A black and white drawing of a building

Description automatically generated

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 і засоби управління**

*на тему: “Засоби оптимізації роботи СУБД PostgreSQL”*

Виконав:

студент ІІI курсу

групи КВ-22

Вакульчук Ярослав

Перевірив:

Павловский В. І.

Київ – 2024

**Мета:** здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

**Виконання роботи**

Нижче будуть наведені сутності предметної області:

**Сутності предметної області**

Для побудови концептуальної моделі обраної предметної області, були виділені такі сутності:

1. Користувач (User)

Атрибути ідентифікатор користувача, ім’я, прізвище, електронна пошта.

Призначення: збереження інформації про зареєстрованих користувачів платформи.

1. Книга (Book)

Атрибути: ідентифікатор книги, назва, автор, жанр, стан.

Призначення: збереження інформації про книги, доступні для обміну.

1. Оголошення (Listing)

Атрибути: ідентифікатор оголошення, статус оголошення, дата створення.

Призначення: збереження інформації про оголошення, створені користувачами, статус оголошення.

Нижче будуть наведені зв’язки між сутностями предметної області:

**Зв’язки між сутностями предметної області**

Зв’язок-сутність «Користувач» - «Книга» є зв’язком 1:N. Один користувач може додати кілька книг для обміну, але кожна книга належить тільки одному користувачу.

Зв’язок-сутність «Книга» - «Оголошеня» є зв’язком 1:1. Одна книга може бути пов’язана лише з одним оголошенням.

Зв’язок-сутність «Користувач» - «Оголошення» є зв’язком M:N. Один користувач може створити кілька оголошень на обмін та кожне оголошення може прийняти декілька користувачів.

Графічне подання концептуальної моделі «Сутність-зв’язок» зображено на рисунку 1.

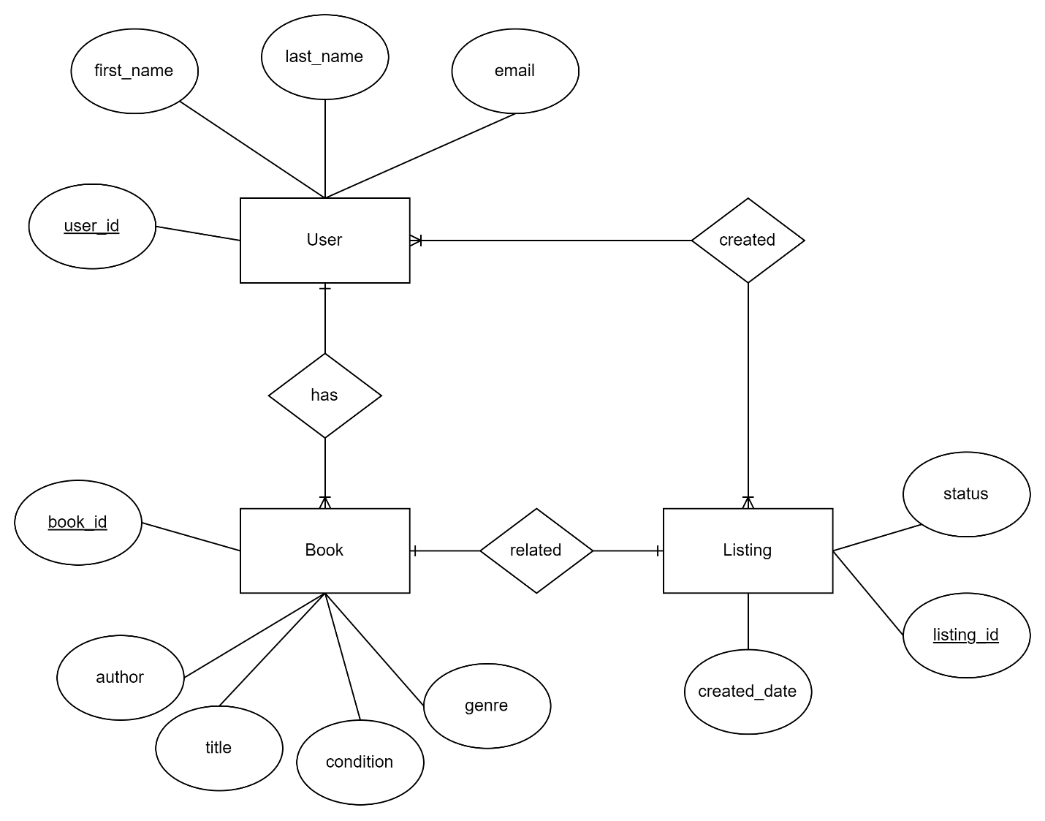


Рисунок 1 – ER-діаграма, побудована за нотацією Чена

Графічне подання логічної моделі «Сутність-зв’язок» зображено на рисунку 2.

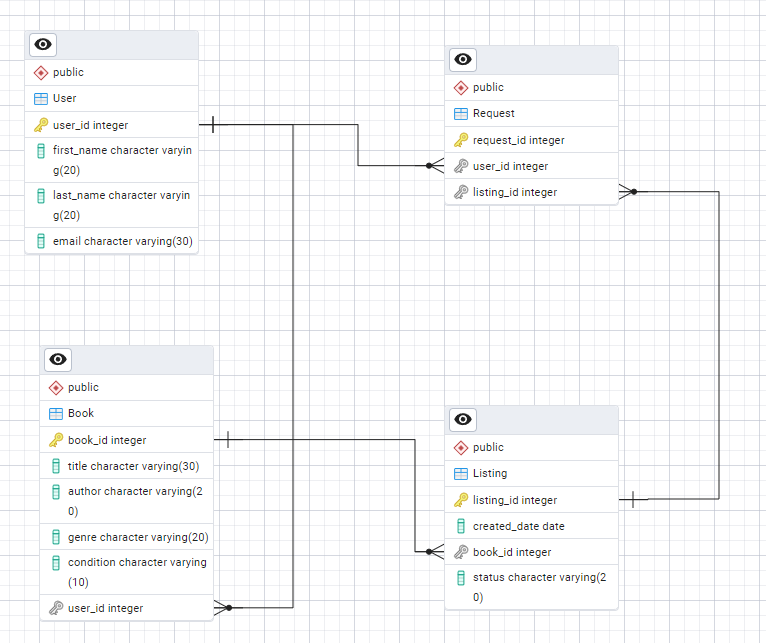


Рисунок 2 – Логічна модель

**Середовище та компоненти розробки**

У процесі розробки була використана мова програмування Python, інтегроване середовище розробки PyCharm, а також була використана бібліотека SQLAlchemy, яка надає API для взаємодії з базою даних PostgreSQL.

**Шаблон проектування**

Модель-представлення-контролер (MVC) - це шаблон проектування, який використовується у розробці програм. Кожен компонент відповідає за конкретну функціональну частину програми:

Модель (Model) - це клас, який визначає логіку роботи з даними. Він обробляє всі операції з даними, такі як додавання, оновлення та видалення.

Представлення (View) - це клас, який дозволяє користувачеві взаємодіяти з програмою. В даному випадку це консольний інтерфейс, який відображає дані користувачу та приймає введення з екрану.

Контролер (Controller) - це клас, що відповідає за взаємодію між користувачем та системою. Він приймає дані, введені користувачем, та обробляє їх. Залежно від результатів, він викликає відповідні дії в Model або View.

Цей підхід дозволяє розділити логіку програми на логічні компоненти, що полегшує розробку, тестування та підтримку продукту.

**Структура програми та її опис**

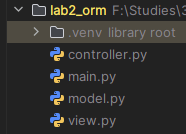
****

Рисунок 3 – Структура програми

З файлу main.py відбувається виклик контролера та передача йому управління.

У файлі model.py описаний клас моделі, який відповідає за управління підключенням до бази даних і виконанням низькорівневих запитів до неї.

У файлі controller.py реалізовано інтерфейс взаємодії з користувачем, включаючи обробку запитів користувача, виконання пошуку, а також інші дії, необхідні для взаємодії з моделлю та представленням.

У файлі view.py описаний клас, який відображає результати виконання різних дій користувача на екрані консолі. Цей компонент відповідає за представлення даних користувачу в зручному для сприйняття вигляді.

Отже, структура програми відповідає патерну MVC.

**Меню програми**

На рисунку 4 зображено меню користувача, яке складається з семи пунктів.

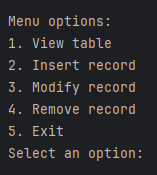


Рисунок 4 – Структура меню користувача

**Фрагмент коду (файл model.py), в якому наведено класи ORM:**

class User(Base):  
 \_\_tablename\_\_ = 'User'  
 user\_id = Column(Integer, primary\_key=True)  
 first\_name = Column(String(20), nullable=False)  
 last\_name = Column(String(20), nullable=False)  
 email = Column(String(50), nullable=False)  
 books = relationship("Book", back\_populates="owner")  
 requests = relationship("Request", back\_populates="user")  
class Book(Base):  
 \_\_tablename\_\_ = 'Book'  
 book\_id = Column(Integer, primary\_key=True)  
 title = Column(String(30), nullable=False)  
 author = Column(String(20), nullable=False)  
 genre = Column(String(40), nullable=False)  
 condition = Column(String(10), nullable=False)  
 user\_id = Column(Integer, ForeignKey('User.user\_id'), nullable=False)  
 owner = relationship("User", back\_populates="books")  
 listings = relationship("Listing", back\_populates="book")  
class Listing(Base):  
 \_\_tablename\_\_ = 'Listing'  
 listing\_id = Column(Integer, primary\_key=True)  
 created\_date = Column(Date, nullable=False)  
 book\_id = Column(Integer, ForeignKey('Book.book\_id'), nullable=False)  
 status = Column(String(30), nullable=False)  
 book = relationship("Book", back\_populates="listings")  
 requests = relationship("Request", back\_populates="listing")  
class Request(Base):  
 \_\_tablename\_\_ = 'Request'  
 request\_id = Column(Integer, primary\_key=True)  
 user\_id = Column(Integer, ForeignKey('User.user\_id'), nullable=False)  
 listing\_id = Column(Integer, ForeignKey('Listing.listing\_id'), nullable=False)  
 user = relationship("User", back\_populates="requests")  
 listing = relationship("Listing", back\_populates="requests")

**Опис класів**

**Клас User (Користувач):**

Представляє користувача у системі.

Зв’язки:

books — Один користувач може володіти кількома книгам (зв’язок з класом Book).

requests — Один користувач може створювати кілька запитів на книги (зв’язок з класом Request).

**Клас Book (Книга):**

Представляє книгу, що належить користувачеві.

Зв’язки:

owner — Посилання на власника книги (зв’язок з класом User).

listings — Оголошення, пов’язані з цією книгою (зв’язок з класом Listing).

**Клас Listing (Оголошення):**

Представляє оголошення про доступність книги.

Зв’язки:

book — Посилання на книгу, яка пропонується в оголошенні (зв’язок з класом Book).

requests — Запити, пов’язані з оголошенням (зв’язок з класом Request).

**Клас Request (Запит):**

Представляє запит на конкретне оголошення користувачем.

Зв’язки:

user — Посилання на користувача, який створив запит (зв’язок з класом User).

listing — Посилання на оголошення, до якого створено запит (зв’язок з класом Listing).

**Фрагмент коду (файл controller.py), в якому наведено головний цикл роботи програми:**

def Run(self):  
 while True:  
 choice = self.dbView.ShowMainMenu()  
 if choice == '1':  
 self.ShowTableData()  
 elif choice == '2':  
 self.InsertNewRecord()  
 elif choice == '3':  
 self.ModifyRecord()  
 elif choice == '4':  
 self.RemoveRecord()  
 elif choice == '5':  
 return  
 else:  
 self.dbView.PrintMessage("Invalid option. Please try again.\n")

**Фрагмент коду (файл model.py), в якому наведено функції перегляду, внесення, редагування, вилучення та генерації у базі даних:**

Функції виведення даних:

def GetTable(self, table\_name):  
 with self.Session() as session:  
 entity\_map = {cls.\_\_tablename\_\_: cls for cls in Base.\_\_subclasses\_\_()}  
 if table\_name not in entity\_map:  
 raise ValueError(f"'{table\_name}' is not a valid table.")  
 entity = entity\_map[table\_name]  
 results = session.query(entity).all()  
 if not results:  
 raise ValueError(f"No entries found in '{table\_name}'.")  
 headers = [col.name for col in entity.\_\_table\_\_.columns]  
 data = [[getattr(record, col) for col in headers] for record in results]  
 return {"headers": headers, "data": data}

Ця функція перевіряє чи є таблиця, якщо перевірка пройдена то функція виводить повністю таблицю даних.

Функції внесення даних:

def AddRecord(self, table\_name, values\_str):  
 with self.Session() as session:  
 table\_map = {cls.\_\_tablename\_\_: cls for cls in Base.\_\_subclasses\_\_()}  
 if table\_name not in table\_map:  
 raise ValueError(f"Table '{table\_name}' does not exist.")  
 model = table\_map[table\_name]  
 values\_list = [val.strip() for val in values\_str.split(",")]  
 table\_columns = [column.name for column in model.\_\_table\_\_.columns]  
 if len(values\_list) != len(table\_columns):  
 raise ValueError(  
 f"The number of provided values ({len(values\_list)}) does not match "  
 f"the number of columns in table '{table\_name}' ({len(table\_columns)})."  
 )  
 data = dict(zip(table\_columns, values\_list))  
 self.ValidateInput(model, data)  
 primary\_keys = {col.name: data[col.name] for col in model.\_\_table\_\_.primary\_key.columns}  
 if session.query(model).filter\_by(\*\*primary\_keys).first():  
 raise ValueError(f"Record with primary keys {primary\_keys} already exists.")  
 new\_record = model(\*\*data)  
 session.add(new\_record)  
 try:  
 session.commit()  
 except Exception as e:  
 session.rollback()  
 raise ValueError(f"Failed to insert record: {e}")

Ця функція перевіряє чи вводить користувач унікальний ідентифікатор для нового запису в таблиці. Також вона перевіряє наявність зовнішнього ключа, якщо такий є в таблиці, щоб упевнитися, що введений ключ існує у відповідній зв’язаній таблиці. Додатково здійснюється перевірка на валідність внесених змін. Якщо всі умови виконуються, дозволяється вставка запису; в іншому випадку користувачу буде запропоновано ввести інші дані.

Функції оновлення даних:

def UpdateRecord(self, table\_name, values\_str, record\_id):  
 with self.Session() as session:  
 table\_map = {cls.\_\_tablename\_\_: cls for cls in Base.\_\_subclasses\_\_()}  
 if table\_name not in table\_map:  
 raise ValueError(f"Table '{table\_name}' does not exist.")  
 model = table\_map[table\_name]  
 primary\_key = list(model.\_\_table\_\_.primary\_key.columns)[0].name  
 record = session.query(model).filter(getattr(model, primary\_key) == record\_id).first()  
 if not record:  
 raise ValueError(f"No record with ID {record\_id} found in table '{table\_name}'.")  
 values\_list = [val.strip() for val in values\_str.split(",")]  
 table\_columns = [column.name for column in model.\_\_table\_\_.columns]  
 if len(values\_list) != len(table\_columns):  
 raise ValueError(  
 f"The number of provided values ({len(values\_list)}) does not match "  
 f"the number of columns in table '{table\_name}' ({len(table\_columns)})."  
 )  
 data = dict(zip(table\_columns, values\_list))  
 self.ValidateInput(model, data)  
 for key, value in data.items():  
 setattr(record, key, value)  
 try:  
 session.commit()  
 except Exception as e:  
 session.rollback()  
 raise ValueError(f"Failed to update record: {e}")

Ця функція перевіряє правильність внесених змін. Якщо користувач намагається змінити поле, яке є зовнішнім ключем, функція перевіряє, чи існує введений зовнішній ключ у відповідній зв’язаній таблиці. Якщо всі перевірки проходять успішно, дозволяється виконати вставку. В іншому випадку, користувачу потрібно ввести інші дані.

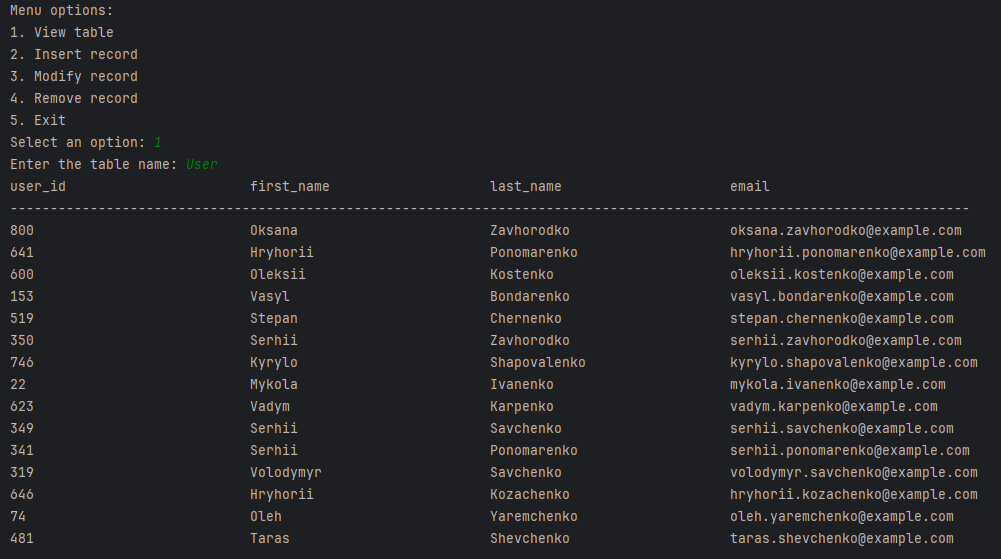
Функції видалення даних:

def DeleteRecord(self, table\_name, record\_id):  
 with self.Session() as session:  
 table\_map = {cls.\_\_tablename\_\_: cls for cls in Base.\_\_subclasses\_\_()}  
 if table\_name not in table\_map:  
 raise ValueError(f"Table '{table\_name}' does not exist.")  
 model = table\_map[table\_name]  
 primary\_key = list(model.\_\_table\_\_.primary\_key.columns)[0].name  
 record = session.query(model).filter(getattr(model, primary\_key) == record\_id).first()  
 if not record:  
 raise ValueError(f"No record with ID {record\_id} found in table '{table\_name}'.")  
 inspector = inspect(self.engine)  
 foreign\_keys = inspector.get\_foreign\_keys(model.\_\_tablename\_\_)  
 for fk in foreign\_keys:  
 child\_table\_name = fk['referred\_table']  
 child\_column\_name = fk['referred\_columns'][0]  
 child\_model = table\_map.get(child\_table\_name)  
 if child\_model:  
 child\_column = getattr(child\_model, child\_column\_name, None)  
 if not child\_column:  
 raise ValueError(  
 f"Foreign key column '{child\_column\_name}' does not exist in table '{child\_table\_name}'."  
 )  
 dependent\_record = session.query(child\_model).filter(child\_column == record\_id).first()  
 if dependent\_record:  
 raise ValueError(  
 f"Cannot delete record; dependent records exist in table '{child\_table\_name}'."  
 )  
 try:  
 session.delete(record)  
 session.commit()  
 except Exception as e:  
 session.rollback()  
 raise ValueError(f"Failed to delete record: {e}")

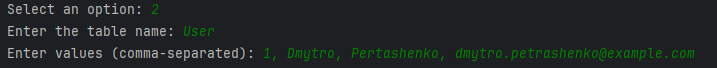
Ця функція перевіряє чи є зв’язані таблиці. Якщо такі таблиці існують, вона перевіряє, чи значення, яке користувач намагається видалити, присутнє в зв’язаній таблиці. Якщо це значення є, видалення не дозволяється.

**Результат виконання програми**

Вивід даних:

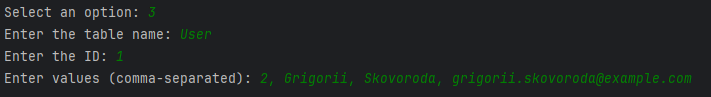
****

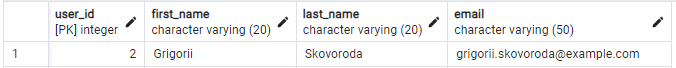
Вставка даних:

****

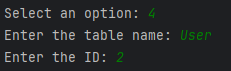
****

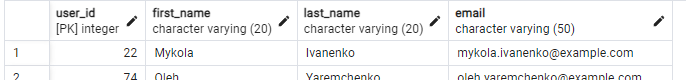
Редагування даних:

****

****

Видалення даних:

****

****

**Створення та аналіз різних типів індексів**

**Hash Index** — це тип індексу в PostgreSQL, який використовує хеш-таблицю для прискорення пошуку значень у базі даних.

Як працює:

1. Коли ви створюєте Hash Index на колонці, для кожного значення цієї колонки обчислюється хеш-значення за допомогою хеш-функції.
2. Ці хеш-значення зберігаються в структурі даних, яка дозволяє швидко знаходити відповідні записи.
3. Hash Index використовується лише для точного пошуку рівності (операція =). Наприклад:
   * Пошук користувача за email.
   * Пошук книги за її title.

Переваги:

* Дуже швидкий пошук, коли потрібно знайти записи за конкретним значенням.
* Простий у використанні для операцій рівності.

Недоліки:

* Не підтримує операції порівняння (>, <, BETWEEN).
* Не допомагає для сортування даних.
* Працює ефективно тільки для точного співставлення.

**BRIN Index** (Block Range INdex) — це тип індексу, який використовується для великих таблиць із впорядкованими даними, наприклад, датами, числовими діапазонами або ID.

Як працює:

1. Таблиця в PostgreSQL ділиться на блоки.
2. BRIN Index зберігає мінімальне та максимальне значення для кожного блоку.
3. Коли виконується запит, BRIN Index швидко визначає, які блоки містять відповідні дані, і лише ці блоки перевіряються на відповідність.

Переваги:

* Дуже ефективний для великих таблиць (мільйони записів).
* Займає дуже мало місця, порівняно з іншими індексами.
* Добре працює з даними, які мають природне впорядкування, наприклад:
  + Дати (created\_date).
  + Порядкові номери (id).

Недоліки:

* Підходить для пошуку діапазонів значень (BETWEEN, >, <).
* Менш ефективний для точного пошуку рівності, порівняно з Hash або B-Tree індексами.
* Якщо дані у таблиці сильно фрагментовані або розташовані у випадковому порядку, ефективність BRIN знижується.

**Створення індексів:**

CREATE INDEX user\_name\_hash\_idx ON "User" USING HASH (first\_name);

CREATE INDEX listing\_created\_date\_brin\_idx ON "Listing" USING BRIN (created\_date);

**Приклади запитів:**

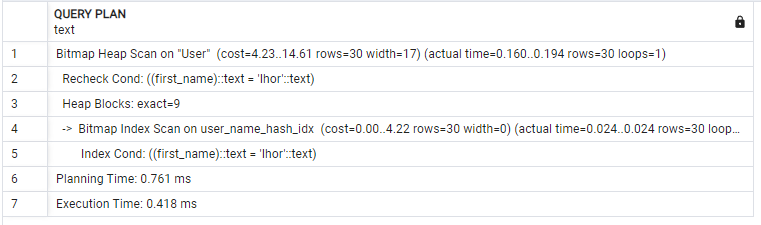
1. Просте фільтрування

EXPLAIN ANALYZE

SELECT first\_name, last\_name

FROM "User"

WHERE first\_name = 'Ihor';



**Пояснення:** У цьому запиті використовується точний пошук за полем first\_name, для якого створено Hash Index.  
Hash Index дозволяє знайти запис без сканування всієї таблиці. PostgreSQL обчислює хеш значення для вказаного email і одразу звертається до потрібного рядка в індексі.

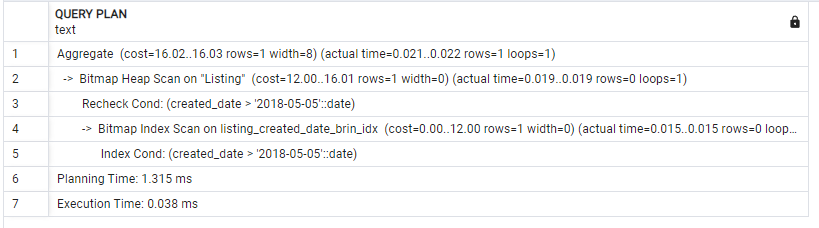
2. Фільтрація за діапазоном

EXPLAIN ANALYZE

SELECT COUNT(\*)

FROM "Listing"

WHERE created\_date > '2018-05-05';



**Пояснення:** BRIN Index на стовпці created\_date дозволяє PostgreSQL обробити лише ті блоки таблиці, які можуть містити записи з датою, більшою за '2018-05-05'. Це скорочує кількість даних для перевірки. BRIN Index зберігає мінімальне та максимальне значення start\_date для кожного блоку таблиці. PostgreSQL обирає лише блоки, де ці значення відповідають умові created\_date > '2018-05-05'. Для відповідних блоків виконується додаткова перевірка рядків (Heap Scan).

1. Об’єднання та групування

EXPLAIN ANALYZE

SELECT u.first\_name, l.created\_date, COUNT(r.request\_id) AS request\_count

FROM "Request" r

JOIN "User" u ON r.user\_id = u.user\_id

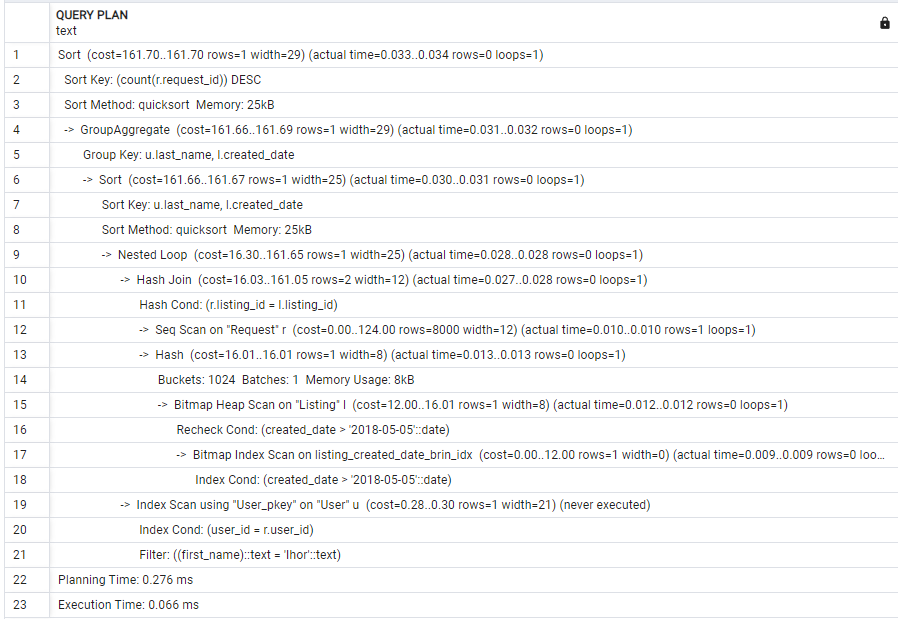
JOIN "Listing" l ON r.listing\_id = l.listing\_id

WHERE l.created\_date > '2018-05-05'

AND u.first\_name = 'Ihor'

GROUP BY u.first\_name, u.last\_name, l.created\_date

ORDER BY request\_count DESC, u.first\_name ASC;



**Пояснення:** Запит виконує об'єднання трьох таблиць (Request, User, Listing), фільтрує дані за датою (created\_date > '2018-05-05') і ім'ям користувача (first\_name = 'Ihor'), групує записи за ім'ям, прізвищем та датою створення оголошення, підраховує кількість запитів для кожної групи, а потім сортує результат за кількістю запитів у спадному порядку. План запиту показує, що використовуються індекси, хеш-з'єднання та сортування для оптимізації виконання, забезпечуючи швидкий результат.

**Розробка тригера**

Створення тригера та прив’язка його до таблиці:

CREATE OR REPLACE FUNCTION trg\_after\_insert\_update\_listing()

RETURNS TRIGGER AS $$

DECLARE

req\_cursor CURSOR FOR

SELECT request\_id, user\_id

FROM "Request"

WHERE listing\_id = NEW.listing\_id;

req\_record RECORD;

BEGIN

-- Умовна обробка нового статусу

IF NEW.status = 'Cancelled' THEN

RAISE NOTICE 'Listing % is now Cancelled.', NEW.listing\_id;

ELSIF NEW.status = 'Approved' THEN

RAISE NOTICE 'Listing % has been Approved.', NEW.listing\_id;

ELSE

RAISE NOTICE 'Listing % status: %', NEW.listing\_id, NEW.status;

END IF;

-- Курсорний цикл: обробка всіх пов'язаних запитів у Request

OPEN req\_cursor;

LOOP

FETCH req\_cursor INTO req\_record;

EXIT WHEN NOT FOUND;

-- Умова для прикладу: повідомлення для кожного запиту

RAISE NOTICE 'Processing Request ID: %, User ID: %', req\_record.request\_id, req\_record.user\_id;

-- Тут можна додати додаткову логіку, наприклад оновлення пов'язаних записів

END LOOP;

CLOSE req\_cursor;

RETURN NEW; -- Обов'язково повертаємо змінений рядок

EXCEPTION

WHEN OTHERS THEN

RAISE WARNING 'Error in trigger function: %', SQLERRM;

RETURN NULL; -- У разі помилки не виконуємо оновлення

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER after\_insert\_update\_listing

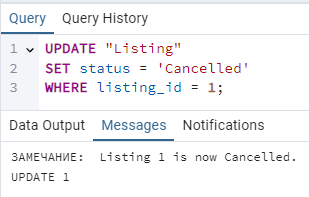
AFTER INSERT OR UPDATE ON "Listing"

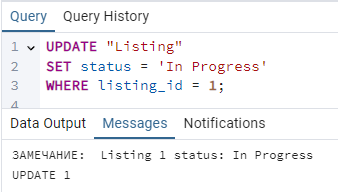
FOR EACH ROW

EXECUTE FUNCTION trg\_after\_insert\_update\_listing();

**Приклади використання:**

****

****

****

Як видно, повідомлення коректно виводяться after insert та update, як задумано.

**Приклади та аналіз рівнів ізоляцій тарнзакцій**

**1. Read Committed (Читання зафіксованих даних)**

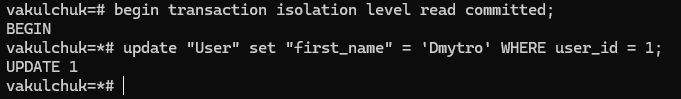
Опис:

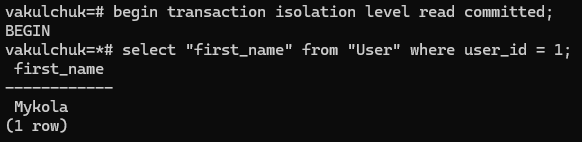
* Транзакція "бачить" тільки зафіксовані зміни інших транзакцій на момент виконання операції.
* Кожен SELECT у межах транзакції отримує новий знімок даних.

Особливості:

* Брудне читання (Dirty Read): Неможливе. Незавершені зміни інших транзакцій недоступні.
* Неповторюване читання (Non-Repeatable Read): Можливе. Інша транзакція може змінити або видалити рядки між різними SELECT у вашій транзакції.

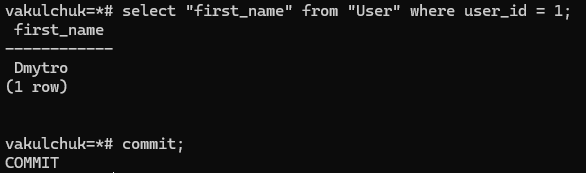
Приклад використання:





Змін не відбулося.





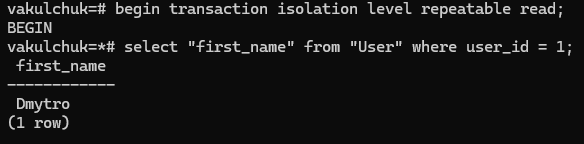
Два SELECT у Транзакції 1 можуть побачити різні значення, якщо Транзакція 2 виконає COMMIT.

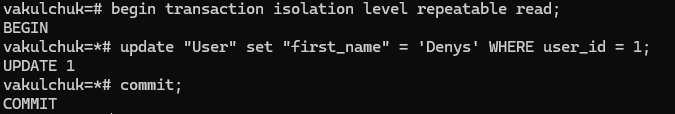
**2. Repeatable Read (Повторюване читання)**

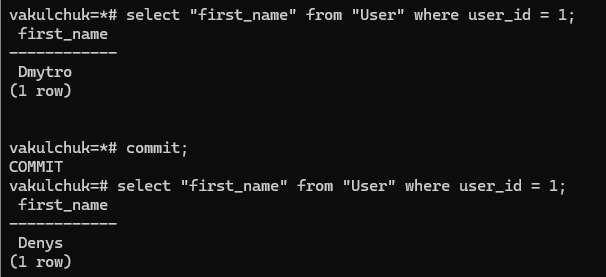
Опис:

* Транзакція отримує стабільний знімок даних на момент першого SELECT.
* Усі наступні SELECT у межах однієї транзакції "бачать" одні й ті ж самі дані.
* Неповторюване читання (Non-Repeatable Read): Неможливе. Інші транзакції не можуть змінювати дані, які читає ваша транзакція.
* Фантомні читання (Phantom Read): Можливі. Інші транзакції можуть додати нові рядки, які відповідають вашим умовам пошуку.

Приклад використання:







Транзакція 1 продовжує "бачити" стару версію даних, незважаючи на зміни у Транзакції 2.

**3. Serializable (Серіалізований рівень ізоляції)**

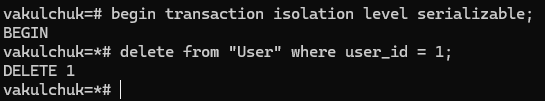
Опис:

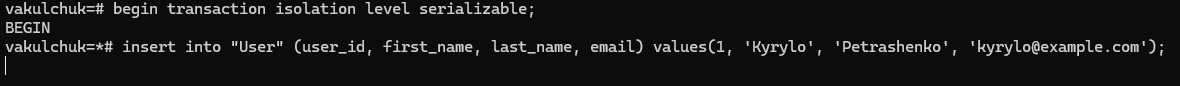
* Найвищий рівень ізоляції, що емулює послідовне виконання транзакцій (одна за одною).
* Гарантує, що кінцевий результат транзакцій буде таким, якби вони виконувались послідовно, а не паралельно.

Особливості:

* Брудне читання (Dirty Read): Неможливе.
* Неповторюване читання (Non-Repeatable Read): Неможливе.
* Фантомне читання (Phantom Read): Неможливе.

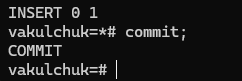
Приклад використання:





Операція була заблокована.





Транзакція 1 завершується тільки після прийняття змін, оскільки Транзакція 2 внесла зміни, які впливають на її знімок даних.

**Повний код програми**

Файл main.py:

from controller import Manager  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 controller = Manager()  
 controller.Run()

Файл model.py:

from sqlalchemy import create\_engine, Column, Integer, String, Date, ForeignKey, inspect  
from sqlalchemy.ext.declarative import declarative\_base  
from sqlalchemy.orm import sessionmaker, relationship  
from datetime import datetime  
  
Base = declarative\_base()  
  
class User(Base):  
 \_\_tablename\_\_ = 'User'  
 user\_id = Column(Integer, primary\_key=True)  
 first\_name = Column(String(20), nullable=False)  
 last\_name = Column(String(20), nullable=False)  
 email = Column(String(50), nullable=False)  
 books = relationship("Book", back\_populates="owner")  
 requests = relationship("Request", back\_populates="user")  
class Book(Base):  
 \_\_tablename\_\_ = 'Book'  
 book\_id = Column(Integer, primary\_key=True)  
 title = Column(String(30), nullable=False)  
 author = Column(String(20), nullable=False)  
 genre = Column(String(40), nullable=False)  
 condition = Column(String(10), nullable=False)  
 user\_id = Column(Integer, ForeignKey('User.user\_id'), nullable=False)  
 owner = relationship("User", back\_populates="books")  
 listings = relationship("Listing", back\_populates="book")  
class Listing(Base):  
 \_\_tablename\_\_ = 'Listing'  
 listing\_id = Column(Integer, primary\_key=True)  
 created\_date = Column(Date, nullable=False)  
 book\_id = Column(Integer, ForeignKey('Book.book\_id'), nullable=False)  
 status = Column(String(30), nullable=False)  
 book = relationship("Book", back\_populates="listings")  
 requests = relationship("Request", back\_populates="listing")  
class Request(Base):  
 \_\_tablename\_\_ = 'Request'  
 request\_id = Column(Integer, primary\_key=True)  
 user\_id = Column(Integer, ForeignKey('User.user\_id'), nullable=False)  
 listing\_id = Column(Integer, ForeignKey('Listing.listing\_id'), nullable=False)  
 user = relationship("User", back\_populates="requests")  
 listing = relationship("Listing", back\_populates="requests")  
  
class Database:  
 def \_\_init\_\_(self):  
 self.engine = create\_engine('postgresql+psycopg2://postgres:1234@localhost/vakulchuk')  
 Base.metadata.create\_all(self.engine)  
 self.Session = sessionmaker(bind=self.engine)  
  
 def GetTable(self, table\_name):  
 with self.Session() as session:  
 entity\_map = {cls.\_\_tablename\_\_: cls for cls in Base.\_\_subclasses\_\_()}  
 if table\_name not in entity\_map:  
 raise ValueError(f"'{table\_name}' is not a valid table.")  
 entity = entity\_map[table\_name]  
 results = session.query(entity).all()  
 if not results:  
 raise ValueError(f"No entries found in '{table\_name}'.")  
 headers = [col.name for col in entity.\_\_table\_\_.columns]  
 data = [[getattr(record, col) for col in headers] for record in results]  
 return {"headers": headers, "data": data}  
  
 def ValidateInput(self, model, data):  
 table\_columns = model.\_\_table\_\_.columns  
 for column in table\_columns:  
 col\_name = column.name  
 col\_type = column.type.python\_type  
 if col\_name not in data:  
 continue  
 if col\_type == int:  
 data[col\_name] = int(data[col\_name])  
 elif col\_type == str:  
 data[col\_name] = str(data[col\_name])  
 elif col\_type == datetime.date:  
 try:  
 data[col\_name] = datetime.strptime(data[col\_name], "%Y-%m-%d").date()  
 except ValueError:  
 raise ValueError(f"Invalid date format for column '{col\_name}', expected 'YYYY-MM-DD'.")  
 inspector = inspect(self.engine)  
 foreign\_keys = inspector.get\_foreign\_keys(model.\_\_tablename\_\_)  
 for fk in foreign\_keys:  
 parent\_table\_name = fk['referred\_table']  
 parent\_column\_name = fk['referred\_columns'][0]  
 child\_column\_name = fk['constrained\_columns'][0]  
 parent\_model = {cls.\_\_tablename\_\_: cls for cls in Base.\_\_subclasses\_\_()}.get(parent\_table\_name)  
 if not parent\_model:  
 raise ValueError(f"Foreign key refers to non-existent table '{parent\_table\_name}'.")  
 with self.Session() as session:  
 if not session.query(parent\_model).filter(  
 getattr(parent\_model, parent\_column\_name) == data[child\_column\_name]).first():  
 raise ValueError(  
 f"No matching record in '{parent\_table\_name}' for value '{data[child\_column\_name]}'.")  
  
 def AddRecord(self, table\_name, values\_str):  
 with self.Session() as session:  
 table\_map = {cls.\_\_tablename\_\_: cls for cls in Base.\_\_subclasses\_\_()}  
 if table\_name not in table\_map:  
 raise ValueError(f"Table '{table\_name}' does not exist.")  
 model = table\_map[table\_name]  
 values\_list = [val.strip() for val in values\_str.split(",")]  
 table\_columns = [column.name for column in model.\_\_table\_\_.columns]  
 if len(values\_list) != len(table\_columns):  
 raise ValueError(  
 f"The number of provided values ({len(values\_list)}) does not match "  
 f"the number of columns in table '{table\_name}' ({len(table\_columns)})."  
 )  
 data = dict(zip(table\_columns, values\_list))  
 self.ValidateInput(model, data)  
 primary\_keys = {col.name: data[col.name] for col in model.\_\_table\_\_.primary\_key.columns}  
 if session.query(model).filter\_by(\*\*primary\_keys).first():  
 raise ValueError(f"Record with primary keys {primary\_keys} already exists.")  
 new\_record = model(\*\*data)  
 session.add(new\_record)  
 try:  
 session.commit()  
 except Exception as e:  
 session.rollback()  
 raise ValueError(f"Failed to insert record: {e}")  
  
 def UpdateRecord(self, table\_name, values\_str, record\_id):  
 with self.Session() as session:  
 table\_map = {cls.\_\_tablename\_\_: cls for cls in Base.\_\_subclasses\_\_()}  
 if table\_name not in table\_map:  
 raise ValueError(f"Table '{table\_name}' does not exist.")  
 model = table\_map[table\_name]  
 primary\_key = list(model.\_\_table\_\_.primary\_key.columns)[0].name  
 record = session.query(model).filter(getattr(model, primary\_key) == record\_id).first()  
 if not record:  
 raise ValueError(f"No record with ID {record\_id} found in table '{table\_name}'.")  
 values\_list = [val.strip() for val in values\_str.split(",")]  
 table\_columns = [column.name for column in model.\_\_table\_\_.columns]  
 if len(values\_list) != len(table\_columns):  
 raise ValueError(  
 f"The number of provided values ({len(values\_list)}) does not match "  
 f"the number of columns in table '{table\_name}' ({len(table\_columns)})."  
 )  
 data = dict(zip(table\_columns, values\_list))  
 self.ValidateInput(model, data)  
 for key, value in data.items():  
 setattr(record, key, value)  
 try:  
 session.commit()  
 except Exception as e:  
 session.rollback()  
 raise ValueError(f"Failed to update record: {e}")  
  
 def DeleteRecord(self, table\_name, record\_id):  
 with self.Session() as session:  
 table\_map = {cls.\_\_tablename\_\_: cls for cls in Base.\_\_subclasses\_\_()}  
 if table\_name not in table\_map:  
 raise ValueError(f"Table '{table\_name}' does not exist.")  
 model = table\_map[table\_name]  
 primary\_key = list(model.\_\_table\_\_.primary\_key.columns)[0].name  
 record = session.query(model).filter(getattr(model, primary\_key) == record\_id).first()  
 if not record:  
 raise ValueError(f"No record with ID {record\_id} found in table '{table\_name}'.")  
 inspector = inspect(self.engine)  
 foreign\_keys = inspector.get\_foreign\_keys(model.\_\_tablename\_\_)  
 for fk in foreign\_keys:  
 child\_table\_name = fk['referred\_table']  
 child\_column\_name = fk['referred\_columns'][0]  
 child\_model = table\_map.get(child\_table\_name)  
 if child\_model:  
 child\_column = getattr(child\_model, child\_column\_name, None)  
 if not child\_column:  
 raise ValueError(  
 f"Foreign key column '{child\_column\_name}' does not exist in table '{child\_table\_name}'."  
 )  
 dependent\_record = session.query(child\_model).filter(child\_column == record\_id).first()  
 if dependent\_record:  
 raise ValueError(  
 f"Cannot delete record; dependent records exist in table '{child\_table\_name}'."  
 )  
 try:  
 session.delete(record)  
 session.commit()  
 except Exception as e:  
 session.rollback()  
 raise ValueError(f"Failed to delete record: {e}")

Файл view.py:

class Display:  
 def PrintMessage(self, msg):  
 print(msg)  
  
 def PrintTable(db\_handler, table\_data):  
 headers = table\_data['headers']  
 data = table\_data['data']  
 row\_format = "{:<30}" \* len(headers)  
 print(row\_format.format(\*headers))  
 print("-" \* (30 \* len(headers)))  
 for row in data:  
 print(row\_format.format(\*map(str, row)))  
  
 def ShowMainMenu(self):  
 print("\nMenu options:")  
 print("1. View table\n2. Insert record\n3. Modify record\n4. Remove record\n5. Exit")  
 choice = input("Select an option: ")  
 return choice  
  
 def PromtTableName(self):  
 tableName = input("Enter the table name: ")  
 return tableName  
  
 def PromtID(self):  
 id = input("Enter the ID: ")  
 return id  
  
 def PromtValues(self):  
 enteries = input("Enter values (comma-separated): ")  
 return enteries

Файл controller.py:

from view import Display  
from model import Database  
  
class Manager:  
 def \_\_init\_\_(self):  
 self.dbView = Display()  
 self.dbModel = Database()  
  
 def Run(self):  
 while True:  
 choice = self.dbView.ShowMainMenu()  
 if choice == '1':  
 self.ShowTableData()  
 elif choice == '2':  
 self.InsertNewRecord()  
 elif choice == '3':  
 self.ModifyRecord()  
 elif choice == '4':  
 self.RemoveRecord()  
 elif choice == '5':  
 return  
 else:  
 self.dbView.PrintMessage("Invalid option. Please try again.\n")  
  
 def ShowTableData(self):  
 tableName = self.dbView.PromtTableName()  
 try:  
 tableData = self.dbModel.GetTable(tableName)  
 self.dbView.PrintTable(tableData)  
 except ValueError as e:  
 self.dbView.PrintMessage(str(e))  
  
 def InsertNewRecord(self):  
 tableName = self.dbView.PromtTableName()  
 inputValues = self.dbView.PromtValues()  
 try:  
 self.dbModel.AddRecord(tableName, inputValues)  
 except ValueError as e:  
 self.dbView.PrintMessage(str(e))  
  
 def ModifyRecord(self):  
 tableName = self.dbView.PromtTableName()  
 recordID = self.dbView.PromtID()  
 inputValues = self.dbView.PromtValues()  
 try:  
 self.dbModel.UpdateRecord(tableName, inputValues, recordID)  
 except ValueError as e:  
 self.dbView.PrintMessage(str(e))  
  
 def RemoveRecord(self):  
 tableName = self.dbView.PromtTableName()  
 recordID = self.dbView.PromtID()  
 try:  
 self.dbModel.DeleteRecord(tableName, recordID)  
 except ValueError as e:  
 self.dbView.PrintMessage(str(e))

**Репозиторій Git:** <https://github.com/Yaroslavvak/lab-2>