективної роботи індексу.

Однією з особливостей GIN є те, що елементи не видаляються з індексу. Навіть якщо значення, що містять ці елементи, змінюються, зникають чи з’являються, набір елементів залишається досить стабільним. Це спрощує алгоритми роботи з індексом, дозволяючи ефективно працювати з ним кільком процесам одночасно.

Для дослідження індексу була створена таблиця gin\_test, яка має дві колонки: “id” та “string”:







*Для тестування візьмемо 4 запити:*

1. EXPLAIN ANALYZE SELECT \* FROM btree\_test WHERE string LIKE 'abc%';
2. EXPLAIN ANALYZE SELECT COUNT(\*), MAX(id), MIN(id) FROM btree\_test;
3. EXPLAIN ANALYZE SELECT string, COUNT(\*) FROM btree\_test GROUP BY string ORDER BY string DESC;
4. EXPLAIN ANALYZE SELECT LEFT(string, 1) AS first\_letter, COUNT(\*), MAX(id) FROM btree\_test WHERE id < 500000 GROUP BY first\_letter ORDER BY first\_letter;

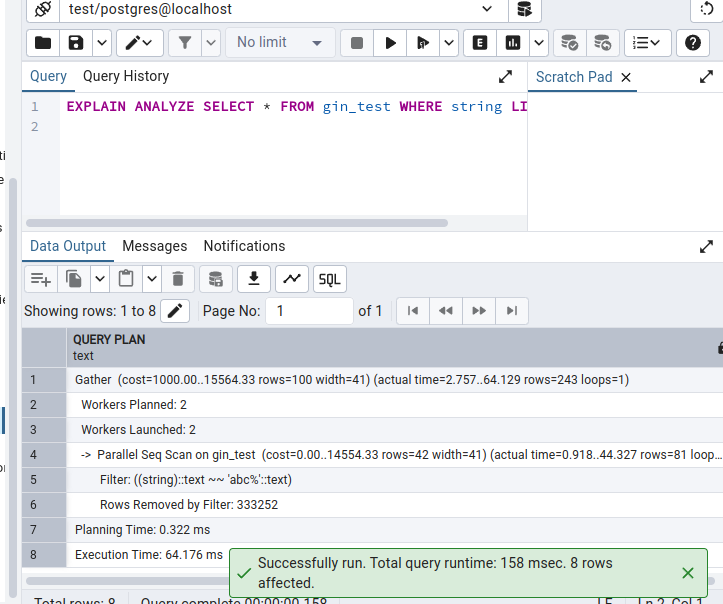
*Створення індексу:*

CREATE INDEX gin\_index ON btree\_test USING gin (string gin\_trgm\_ops);

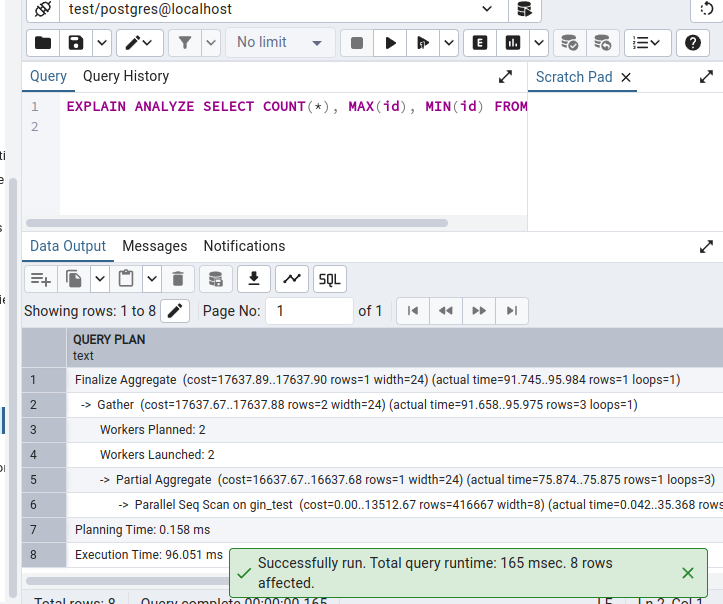
Результати виконання запитів

*Без індекса GIN*

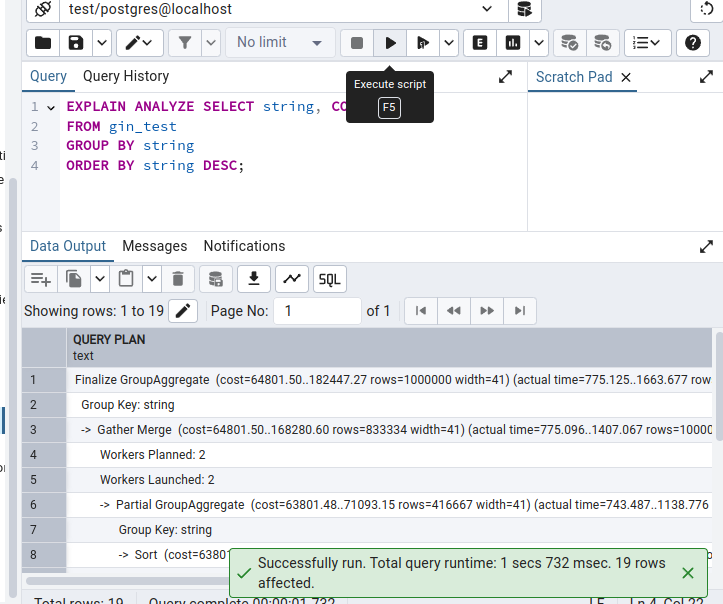
Запит №1



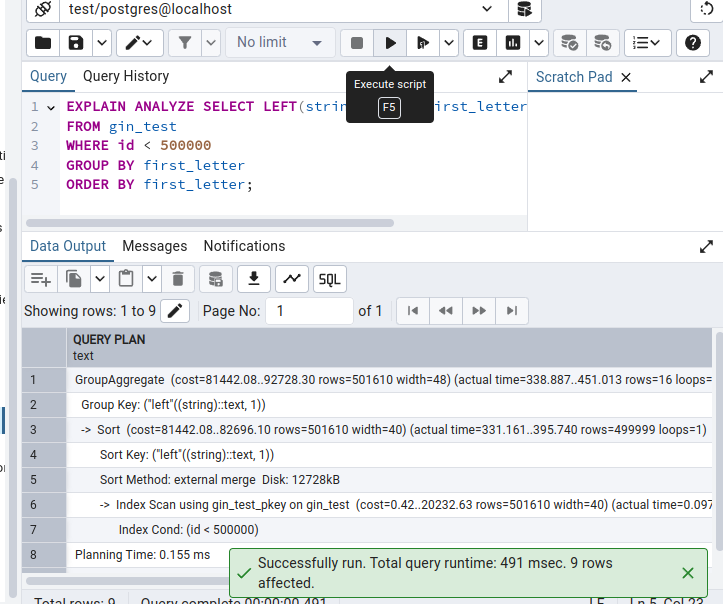
Запит №2



Запит №3

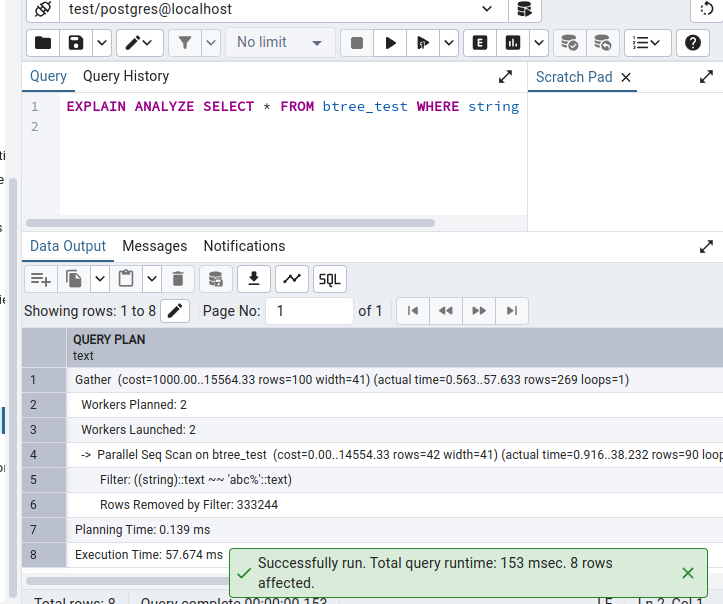


Запит №4



З індексом GIN

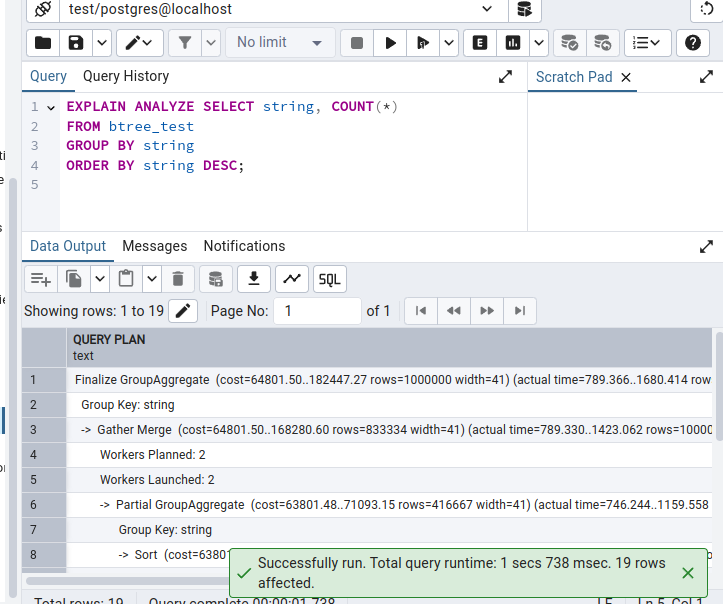
Запит №1



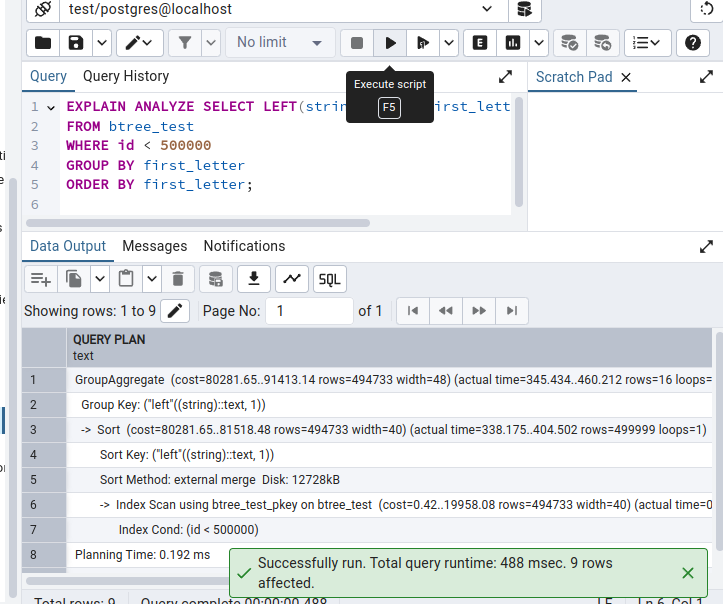
Запит №2



Запит №3



Запит №4



### *Висновок по результатах виконання запитів з індексом GIN*

При порівнянні запитів з індексом GIN можна зробити кілька важливих висновків, особливо для запитів, що працюють із текстовими даними або великими наборами елементів, як-от за допомогою оператора LIKE.

1. Запит із фільтрацією:
2. Інтерфейс GIN є дуже ефективним для запитів з шаблонами типу LIKE '%abc%', оскільки індекс індексує не самі значення, а окремі елементи (наприклад, літери або частини слів). Завдяки цьому індекс GIN швидко знаходить рядки, які містять потрібний елемент, навіть якщо він з’являється на початку чи в середині значення.
3. Запит із шаблоном LIKE '%abc%' буде виконуватися значно швидше при використанні індексу GIN порівняно з відсутністю індексу, оскільки індекс дозволяє швидко знайти елементи, що входять у значення, які відповідають шаблону.
4. Запит із агрегатними функціями:
5. Інтерфейс GIN також може пришвидшити виконання запитів з агрегатними функціями, якщо ці функції працюють із великою кількістю елементів або текстових даних. Завдяки використанню індексу, СУБД може швидше знаходити потрібні елементи та обробляти їх, а не виконувати повне сканування таблиці.
6. Запит із групуванням і сортуванням:
7. Для запитів з групуванням і сортуванням, які працюють із великою кількістю текстових даних, індекс GIN також може покращити продуктивність, оскільки індекс зберігає елементи у вигляді впорядкованих посилань на рядки, що дозволяє значно зменшити час на групування та сортування даних.
8. Однак для запитів, що включають сортування за текстовими полями, індекс GIN може бути менш ефективним, якщо порівняння виконується по всьому рядку, а не за окремими елементами.
9. Запит із умовами, групуванням і сортуванням:
10. Запити, що включають умовні фільтри, групування та сортування, можуть виграти від використання GIN-індексу, особливо якщо умови включають пошук за шаблоном або роботу з великими наборами даних.
11. Наприклад, для запиту з групуванням за першою літерою та умовою WHERE id < 500000, GIN індекс дозволяє швидше обробляти дані завдяки ефективному пошуку елементів, що задовольняють умови.

### *Загальний висновок:*

Індекс GIN є дуже ефективним для роботи з великими обсягами текстових даних або елементів, де потрібен пошук за шаблонами типу LIKE '%abc%'. Він значно покращує швидкість виконання запитів, що включають фільтрацію, агрегатні функції, групування та сортування. Основна перевага GIN полягає в тому, що він індексує окремі елементи даних (як, наприклад, окремі слова чи літери), що дозволяє швидко знаходити відповідні рядки. Однак для деяких складних запитів із сортуванням чи групуванням за повними рядками ефективність GIN може бути обмежена порівняно з іншими типами індексів, такими як BTree.

**Розробка тригера бази даних PostgreSQL**

*Створення таблиці для демонстрації*

Для прикладу створимо таблицю employees з деякими даними:

CREATE TABLE employees (

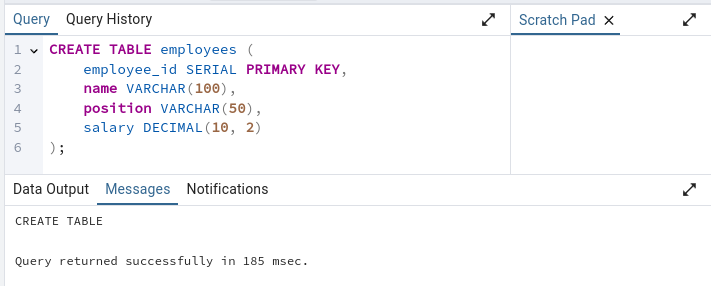
employee\_id SERIAL PRIMARY KEY,

name VARCHAR(100),

position VARCHAR(50),

salary DECIMAL(10, 2)

);



*Створення тригерної функції*

Тригерна функція буде виконуватися після вставки або оновлення записів у таблиці employees. Вона також використовуватиме курсор для ітерації по оновлених чи вставлених записах, а також оброблятиме виключні ситуації.

CREATE OR REPLACE FUNCTION employee\_trigger\_function()

RETURNS TRIGGER AS $$

DECLARE

rec RECORD;

BEGIN

IF (TG\_OP = 'INSERT') THEN

RAISE NOTICE 'New employee added: Name: %, Position: %, Salary: %', NEW.name, NEW.position, NEW.salary;

END IF;

IF (TG\_OP = 'UPDATE') THEN

IF NEW.salary <> OLD.salary THEN

RAISE NOTICE 'Employee updated: Name: %, Old salary: %, New salary: %', NEW.name, OLD.salary, NEW.salary;

FOR rec IN SELECT \* FROM employees WHERE salary > OLD.salary LOOP

RAISE NOTICE 'Processing employee: ID %, Name %, Salary %', rec.employee\_id, rec.name, rec.salary;

END LOOP;

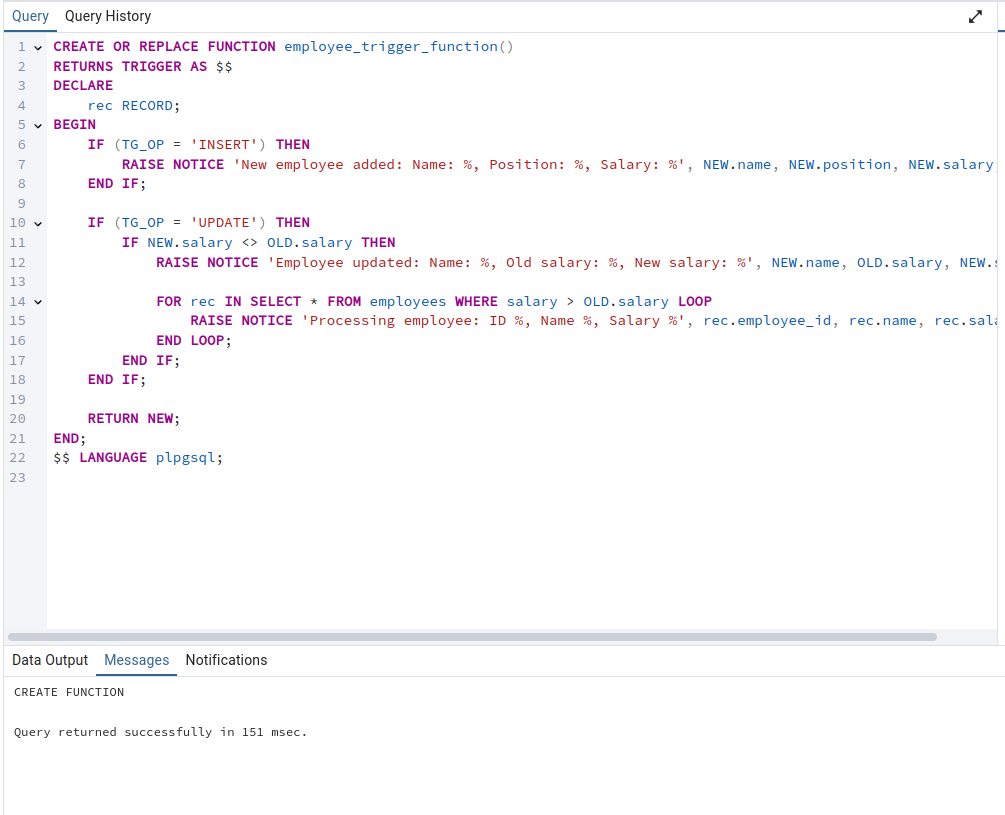
END IF;

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;



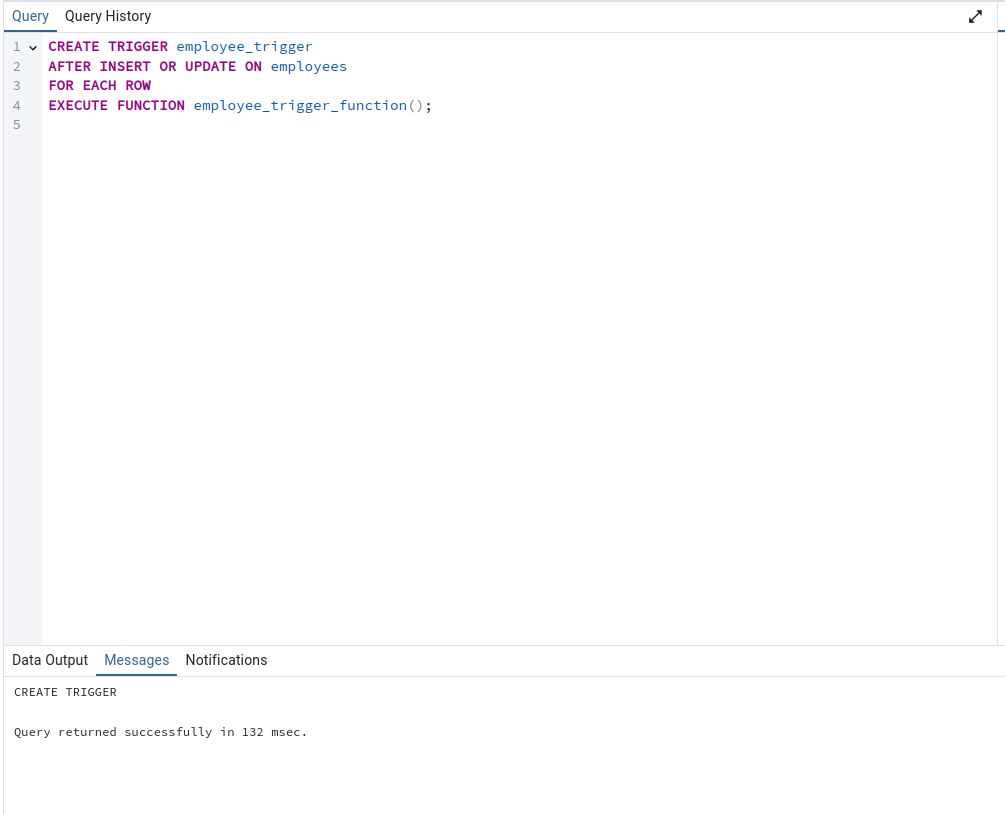
*Створення тригера*

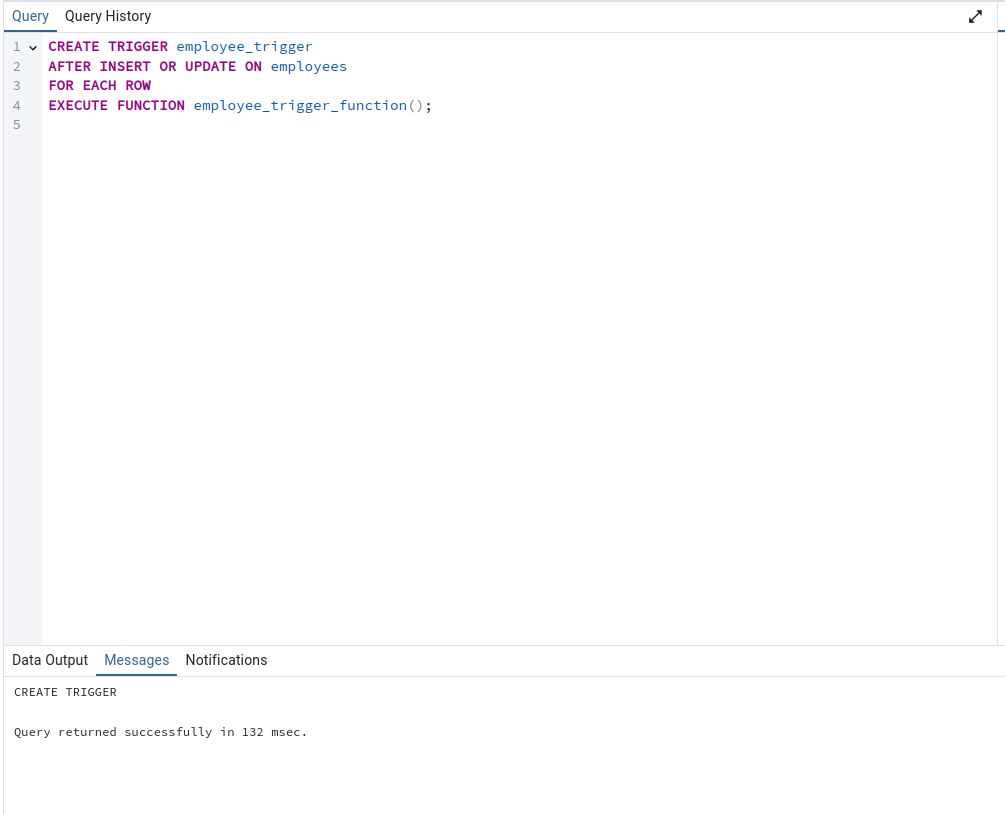
CREATE TRIGGER employee\_trigger

AFTER INSERT OR UPDATE ON employees

FOR EACH ROW

EXECUTE FUNCTION employee\_trigger\_function();





*Тестування тригера*

Тепер, коли тригер і функція створені, можна протестувати їх, виконуючи запити на вставку та оновлення даних

INSERT INTO employees (name, position, salary)

VALUES ('John Doe', 'Software Engineer', 50000.00);

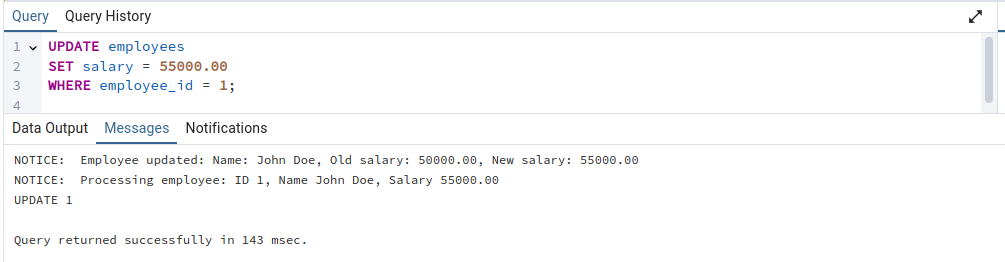


UPDATE employees

SET salary = 55000.00

WHERE employee\_id = 1;

*Результат: Якщо зарплата змінилась, виведеться повідомлення:*



### Висновок щодо створеного тригера

1. Функціональність тригера:
2. Тригер успішно виконує завдання, пов’язані з обробкою даних у таблиці employees.
3. Він спрацьовує після подій INSERT та UPDATE, що дозволяє автоматично обробляти нові записи та оновлення.
4. Реалізовані можливості:
5. Для події INSERT: Тригер записує повідомлення з інформацією про нового працівника, включаючи його ім’я, посаду та зарплату.
6. Для події UPDATE: Тригер відстежує зміни у зарплаті працівника, записуючи старе та нове значення.
7. Коректність реалізації:
8. Тригерна функція використовує ключові конструкції PostgreSQL, такі як NEW і OLD, що забезпечує доступ до нових і старих значень зміненого запису.
9. Форматування повідомлень у функції є точним, а кількість параметрів у RAISE NOTICE відповідає кількості змінних, що виключає синтаксичні помилки.
10. Ефективність:
11. Використання тригера забезпечує автоматизацію процесів, таких як логування або моніторинг змін, без необхідності виконувати ці операції вручну.
12. Це дозволяє зберігати контроль за змінами даних у таблиці employees і підвищує інтеграцію між процесами в базі даних.

**Рівні ізоляції транзакцій у PostgreSQL**

У PostgreSQL доступні чотири рівні ізоляції транзакцій, які визначають, як транзакції можуть взаємодіяти одна з одною під час роботи з даними. Рівні ізоляції впливають на можливість виникнення таких феноменів, як "брудне читання", "фантомне читання" або "неповторюване читання".

### *Рівні ізоляції транзакцій*

1. Read Uncommitted (Читання непідтверджених даних)
2. У цьому режимі транзакція може читати зміни, виконані іншими транзакціями, навіть якщо ці зміни ще не підтверджені (COMMIT).
3. Феномени: можливі "брудні читання", "неповторювані читання" і "фантомні читання".
4. PostgreSQL піднімає цей рівень до Read Committed, оскільки не дозволяє читати непідтверджені зміни.
5. Read Committed (Читання підтверджених даних)
6. За замовчуванням у PostgreSQL.
7. Кожен запит бачить тільки ті зміни, які були підтверджені іншими транзакціями до початку виконання запиту.
8. Феномени: можливі "неповторювані читання" і "фантомні читання".
9. Repeatable Read (Повторюване читання)
10. У цьому режимі транзакція бачить стан бази даних на момент її початку.
11. Транзакція не побачить змін, виконаних іншими транзакціями, навіть якщо вони були підтверджені.
12. Феномени: можливі "фантомні читання".
13. Serializable (Серіалізований доступ)
14. Найвищий рівень ізоляції, де транзакції виконуються так, ніби вони серіалізовані (виконуються одна за одною).
15. Феномени: жодні феномени не можливі.
16. Використовує блокування або додаткову перевірку, щоб забезпечити цілісність.

*Приклади:*

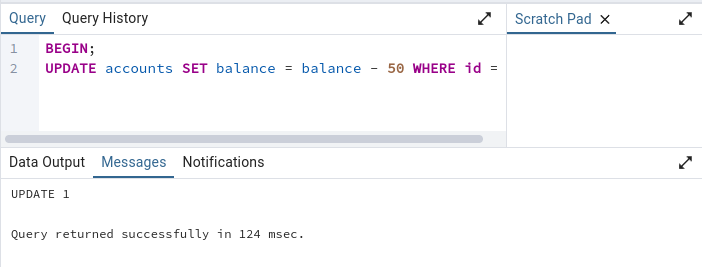
*Таблиця accounts  
CREATE TABLE accounts (id SERIAL PRIMARY KEY, balance DECIMAL);*

*INSERT INTO accounts (balance) VALUES (100), (200), (300);*

**Read Committed:**

BEGIN;

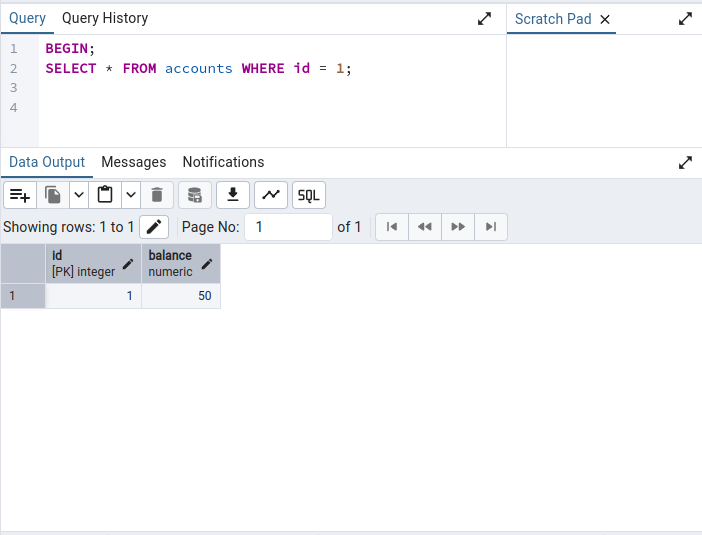
UPDATE accounts SET balance = balance - 50 WHERE id = 1;

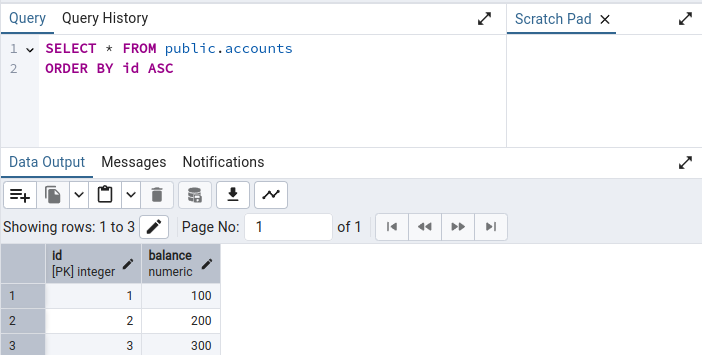


BEGIN;

SELECT \* FROM accounts WHERE id = 1;

COMMIT;



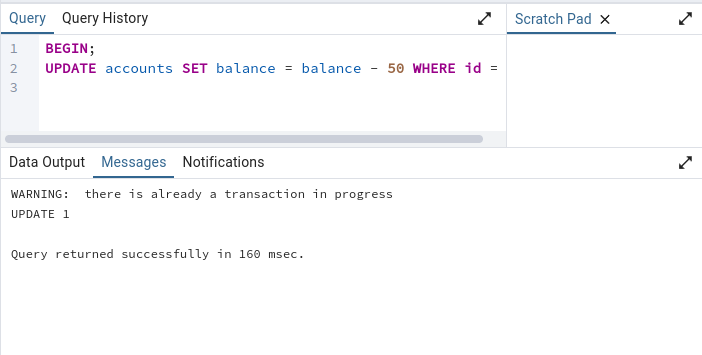


У цьому режимі транзакція 2 читає тільки підтверджені зміни, тому "брудні читання" неможливі. Але повторний запит у тій самій транзакції може дати інший результат, якщо транзакція 1 виконає COMMIT

**Repeatable Read:**

BEGIN;

UPDATE accounts SET balance = balance - 50 WHERE id = 2;

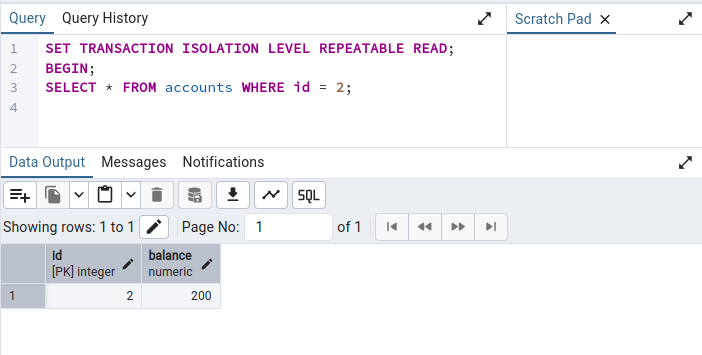


SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN;

SELECT \* FROM accounts WHERE id = 2;

COMMIT;



Транзакція 2 бачить знімок таблиці на момент її початку. Навіть якщо транзакція 1 виконає COMMIT, транзакція 2 працює із зафіксованим знімком, що виключає "неповторювані читання".

**Serializable:**

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN;

INSERT INTO accounts (balance) VALUES (400);

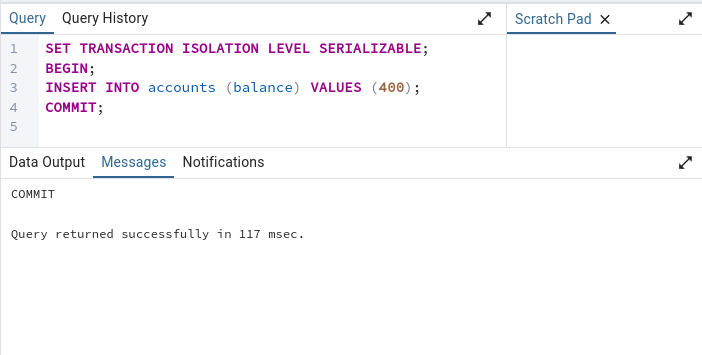
COMMIT;

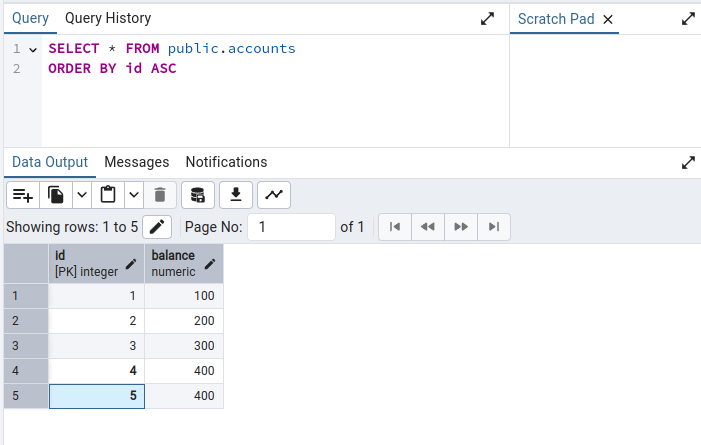
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN;

INSERT INTO accounts (balance) VALUES (400);

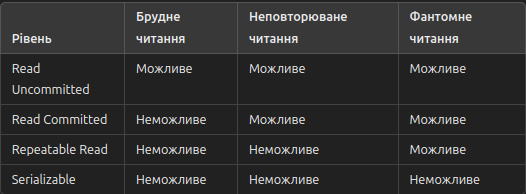
COMMIT;





PostgreSQL гарантує, що жодна з транзакцій не впливатиме на результати іншої. Якщо виникає конфлікт, одна з транзакцій завершується помилкою.

**Порівняння рівнів ізоляції**



### **Висновок**

Рівні ізоляції в PostgreSQL дозволяють керувати взаємодією транзакцій, залежно від вимог до цілісності даних. Вибір рівня ізоляції має базуватися на компромісі між продуктивністю та безпекою даних:

1. **Read Committed** є достатнім для більшості прикладних систем.
2. **Serializable** варто використовувати, коли необхідна максимальна цілісність, але це може вплинути на продуктивність.
3. **Repeatable Read** цей рівень ізоляції забезпечує, що всі прочитані дані залишаються незмінними протягом транзакції. Неможливі "брудні читання" та "неповторювані читання", але можливі "фантомні читання". Використовується для аналітичних запитів, де важлива стабільність даних.
4. **Read Uncommitted** цей рівень ізоляції дозволяє транзакціям читати дані, які ще не були підтверджені іншими транзакціями. Це призводить до можливих "брудних читань", коли транзакція може бачити зміни, які згодом будуть скасовані. У PostgreSQL рівень Read Uncommitted фактично підвищується до Read Committed, оскільки PostgreSQL не дозволяє читати непідтверджені зміни.