

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування та спеціалізованих**

**комп’ютерних систем**

**ЛАБОРАТОРНА РОБОТА №2**

**«Засоби оптимізації роботи СУБД PostgreSQL»**

з дисципліни

**«Бази даних та засоби управління»**

Виконав:

студент III курсу

групи КВ-12

Чернишков О.О.

Перевірив:

Павловський В.І.

*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 №24

|  |  |  |
| --- | --- | --- |
| *№ варіанта* | *Види індексів* | *Умови для тригера* |
| *24* | *GIN, BRIN* | *after update, insert* |

*Посилання на репозиторій GitHub:* <https://github.com/SashaBlack01/DBLabs>

*Контакт студента в Телеграм:* @Sasha\_Black01

**Сутності**

Згідно з обраною галуззю для побудови бази даних було виділено наступні сутності:

* Посилка (Parcel) з атрибутами: номер посилки, статус, дата створення, вартість, ідентифікатор кур’єра, ідентифікатор отримувача, ідентифікатор складу. Призначення: збереження даних щодо посилок.
* Отримувач (Recipient) з атрибутами: ідентифікатор, ім’я, адреса. Призначення: збереження даних отримувачів посилок.
* Відправник (Sender) з атрибутами: ідентифікатор, ім’я, адреса. Призначення: збереження даних відправників посилок.
* Кур’єр (Courier) з атрибутами: службовий номер, ім’я, номер телефону.
* Склад (Warehouse) з атрибутами: номер складу, адреса, номер телефону.

**Опис зв’язків**

Посилка має найбільшу кількість зв’язків з іншими сутностями:

* Кожна посилка має зв’язок з кур’єром який доставляє посилку, причому тільки з одним, але кожен кур’єр може доставляти декілька посилок, тому зв’язок 1:N.
* Кожна посилка має зв’язок з отримувачем, до якого вона прямує, при чому тільки з одним, але до отримувача може прямувати багато посилок, тому зв’язок 1:N.
* Кожна посилка має зв’язок з відправником, який створив та надіслав посилку, при чому відправник може бути тільки один для кожної посилки, тому зв’язок 1:N.
* Кожна посилка має зв’язок зі складом, на якому вона тимчасово зберігається, при чому склад для конкретної посилки може бути тільки один, але на одному складі може бути декілька посилок, тому зв’язок 1:N.
* Кожен кур’єр має зв’язок зі складом, до якого від закріплений, щоб доставляти посилки з конкретного складу, при чому кожен кур’єр може бути закріпленим за декількома складами, а за кожним складом може бути закріплено декілька кур’єрів, тому зв’язок M:N.

Графічне подання моделі «сутність зв’язок» побудованої за нотацією Чена представлена на рисунку 1.

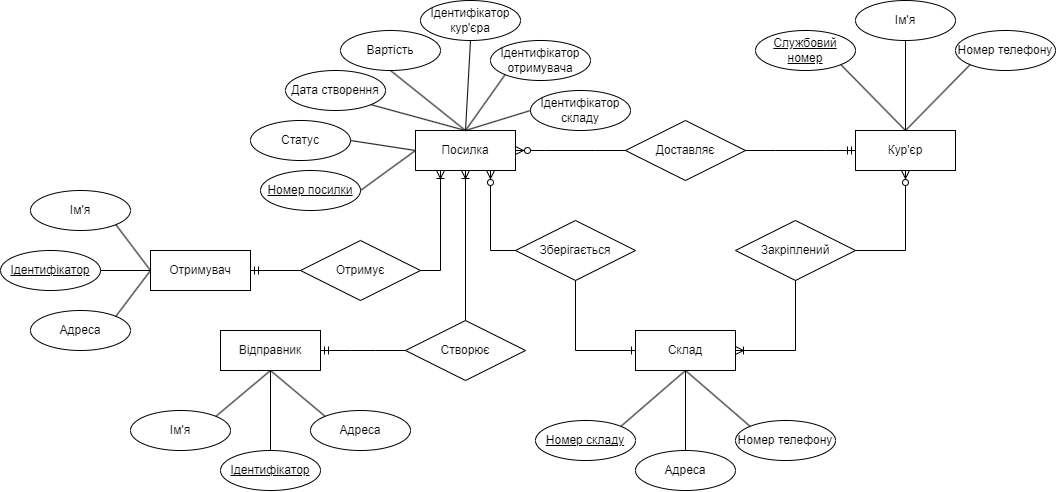


Рисунок 1 – ER-діаграма, побудована за нотацією Чена

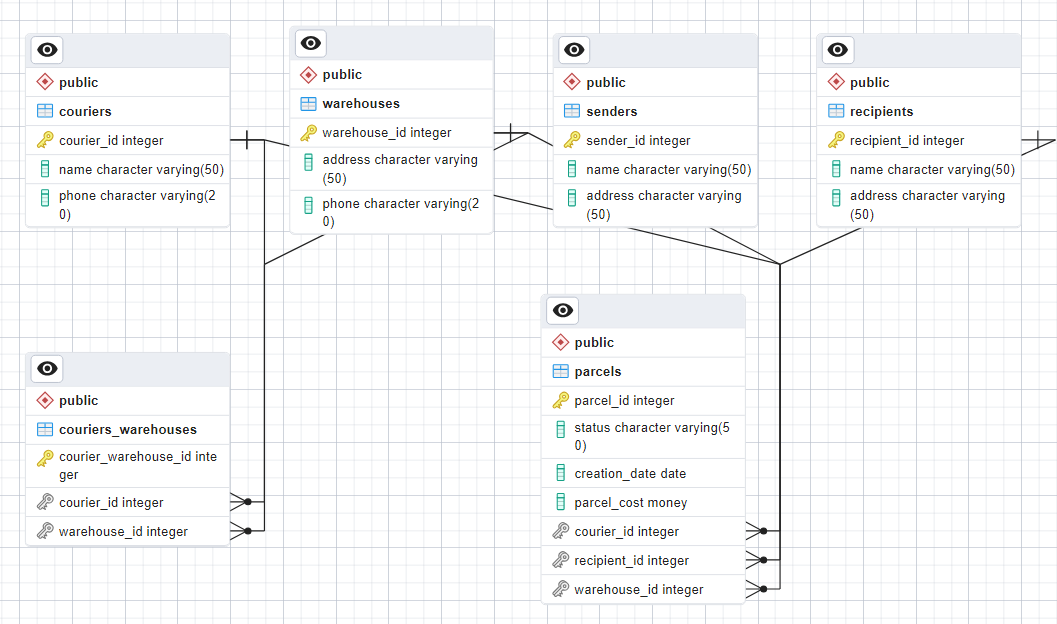
****

Рисунок 2 – Схема бази даних у графічному вигляді

**Завдання 1**

У даній лабораторній роботі було реалізовано 6 класів відповідно до 6 існуючих таблиць у розробленій базі даних, а саме:

1. Recipient
2. Sender
3. Parcel
4. Courier
5. Warehouse
6. CourierWarehouse

**Recipient**

Таблиця Recipients має такі стовпці: recipient\_id (ідентифікатор отримувача), name (ім’я отримувача), address (адреса отримувача). Також наявний зв’язок із таблицею Parcels, тому в цьому класі встановлений зв’язок relationship("Parcel").

Програмна реалізація класу **Recipient**:

**class** Recipient(Base):

\_\_tablename\_\_ = 'recipients'

recipient\_id = Column(Integer, primary\_key=True)

name = Column(String)

address = Column(String)

parcel = relationship("Parcel")

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

self.name = name

self.address = address

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

**return** f"<Recipients(recipient\_id={self.recipient\_id}, name={self.name}, address={self.address})>"

**Sender**

Таблиця Senders має такі стовпці: sender\_id (ідентифікатор відправника), name (ім’я відправника), address (адреса відправника).

Програмна реалізація класу **Sender**:

**class** Sender(Base):

\_\_tablename\_\_ = 'senders'

sender\_id = Column(Integer, primary\_key=True)

name = Column(String)

address = Column(String)

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

self.name = name

self.address = address

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

**return** f"<Senders(sender\_id={self.sender\_id}, name={self.name}, address={self.address})>"

**Parcel**

Таблиця Parcels має такі стовпці: parcel\_id (ідентифікатор посилки), status (статус посилки), creation\_date (дата створення посилки), parcel\_cost (ціна посилки), courier\_id (зовнішній ключ, який пов’язує посилку із кур’єром), recipient\_id (зовнішній ключ, який пов’язує посилку із отримувачем), warehouse\_id (зовнішній ключ, який пов’язує посилку із складом).

Програмна реалізація класу **Parcel**:

**class** Parcel(Base):

\_\_tablename\_\_ = 'parcels'

parcel\_id = Column(Integer, primary\_key=True)

status = Column(String)

creation\_date = Column(Date)

parcel\_cost = Column(Numeric)

courier\_id = Column(Integer, ForeignKey('couriers.courier\_id'))

recipient\_id = Column(Integer, ForeignKey('recipients.recipient\_id'))

warehouse\_id = Column(Integer, ForeignKey('warehouses.warehouse\_id'))

**def** \_\_init\_\_(self, status, creation\_date, parcel\_cost, courier\_id, recipient\_id, warehouse\_id):

self.status = status

self.creation\_date = creation\_date

self.parcel\_cost = parcel\_cost

self.courier\_id = courier\_id

self.recipient\_id = recipient\_id

self.warehouse\_id = warehouse\_id

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

**return** f"<Parcels(parcel\_id={self.parcel\_id}, status={self.status}, creation\_date={self.creation\_date}, parcel\_cost={self.parcel\_cost}, courier\_id={self.courier\_id}, recipient\_id={self.recipient\_id}, warehouse\_id={self.warehouse\_id})>"

**Courier**

Таблиця Couriers має такі стовпці: courier\_id (ідентифікатор кур’єра), name (ім’я кур’єра), phone (номер телефону кур’єра). Також наявні зв’язки із таблицями Parcels та Couriers\_Warehouses, тому в цьому класі встановлені зв’язки relationship("Parcel") та relationship("CourierWarehouse").

Програмна реалізація класу **Courier**:

**class** Courier(Base):

\_\_tablename\_\_ = 'couriers'

courier\_id = Column(Integer, primary\_key=True)

name = Column(String)

phone = Column(String)

parcel = relationship("Parcel")

courier\_warehouse = relationship("CourierWarehouse")

**def** \_\_init\_\_(self, name, phone):

self.name = name

self.phone = phone

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

**return** f"<Couriers(courier\_id={self.courier\_id}, name={self.name}, phone={self.phone})>"

**Warehouse**

Таблиця Warehouses має такі стовпці: warehouse\_id (ідентифікатор складу), address (адреса складу), phone (номер телефону складу). Також наявні зв’язки із таблицями Parcels та Couriers\_Warehouses, тому в цьому класі встановлені зв’язки relationship("Parcel") та relationship("CourierWarehouse").

Програмна реалізація класу **Warehouse**:

**class** Warehouse(Base):

\_\_tablename\_\_ = 'warehouses'

warehouse\_id = Column(Integer, primary\_key=True)

address = Column(String)

phone = Column(String)

parcel = relationship("Parcel")

courier\_warehouse = relationship("CourierWarehouse")

**def** \_\_init\_\_(self, address, phone):

self.address = address

self.phone = phone

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

**return** f"<Warehouses(warehouse\_id={self.warehouse\_id}, address={self.address}, phone={self.phone})>"

**CourierWarehouse**

Таблиця Couriers\_Warehouses має такі стовпці: courier\_warehouse\_id (ідентифікатор рядка), courier\_id (зовнішній ключ, який посилається на кур’єра), warehouse\_id (зовнішній ключ, який посилається на склад).

Програмна реалізація класу **CourierWarehouse**:

**class** CourierWarehouse(Base):

\_\_tablename\_\_ = 'couriers\_warehouses'

courier\_warehouse\_id = Column(Integer, primary\_key=True)

courier\_id = Column(Integer, ForeignKey('couriers.courier\_id'))

warehouse\_id = Column(Integer, ForeignKey('warehouses.warehouse\_id'))

**def** \_\_init\_\_(self, courier\_id, warehouse\_id):

self.courier\_id = courier\_id

self.warehouse\_id = warehouse\_id

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

**return** f"<CouriersWarehouses(courier\_warehouse\_id={self.courier\_warehouse\_id}, courier\_id={self.courier\_id}, warehouse\_id={self.warehouse\_id})>"

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

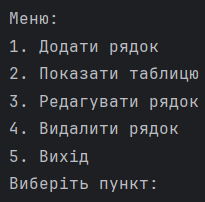


Рисунок 3 – Відображення меню

Пункт №1 – «*Додати рядок*»

Після вибору цього пункту, відкривається список доступних таблиць (рисунок 4), де потрібно вибрати таблицю, до якої бажаємо додати рядок:

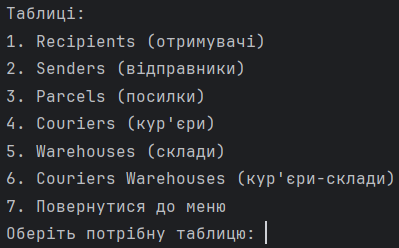


Рисунок 4 – Мені вибору необхідної таблиці

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 4), 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 4), де потрібно вибрати таблицю, в якій бажаємо зробити зміну. Після вибору таблиці, користувачу потрібно ввести ідентифікатор існуючого рядку в таблиці. Потім відповідно ввести нові дані, після чого буде повідомлено про успішну дію або помилку.

Пункт №4 – «*Видалити рядок*»

Після вибору цього пункту, відкривається список доступних таблиць (рисунок 4), 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.

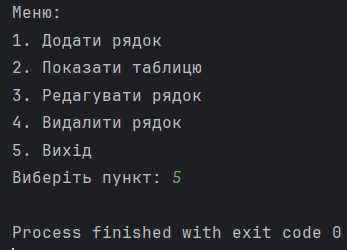


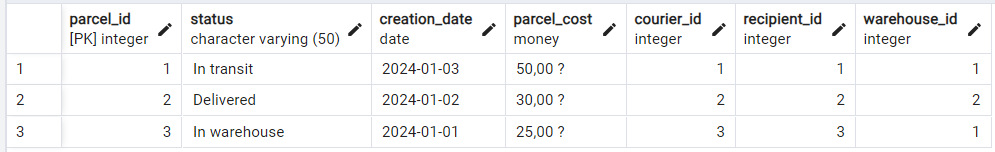
Рисунок 5 – Відображення процесу виходу з програми

***Приклади запитів у вигляді ORM***

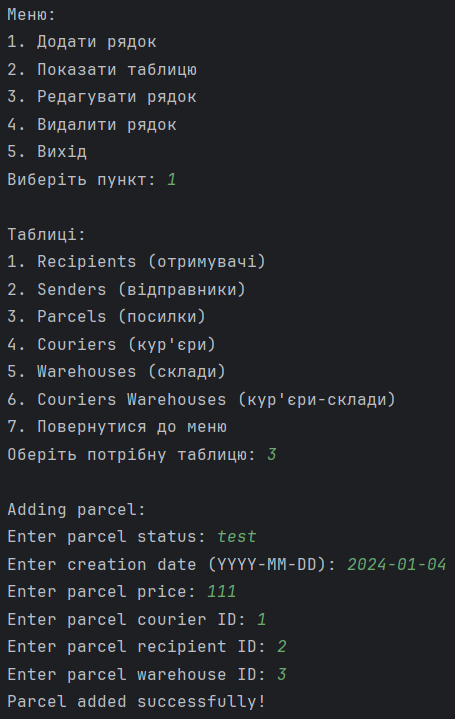
Для демонстрації запитів виберемо по 1-2 таблиці до кожного.

***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.

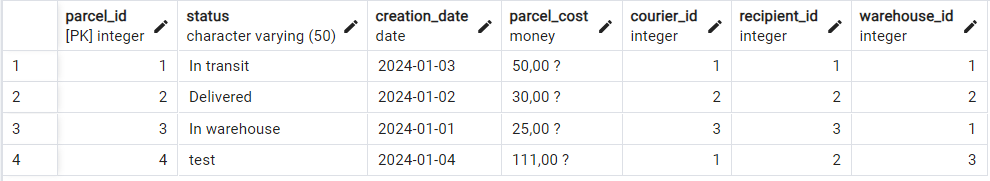
***Таблиця “parcels” до вставки:***

******

***Процес додавання рядка:***

******

***Таблиця “parcels” після вставки:***



Лістинг функцій **insert** для кожної таблиці:

**def** insert\_recipient(self, name: str, address: float) -> None:

recipient = Recipient(name=name, address=address)

s.add(recipient)

s.commit()

**def** insert\_sender(self, name: str, address: float) -> None:

sender = Sender(name=name, address=address)

s.add(sender)

s.commit()

**def** insert\_parcel(self, status: str, creation\_date: Date, parcel\_cost: Numeric, courier\_id: int, recipient\_id: int, warehouse\_id: int) -> None:

parcel = Parcel(status=status, creation\_date=creation\_date, parcel\_cost=parcel\_cost, courier\_id=courier\_id, recipient\_id=recipient\_id, warehouse\_id=warehouse\_id)

s.add(parcel)

s.commit()

**def** insert\_courier(self, name: str, phone: str) -> None:

courier = Courier(name=name, phone=phone)

s.add(courier)

s.commit()

**def** insert\_warehouse(self, address: str, phone: str) -> None:

warehouse = Warehouse(address=address, phone=phone)

s.add(warehouse)

s.commit()

**def** insert\_courier\_warehouse(self, courier\_id: int, warehouse\_id: int) -> None:

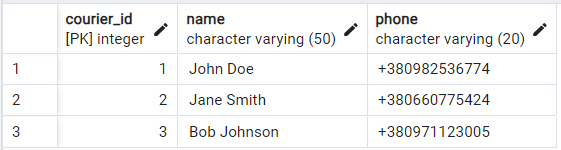
courier\_warehouse = CourierWarehouse(courier\_id=courier\_id, warehouse\_id=warehouse\_id)

s.add(courier\_warehouse)

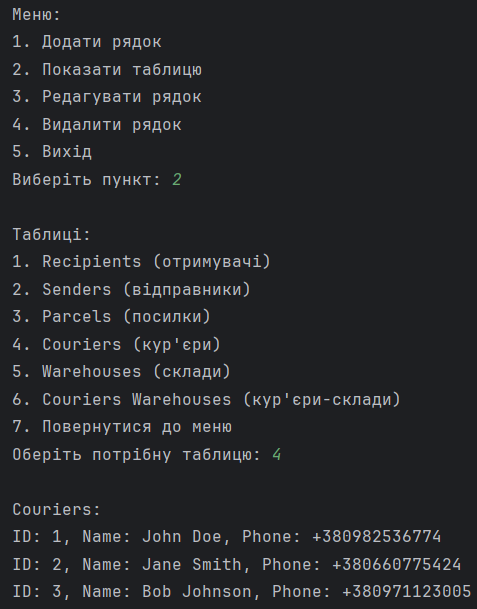
s.commit()

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

***Таблиця “couriers”:***

******

***Процес показу таблиці:***



Лістинг функцій **show** для кожної таблиці:

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

**return** s.query(Recipient.recipient\_id, Recipient.name, Recipient.address).all()

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

**return** s.query(Sender.sender\_id, Sender.name, Sender.address).all()

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

**return** s.query(Parcel.parcel\_id, Parcel.status, Parcel.creation\_date, Parcel.parcel\_cost, Parcel.courier\_id, Parcel.recipient\_id, Parcel.warehouse\_id).all()

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

**return** s.query(Courier.courier\_id, Courier.name, Courier.phone).all()

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

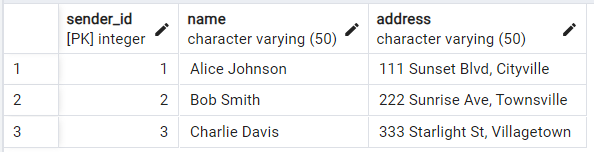
**return** s.query(Warehouse.warehouse\_id, Warehouse.address, Warehouse.phone).all()

**def** show\_couriers\_warehouses(self):

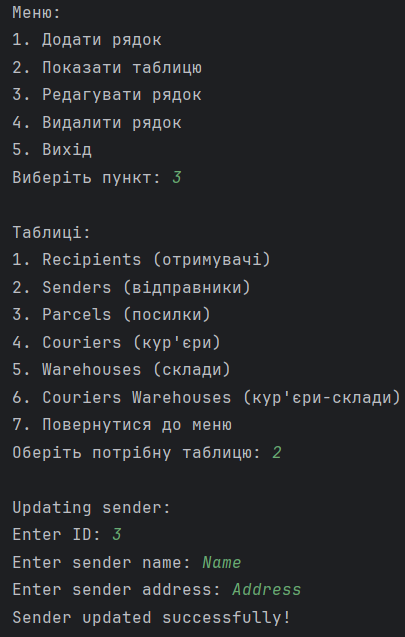
**return** s.query(CourierWarehouse.courier\_warehouse\_id, CourierWarehouse.courier\_id, CourierWarehouse.warehouse\_id).all()

***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 редагування рядка.

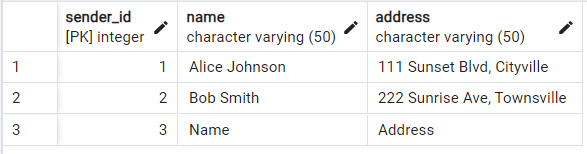
***Таблиця “senders” до редагування:***

******

***Процес редагування рядка:***

******

***Таблиця “senders” після редагування:***



Лістинг функцій **update** для кожної таблиці:

**def** update\_recipient(self, name: str, address: str, recipient\_id: int) -> None:

s.query(Recipient).filter\_by(recipient\_id=recipient\_id).update({Recipient.name: name, Recipient.address: address})

s.commit()

**def** update\_sender(self, name: str, address: str, sender\_id: int) -> None:

s.query(Sender).filter\_by(sender\_id=sender\_id).update({Sender.name: name, Sender.address: address})

s.commit()

**def** update\_parcel(self, status: str, creation\_date: Date, parcel\_cost: Numeric, courier\_id: int, recipient\_id: int, warehouse\_id: int, parcel\_id: int, ) -> None:

s.query(Parcel).filter\_by(parcel\_id=parcel\_id).update({Parcel.status: status, Parcel.creation\_date: creation\_date, Parcel.parcel\_cost: parcel\_cost, Parcel.courier\_id: courier\_id, Parcel.recipient\_id: recipient\_id, Parcel.warehouse\_id: warehouse\_id})

s.commit()

**def** update\_courier(self, name: str, phone: str, courier\_id: int) -> None:

s.query(Courier).filter\_by(courier\_id=courier\_id).update({Courier.name: name, Courier.phone: phone})

s.commit()

**def** update\_warehouse(self, address: str, phone: str, warehouse\_id: int) -> None:

s.query(Warehouse).filter\_by(warehouse\_id=warehouse\_id).update({Warehouse.address: address, Warehouse.phone: phone})

s.commit()

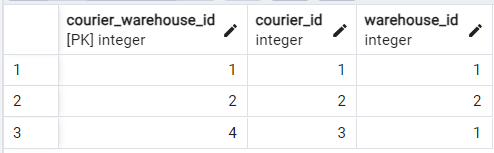
**def** update\_courier\_warehouse(self, courier\_id: int, warehouse\_id: int, courier\_warehouse\_id: int) -> None:

s.query(CourierWarehouse).filter\_by(courier\_warehouse\_id=courier\_warehouse\_id).update({CourierWarehouse.courier\_id: courier\_id, CourierWarehouse.warehouse\_id: warehouse\_id})

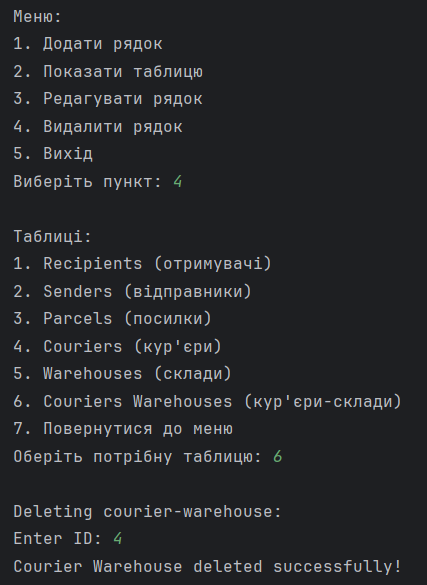
s.commit()

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

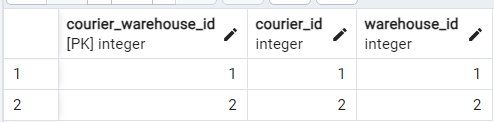
***Таблиця “couriers\_warehouses” до видалення:***

******

***Процес видалення рядка:***

******

***Таблиця “couriers\_warehouses” після видалення:***



Лістинг функцій **delete** для кожної таблиці:

**def** delete\_recipient(self, recipient\_id) -> None:

recipient = s.query(Recipient).filter\_by(recipient\_id=recipient\_id).one()

s.delete(recipient)

s.commit()

**def** delete\_sender(self, sender\_id) -> None:

sender = s.query(Sender).filter\_by(sender\_id=sender\_id).one()

s.delete(sender)

s.commit()

**def** delete\_parcel(self, parcel\_id) -> None:

parcel = s.query(Parcel).filter\_by(parcel\_id=parcel\_id).one()

s.delete(parcel)

s.commit()

**def** delete\_courier(self, courier\_id) -> None:

courier = s.query(Courier).filter\_by(courier\_id=courier\_id).one()

s.delete(courier)

s.commit()

**def** delete\_warehouse(self, warehouse\_id) -> None:

warehouse = s.query(Warehouse).filter\_by(warehouse\_id=warehouse\_id).one()

s.delete(warehouse)

s.commit()

**def** delete\_courier\_warehouse(self, courier\_warehouse\_id) -> None:

couriers\_warehouses = s.query(CourierWarehouse).filter\_by(courier\_warehouse\_id=courier\_warehouse\_id).all()

**for** courier\_warehouse **in** couriers\_warehouses:

s.delete(courier\_warehouse)

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”:

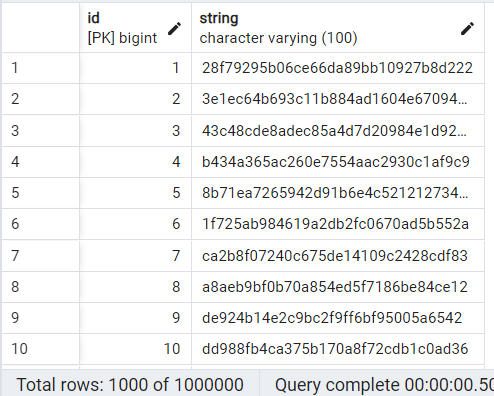
**CREATE** **TABLE** "gin\_test"("id" bigserial **PRIMARY** **KEY**, "string" **varchar**(100));

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

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

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

Рядки таблиці *gin\_test:*



Для тестування візьмемо 5 запитів:

**SELECT** \* **FROM** gin\_test **WHERE** "string" = 'de924b14e2c9bc2f9ff6bf95005a6542';

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

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

**SELECT** **COUNT**(\*), "string" **FROM** gin\_test **WHERE** "id" **BETWEEN** 100 **AND** 1000 **GROUP** **BY** "string";

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

Створення індексу:

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

Результати виконання запитів:

Без індексу gin:

Запит №1



Запит №2



Запит №3



Запит №4



Запит №5



З індексом gin:

Запит №1



Запит №2



Запит №3



Запит №4



Запит №5



Отже, з отриманих результатів можна побачити, що запити №2, 3, 5 спрацювали швидше без індексу gin. Розглянемо кожен із них:

1. Запит №2 - GROUP BY

Запит з агрегацією та GROUP BY може працювати ефективніше без індексу, оскільки база даних може проводити агрегацію на основі сортованого порядку стовпців без використання індексу. GIN-індекс може займати додатковий ресурс для таких запитів.

1. Запит №3 - ORDER BY

Запити з сортуванням можуть працювати швидше без індексу, оскільки база даних може використовувати сортований порядок на основі фізичного розташування даних у таблиці. Використання індексу може вимагати додаткових операцій.

1. Запит №5 - GROUP BY та ORDER BY COUNT(\*)

Запит з використанням GROUP BY і ORDER BY COUNT(\*) може бути швидше оброблений без індексу, оскільки GIN-індекс може вимагати додаткових операцій для обробки агрегаційних функцій та сортування.

***BRIN***

Індекс BRIN (Block Range INdex) - це інший тип індексу в PostgreSQL, який призначений для оптимізації пошуку та фільтрації даних великих таблиць, особливо тих, які мають упорядкований порядок значень за деяким стовпцем. BRIN індекси призначені для ефективної обробки діапазонних запитів.

Основна ідея BRIN індексу полягає в тому, щоб розділити дані на блоки та зберігати індексовані значення для кожного блоку. Це дозволяє базі даних швидко визначати, які блоки можуть містити потрібні рядки, і використовувати індекс для обробки тільки цих блоків.

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

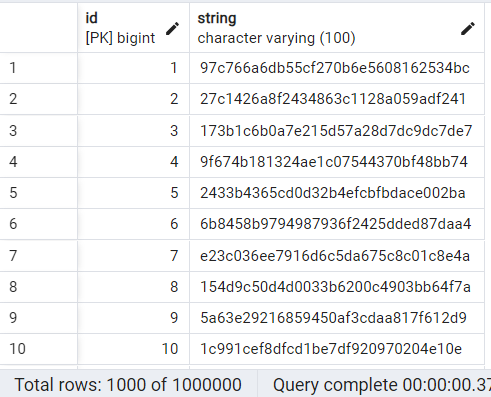
**CREATE** **TABLE** "brin\_test"("id" bigserial **PRIMARY** **KEY**, "string" **varchar**(100));

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

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

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

Рядки таблиці *brin\_test:*



Для тестування візьмемо 5 запитів:

**SELECT** \* **FROM** brin\_test **WHERE** "string" = '2433b4365cd0d32b4efcbfbdace002ba';

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

**SELECT** \* **FROM** brin\_test **ORDER** **BY** "string" **DESC**;

**SELECT** **COUNT**(\*), "string" **FROM** brin\_test **WHERE** "id" **BETWEEN** 100 **AND** 1000 **GROUP** **BY** "string";

**SELECT** **COUNT**(\*), "string" **FROM** brin\_test **GROUP** **BY** "string" **ORDER** **BY** **COUNT**(\*) **DESC**;

Створення індексу:

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

Результати виконання запитів:

Без індексу brin:

Запит №1



Запит №2



Запит №3



Запит №4



Запит №5



З індексом brin:

Запит №1



Запит №2



Запит №3



Запит №4



Запит №5



Отже, з отриманих результатів можна побачити, що запити №3, 5 спрацювали швидше без індексу brin. Розглянемо кожен із них:

1. Запит №3 - ORDER BY

Запит з сортуванням (ORDER BY) може працювати ефективніше без індексу, особливо якщо дані вже відсортовані фізично в таблиці. BRIN-індекс, який орієнтований на діапазонні значення, може бути менш ефективним для сортування, особливо якщо діапазони не відображають точно впорядкованих значень.

1. Запит №5 - GROUP BY та ORDER BY COUNT(\*)

Схоже на випадок з GIN-індексом, GROUP BY і ORDER BY COUNT(\*) можуть працювати ефективніше без BRIN-індексу. BRIN, орієнтований на блоки, може мати обмежену точність для агрегаційних функцій та сортування в порівнянні з B-деревооб'єктними індексами.

Ефективність індексів може сильно залежати від конкретного сценарію використання, обсягу даних та конкретного плану виконання, який вибирає оптимізатор запитів бази даних.