

**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»

Виконала:
студентка групи КП-72
Завгородня Анна

Перевірив:

Завдання

Завдання роботи полягає у наступному:

- 1.Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проєкції (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 " ... ". Для взаємодії з БД використано ORM модуль SQLAlchemy.

Логічна модель бази даних наведена на Рис 1.

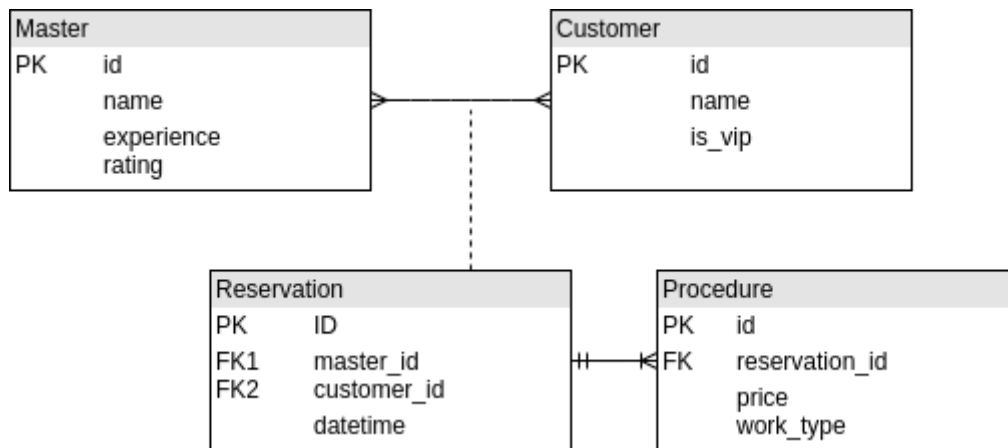


Рис 1. Логічна модель бази даних

Сутнісні класи програми наведені на Рис 2.

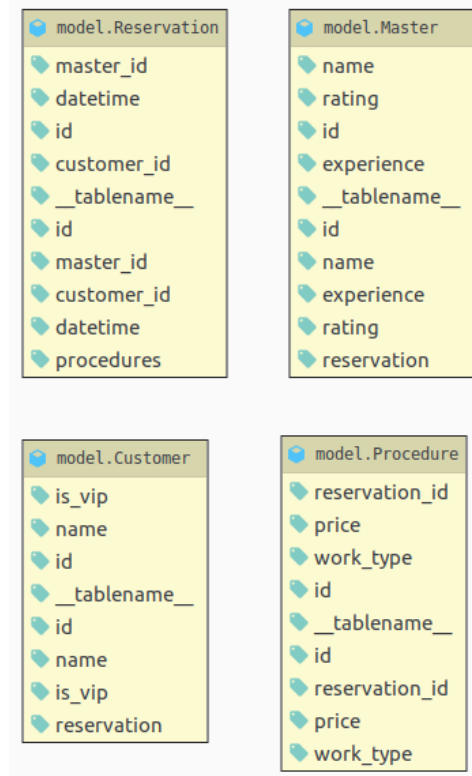


Рис 2. Фрагмент UML

Зв'язки між сутнісними класами, сгенеровані за допомогою SQLAlchemy, наведені на Рис 3.

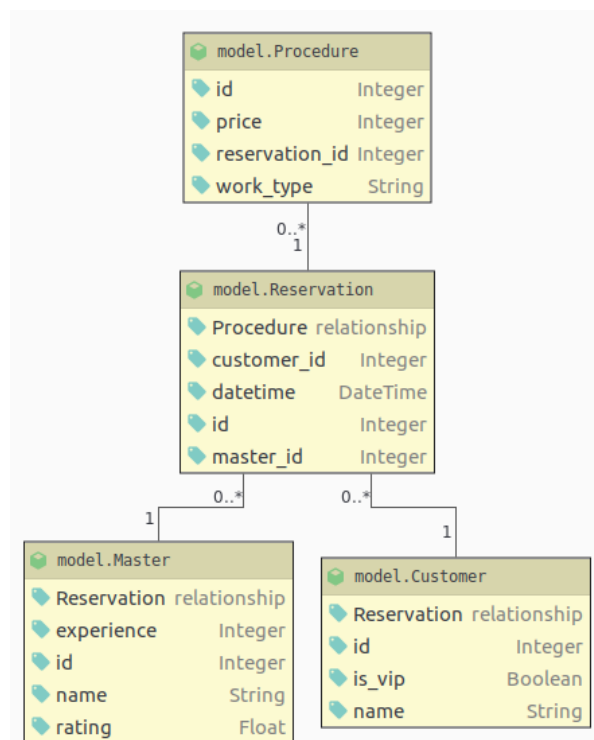


Рис 3. Зв'язки між сутнісними класами

Меню програми наведене на схемі нижче.

Оберіть таблицю або дію:

- 1 - master
- 2 - customer
- 3 - reservation
- 4 - procedure
- 5 - Створити 10 000 рандомних майстрів
- 6 - Зробити коміт
- 7 - Exit

Пункт 1-4:

Обрано таблицю `master`

- 1 - GET
- 2 - INSERT
- 3 - UPDATE
- 4 - DELETE
- 5 - Назад

Пункт 5:

10 000 випадкових майстрів додано

Пункт 6:

Зміни успішно збережені

Пункт 7 (вихід з програми):

Пака!!!!

Лістинг програми

```
import controller

if __name__ == '__main__':
    controller.Controller().show_start_menu()
```

controller.py

```
from consolemenu import SelectionMenu

import model
import view
import scanner

class Controller:
    def __init__(self):
        self.model = model.Model()
        self.view = view.View()

        self.tables = list(model.TABLES.keys())

    def get_table_name(self, index):
        try:
            return self.tables[index]
        except IndexError:
            return None

    def show_start_menu(self, subtitle='', **kwargs):
        menu_options = self.tables + ['Створити 10 000 рандомних майстрів',
                                       'Зробити коміт']
        next_steps = [self.show_table_menu] * len(self.tables) + [
            self.create_random_masters,
            self.commit
        ]
        menu = SelectionMenu(menu_options, subtitle=subtitle,
                              title="Оберіть таблицю або дію:")
        menu.show()

        index = menu.selected_option
        if index < len(menu_options):
            table_name = self.get_table_name(index)
            next_step = next_steps[index]
            try:
                next_step(table_name=table_name)
            except Exception as err:
                self.show_start_menu(subtitle=str(err))
        else:
            print('Пака!!!!')

    def show_table_menu(self, table_name, subtitle=''):
        next_steps = [self.get, self.insert, self.update, self.delete,
self.show_start_menu]
        menu = SelectionMenu(
```

```

        ['GET', 'INSERT', 'UPDATE', 'DELETE'], subtitle=subtitle,
        title=f'Обрано таблицю `{table_name}`', exit_option_text='Назад', )
    menu.show()

    next_step = next_steps[menu.selected_option]
    next_step(table_name=table_name)

    def get(self, table_name):
        filter_by = scanner.input_dict(table_name, 'За чим фільтрувати запит? Залиште
пустим щоб отримати всі рядки:')
        data = self.model.get(table_name, **filter_by)
        self.view.print_entities(table_name, data)
        scanner.press_enter()
        self.show_table_menu(table_name)

    def insert(self, table_name):
        new_values = scanner.input_dict(table_name, 'Введіть нові значення:')
        self.model.insert(table_name, **new_values)
        self.show_table_menu(table_name, 'Вставка відбулася успішно')

    def update(self, table_name):
        filter_by = scanner.input_dict(table_name, 'Який рядок треба змінити?:',
limit=1)
        new_values = scanner.input_dict(table_name, 'Введіть нові значення:')
        self.model.update(table_name, list(filter_by.items())[0], **new_values)
        self.show_table_menu(table_name, 'Оновлення відбулося успішно')

    def delete(self, table_name):
        filter_by = scanner.input_dict(table_name, 'Які рядки треба видалити?')
        self.model.delete(table_name, **filter_by)
        self.show_table_menu(table_name, 'Видалення відбулося успішно')

    def create_random_masters(self, **kwargs):
        self.model.create_random_masters()
        self.show_start_menu('10 000 випадкових майстрів додано')

    def commit(self, **kwargs):
        self.model.commit()
        self.show_start_menu(subtitle='Зміни успішно збережені')

```

model.py

```

from sqlalchemy import Column, Integer, String, DateTime, \
    Boolean, ForeignKey, Float, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, sessionmaker

db_str = 'postgres://admin:admin@localhost:5432/kpi'
db = create_engine(db_str)
Base = declarative_base()

class ReprMixin:
    def __repr__(self):

```

```
clean_dict = self.__dict__.copy()
clean_dict.pop('_sa_instance_state')
return f'<{self.__class__.__name__}>{clean_dict}'
```

```
class Master(Base, ReprMixin):
```

```
    __tablename__ = 'master'
```

```
    id = Column(Integer, primary_key=True)
```

```
    name = Column(String)
```

```
    experience = Column(Integer)
```

```
    rating = Column(Float)
```

```
    reservation = relationship('Reservation')
```

```
    def __init__(self, id=None, name=None, experience=None, rating=None):
```

```
        self.id = id
```

```
        self.name = name
```

```
        self.experience = experience
```

```
        self.rating = rating
```

```
class Customer(Base, ReprMixin):
```

```
    __tablename__ = 'customer'
```

```
    id = Column(Integer, primary_key=True)
```

```
    name = Column(String)
```

```
    is_vip = Column(Boolean)
```

```
    reservation = relationship('Reservation')
```

```
    def __init__(self, id=None, name=None, is_vip=None):
```

```
        self.id = id
```

```
        self.name = name
```

```
        self.is_vip = is_vip
```

```
class Reservation(Base, ReprMixin):
```

```
    __tablename__ = 'reservation'
```

```
    id = Column(Integer, primary_key=True)
```

```
    master_id = Column(Integer, ForeignKey('master.id'))
```

```
    customer_id = Column(Integer, ForeignKey('customer.id'))
```

```
    datetime = Column(DateTime)
```

```
    procedures = relationship('Procedure')
```

```
    def __init__(self, id=None, master_id=None, customer_id=None, datetime=None):
```

```
        self.id = id
```

```
        self.master_id = master_id
```

```
        self.customer_id = customer_id
```

```
        self.datetime = datetime
```

```

class Procedure(Base, ReprMixin):
    __tablename__ = 'procedure'

    id = Column(Integer, primary_key=True)
    reservation_id = Column(Integer, ForeignKey('reservation.id'))
    price = Column(Integer)
    work_type = Column(String)

    def __init__(self, id=None, reservation_id=None, price=None, work_type=None):
        self.id = id
        self.reservation_id = reservation_id
        self.price = price
        self.work_type = work_type

```

```
Base.metadata.create_all(db)
```

```

TABLES = {
    'master': ('id', 'name', 'experience', 'rating'),
    'customer': ('id', 'name', 'is_vip'),
    'reservation': ('id', 'master_id', 'customer_id', 'datetime'),
    'procedure': ('id', 'reservation_id', 'price', 'work_type')
}

```

```

CLASSES = {
    'procedure': Procedure, 'customer': Customer,
    'reservation': Reservation, 'master': Master
}

```

```

class Model:
    def __init__(self):
        self.session = sessionmaker(db)()

    def create_tables(self):
        with open('scripts/create.sql') as file:
            command = file.read()
            self.session.execute(command)
            self.session.commit()

    def get(self, table_name, **filter_by):
        objects_class = CLASSES[table_name]
        objects = self.session.query(objects_class)
        for key, item in filter_by.items():
            objects = objects.filter(getattr(objects_class, key) == item)

        return list(objects)

    def insert(self, table_name, **new_values):
        object_class = CLASSES[table_name]
        obj = object_class(**new_values)
        self.session.add(obj)

```



```

def update(self, table_name, condition, **new_values):
    if not new_values:
        raise Exception('Не вказані поля, які треба оновити')

    column, value = condition
    object_class = CLASSES[table_name]
    filter_attr = getattr(object_class, column)
    objects = self.session.query(object_class).filter(filter_attr == value)

    for obj in objects:
        for key, item in new_values.items():
            setattr(obj, key, item)

def delete(self, table_name, **filter_by):
    if not filter_by:
        raise Exception('Не вказані умови для рядків, які треба видалити')

    objects_class = CLASSES[table_name]
    objects = self.session.query(objects_class)
    for key, item in filter_by.items():
        objects = objects.filter(getattr(objects_class, key) == item)

    objects.delete()

def commit(self):
    self.session.commit()

def create_random_masters(self):
    with open('scripts/random.sql') as file:
        sql = file.read()
        self.session.execute(sql)

```

view.py

```

class View:
    def __init__(self):
        self.SEPARATOR_WIDTH = 30

    def print_entities(self, table_name, data):
        separator_line = '-' * self.SEPARATOR_WIDTH

        print(f'Результат для таблиці `{table_name}`', end='\n\n')
        for entity in data:
            print(entity)

```

Індекси

Gin індекс:

```
ALTER TABLE master ADD COLUMN document tsvector;  
UPDATE master SET document = to_tsvector(name) WHERE true;  
CREATE INDEX gin_index ON master using gin(document);
```

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

```
7  EXPLAIN SELECT * FROM master;
```

Output		Result 3 x
1 row		
QUERY PLAN		
1	Seq Scan on master (cost=0.00..258.84 rows=6084 width=61)	

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

```
8  EXPLAIN SELECT * FROM master WHERE document @@ to_tsquery('foobar');
```

Output		Result 5 x
4 rows		
QUERY PLAN		
1	Bitmap Heap Scan on master (cost=12.25..16.52 rows=1 width=61)	
2	Recheck Cond: (document @@ to_tsquery('foobar'::text))	
3	-> Bitmap Index Scan on gin_index (cost=0.00..12.25 rows=1 width=0)	
4	Index Cond: (document @@ to_tsquery('foobar'::text))	

Btree індекс:

```
CREATE INDEX btree_index ON master using btree(id);
```

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

```
4  EXPLAIN SELECT * FROM master;
```

Output	
Result 1 ×	
1 row	
QUERY PLAN	
1	Seq Scan on master (cost=0.00..298.06 rows=10006 width=61)

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

```
5  EXPLAIN SELECT * FROM master WHERE id = 10000;
```

Output	
Result 1 ×	
2 rows	
QUERY PLAN	
1	Index Scan using btree_index on master (cost=0.29..8.30 rows=1 width=61)
2	Index Cond: (id = 10000)

Тригер

AFTER INSERT





Якщо для до броні додається нова процедура, то всі процедури в цій броні отримують знижку 10%.

Код:

```
CREATE OR REPLACE FUNCTION after_insert_procedure()  
RETURNS TRIGGER LANGUAGE PLPGSQL AS $$  
DECLARE  
    other_procedures CURSOR IS SELECT * FROM procedure WHERE reservation_id =  
NEW.reservation_id;  
BEGIN  
    FOR p IN other_procedures LOOP  
        UPDATE procedure SET price = price * 0.9 WHERE id = p.id;  
    END LOOP;  
    RETURN NEW;  
END;  
$$;
```

Приклади результатів:





Процедури для броні з id=1;

	 id	 reservation_id	 price	 work_type
1	10	1	150	skin cryotherapy
2	1	1	100	makeup cosmetic

Додамо нову процедуру:

```
INSERT INTO procedure(reservation_id, price, work_type)  
VALUES (1, 100, 'something else');
```

Процедури після цього:

	 id	 reservation_id	 price	 work_type
1	1	1	100	makeup cosmetic
2	10	1	135	skin cryotherapy
3	18	1	90	something else

AFTER UPDATE

Якщо оновлюється бронь і їй у відповідність встановлюється ВІП-клієнт, то майстер, до якого записаний цей клієнт отримує +10% до рейтингу.

Код:

```
CREATE OR REPLACE FUNCTION after_update_reservation()  
RETURNS TRIGGER LANGUAGE PLPGSQL AS $$  
DECLARE  
BEGIN  
    IF true IN (SELECT is_vip FROM customer WHERE id = NEW.customer_id) THEN  
        UPDATE master SET rating = rating * 1.1 WHERE id = NEW.master_id;  
    END IF;  
    RETURN NEW;  
END;  
$;
```

Приклади результатів:

```
INSERT INTO master(id, name, experience, rating) VALUES (1337, 'Sample Master', 15, 4);  
INSERT INTO customer(id, name, is_vip) VALUES (42, 'Simple Customer', false);  
INSERT INTO customer(id, name, is_vip) VALUES (1337, 'Vip Customer', true);  
INSERT INTO reservation(master_id, customer_id) VALUES (1337, 42);  
UPDATE reservation SET customer_id = 1337 WHERE master_id = 1337;  
SELECT * FROM master WHERE name = 'Sample Master';
```

	id	name	experience	rating
1	1337	Sample Master	15	4.4

Початковий рейтинг майстра був 4. В його броні змінився клієнт (новий клієнт ВІП). Майстер отримав збільшення рейтингу до 4.4.

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

Створює нового Клієнта і перевіряє його наявність для свого запиту SELECT

```
Введіть нові значення:  
(у форматі <attribute>=<value>)  
Допустимі колонки: (id/name/is_vip)  
  
name=Read Committed Customer
```

Результат для таблиці `customer`

```
<Customer>{'id': 1, 'name': 'William Red', 'is_vip': False}  
<Customer>{'id': 2, 'name': 'Peter Snow', 'is_vip': False}  
<Customer>{'id': 3, 'name': 'Johnny Good', 'is_vip': False}  
<Customer>{'id': 4, 'name': 'Peter Rainy', 'is_vip': True}  
<Customer>{'id': 5, 'name': 'Crusty Crab', 'is_vip': True}  
<Customer>{'id': 1337, 'name': 'Sample Customer', 'is_vip': True}  
<Customer>{'id': 42, 'name': 'Simple Customer', 'is_vip': False}  
<Customer>{'id': 7, 'name': 'Read Committed Customer', 'is_vip': None}
```

Транзакція №2

Отримує список всіх Клієнтів.

Результат для таблиці `customer`

```
<Customer>{'name': 'William Red', 'is_vip': False, 'id': 1}  
<Customer>{'name': 'Peter Snow', 'is_vip': False, 'id': 2}  
<Customer>{'name': 'Johnny Good', 'is_vip': False, 'id': 3}  
<Customer>{'name': 'Peter Rainy', 'is_vip': True, 'id': 4}  
<Customer>{'name': 'Crusty Crab', 'is_vip': True, 'id': 5}  
<Customer>{'name': 'Sample Customer', 'is_vip': True, 'id': 1337}  
<Customer>{'name': 'Simple Customer', 'is_vip': False, 'id': 42}
```

Транзакція №1

Робить коміт.

```
Зміни успішно збережені
```

Транзакція №2

Отримує список всіх Клієнтів.

Результат для таблиці `customer`

```
<Customer>{'id': 1, 'name': 'William Red', 'is_vip': False}
<Customer>{'id': 2, 'name': 'Peter Snow', 'is_vip': False}
<Customer>{'id': 3, 'name': 'Johnny Good', 'is_vip': False}
<Customer>{'id': 4, 'name': 'Peter Rainy', 'is_vip': True}
<Customer>{'id': 5, 'name': 'Crusty Crab', 'is_vip': True}
<Customer>{'id': 1337, 'name': 'Sample Customer', 'is_vip': True}
<Customer>{'id': 42, 'name': 'Simple Customer', 'is_vip': False}
<Customer>{'id': 7, 'name': 'Read Committed Customer', 'is_vip': None}
```

Як бачимо Транзакція №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 Клієнтів.

Результат для таблиці `customer`

```
<Customer>{'is_vip': False, 'id': 1, 'name': 'William Red'})
<Customer>{'is_vip': False, 'id': 2, 'name': 'Peter Snow'})
<Customer>{'is_vip': False, 'id': 3, 'name': 'Johnny Good'})
<Customer>{'is_vip': True, 'id': 4, 'name': 'Peter Rainy'})
<Customer>{'is_vip': True, 'id': 5, 'name': 'Crusty Crab'})
<Customer>{'is_vip': True, 'id': 1337, 'name': 'Sample Customer'})
<Customer>{'is_vip': False, 'id': 42, 'name': 'Simple Customer'})
<Customer>{'is_vip': None, 'id': 7, 'name': 'Read Committed Customer'})
```

Транзакція №2

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

```
Введіть нові значення:
(у форматі <attribute>=<value>)
Допустимі колонки: (id/name/is_vip)

name=REPEATABLE READ customer
```

Зміни успішно збережені

Результат для таблиці `customer`

```
<Customer>{'is_vip': False, 'name': 'William Red', 'id': 1})
<Customer>{'is_vip': False, 'name': 'Peter Snow', 'id': 2})
<Customer>{'is_vip': False, 'name': 'Johnny Good', 'id': 3})
<Customer>{'is_vip': True, 'name': 'Peter Rainy', 'id': 4})
<Customer>{'is_vip': True, 'name': 'Crusty Crab', 'id': 5})
<Customer>{'is_vip': True, 'name': 'Sample Customer', 'id': 1337})
<Customer>{'is_vip': False, 'name': 'Simple Customer', 'id': 42})
<Customer>{'is_vip': None, 'name': 'Read Committed Customer', 'id': 7})
<Customer>{'is_vip': None, 'name': 'REPEATABLE READ customer', 'id': 8})
```

Транзакція №1

Заново отримує список усіх клієнтів.

Результат для таблиці `customer`

```
<Customer>{'is_vip': False, 'id': 1, 'name': 'William Red'})
<Customer>{'is_vip': False, 'id': 2, 'name': 'Peter Snow'})
<Customer>{'is_vip': False, 'id': 3, 'name': 'Johnny Good'})
<Customer>{'is_vip': True, 'id': 4, 'name': 'Peter Rainy'})
<Customer>{'is_vip': True, 'id': 5, 'name': 'Crusty Crab'})
<Customer>{'is_vip': True, 'id': 1337, 'name': 'Sample Customer'})
<Customer>{'is_vip': False, 'id': 42, 'name': 'Simple Customer'})
<Customer>{'is_vip': None, 'id': 7, 'name': 'Read Committed Customer'})
```

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 REPEATABLE 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 нову Процедуру “test manicure”.

```
Введіть нові значення:
(у форматі <attribute>=<value>)
Допустимі колонки: (id/reservation_id/price/work_type)

reservation_id=1
work_type=test manicure
```


Транзакція №2

Змінює ціну для всіх процедур “test manicure”.

```
Який рядок треба змінити?:  
(у форматі <attribute>=<value>)  
Допустимі колонки: (id/reservation_id/price/work_type)  
  
work_type=test manicure  
Введіть нові значення:  
(у форматі <attribute>=<value>)  
Допустимі колонки: (id/reservation_id/price/work_type)  
  
price=7777
```

Транзакція №1

Намагається закомітити зміни

```
Зміни успішно збережені
```

Транзакція №2

Намагається закомітити зміни

```
(psycopg2.errors.SerializationFailure) could not serialize access due to concurrent update  
[SQL: UPDATE procedure SET price=%(price)s WHERE procedure.id = %(procedure_id)s]  
[parameters: {'price': '1548', 'procedure_id': 19}]  
(Background on this error at: http://sqlalche.me/e/e3q8)
```

Як бачимо Транзакція №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”.