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

Деркач А. І.

Перевірив:

Павловский В. І.

Київ – 2024

**Мета:** здобуття вмінь програмування прикладних додатків баз даних PostgreSQL.

**Виконання роботи**

Нижче будуть наведені сутності предметної області:

**Сутності предметної області**

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

Рахунок з атрибутами: ідентифікатор рахунку, баланс, тип валюти, ідентифікатор власнику рахунку. Призначення: збереження інформації про рахунки які беруть участь в транзакціях.

Клієнт з атрибутами ідентифікатор клієнта, ім’я клієнта, адреса, номер телефону. Призначення: зберігання інформації про клієнтів фінансової системи.

Транзакція з атрибутами: ідентифікатор транзакції, дата транзакції, сума транзакції, ідентифікатор рахунку-відправника, ідентифікатор рахунку-отримувача. Призначення: збереження інформації про транзакції які відбулися.

Нижче будуть наведені зв’язки між сутностями предметної області:

**Зв’язки між сутностями предметної області**

Кожен рахунок має одного клієнта-власника. Інші клієнти також можуть користуватися цим рахунком, якщо їм було надано доступ. Таким чином, між сутностями Клієнт і Рахунок існує зв'язок M:N.

Один рахунок може здійснити багато транзакцій і одна транзакція може стосуватись тільки одного рахунку. Тому між сутностями Рахунок і Транзакція існує зв’язок 1:N.

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

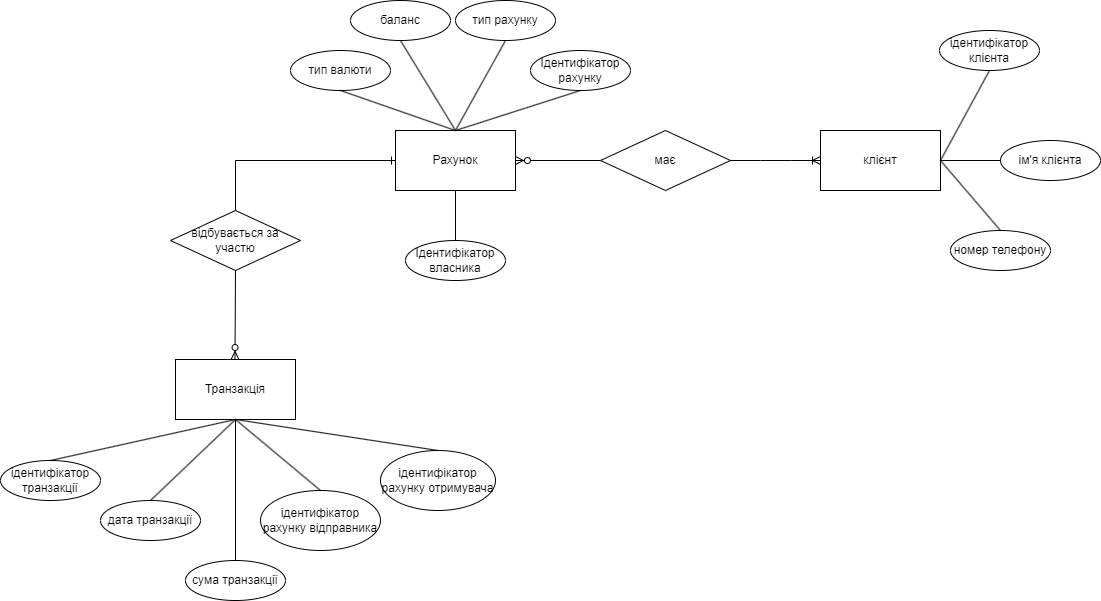


Рисунок 1 – ER-діаграма, побудована за нотацією Чена

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

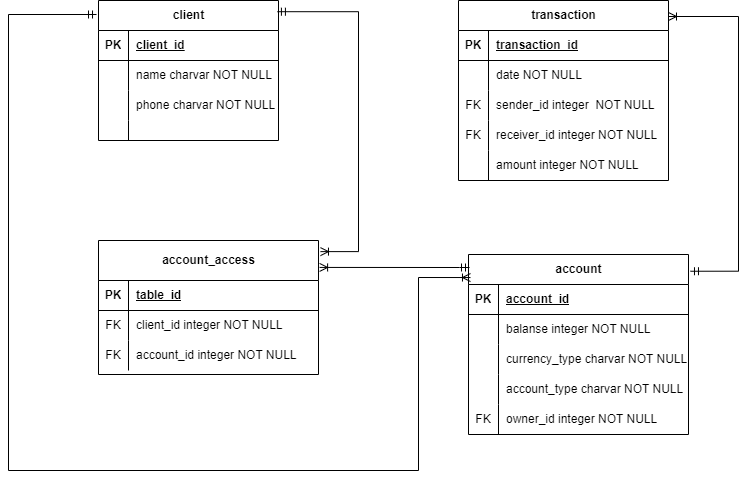


Рисунок 2 – Логічна модель

**Середовище та компоненти розробки**

У процесі розробки була використана мова програмування Python, інтегроване середовище розробки PyCharm, а також була використана бібліотека ORM 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**

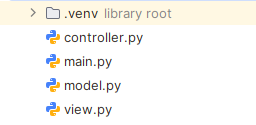
****

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 п’яти пунктів.

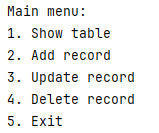


Рисунок 4 – Структура меню користувача

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

def Start(self):  
 while True:  
 input = self.view.MainMenu()  
 if input == '1':  
 self.OutputTable()  
 elif input == '2':  
 self.AddRecord()  
 elif input == '3':  
 self.UpdateRecord()  
 elif input == '4':  
 self.DeleteRecord()  
 elif input == '5':  
 exit(1)  
 else:  
 self.view.ShowMessage('Incorrect input. 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 ShowTable(self, table\_name):  
 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.")  
  
 table\_class = table\_map[table\_name]  
 records = session.query(table\_class).all()  
  
 if not records:  
 raise ValueError(f"No records found in table '{table\_name}'.")  
  
 columns = [col.name for col in table\_class.\_\_table\_\_.columns]  
 rows = [tuple(getattr(record, col) for col in columns) for record in records]  
  
 return {"field\_names": columns, "rows": rows}

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.ensure\_valid\_data(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.ensure\_valid\_data(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}")

Ці функції перевіряють, 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.

**Повний код програми**

Файл main.py:

from controller import Controller  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 controller = Controller()  
 controller.Start()

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 Client(Base):  
 \_\_tablename\_\_ = 'client'  
 client\_id = Column(Integer, primary\_key=True)  
 name = Column(String, nullable=False)  
 phone = Column(String, nullable=False)  
 accounts = relationship("Account", back\_populates="owner")  
 accesses = relationship("AccountAccess", back\_populates="client")  
  
class Account(Base):  
 \_\_tablename\_\_ = 'account'  
 account\_id = Column(Integer, primary\_key=True)  
 balance = Column(Integer, nullable=False)  
 currency\_type = Column(String, nullable=False)  
 account\_type = Column(String, nullable=False)  
 owner\_id = Column(Integer, ForeignKey('client.client\_id'), nullable=False)  
 owner = relationship("Client", back\_populates="accounts")  
 accesses = relationship("AccountAccess", back\_populates="account")  
 sent\_transactions = relationship("Transaction", foreign\_keys='Transaction.sender\_id', back\_populates="sender")  
 received\_transactions = relationship("Transaction", foreign\_keys='Transaction.receiver\_id', back\_populates="receiver")  
  
class AccountAccess(Base):  
 \_\_tablename\_\_ = 'account\_access'  
 table\_id = Column(Integer, primary\_key=True)  
 client\_id = Column(Integer, ForeignKey('client.client\_id'), nullable=False)  
 account\_id = Column(Integer, ForeignKey('account.account\_id'), nullable=False)  
 client = relationship("Client", back\_populates="accesses")  
 account = relationship("Account", back\_populates="accesses")  
  
class Transaction(Base):  
 \_\_tablename\_\_ = 'transaction'  
 transaction\_id = Column(Integer, primary\_key=True)  
 date = Column(Date, nullable=False)  
 sender\_id = Column(Integer, ForeignKey('account.account\_id'), nullable=False)  
 receiver\_id = Column(Integer, ForeignKey('account.account\_id'), nullable=False)  
 amount = Column(Integer, nullable=False)  
 sender = relationship("Account", foreign\_keys=[sender\_id], back\_populates="sent\_transactions")  
 receiver = relationship("Account", foreign\_keys=[receiver\_id], back\_populates="received\_transactions")  
  
class Model:  
 def \_\_init\_\_(self):  
 self.engine = create\_engine('postgresql+psycopg2://postgres:1234@localhost/derkach')  
 Base.metadata.create\_all(self.engine)  
 self.Session = sessionmaker(bind=self.engine)  
  
 def ShowTable(self, table\_name):  
 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.")  
  
 table\_class = table\_map[table\_name]  
 records = session.query(table\_class).all()  
  
 if not records:  
 raise ValueError(f"No records found in table '{table\_name}'.")  
  
 columns = [col.name for col in table\_class.\_\_table\_\_.columns]  
 rows = [tuple(getattr(record, col) for col in columns) for record in records]  
  
 return {"field\_names": columns, "rows": rows}  
  
 def ensure\_valid\_data(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.ensure\_valid\_data(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.ensure\_valid\_data(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:

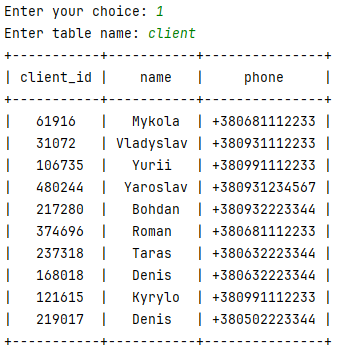
from prettytable import PrettyTable  
  
class View:  
 def ShowMessage(self, message):  
 print(message)  
  
 def ShowTable(self, result\_proxy):  
 table = PrettyTable()  
 table.field\_names = result\_proxy["field\_names"]  
 for row in result\_proxy["rows"]:  
 table.add\_row(row)  
 print(table)  
  
 def MainMenu(self):  
 print("\nMain menu:")  
 print("1. Show table\n2. Add record\n3. Update record\n4. Delete record\n5. Exit")  
 choice = input("Enter your choice: ")  
 return choice  
  
 def GetTableName(self):  
 name = input("Enter table name: ")  
 return name  
  
 def GetValues(self):  
 values = input("Enter values (separated by commas): ")  
 return values  
  
 def GetID(self):  
 id = input("Enter ID: ")  
 return id

Файл controller.py:

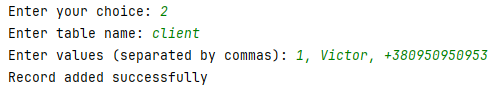
from model import Model  
from view import View  
  
class Controller:  
 def \_\_init\_\_(self):  
 self.model = Model()  
 self.view = View()  
 self.view.ShowMessage('Database connected successfully')  
  
 def Start(self):  
 while True:  
 input = self.view.MainMenu()  
 if input == '1':  
 self.OutputTable()  
 elif input == '2':  
 self.AddRecord()  
 elif input == '3':  
 self.UpdateRecord()  
 elif input == '4':  
 self.DeleteRecord()  
 elif input == '5':  
 exit(1)  
 else:  
 self.view.ShowMessage('Incorrect input. Try again\n')  
  
 def OutputTable(self):  
 name = self.view.GetTableName()  
 try:  
 table = self.model.ShowTable(name)  
 self.view.ShowTable(table)  
 except Exception as e:  
 self.view.ShowMessage(str(e))  
  
 def AddRecord(self):  
 name = self.view.GetTableName()  
 values = self.view.GetValues()  
 try:  
 self.model.AddRecord(name, values)  
 self.view.ShowMessage('Record added successfully')  
 except Exception as e:  
 self.view.ShowMessage(str(e))  
  
 def UpdateRecord(self):  
 name = self.view.GetTableName()  
 id = self.view.GetID()  
 values = self.view.GetValues()  
 try:  
 self.model.UpdateRecord(name, values, id)  
 self.view.ShowMessage('Record updated successfully')  
 except Exception as e:  
 self.view.ShowMessage(str(e))  
  
 def DeleteRecord(self):  
 name = self.view.GetTableName()  
 id = self.view.GetID()  
 try:  
 self.model.DeleteRecord(name, id)  
 self.view.ShowMessage('Record deleted successfully')  
 except Exception as e:  
 self.view.ShowMessage(str(e))

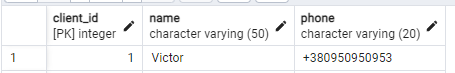
**Результат виконання програми**

Вивід даних:

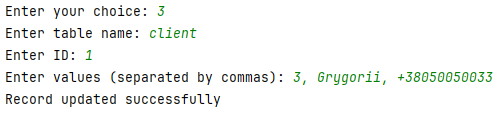
****

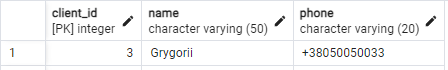
Вставка даних:

****

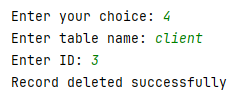
****

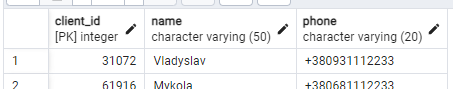
Редагування даних:

****

****

Видалення даних:

****

****

**Створення та аналіз різних типів індексів**

Задані індекси за варіантом: GIN, BRIN

GIN-індекси демонструють високу ефективність для текстового пошуку завдяки можливості швидко обробляти складні запити, що базуються на повнотекстовому пошуку або фільтрації за ключовими словами. Вони є ідеальними для полів, які містять текстові дані великого обсягу (наприклад, опис або типи рахунків).

BRIN-індекси є оптимальними для стовпців, що мають значення, які природно впорядковуються (наприклад, дати, географічні координати). Вони дозволяють суттєво зменшити кількість оброблюваних блоків, що прискорює запити для великих таблиць, хоча можуть потребувати додаткової перевірки через втрату точності (lossy blocks).

Створення індекса GIN:

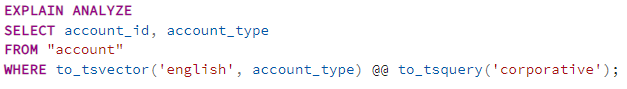


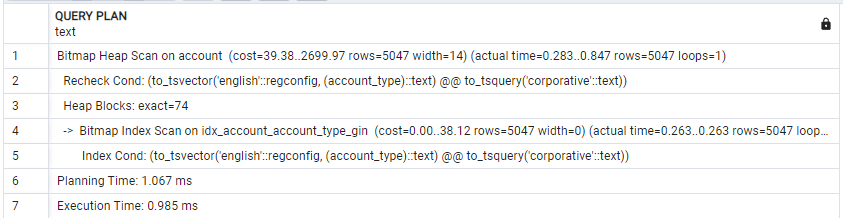
Створення індекса BRIN:



Приклади запитів:

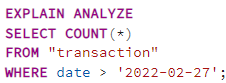
1. Просте фільтрування

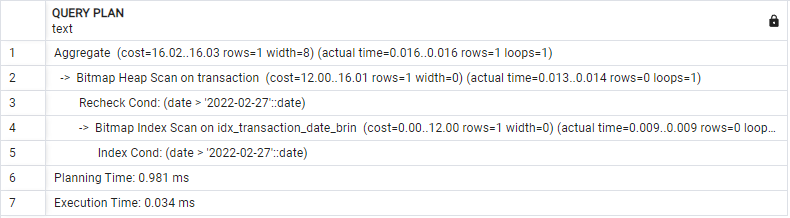




GIN-індекс на полі account\_type прискорює пошук за текстовими даними. Це ефективно для пошукових запитів, таких як @@ для повнотекстового пошуку.

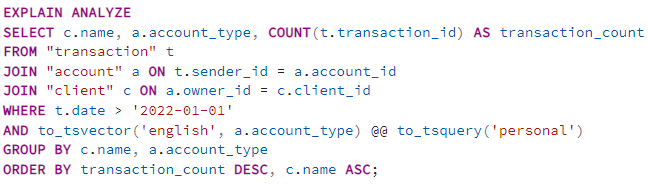
1. Агрегатна функція

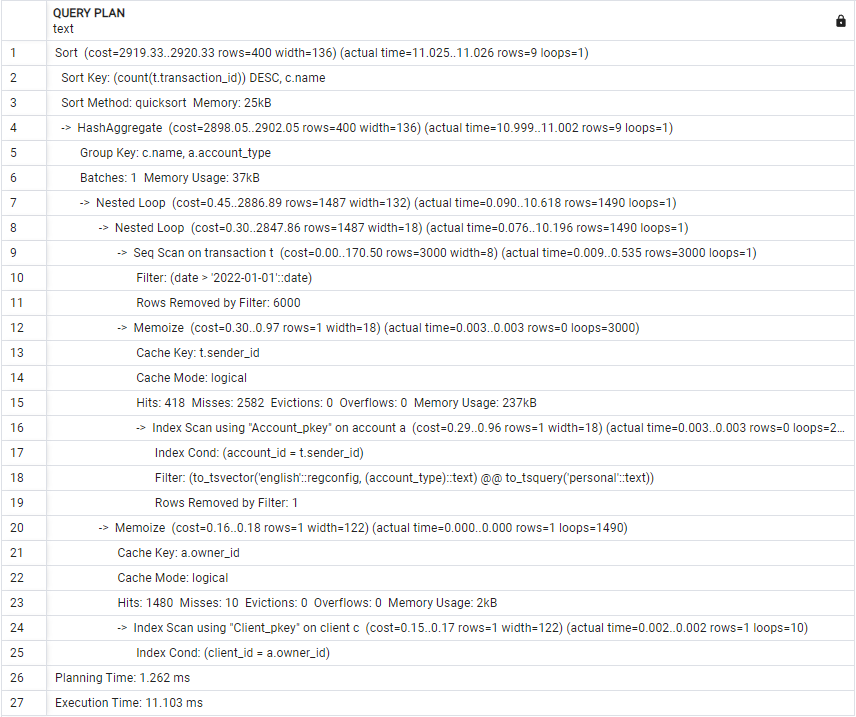




BRIN-індекс на полі date допомагає швидко знаходити відповідні записи, оскільки обмежує перевірку лише до блоків, які потенційно містять дані, що відповідають умові.

1. Групування та сортування

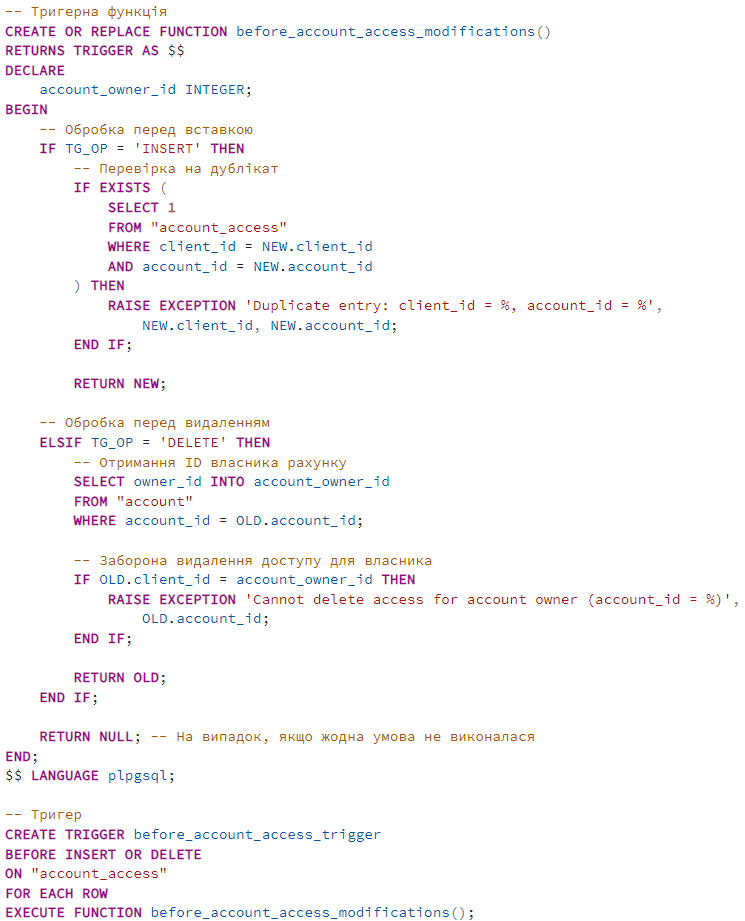




GIN-індекс для текстового поля account\_type та BRIN-індекс для дати дозволяють оптимізувати запит завдяки прискоренню фільтрації та об'єднань.

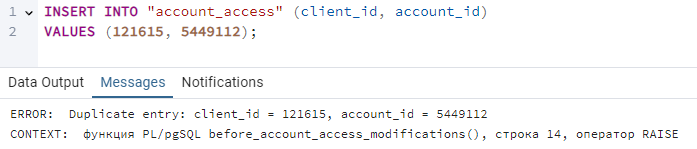
**Розробка тригера**

Створення тригера:

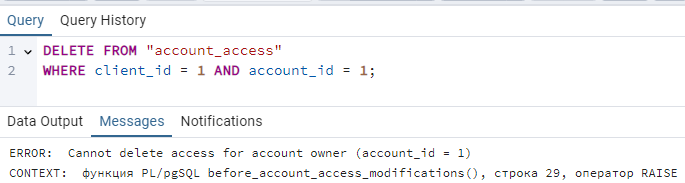


Тестування тригера:

Вставка даних із дублікатом:



Спроба видалення доступу власника рахунку:

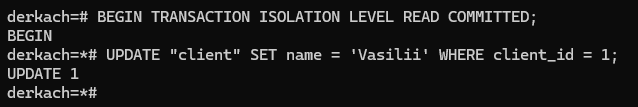


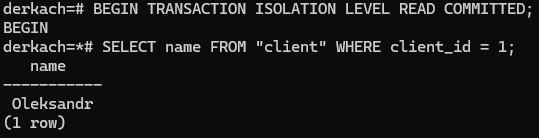
Як видно з поля ERROR, консоль виводить повідомлення задане тригером, одже тригер працює коректно.

**Приклади та аналіз рівнів ізоляцій тарнзакцій**

1. **READ COMMITTED**

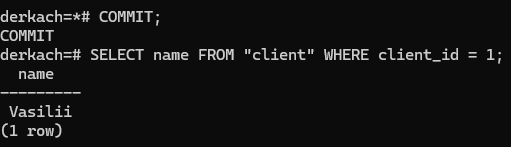
Цей рівень ізоляції гарантує, що транзакція читає лише ті дані, які були закомічені іншими транзакціями та уникає "брудного читання" (dirty reads), але дозволяє неповторюване читання (non-repeatable reads), тобто дані можуть змінитися між запитами в межах однієї транзакції.





Зміни не відображаются в другому вікні.

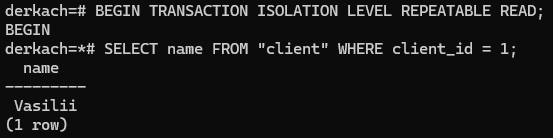


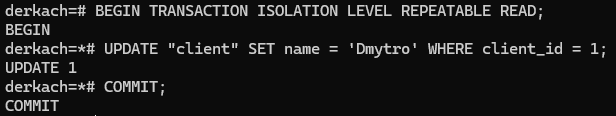


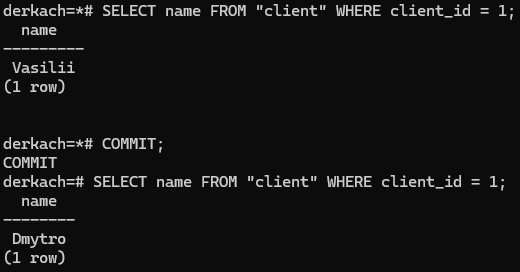
Зміни почали відображатися в другому вікні після фіксування змін в першому вікні.

1. **REPEATABLE READ**

Цей рівень ізоляції гарантує, що дані, які були прочитані в межах транзакції, не зміняться іншими транзакціями до завершення поточної та уникає "брудного читання" і "неповторюваного читання", але допускає фантомне читання (phantom reads), коли нові рядки можуть бути додані іншими транзакціями.



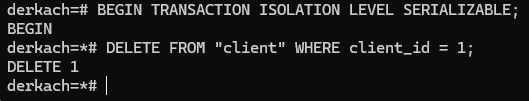


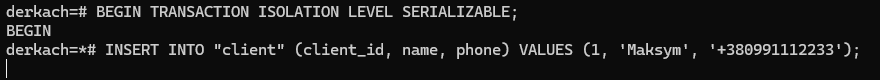


Після оновлення у другому вікні, другий запит у першому вікні все ще повертає оригінальне значення до прийняття змін.

1. **SERIALIZABLE**

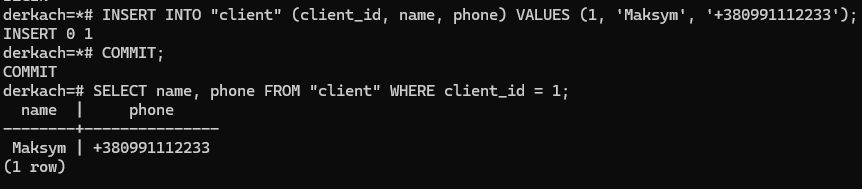
Найвищий рівень ізоляції. Транзакції виконуються так, ніби вони були послідовно ізольовані одна від одної. Уникає "брудного читання", "неповторюваного читання" та "фантомного читання". Найповільніший через блокування ресурсів, але забезпечує максимальну цілісність даних.





Операція була заблокована через незафіксоване видалення в першому вікні.





Операція виконалася після фіксування змін в першому вікні.

**Висновок:**

Вибір рівня ізоляції залежить від конкретних вимог до системи:

* READ COMMITTED оптимальний для сценаріїв із високою продуктивністю, де точність не є критично важливою.
* REPEATABLE READ підходить для аналітичних запитів, де необхідно отримати узгоджені дані.
* SERIALIZABLE слід використовувати для фінансових операцій чи інших завдань, які вимагають максимальної цілісності даних.

Грамотне налаштування рівнів ізоляції дозволяє забезпечити баланс між продуктивністю і коректністю в багатокористувацьких системах.

Репозиторій Github**:** [**https://github.com/andriyderkach1/KV22\_Derkach\_SQL\_Lab2.git**](https://github.com/andriyderkach1/KV22_Derkach_SQL_Lab2.git)