

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

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

**Лабораторна робота № 2**

з дисципліни

**«Бази даних та засоби управління»**

**Тема: «Засоби оптимізації роботи СУБД PostgreSQL»**

Виконала: студентка 3 курсу

ФПМ групи КВ-12

Бурда Є. А.

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.

*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 №3

|  |  |  |
| --- | --- | --- |
| *№ варіанта* | *Види індексів* | *Умови для тригера* |
| 3 | *GIN, Hash* | *before delete, update* |

Github: <https://github.com/evaburda/lab2->

Telegram: https://t.me/night\_fool

**Сутності**

**Patient** з атрибутами : ID пацієнта, ім’я, прізвище, дата народження, адреса.

**Medical Worker** з атрибутами: ID працівника, ім'я, прізвище, спеціальність.

**Medical Record** з атрибутами: ID запису, дата та час, опис симптомів, діагноз.

**Medicine** з атрибутами: ID лікарського засобу, назва, дозування

**Опис зв’язків**

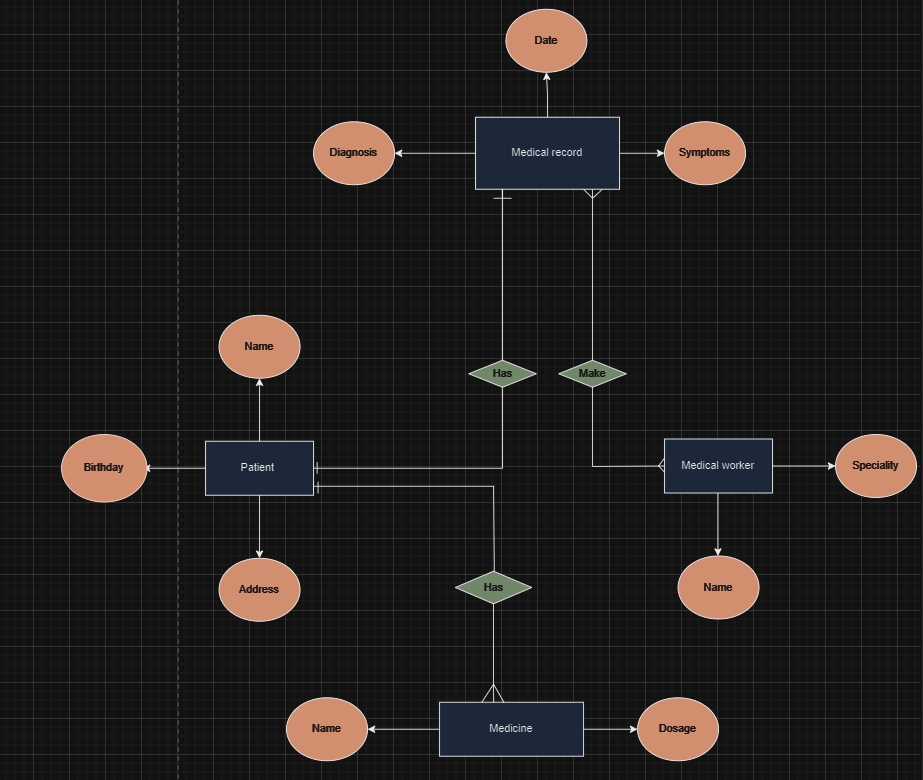
Patient має medical record: вказує на те, що у кожного пацієнта є один медичний запис, та медичний запис відноситься до одного пацієнта.

Medical worker створює medical record: вказує на те, що медичний працівник може створювати багато медичних записів і багато медичних записів створені працівниками

Patient має medicine: вказує на те, що пацієнт може мати декілька видів ліків, але кожні ліки відносяться лише до одного пацієнта.

Medicine – це ліки для кожного пацієнта, вони можуть існувати без пацієнта так само як пацієнт може існувати без ліків.

***ER-діаграма, побудована за нотацією Чена***



***Схема бази даних у графічному вигляді***

**Зображення, що містить текст, знімок екрана, схема, ряд

Автоматично згенерований опис**

**Завдання №1**

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

1. Patient
2. Medicine
3. Medical\_Record
4. Medical\_Worker
5. Worker\_Record

**Patient**

Таблиця Patient має такі стовпці: Patient\_ID (ідентифікатор пацієнта), Name (ім’я пацієнта), Birthday (дата народження пацієнта), Address (адреса проживання пацієнта). Також наявні зв’язки із таблицями Medicine та Medical\_Record, тому в цьому класі встановлені зв’язки relationship("Medicine") та relationship("Medical\_Record").

Програмна реалізація класу Patient:

**class** Patient(Base):

\_\_tablename\_\_ = 'Patient'

Patient\_ID = Column(Integer, primary\_key=True)

Name = Column(String)

Birthday = Column(Date)

Address = Column(String)

medicine = relationship("Medicine")

medical\_record = relationship("Medical\_Record")

**def** \_\_init\_\_(self, patient\_id, name, birthday, address):

self.Patient\_ID = patient\_id

self.Name = name

self.Birthday = birthday

self.Address = address

**def** \_\_repr\_\_(self):

**return** f"<Patients(patient\_id={self.Patient\_ID}, name={self.Name}, birthday={self.Birthday}, address={self.Address})>"

**Medicine**

Таблиця Medicine має такі стовпці: Medicine\_ID (ідентифікатор рецепту ліків), Patient\_ID (зовнішній ключ, який пов’язує рецепт ліків із пацієнтом), Name (назва ліків), Dosage (дозування ліків).

Програмна реалізація класу Medicine:

**class** Medicine(Base):

\_\_tablename\_\_ = 'Medicine'

Medicine\_ID = Column(Integer, primary\_key=True)

Name = Column(String)

Dosage = Column(String)

Patient\_ID = Column(Integer, ForeignKey('Patient.Patient\_ID'))

**def** \_\_init\_\_(self, medicine\_id, patient\_id, name, dosage):

self.Medicine\_ID = medicine\_id

self.Patient\_ID = patient\_id

self.Name = name

self.Dosage = dosage

**def** \_\_repr\_\_(self):

**return** f"<Medicines(medicine\_id={self.Medicine\_ID}, patient\_id={self.Patient\_ID}, name={self.Name}, dosage={self.Dosage})>"

**Medical\_Record**

Таблиця Medical\_Record має такі стовпці: Record\_ID (ідентифікатор запису пацієнта), Patient\_ID (зовнішній ключ, який пов’язує запис із пацієнтом), Date (дата запису), Symptoms (симптоми), Diagnosis (діагноз). Також наявний зв’язок із таблицею Worker\_Record, тому в цьому класі встановлений зв’язок relationship("Worker\_Record").

Програмна реалізація класу Medical\_Record:

**class** Medical\_Record(Base):

\_\_tablename\_\_ = 'Medical\_Record'

Record\_ID = Column(Integer, primary\_key=True)

Date = Column(Date)

Symptoms = Column(String)

Diagnosis = Column(String)

Patient\_ID = Column(Integer, ForeignKey('Patient.Patient\_ID'))

worker\_record = relationship("Worker\_Record")

**def** \_\_init\_\_(self, record\_id, patient\_id, date, symptoms, diagnosis):

self.Record\_ID = record\_id

self.Patient\_ID = patient\_id

self.Date = date

self.Symptoms = symptoms

self.Diagnosis = diagnosis

**def** \_\_repr\_\_(self):

**return** f"<Medical\_Records(record\_id={self.Record\_ID}, patient\_id={self.Patient\_ID}, date={self.Date}, symptoms={self.Symptoms}, diagnosis={self.Diagnosis})>"

**Medical\_Worker**

Таблиця Medical\_Worker має такі стовпці: Worker\_ID (ідентифікатор лікаря), Name (ім’я лікаря), Speciality (спеціальність лікаря). Також наявний зв’язок із таблицею Worker\_Record, тому в цьому класі встановлений зв’язок relationship("Worker\_Record").

Програмна реалізація класу Medical\_Worker:

**class** Medical\_Worker(Base):

\_\_tablename\_\_ = 'Medical\_Worker'

Worker\_ID = Column(Integer, primary\_key=True)

Name = Column(String)

Speciality = Column(String)

worker\_record = relationship("Worker\_Record")

**def** \_\_init\_\_(self, worker\_id, name, speciality):

self.Worker\_ID = worker\_id

self.Name = name

self.Speciality = speciality

**def** \_\_repr\_\_(self):

**return** f"<Medical\_Worker(worker\_id={self.Worker\_ID}, name={self.Name}, speciality={self.Speciality})>"

**Worker\_Record**

Таблиця Worker\_Record має такі стовпці: Tab\_ID (ідентифікатор поля), Worker\_ID (зовнішній ключ, який посилається на лікаря), Record\_ID (зовнішній ключ, який посилається на запис).

Програмна реалізація класу Worker\_Record:

**class** Worker\_Record(Base):

\_\_tablename\_\_ = 'Worker\_Record'

Tab\_ID = Column(Integer, primary\_key=True)

Worker\_ID = Column(Integer, ForeignKey('Medical\_Worker.Worker\_ID'))

Record\_ID = Column(Integer, ForeignKey('Medical\_Record.Record\_ID'))

**def** \_\_init\_\_(self, tab\_id, worker\_id, record\_id):

self.Tab\_ID = tab\_id

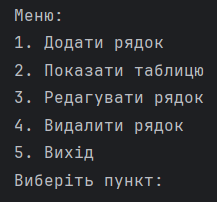
self.Worker\_ID = worker\_id

self.Record\_ID = record\_id

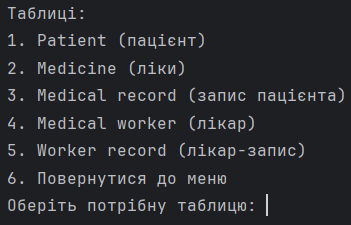
**def** \_\_repr\_\_(self):

**return** f"<Worker\_Record(tab\_id={self.Tab\_ID}, worker\_id={self.Worker\_ID}, record\_id={self.Record\_ID})>"

Меню виглядає наступним чином і включає в себе 5 пунктів. Розглянемо їх далі.



*Рисунок 2.1 – Вигляд меню*



*Рисунок 2.2 – Список таблиць*

Пункт №1 – «Додати рядок»

Після вибору цього пункту, відкривається список доступних таблиць (рисунок 2.2), де потрібно вибрати таблицю, до якої бажаємо додати рядок.

Після вибору таблиці, користувачу потрібно ввести всі необхідні дані для нового рядка, після чого буде повідомлено про успішну дію або помилку.

Пункт №2 – «Показати таблицю»

Після вибору цього пункту, відкривається список доступних таблиць (рисунок 2.2), де потрібно вибрати таблицю, яку бажаємо побачити.

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

Пункт №3 – «Редагувати рядок»

Після вибору цього пункту, відкривається список доступних таблиць (рисунок 2.2), де потрібно вибрати таблицю, в якій бажаємо зробити зміну.

Після вибору таблиці, користувачу потрібно ввести ідентифікатор існуючого рядку в таблиці. Потім відповідно внести нові дані, після чого буде повідомлено про успішну дію або помилку.

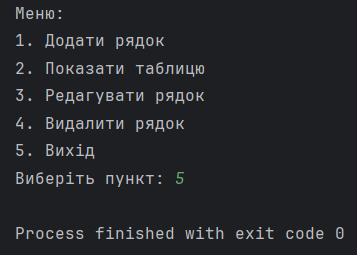
Пункт №4 – «Видалити рядок»

Після вибору цього пункту, відкривається список доступних таблиць (рисунок 2.2), де потрібно вибрати таблицю, в якій бажаємо видалити рядок.

Після вибору таблиці, користувачу потрібно ввести ідентифікатор існуючого рядку в таблиці, після чого рядок буде видалено в разі успіху або повідомлено про помилку.

Пункт №5 – «Вихід»

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

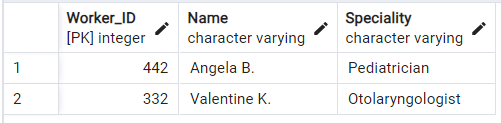


***Приклади запитів у вигляді ORM***

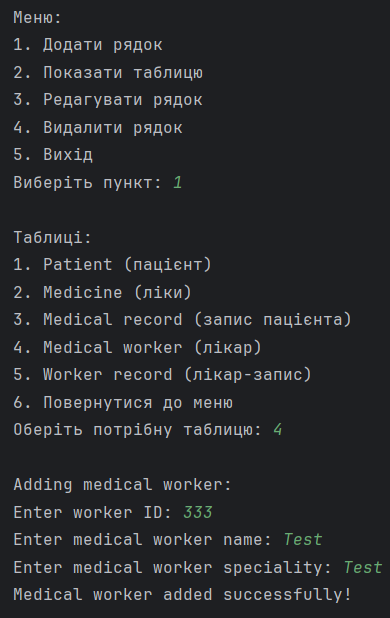
Для демонстрації запитів виберемо по 1-2 таблиці до кожного.

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

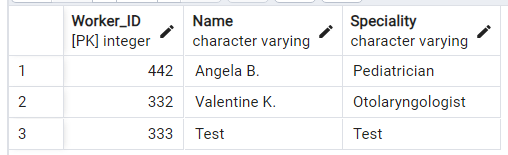
***Таблиця “Medical\_Worker” до вставки:***

******

***Процес додавання рядка:***

******

***Таблиця “Medical\_Worker” після вставки:***



Лістинг функцій insert для кожної таблиці:

**def** insert\_patient(self, patient\_id: int, name: str, birthday: Date, address: str) -> None:

patient = Patient(patient\_id=patient\_id, name=name, birthday=birthday, address=address)

s.add(patient)

s.commit()

**def** insert\_medicine(self, medicine\_id: int, patient\_id: int, name: str, dosage: str) -> None:

medicine = Medicine(medicine\_id=medicine\_id, patient\_id=patient\_id, name=name, dosage=dosage)

s.add(medicine)

s.commit()

**def** insert\_medical\_record(self, record\_id: int, patient\_id: int, date: Date, symptoms: str, diagnosis: str) -> None:

medical\_record = Medical\_Record(record\_id=record\_id, patient\_id=patient\_id, date=date, symptoms=symptoms, diagnosis=diagnosis)

s.add(medical\_record)

s.commit()

**def** insert\_medical\_worker(self, worker\_id: int, name: str, speciality: str) -> None:

medical\_worker = Medical\_Worker(worker\_id=worker\_id, name=name, speciality=speciality)

s.add(medical\_worker)

s.commit()

**def** insert\_worker\_record(self, tab\_id: int, worker\_id: int, record\_id: int) -> None:

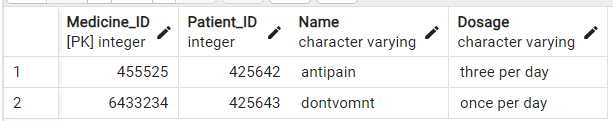
worker\_record = Worker\_Record(tab\_id=tab\_id, worker\_id=worker\_id, record\_id=record\_id)

s.add(worker\_record)

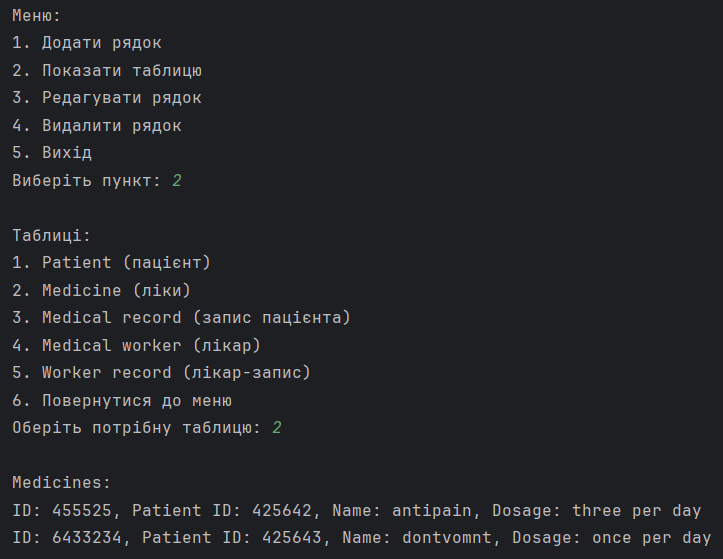
s.commit()

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

***Таблиця “Medicine”:***

******

***Процес показу таблиці:***



Лістинг функцій show для кожної таблиці:

**def** show\_patients(self):

**return** s.query(Patient.Patient\_ID, Patient.Name, Patient.Birthday, Patient.Address).all()

**def** show\_medicines(self):

**return** s.query(Medicine.Medicine\_ID, Medicine.Patient\_ID, Medicine.Name, Medicine.Dosage).all()

**def** show\_medical\_records(self):

**return** s.query(Medical\_Record.Record\_ID, Medical\_Record.Patient\_ID, Medical\_Record.Date, Medical\_Record.Symptoms, Medical\_Record.Diagnosis).all()

**def** show\_medical\_workers(self):

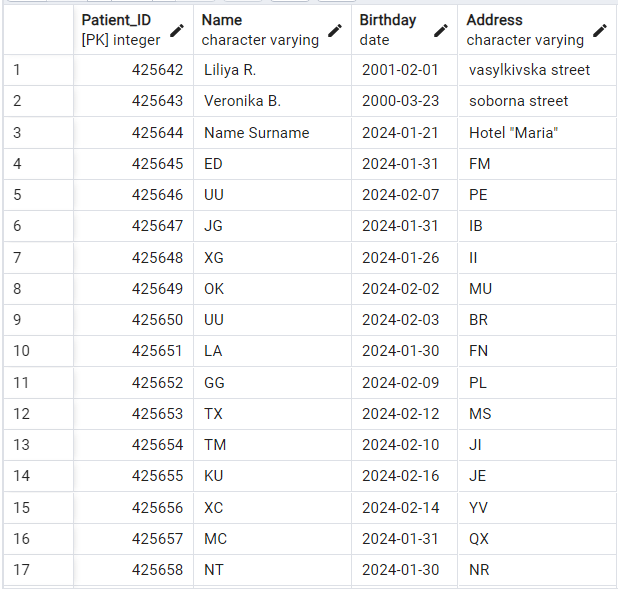
**return** s.query(Medical\_Worker.Worker\_ID, Medical\_Worker.Name, Medical\_Worker.Speciality).all()

**def** show\_workers\_records(self):

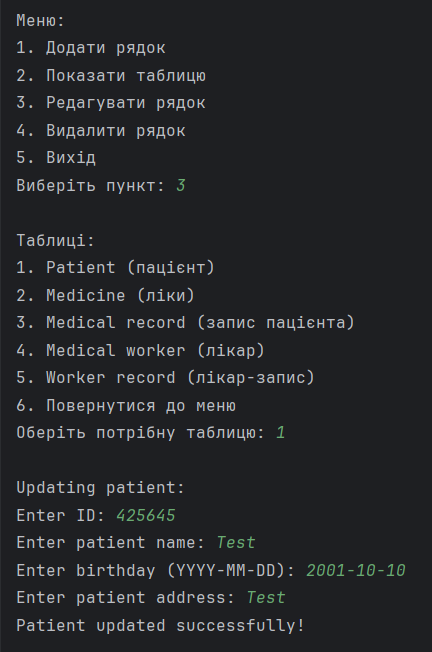
**return** s.query(Worker\_Record.Tab\_ID, Worker\_Record.Worker\_ID, Worker\_Record.Record\_ID).all()

***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 реадгування рядка.

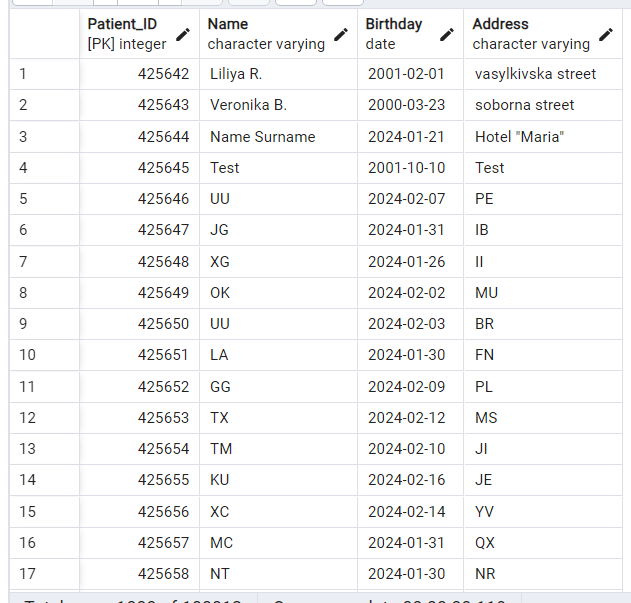
***Таблиця “Patient” до редагування:***

******

***Процес редагування рядка:***

******

***Таблиця “Patient” після редагування:***



Лістинг функцій update для кожної таблиці:

**def** update\_patient(self, name: str, birthday: Date, address: str, patient\_id: int) -> None:

s.query(Patient).filter\_by(Patient\_ID=patient\_id).update({Patient.Name: name, Patient.Birthday: birthday, Patient.Address: address})

s.commit()

**def** update\_medicine(self, patient\_id: int, name: str, dosage: str, medicine\_id: int) -> None:

s.query(Medicine).filter\_by(Medicine\_ID=medicine\_id).update({Medicine.Patient\_ID: patient\_id, Medicine.Name: name, Medicine.Dosage: dosage})

s.commit()

**def** update\_medical\_record(self, patient\_id: int, date: Date, symptoms: str, diagnosis: str, record\_id: int) -> None:

s.query(Medical\_Record).filter\_by(Record\_ID=record\_id).update({Medical\_Record.Patient\_ID: patient\_id, Medical\_Record.Date: date, Medical\_Record.Symptoms: symptoms, Medical\_Record.Diagnosis: diagnosis})

s.commit()

**def** update\_medical\_worker(self, name: str, speciality: str, worker\_id: int) -> None:

s.query(Medical\_Worker).filter\_by(Worker\_ID=worker\_id).update({Medical\_Worker.Name: name, Medical\_Worker.Speciality: speciality})

s.commit()

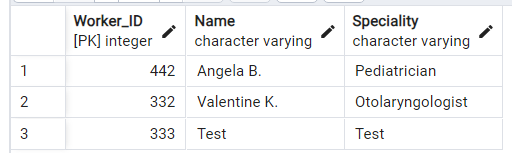
**def** update\_worker\_record(self, worker\_id: int, record\_id: int, tab\_id: int) -> None:

s.query(Worker\_Record).filter\_by(Tab\_ID=tab\_id).update({Worker\_Record.Worker\_ID: worker\_id, Worker\_Record.Record\_ID: record\_id})

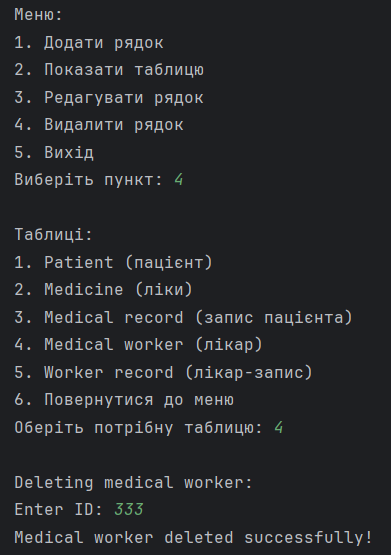
s.commit()

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

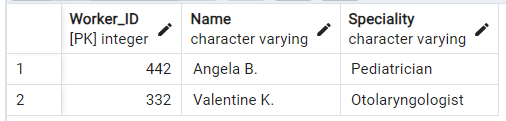
***Таблиця “Medical\_Worker” до видалення:***

******

***Процес видалення рядка:***

******

***Таблиця “Medical\_Worker” після видалення:***



Лістинг функцій delete для кожної таблиці:

**def** delete\_patient(self, patient\_id) -> None:

patient = s.query(Patient).filter\_by(Patient\_ID=patient\_id).one()

s.delete(patient)

s.commit()

**def** delete\_medicine(self, medicine\_id) -> None:

medicine = s.query(Medicine).filter\_by(Medicine\_ID=medicine\_id).one()

s.delete(medicine)

s.commit()

**def** delete\_medical\_record(self, record\_id) -> None:

medical\_record = s.query(Medical\_Record).filter\_by(Record\_ID=record\_id).one()

s.delete(medical\_record)

s.commit()

**def** delete\_medical\_worker(self, worker\_id) -> None:

medical\_worker = s.query(Medical\_Worker).filter\_by(Worker\_ID=worker\_id).one()

s.delete(medical\_worker)

s.commit()

**def** delete\_worker\_record(self, tab\_id) -> None:

worker\_record = s.query(Worker\_Record).filter\_by(Tab\_ID=tab\_id).one()

s.delete(worker\_record)

s.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 записів.

***GIN***

Індекс GIN (Generalized Inverted Index) — це тип індексу у системі управління базами даних, який дозволяє ефективно виконувати пошук та фільтрацію для деяких типів даних, які не обробляються традиційними B-деревами (Binary Tree Index).

GIN особливо корисний для оптимізації запитів, які використовують рядкові масиви, текстові дані, JSON-структури та інші складні типи даних. Онлайн-пошук, фільтрація, і взаємодія з елементами цих типів стає значно ефективнішою завдяки GIN індексу.

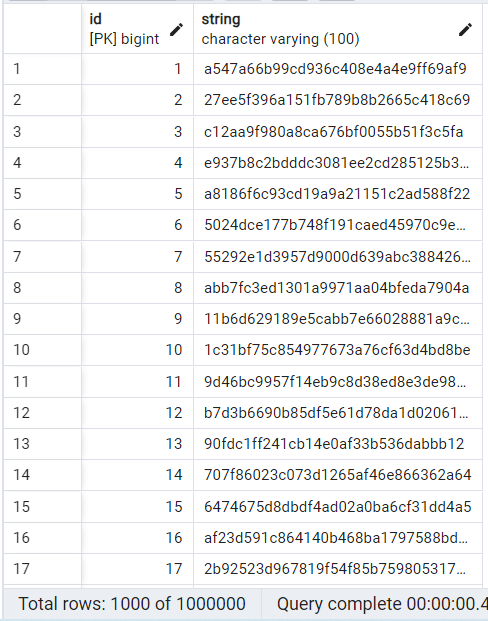
Для дослідження індексу була створена таблиця gin\_test, яка має дві колонки: “id” та “string”:

**CREATE** **TABLE** "gin\_test"("id" bigserial **PRIMARY** **KEY**, "string" **varchar**(100));

**INSERT** **INTO** "gin\_test"("id", "string")

**SELECT** generate\_series **as** "id", md5(**random**()::text)

**FROM** generate\_series(1, 1000000)



Для тестування візьмемо 4 запити:

**SELECT** \* **FROM** gin\_test **WHERE** "string" LIKE 'a%';

**SELECT** COUNT(\*) **FROM** gin\_test;

**SELECT** "string", COUNT(\*) **FROM** gin\_test **GROUP** **BY** "string";

**SELECT** \* **FROM** gin\_test **ORDER** **BY** "string" **ASC**;

Створення індексу:

**CREATE** **INDEX** gin\_index **ON** gin\_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



Отже, з отриманих результатів можна побачити, що лише запит №2 виконався швидше без індексу gin. Цей запит не використовує стовпець "string" індексу GIN, оскільки ми використовуємо COUNT(\*), що означає підрахунок усіх рядків у таблиці. Індекс GIN створений для полегшення пошуку та фільтрації даних в текстових стовпцях, але в даному випадку він не задіяний, оскільки ми просто отримуємо загальну кількість рядків.

Якщо ми використовуємо COUNT(\*) без фільтрації за текстовим стовпцем, база даних може вибрати інший метод виконання запиту, який може бути більш оптимальним в даному випадку.

***Hash***

Хеш-індекс (hash) використовує хеш-функцію для створення внутрішнього представлення ключа і визначення позиції (адреси) у віртуальному масиві чи таблиці. Основна ідея полягає в тому, що за допомогою хеш-функції можна ефективно призначити унікальний код (хеш) кожному ключевому значенню. Потім цей хеш використовується для знаходження місця зберігання в пам'яті чи на диску.

Одна з особливостей хеш-індексу - це швидкий пошук за значенням ключа, оскільки хеш-функція дозволяє швидко визначити адресу в пам'яті чи на диску, де повинно знаходитися відповідне значення.

Проте, є деякі обмеження та проблеми при використанні хеш-індексів. Наприклад, можливі колізії, коли різні ключі мають однаковий хеш-код і відображаються на одну і ту ж саму адресу. Для вирішення цього можливі проблеми використовуються різні техніки, такі як використання відкритої адресації, використання ланцюгів для управління колізіями тощо.

Хеш-індекси ефективні при швидкому пошуку великої кількості даних за ключем, але їх використання обмежується конкретними властивостями хеш-функції та можливими колізіями.

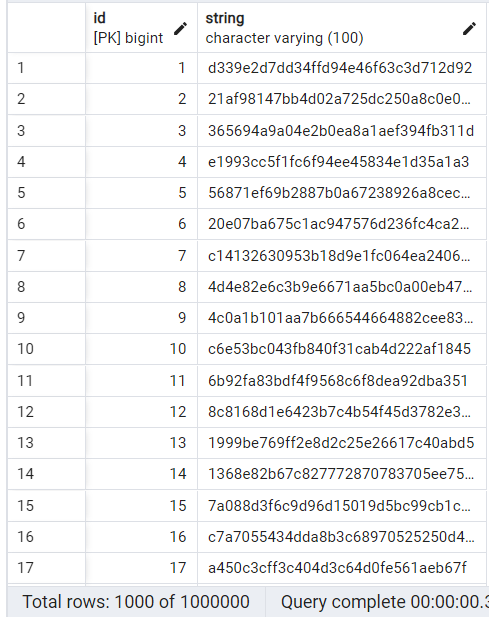
Для дослідження індексу була створена таблиця hash\_test, яка має дві колонки: “id” та “string”:

**CREATE** **TABLE** "hash\_test"("id" bigserial **PRIMARY** **KEY**, "string" **varchar**(100));

**INSERT** **INTO** "hash\_test"("id", "string")

**SELECT** generate\_series **as** "id", md5(**random**()::text)

**FROM** generate\_series(1, 1000000)



Для тестування візьмемо 4 запити:

**SELECT** \* **FROM** hash\_test **WHERE** "string" LIKE 'a%';

**SELECT** COUNT(\*) **FROM** hash\_test;

**SELECT** "string", COUNT(\*) **FROM** hash\_test **GROUP** **BY** "string";

**SELECT** \* **FROM** hash\_test **ORDER** **BY** "string" **ASC**;

Створення індексу:

**CREATE** **INDEX** hash\_index **ON** hash\_test **USING** hash("string");

Результати виконання запитів

Без індекса hash

Запит №1



Запит №2



Запит №3



Запит №4



З індексом hash

Запит №1



Запит №2



Запит №3



Запит №4



Отже, з отриманих результатів можна побачити, що запити №1, 3 і 4 спрацювали швидше без індексу hash. Розглянемо кожен із них:

1. Запит №1 (SELECT \* FROM hash\_test WHERE "string" LIKE 'a%')

У цьому запиті вибираються всі рядки, де колонка "string" починається з літери 'a'. Враховуючи, що в таблиці мільйон записів, і в індексі GIN зберігається інформація про кожне унікальне значення хешу, операція LIKE може виявитися швидше при пошуку в повному тексті, не використовуючи індекс.

1. Запит №3 (SELECT "string", COUNT(\*) FROM hash\_test GROUP BY "string")

У цьому запиті використовується групування за значенням колонки "string" і підрахунок кількості записів для кожного унікального значення. Якщо велика кількість унікальних значень "string" у таблиці, то може виявитися, що використання індексу GIN для такого запиту не є оптимальним, оскільки індекс повинен бути просканований.

1. Запит №4 (SELECT \* FROM hash\_test ORDER BY "string" ASC)

У цьому запиті виконується сортування за колонкою "string". Якщо дані вже впорядковані у таблиці або якщо обсяг даних досить малий, то використання індексу GIN може бути надмірним і може виявитися швидше використовувати простий підхід до сортування без індексів.

Загалом, ефективність використання індексів залежить від конкретних характеристик даної таблиці та запитів. У деяких випадках індекси можуть прискорити операції, а в інших навпаки уповільнити через додатковий обсяг даних, що пов'язаний з індексом.