

**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 проєкції (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

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:M, M:M 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

<i>№ варіанта</i>	<i>Види індексів</i>	<i>Умови для тригера</i>
<i>12</i>	<i>BTree, GIN</i>	<i>after update, insert</i>

Графічне подання концептуальної моделі зображено на рисунку 1

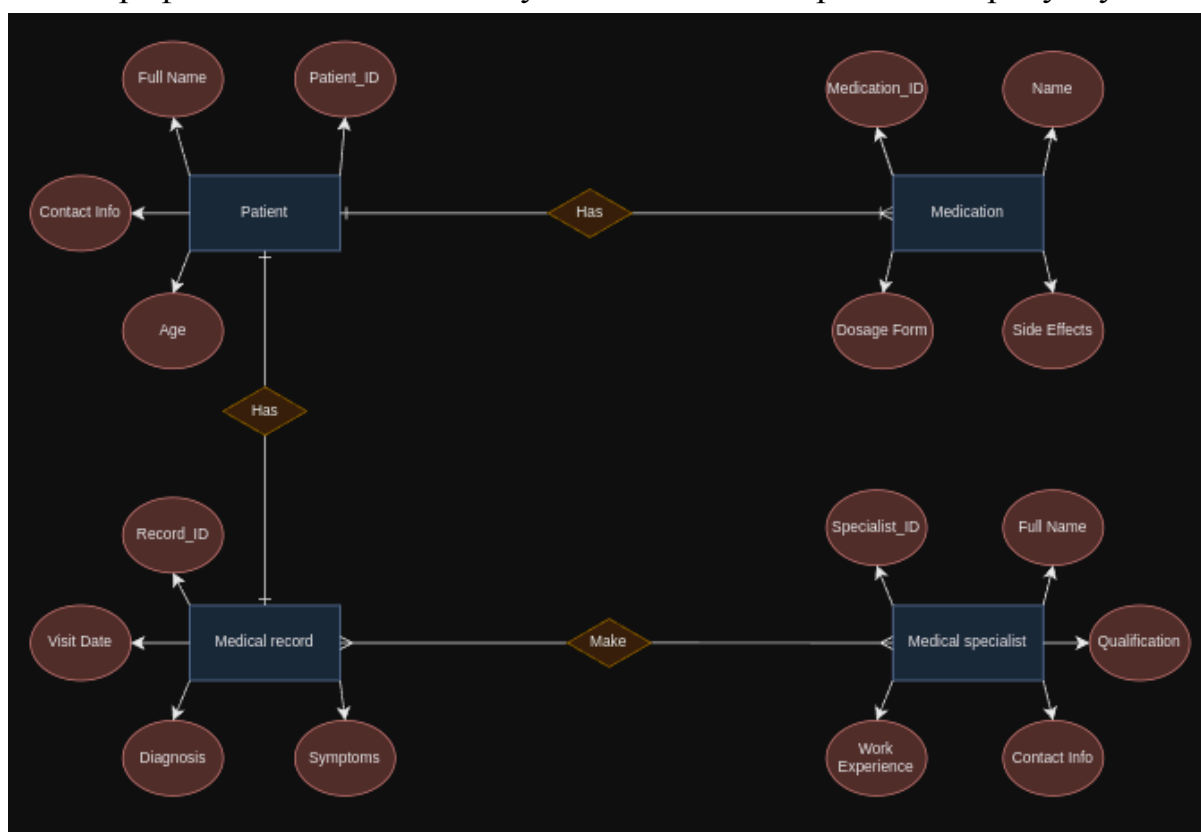


Рис. 1 ER-діаграма, побудована за нотацією Чена

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 і 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	Record_ID – ID запису Patient_ID – ID пацієнта	integer (числовий) integer (числовий)

медичним працівником	Visit date – дата візиту пацієнта Diagnosis – діагноз Symptoms – симптоми	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

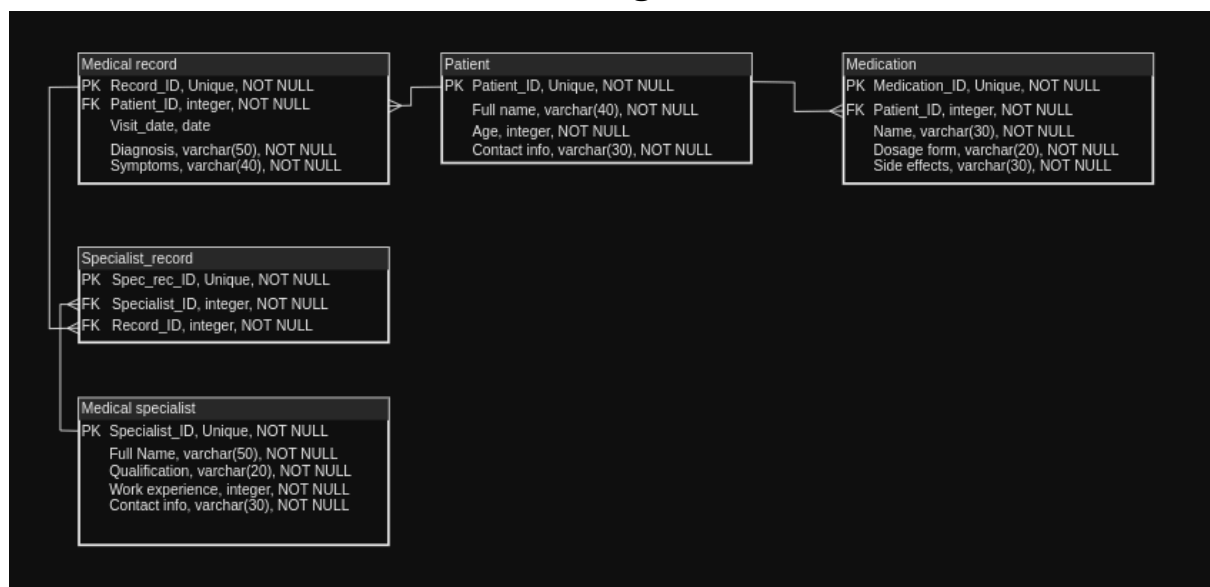


Рис 2. Схема бази даних у графічному вигляді

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 зовнішні 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

```
Menu:
1. Add row
2. Show table
3. Update row
4. Delete row
5. Exit
Select your choice: |
```

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:

```
Menu:
1. Add row
2. Show table
3. Update row
4. Delete row
5. Exit
Select your choice: 1

Tables:
1. Patient
2. Medication
3. Medical record
4. Medical specialist
5. Specialist record
6. Back to menu
Select table:
```

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

2) 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.

```
Menu:
1. Add row
2. Show table
3. Update row
4. Delete row
5. Exit
Select your choice: 2

Tables:
1. Patient
2. Medication
3. Medical record
4. Medical specialist
5. Specialist record
6. Back to menu
Select table: |
```

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

3) 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.

```
Menu:
1. Add row
2. Show table
3. Update row
4. Delete row
5. Exit
Select your choice: 3

Tables:
1. Patient
2. Medication
3. Medical record
4. Medical specialist
5. Specialist record
6. Back to menu
Select table: |
```

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

4) 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.

```
Menu:
1. Add row
2. Show table
3. Update row
4. Delete row
5. Exit
Select your choice: 4

Tables:
1. Patient
2. Medication
3. Medical record
4. Medical specialist
5. Specialist record
6. Back to menu
Select table: |
```

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

5) Exit (Вихід)

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

```
Menu:
1. Add row
2. Show table
3. Update row
4. Delete row
5. Exit
Select your choice: 5

Process finished with exit code 0
```

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:

	Id [PK] integer	full_name character varying (40)	age integer	contact_info character varying (30)
1	1	Osoka Oleg	22	osol22@gmail.com
2	2	Chimin Bohdan	30	chim1994@gmail.com
3	3	HC WF	30	contact558@example.com
4	4	IZ HW	62	contact869@example.com
5	5	XK DB	41	contact390@example.com
6	6	IR DK	60	contact726@example.com
7	7	XQ ZU	59	contact823@example.com
8	8	RE DD	34	wef12@gmail.com

Таблиця Patient після вставки:

	Id [PK] integer	full_name character varying (40)	age integer	contact_info character varying (30)
1	1	Osoka Oleg	22	osol22@gmail.com
2	2	Chimin Bohdan	30	chim1994@gmail.com
3	3	HC WF	30	contact558@example.com
4	4	IZ HW	62	contact869@example.com
5	5	XK DB	41	contact390@example.com
6	6	IR DK	60	contact726@example.com
7	7	XQ ZU	59	contact823@example.com
8	8	RE DD	34	wef12@gmail.com
9	9	Tokar Gleb	45	tok333@gmail.com

Робота програми:

```
Tables:
1. Patient
2. Medication
3. Medical record
4. Medical specialist
5. Specialist record
6. Back to menu
Select table: 1

Adding patient:
Enter patient ID: 9
Enter full name: Tokar Gleb
Enter age: 45
Enter contact info: tok333@gmail.com
Patient added successfully!
```

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:

	Id [PK] integer	Id_patient integer	name character varying (30)	dosage_form character varying (20)	side_effect character varying (30)
1	1	1	Somnivol	capsules	insomnia or nervousness

```
Tables:
1. Patient
2. Medication
3. Medical record
4. Medical specialist
5. Specialist record
6. Back to menu
Select table: 2

Medication:
Medication ID: 1, Name: Somnivol, Dosage form: capsules, Side effect: insomnia or nervousness
```

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:

	Id [PK] integer	id_patient integer	visit_date date	diagnosis character varying (50)	symptoms character varying (40)
1	1	1	2023-09-12	Sleep dystrophy	constant fatigue
2	2	2	2020-12-01	Digestive myiasis	bloating, abdominal pain
3	3	3	2025-12-24	Sleep dystrophy	constant fatigue

Таблиця Medical record після редагування:

	Id [PK] integer	id_patient integer	visit_date date	diagnosis character varying (50)	symptoms character varying (40)
1	1	1	2024-01-05	none	none
2	2	2	2020-12-01	Digestive myiasis	bloating, abdominal pain
3	3	3	2025-12-24	Sleep dystrophy	constant fatigue

Робота програми:

```
Tables:
1. Patient
2. Medication
3. Medical record
4. Medical specialist
5. Specialist record
6. Back to menu
Select table: 3

Updating medical record:
Enter medical record ID: 1
Enter medical record ID: 1
Enter Patient ID: 1
Enter state visit date (YYYY-MM-DD): 2024-01-05
Enter diagnosis: none
Enter symptoms: none
Medical record updated successfully!
```


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:

	id [PK] integer	full_name character varying (50)	qualification character varying (20)	work_experience integer	contact_info character varying (30)
1	1	Marsh Maxim	intern	1	marshm2000@gmail.c

Таблиця Medical specialist після видалення:

	id [PK] integer	full_name character varying (50)	qualification character varying (20)	work_experience integer	contact_info character varying (30)

Робота програми:

```
Menu:
1. Add row
2. Show table
3. Update row
4. Delete row
5. Exit
Select your choice: 4

Tables:
1. Patient
2. Medication
3. Medical record
4. Medical specialist
5. Specialist record
6. Back to menu
Select table: 4

Deleting medical specialist:
Enter medical specialist ID: 1
Medical specialist deleted successfully!
```

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 В-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”:

Query	Query History	Scratch Pad
1 CREATE TABLE btree_test (2 id BIGSERIAL PRIMARY KEY , 3 string VARCHAR(100) 4); 5		
Data Output Messages Notifications		
CREATE TABLE Query returned successfully in 117 msec.		

Query	Query History	Scratch Pad
1 INSERT INTO btree_test (string) 2 SELECT md5(random()::text) 3 FROM generate_series(1, 1000000); 4 5		
Data Output Messages Notifications		
INSERT 0 1000000 Query returned successfully in 4 secs 267 msec.		

	Id [PK] bigint	string character varying (100)
1	1	c034d684144392e7c0e2e6d7b3052...
2	2	c9e876d78c6671da76099bb881797...
3	3	0706a44bdf895d1cc08c175c8c9aafc1
4	4	8d8bd72ff83c996bfdd230f4a757021c
5	5	e49ba87f113228e25b9dca5d8be3de...
6	6	ae7e896a649acf80919616a5612503...
7	7	2c63774238df8b4c95c389c19075de...
8	8	f8311fb0996b3c6d6008df8e391e288c
9	9	f5fb680e10c0b99247d3e39e4a3cc1...
10	10	6e36d2573be23112c92f8bae8a5a67...
11	11	d9e08e4ed626157b52b7d1dbd72bd...
12	12	2436e9f8144040d6306dff3d1df6277b
13	13	75329b2f51e350ebac9653002d4cd4...
14	14	8f93d47c68d39d5c847c3dbd42a33f...
15	15	a76b75f39bee75ccc1c736fc43f80a9f
16	16	02f10888dccc867f0dfef22ba5cc05d2

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);
```

Query

Query History

Scratch Pad x

1

CREATE INDEX btree_index ON btree_test (string);

Data Output

Messages

Notifications

CREATE INDEX

Query returned successfully in 1 secs 913 msec.

Результати виконання запитів

Без індекса BTree

Запит №1

✓ Successfully run. Total query runtime: 162 msec. 8 rows affected. ✕

Запит №2

✓ Successfully run. Total query runtime: 170 msec. 8 rows affected. ✕

Запит №3

✓ Successfully run. Total query runtime: 1 secs 807 msec. 19 rows affected. ✕

Запит №4

✓ Successfully run. Total query runtime: 535 msec. 9 rows affected. ✕

З індексом BTree

Запит №1

✓ Successfully run. Total query runtime: 165 msec. 8 rows affected. ✕

Запит №2

✓ Successfully run. Total query runtime: 163 msec. 8 rows affected. ✕

Запит №3

✓ Successfully run. Total query runtime: 653 msec. 6 rows affected. ✕

Запит №4

✓ Successfully run. Total query runtime: 582 msec. 9 rows affected. ✕

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:

- 1) Без індексу кожен запит вимушений сканувати всю таблицю, що може значно збільшити час виконання, особливо при великій кількості рядків.
- 2) Запити, які містять фільтрацію, агрегацію або сортування, можуть бути повільнішими, оскільки СУБД не може швидко знайти потрібні рядки без індексу.
- 3) Наприклад, запит із LIKE 'abc%' без індексу потребує повного сканування таблиці для пошуку всіх рядків, що починаються на 'abc'.

2. Запити з індексом BTree:

- 1) Індекс BTree значно покращує швидкість виконання запитів, оскільки дозволяє СУБД швидше знаходити відповідні рядки завдяки сортуванню даних та оптимізації пошуку через дерево.
- 2) Запити, що містять порівняння за операторами (наприклад, =, >, <, >=), набагато швидші при використанні BTree індексу. Індекс дає змогу уникнути сканування всіх рядків таблиці.
- 3) 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.
- 4) Агрегатні функції (COUNT(), MAX(), MIN()) також працюють значно швидше при використанні індексу, оскільки індекс дає змогу обробляти лише частину таблиці, що відповідає умовам запиту.

3. Запити з групуванням та сортуванням:

- 1) Без індексу, групування та сортування можуть бути дуже повільними, оскільки СУБД повинна повністю обробити всі рядки таблиці.
- 2) 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.

4. Запити з умовами, групуванням і сортуванням:

- 1) Запити, що включають складніші умови (наприклад, WHERE $id < 500000$ з групуванням та сортуванням), за відсутності індексу можуть бути дуже повільними.
- 2) 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 В-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`:

Query	Query History	Scratch Pad
1 CREATE TABLE <code>gin_test</code> (2 id <code>BIGSERIAL PRIMARY KEY</code> , 3 string <code>VARCHAR(100)</code> 4);		
Data Output		Messages
CREATE TABLE		
Query returned successfully in 115 msec.		

Query	Query History	Scratch Pad
1 INSERT INTO <code>gin_test</code> (<code>string</code>) 2 SELECT <code>md5(random()::text)</code> 3 FROM <code>generate_series(1, 1000000)</code> ; 4		
Data Output		Messages
INSERT 0 1000000		
Query returned successfully in 4 secs 35 msec.		

	id [PK] bigint	string character varying (100)
1	1	c89915489e9244814a3fe36315abd3...
2	2	90fc03f3cc7a55c726c4aad1cc4bb0f
3	3	b357f9185d363090e11a23deb30d7f45
4	4	cd14aec256549d6eae551408324c2...
5	5	4d683ce38d4e0991a615783a655e26...
6	6	415fdb06bf10059051751407dc134382
7	7	7096eb605220d9ee4a8b5901f69ab2eb
8	8	5b26a68a3004cfbb22dac2f3093b8913
9	9	90de011fa4434104e2de651ba8922fff
10	10	28a2e21efc8fd6b74cfb0498bba60377
11	11	90274051ea4a1c57f1b1bb85829267...
12	12	031d54b8bf89d2448c01876a98705b...
13	13	f50af48e10a7a95fb9d806fe4ecb2f0e
14	14	b2b85f7e8aa5c9d84a594153a21290bc
15	15	68cd0ed7af4a24b924eeae598a216f74
16	16	510202ff0f055024052f00fd8c620570

Для тестування візьмемо 4 запити:

5. EXPLAIN ANALYZE SELECT * FROM btree_test WHERE string LIKE 'abc%';
6. EXPLAIN ANALYZE SELECT COUNT(*), MAX(id), MIN(id) FROM btree_test;
7. EXPLAIN ANALYZE SELECT string, COUNT(*) FROM btree_test GROUP BY string ORDER BY string DESC;
8. 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

✓
Successfully run. Total query runtime: 158 msec. 8 rows affected.
✕

Запит №2

✓ Successfully run. Total query runtime: 165 msec. 8 rows affected. ✕

Запит №3

✓ Successfully run. Total query runtime: 1 secs 732 msec. 19 rows affected. ✕

Запит №4

✓ Successfully run. Total query runtime: 491 msec. 9 rows affected. ✕

З індексом GIN

Запит №1

✓ Successfully run. Total query runtime: 153 msec. 8 rows affected. ✕

Запит №2

✓ Successfully run. Total query runtime: 165 msec. 8 rows affected. ✕

Запит №3

✓ Successfully run. Total query runtime: 1 secs 738 msec. 19 rows affected. ✕

Запит №4

✓ Successfully run. Total query runtime: 488 msec. 9 rows affected. ✕

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:

- 1) 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.

- 2) 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.

2. Запит із агрегатними функціями:

- 1) 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.

3. Запит із групуванням і сортуванням:

- 1) 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.
- 2) 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.

4. Запит із умовами, групуванням і сортуванням:

- 1) 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.
- 2) 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;

```

The screenshot shows a SQL IDE with a query editor and a messages pane. The query editor contains the following SQL code:

```

1 CREATE OR REPLACE FUNCTION employee_trigger_function()
2 RETURNS TRIGGER AS $$
3 DECLARE
4     rec RECORD;
5 BEGIN
6     IF (TG_OP = 'INSERT') THEN
7         RAISE NOTICE 'New employee added: Name: %, Position: %, Salary: %', NEW.name, NEW.position, NEW.salary;
8     END IF;
9
10    IF (TG_OP = 'UPDATE') THEN
11        IF NEW.salary <> OLD.salary THEN
12            RAISE NOTICE 'Employee updated: Name: %, Old salary: %, New salary: %', NEW.name, OLD.salary, NEW.salary;
13        END IF;
14
15        FOR rec IN SELECT * FROM employees WHERE salary > OLD.salary LOOP
16            RAISE NOTICE 'Processing employee: ID %, Name %, Salary %', rec.employee_id, rec.name, rec.salary;
17        END LOOP;
18    END IF;
19
20    RETURN NEW;
21 END;
22 $$ LANGUAGE plpgsql;
23

```

The messages pane shows the following output:

```

CREATE FUNCTION
Query returned successfully in 151 msec.

```

Створення тригера

```

CREATE TRIGGER employee_trigger
AFTER INSERT OR UPDATE ON employees
FOR EACH ROW
EXECUTE FUNCTION employee_trigger_function();

```

The screenshot shows a SQL IDE with a query editor. The query editor contains the following SQL code:

```

1 CREATE TRIGGER employee_trigger
2 AFTER INSERT OR UPDATE ON employees
3 FOR EACH ROW
4 EXECUTE FUNCTION employee_trigger_function();
5

```

```
Data Output Messages Notifications
CREATE TRIGGER

Query returned successfully in 132 msec.
```

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);
```

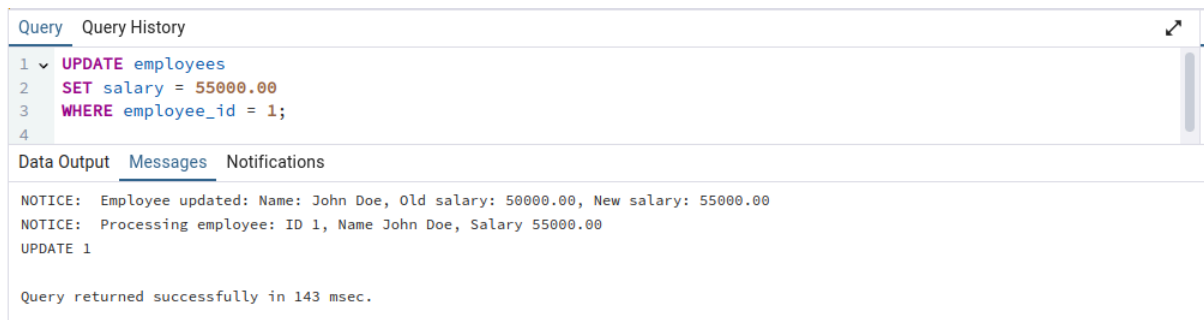
```
Query Query History
1 INSERT INTO employees (name, position, salary)
2 VALUES ('John Doe', 'Software Engineer', 50000.00);
3

Data Output Messages Notifications
NOTICE: New employee added: Name: John Doe, Position: Software Engineer, Salary: 50000.00
INSERT 0 1

Query returned successfully in 147 msec.
```

```
UPDATE employees
SET salary = 55000.00
WHERE employee_id = 1;
```


Результат: Якщо зарплата змінилась, виведеться повідомлення:



```
Query History
1 UPDATE employees
2 SET salary = 55000.00
3 WHERE employee_id = 1;
4

Data Output Messages Notifications
NOTICE: Employee updated: Name: John Doe, Old salary: 50000.00, New salary: 55000.00
NOTICE: Processing employee: ID 1, Name John Doe, Salary 55000.00
UPDATE 1

Query returned successfully in 143 msec.
```

Висновок щодо створеного тригера

1. Функціональність тригера:

- 1) Тригер успішно виконує завдання, пов'язані з обробкою даних у таблиці employees.
- 2) Він спрацьовує після подій INSERT та UPDATE, що дозволяє автоматично обробляти нові записи та оновлення.

2. Реалізовані можливості:

- 1) Для події INSERT: Тригер записує повідомлення з інформацією про нового працівника, включаючи його ім'я, посаду та зарплату.
- 2) Для події UPDATE: Тригер відстежує зміни у зарплаті працівника, записуючи старе та нове значення.

3. Коректність реалізації:

- 1) Тригерна функція використовує ключові конструкції PostgreSQL, такі як NEW і OLD, що забезпечує доступ до нових і старих значень зміненого запису.
- 2) Форматування повідомлень у функції є точним, а кількість параметрів у RAISE NOTICE відповідає кількості змінних, що виключає синтаксичні помилки.

4. Ефективність:

- 1) Використання тригера забезпечує автоматизацію процесів, таких як логування або моніторинг змін, без необхідності виконувати ці операції вручну.
- 2) 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)
 - 1) У цьому режимі транзакція може читати зміни, виконані іншими транзакціями, навіть якщо ці зміни ще не підтверджені (COMMIT).
 - 2) Феномени: можливі "брудні читання", "неповторювані читання" і "фантомні читання".
 - 3) PostgreSQL піднімає цей рівень до Read Committed, оскільки не дозволяє читати непідтверджені зміни.
2. Read Committed (Читання підтверджених даних)
 - 1) За замовчуванням у PostgreSQL.
 - 2) Кожен запит бачить тільки ті зміни, які були підтверджені іншими транзакціями до початку виконання запиту.
 - 3) Феномени: можливі "неповторювані читання" і "фантомні читання".
3. Repeatable Read (Повторюване читання)
 - 1) У цьому режимі транзакція бачить стан бази даних на момент її початку.
 - 2) Транзакція не побачить змін, виконаних іншими транзакціями, навіть якщо вони були підтверджені.
 - 3) Феномени: можливі "фантомні читання".
4. Serializable (Серіалізований доступ)
 - 1) Найвищий рівень ізоляції, де транзакції виконуються так, ніби вони серіалізовані (виконуються одна за одною).
 - 2) Феномени: жодні феномени не можливі.
 - 3) 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;
```

Query

Query History

Scratch Pad x

1

BEGIN;

2

UPDATE accounts SET balance = balance - 50 WHERE id =

Data Output

Messages

Notifications

UPDATE 1

Query returned successfully in 124 msec.

```
BEGIN;
```

```
SELECT * FROM accounts WHERE id = 1;
```

```
COMMIT;
```

Query

Query History

Scratch Pad x

1

BEGIN;

2

SELECT * FROM accounts WHERE id = 1;

3

4

Data Output

Messages

Notifications

Showing rows: 1 to 1

Page No: 1 of 1

	Id [PK] integer	balance numeric
1	1	50

Query

Query History

Scratch Pad

1

SELECT * FROM public.accounts

2

ORDER BY id ASC

Data Output

Messages

Notifications

Showing rows: 1 to 3

Page No: 1 of 1

	id [PK] integer	balance numeric
1	1	100
2	2	200
3	3	300

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;
```

Query

Query History

Scratch Pad

1

BEGIN;

2

UPDATE accounts SET balance = balance - 50 WHERE id =

3

Data Output

Messages

Notifications

WARNING: there is already a transaction in progress

UPDATE 1

Query returned successfully in 160 msec.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;
SELECT * FROM accounts WHERE id = 2;
```

COMMIT;

The screenshot shows a SQL IDE interface. The top bar has tabs for 'Query', 'Query History', and 'Scratch Pad'. The 'Query' tab is active, displaying the following SQL code:

```
1 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
2 BEGIN;  
3 SELECT * FROM accounts WHERE id = 2;  
4
```

Below the query window, there are tabs for 'Data Output', 'Messages', and 'Notifications'. The 'Data Output' tab is active, showing a table with the following data:

	Id [PK] integer	balance numeric
1	2	200

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;
```

The screenshot shows a SQL IDE interface. The top bar has tabs for 'Query', 'Query History', and 'Scratch Pad'. The 'Query' tab is active, displaying the following SQL code:

```
1 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
2 BEGIN;  
3 INSERT INTO accounts (balance) VALUES (400);  
4 COMMIT;  
5
```

Below the query window, there are tabs for 'Data Output', 'Messages', and 'Notifications'. The 'Messages' tab is active, showing the following message:

```
COMMIT  
  
Query returned successfully in 117 msec.
```

Query

Query History

Scratch Pad x

1

SELECT * FROM public.accounts

2

ORDER BY id ASC

Data Output

Messages

Notifications

Showing rows: 1 to 5

Page No: 1 of 1

	id [PK] integer	balance numeric
1	1	100
2	2	200
3	3	300
4	4	400
5	5	400

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

Порівняння рівнів ізоляції

Рівень	Брудне читання	Неповторюване читання	Фантомне читання
Read Uncommitted	Можливе	Можливе	Можливе
Read Committed	Неможливе	Можливе	Можливе
Repeatable Read	Неможливе	Неможливе	Можливе
Serializable	Неможливе	Неможливе	Неможливе

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.