**Міністерство освіти і науки України**

**Національний технічний університет України**

**“Київський політехнічний інститут ім. Ігоря Сікорського”**

**Факультет прикладної математики**

**Кафедра спеціалізованих комп’ютерних систем**

**Лабораторна робота № 3**

з дисципліни «Бази даних і засоби управління»

«Ознайомлення з базовими операціями СУБД PostgreSQL»

Виконав:

студент групи КП-73

Литвиненко Антон

Перевірив:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Завдання

Завдання роботи полягає у наступному:

1.Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM).

2.Створити та проаналізувати різні типи індексів у PostgreSQL.

3.Розробити тригер бази даних PostgreSQL.

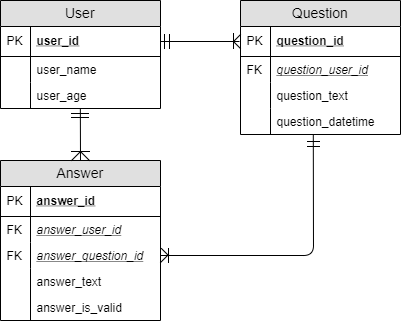
4.Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

Порядок виконання роботи

В ході роботи розроблено:

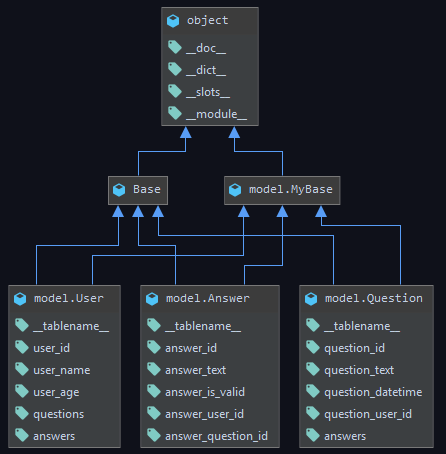
1. Логічну модель БД та Діаграму класів;
2. Функціонал програмного додатку;
3. ОО програмний додаток роботи з БД "Система питання-відповідь". Для взаємодії з БД використано ORM модуль SQLAlchemy.

Логічна модель бази даних наведена на Рис 1.



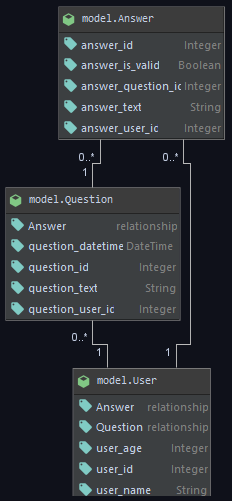
*Рис 1. Логічна модель бази даних*

Сутнісні класи програми наведені на Рис 2.



*Рис 2. Фрагмент UML діаграми сутнісних класів*

Зв’язки між сутнісними класами, сгенеровані за допомогою PyCharm, наведені на Рис 3.



*Рис 3. Зв’язки між сутнісними класами*

Лістинг програми

*from* controller *import* display\_main\_menu  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 display\_main\_menu()

controller.py

*from* consolemenu *import* SelectionMenu  
  
*import* model  
*import* view  
  
  
*def* display\_main\_menu(err='', table=*None*):  
 tables = *list*(model.TABLES.keys())  
  
 menu = SelectionMenu(tables + ['Make commit'], subtitle=err,  
 title="Select a table to work with:")  
 menu.show()  
  
  
 index = menu.selected\_option  
 *if* index < *len*(tables):  
 table = tables[index]  
 display\_secondary\_menu(table)  
 *elif* index == *len*(tables):  
 model.commit()  
 display\_main\_menu('Commit was made successful')  
  
  
*def* display\_secondary\_menu(table, subtitle=''):  
 opts = ['Select', 'Insert', 'Update', 'Delete']  
 steps = [select, insert, update, delete, display\_main\_menu]  
  
 menu = SelectionMenu(  
 opts, subtitle=subtitle,  
 title=f'Selected table "{table}"', exit\_option\_text='Go back',)  
 menu.show()  
 index = menu.selected\_option  
 steps[index](table=table)  
  
  
*def* select(table):  
 query = view.multiple\_input(table, 'Enter requested fields:')  
 data = model.get(table, query)  
 view.print\_entities(table, data)  
 view.press\_enter()  
 display\_secondary\_menu(table)  
  
  
*def* insert(table):  
 data = view.multiple\_input(table, 'Enter new fields values:')  
 model.insert(table, data)  
 display\_secondary\_menu(table, 'Insertion was made successfully')  
  
  
*def* update(table):  
 condition = view.single\_input(  
 table, 'Enter requirement of row to be changed:')  
 query = view.multiple\_input(table, 'Enter new fields values:')  
  
 model.update(table, condition, query)  
 display\_secondary\_menu(table, 'Update was made successfully')  
  
  
*def* delete(table):  
 query = view.multiple\_input(  
 table, 'Enter requirement of row to be deleted:')  
  
 model.delete(table, query)  
 display\_secondary\_menu(table, 'Deletion was made successfully')

model.py

*from* sqlalchemy *import* Column, Integer, String, DateTime, \  
 Boolean, ForeignKey, create\_engine  
*from* sqlalchemy.ext.declarative *import* declarative\_base  
*from* sqlalchemy.orm *import* relationship  
*from* sqlalchemy.orm *import* sessionmaker  
  
COLUMN\_WIDTH = 25  
db\_str = 'postgres://anton:password@localhost:5432/labs'  
db = create\_engine(db\_str)  
Base = declarative\_base()  
  
  
*class* MyBase:  
 *def \_\_init\_\_*(*self*, \*\*kwargs):  
 *for* attr, val *in* kwargs.items():  
 *setattr*(*self*, attr, val)  
  
 *def* \_\_clean\_dict(*self*):  
 clean = *self*.*\_\_dict\_\_*.copy()  
 clean.pop('\_sa\_instance\_state')  
 *return* clean  
  
 *def* get\_columns(*self*):  
 *return self*.\_\_clean\_dict().keys()  
  
 *def* get\_values(*self*):  
 *return self*.\_\_clean\_dict().values()  
  
  
*class* User(MyBase, Base):  
 \_\_tablename\_\_ = 'user'  
 user\_id = Column(Integer, primary\_key=*True*)  
 user\_name = Column(String, nullable=*False*)  
 user\_age = Column(Integer)  
  
 questions = relationship('Question')  
 answers = relationship('Answer')  
  
  
*class* Question(MyBase, Base):  
 \_\_tablename\_\_ = 'question'  
  
 question\_id = Column(Integer, primary\_key=*True*)  
 question\_text = Column(String, nullable=*False*)  
 question\_datetime = Column(DateTime)  
 question\_user\_id = Column(Integer, ForeignKey('user.user\_id'))  
  
 answers = relationship('Answer')  
  
  
*class* Answer(MyBase, Base):  
 \_\_tablename\_\_ = 'answer'  
  
 answer\_id = Column(Integer, primary\_key=*True*)  
 answer\_text = Column(String, nullable=*False*)  
 answer\_is\_valid = Column(Boolean, default=*False*)  
 answer\_user\_id = Column(Integer, ForeignKey('user.user\_id'))  
 answer\_question\_id = Column(Integer, ForeignKey('question.question\_id'))  
  
  
session = sessionmaker(db)()  
Base.metadata.create\_all(db)  
  
TABLES = {  
 'user': ('user\_id', 'user\_name', 'user\_age'),  
 'question': ('question\_id', 'question\_text', 'question\_datetime', 'question\_user\_id'),  
 'answer': ('answer\_id', 'answer\_text', 'answer\_is\_valid', 'answer\_user\_id', 'answer\_question\_id')  
}  
  
MODELS = {'user': User, 'question': Question, 'answer': Answer}  
  
*def* insert(table, opts):  
 object\_class = MODELS[table]  
 obj = object\_class(\*\*opts)  
 session.add(obj)  
  
  
*def* get(table, opts=*None*):  
 objects\_class = MODELS[table]  
 objects = session.query(objects\_class)  
 *for* key, item *in* opts.items():  
 objects = objects.filter(*getattr*(objects\_class, key) == item)  
  
 *return list*(objects)  
  
  
*def* update(table, condition, opts):  
 column, value = condition  
 object\_class = MODELS[table]  
 filter\_attr = *getattr*(object\_class, column)  
 objects = session.query(object\_class).filter(filter\_attr == value)  
  
 *for* obj *in* objects:  
 *for* key, item *in* opts.items():  
 *setattr*(obj, key, item)  
  
  
*def* delete(table, opts):  
 objects\_class = MODELS[table]  
 objects = session.query(objects\_class)  
 *for* key, item *in* opts.items():  
 objects = objects.filter(*getattr*(objects\_class, key) == item)  
  
 objects.delete()  
  
  
*def* commit():  
 session.commit()

view.py

*import* model  
  
COLUMN\_WIDTH = 25  
  
  
*def* print\_entities(table, data):  
 entities = data  
 *if not* entities:  
 *return* cols = entities[0].get\_columns()  
 separator\_line = '-' \* COLUMN\_WIDTH \* *len*(cols)  
  
 *print*(f'Working with table "{table}"', end='\n\n')  
 *print*(separator\_line)  
 *print*(''.join([f'{col} |'.rjust(COLUMN\_WIDTH, ' ') *for* col *in* cols]))  
 *print*(separator\_line)  
  
 *for* entity *in* entities:  
 *print*(''.join([f'{val} |'.rjust(COLUMN\_WIDTH, ' ') *for* val *in* entity.get\_values()]))  
 *print*(separator\_line)  
  
  
*def* specified\_input(colname=*None*, msg=*None*):  
 *if* msg:  
 *print*(msg)  
 *if* colname:  
 *print*(f'{colname}=', end='')  
 *return input*()  
  
  
*def* single\_input(tname, msg):  
 *print*(msg)  
 *print*('(use format <attribute>=<value>)')  
 *print*(f'({"/".join(model.TABLES[tname])})', end='\n\n')  
  
 *while True*:  
 data = *input*()  
 *if not* data *or* data.count('=') != 1:  
 *print*('Invalid input, try one more time')  
 *continue* data = data.split('=')  
 col, val = data[0].strip(), data[1].strip()  
 *if* col.lower() *in* [tcol.lower() *for* tcol *in* model.TABLES[tname]]:  
 *return* col, val  
 *else*:  
 print(f'Invalid column name "{col}" for table "{tname}"')  
  
  
*def* multiple\_input(tname, msg):  
 print(msg)  
 print('(use format <attribute>=<value>)')  
 print(f'({"/".join(model.TABLES[tname])})', end='\n\n')  
  
 res = {}  
 *while True*:  
 data = input()  
 *if not* data:  
 *break  
 if* data.count('=') != 1:  
 print('Invalid input')  
 *continue* data = data.split('=')  
 col, val = data[0].strip(), data[1].strip()  
 *if* col.lower() *in* [tcol.lower() *for* tcol *in* model.TABLES[tname]]:  
 res[col] = val  
 *else*:  
 print(f'Invalid column name "{col}" for table "{tname}"')  
  
 *return* res  
  
  
*def* multiline\_input(msg):  
 print(msg, end='\n\n')  
  
 lines = []  
 *while True*:  
 line = input()  
 *if not* line:  
 *break* lines.append(line)  
  
 *return* '\n'.join(lines)  
  
  
*def* press\_enter():  
 input()

**Індекси**

**Btree індекс:**

*create index* btree\_index *on* question *using* btree(question\_id);

Порядок звертання до таблиці без використання фільтру по колонці, на яку додано індекс (створений індекс не використовується):

Запит:

*explain select \* from* question;

Результат:

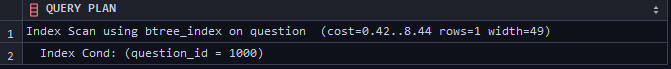


Порядок звертання до таблиці з використанням фільтру по колонці, на яку додано індекс (пошук відбувається за допомогою створеного індексу):

Запит:

*explain select \* from* question *where* question\_id = 1000;

Результат:



**BRIN індекс:**

*create index* brin\_index *on* question *using* brin(question\_datetime);

Порядок звертання до таблиці без використання фільтру по колонці, на яку додано індекс (створений індекс не використовується):

Запит:

*explain select \* from* question;

Результат:



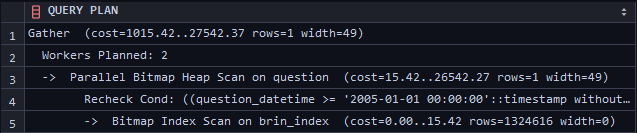
Порядок звертання до таблиці з використанням фільтру по колонці, на яку додано індекс (пошук відбувається за допомогою створеного індексу):

Запит:

*explain select \* from* question

*where* question\_datetime *between* '2005-01-01 00:00:00' *and* '2015-01-01 00:00:00';

Результат:



**Тригер**

**BEFORE INSERT**

Якщо текст питання не закінчується знаком питання(?), то таке питання не буде додане.

Код:

*create or replace function before\_insert\_question*()  
*returns trigger  
language* plpgsql  
*as* $$  
*begin  
 if NEW*.question\_text *LIKE* '%?' *then  
 return NEW*;  
 *end if*;  
 *raise exception* 'question text should end with `?`';  
*end*;  
$$;

*create trigger* before\_insert *before insert on* question  
 *for each row execute procedure before\_insert\_question*();

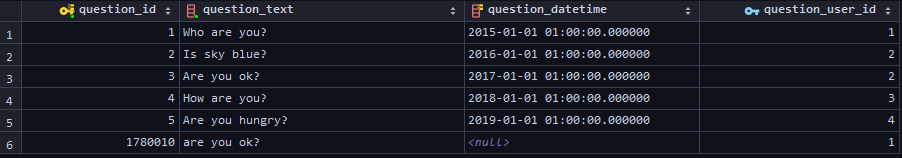
Приклади результатів:

*insert into* question(question\_text, question\_user\_id) *values* ('are you ok', 1);



Або вдале додавання і усі питання після цього:

*insert into* question(question\_text, question\_user\_id) *values* ('are you ok?', 1);



**BEFORE DELETE**

Якщо таблиці менше 10 питань, видалити питання не вдасться. (реалізовано курсорним циклом через вимогу використання в функції тригера подібного циклу, тим не менш підрахунок кількості питань в таблиці можна було б реалізувати SQL запитом).

Код:

*create or replace function before\_delete\_question*()  
*returns trigger  
language* plpgsql  
*as* $$  
*declare* all\_questions *cursor is select \* from* question;  
 question\_count *integer* = 0;  
*begin  
 for* q *in* all\_questions *loop* question\_count := question\_count + 1;  
 *end loop*;  
 *if* question\_count < 10 *then  
 raise exception* 'At least 10 questions should be in table';  
 *end if*;  
 *return old*;  
*end*;  
$$;

*create trigger* before\_delete *before delete on* question  
 *for each row execute procedure before\_delete\_question*();

Приклади результатів:

*select count*(*\**) *from* question;



*delete from* question *where* question\_id = 1780010;  
*select count*(*\**) *from* question;



*delete from* question *where* question\_id = 1780014;



*select count*(*\**) *from* question;



**Дослідження рівнів ізоляції**

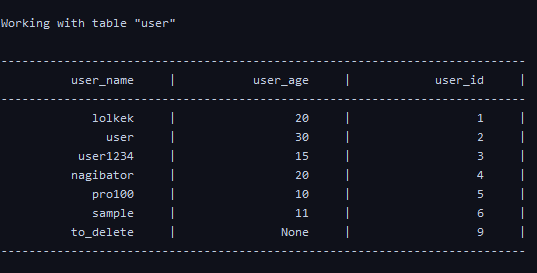
Для перевірки аномалій буде використовуватися розроблений програмний додаток, запущений у двох екземплярах паралельно.

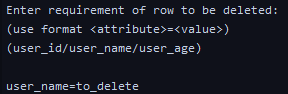
1. **READ COMMITTED**

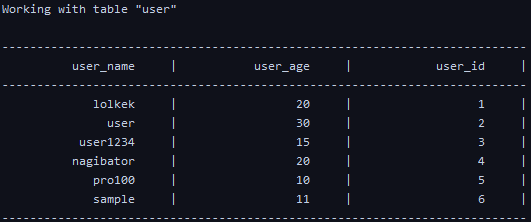
Перевіримо наявність аномалії “dirty read”, коли транзакція читає дані, які ще не були закомічені паралельною транзакцією.

Транзакція №1

Перевіряє список користувачів, видаляє якогось користувача, перевіряє видалення.

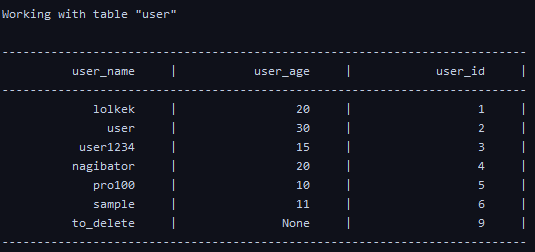






Транзакція №2

Отримує список всіх Користувачів.



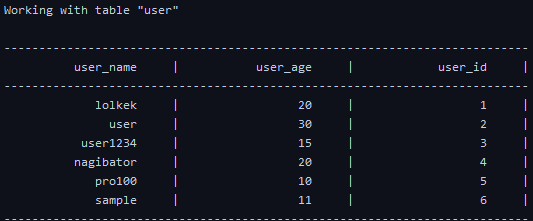
Транзакція №1

Робить коміт.



Транзакція №2

Отримує список всіх Клієнтів.



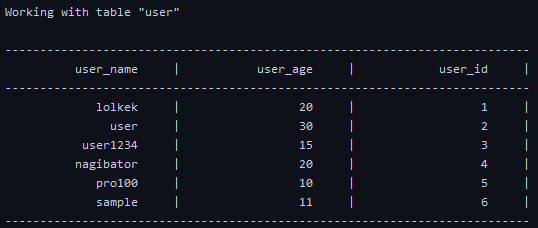
Як бачимо Транзація №2 не бачила зміни, внесені до таблиці Транзакцією №1, до ти пір поки остання не закомітила свої зміни. Отже, було доведено, що рівень ізоляції READ COMMITTED захищає від аномалії “брудного читання”.

2. **REPEATABLE READ**

Перевіримо наявність аномалії “nonrepeatable read”, коли транзакція повторно зчитує дані і вони виявляються модифіковані комітом паралельної транзакції.

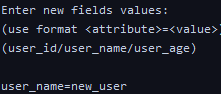
Транзакція №1

Отримує список усіх Користувачів.

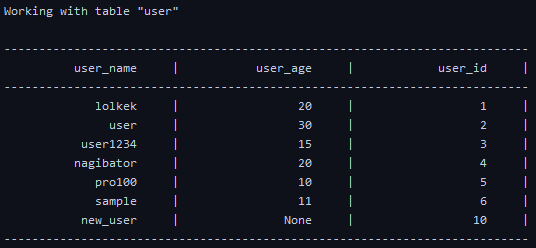


Транзакція №2

Створює нового Клієнта, комітить зміни і перевіряє його наявність в списку усіх клієнтів.

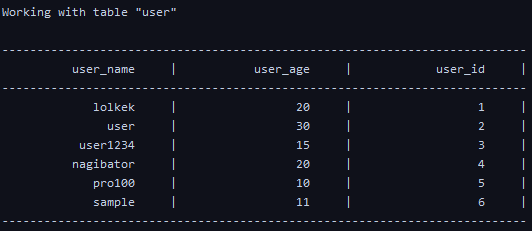






Транзакція №1

Заново отримує список усіх клієнтів.



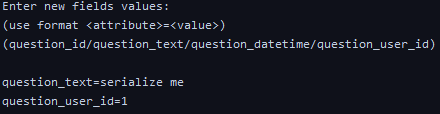
Як бачимо, Транзакція №1 так і не побачила закомічені зміни Транзакції №2, так як перша з них зчитувала дані з таблиці ще до коміту. Отже, було доведено, що рівень ізоляції REPEATBLE READ захищає від аномалії “nonrepeatable read”.

3. **SERIALIZABLE**

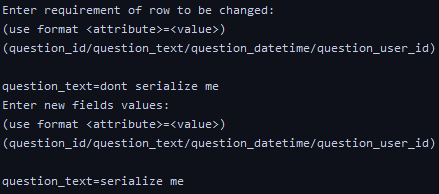
Перевіримо наявність аномалії “serialization anomaly”, коли дві паралельні транзакції хочуть закомітити свої результати і при цьому є різниця, в якому порядку виконувати команди, виконані кожною з транзакцій.

Транзакція №1

Створює нове питання з текстом ‘serialize me’.

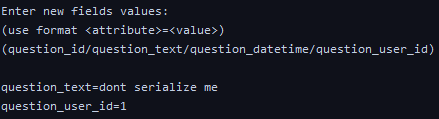


Оновлює всі дописи з текстом ‘dont serialize me’.

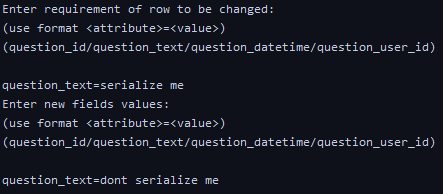


Транзакція №2

Створює допис з текстом ‘dont serialize me’.



Оновлює всі дописи з текстом ‘serialize me’.



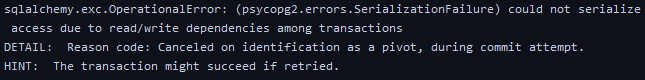
Транзакція №1

Намагається закомітити зміни



Транзакція №2

Намагається закомітити зміни



Як бачимо Транзація №2 не змогла закомітити зміни бо порядок виконання команд з обох транзакцій змінює вихідний результат. Отже, було доведено, що рівень ізоляції SERIALIZABLE захищає від “serialization anomaly”.