**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

Городченко Анна

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.

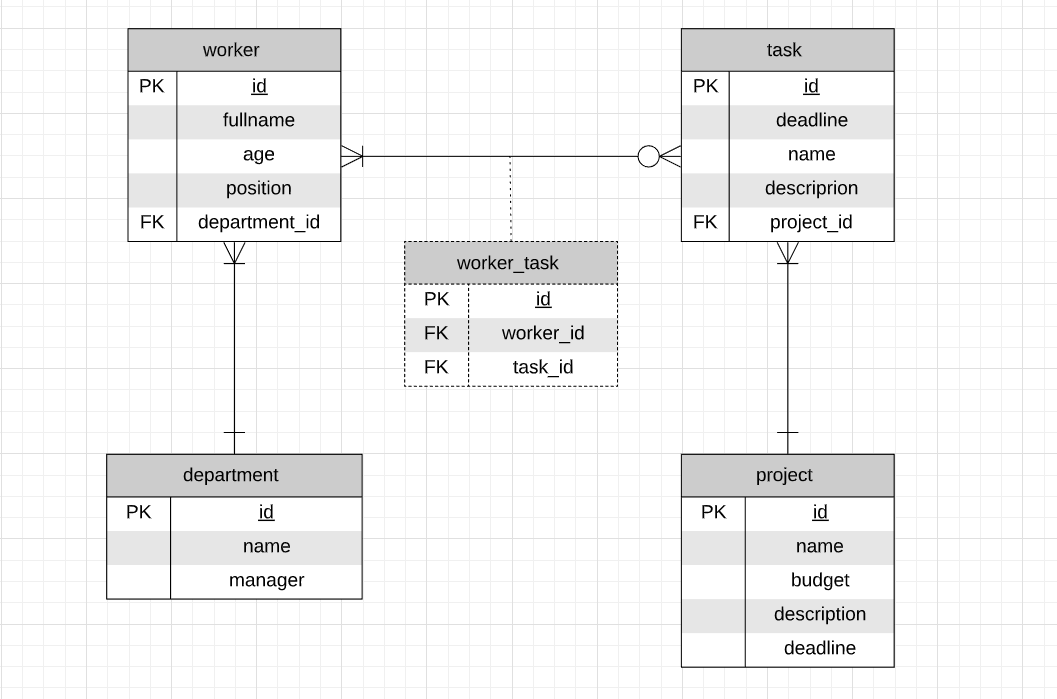
4.Навести приклади та проаналізувати рівні ізоляції транзакцій уPostgreSQL.

Порядок виконання роботи

В ході роботи розроблено:

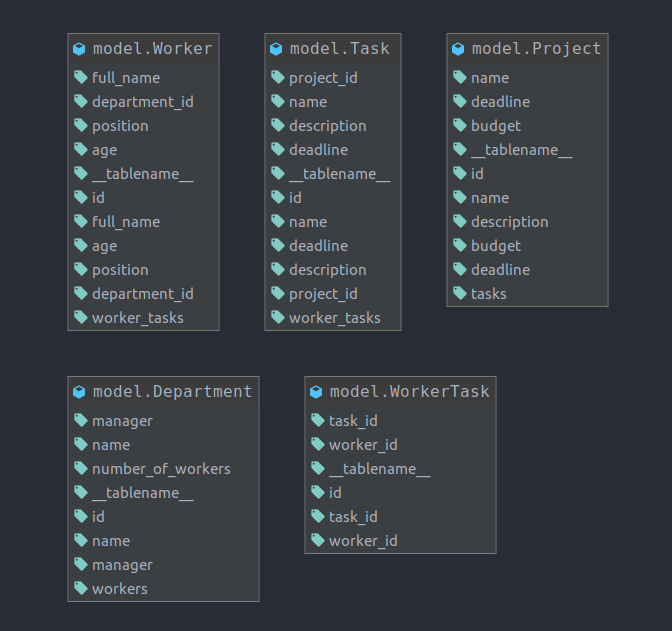
1. Логічну модель БД та Діаграму класів;
2. Функціонал програмного додатку;
3. ОО програмний додаток роботи з БД " … ". Для взаємодії з БД використано ORM модуль SQLAlchemy.

Логічна модель бази даних наведена на Рис 1.



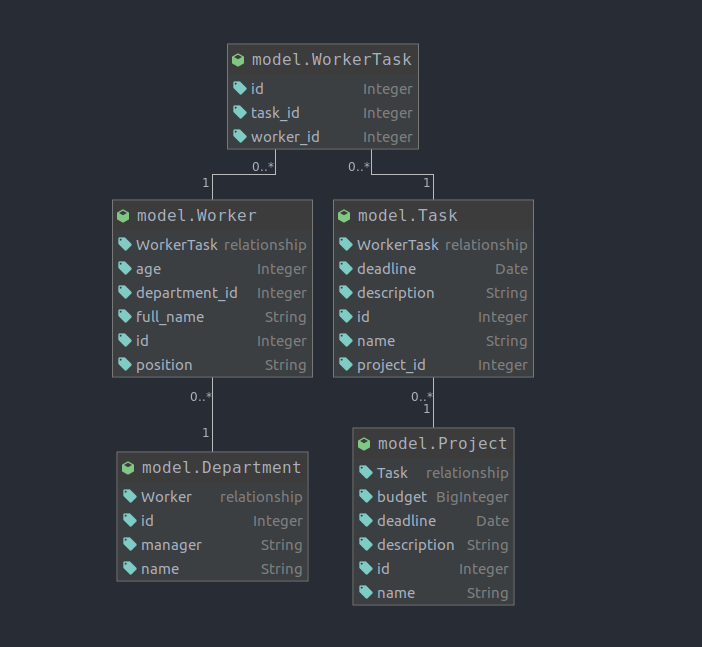
*Рис 1. Логічна модель бази даних*

Сутнісні класи програми наведені на Рис 2.



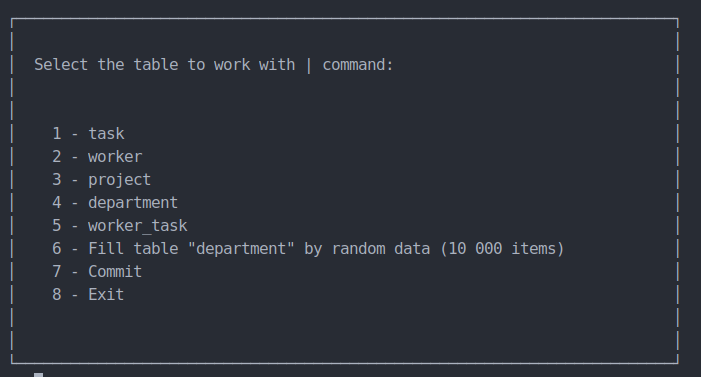
*Рис 2. Фрагмент UML діаграми сутнісних класів*

Зв’язки між сутнісними класами, згенеровані за допомогою SqlAlchemy, наведені на Рис 3.

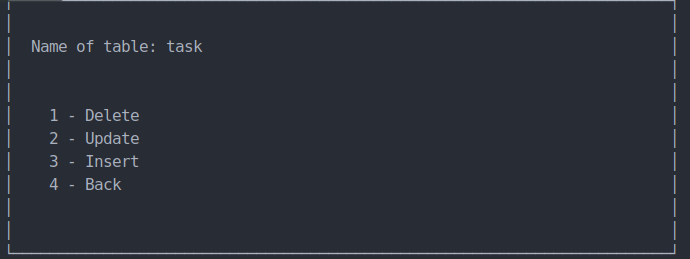


*Рис 3. Зв’язки між сутнісними класами*

Меню програми наведене на схемі нижче.



Пункт 1-5:



Пункт 6:



Пункт 7:



Пункт 8 (вихід з програми):



Лістинг програми

**main.py**

*from* controller *import* Controller

*if* \_\_name\_\_ == '\_\_main\_\_':

Controller().show\_init\_menu()

**controller.py**

***from* consolemenu *import* SelectionMenu**

*from* model *import* Model

TABLES\_NAMES = ['task', 'worker', 'project', 'department', 'worker\_task']

TABLES = {

'task': ['id', 'deadline', 'name', 'description', 'project\_id'],

'worker': ['id', 'full\_name', 'age', 'position', 'department\_id'],

'department': ['id', 'name', 'number\_of\_workers', 'manager'],

'project': ['id', 'name', 'budget', 'deadline'],

'worker\_task': ['id', 'worker\_id', 'task\_id']

}

*def* get\_input(msg, table\_name=''):

*print*(msg)

*if* table\_name:

*print*(' | '.join(TABLES[table\_name]), end='\n\n')

*return input*()

*def* get\_insert\_input(msg, table\_name):

*print*(msg)

*print*(' | '.join(TABLES[table\_name]), end='\n\n')

*return input*(), *input*()

*def* press\_enter():

*input*()

*class* Controller:

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

*self*.model = Model()

*def* show\_init\_menu(*self*, msg=''):

selection\_menu = SelectionMenu(

TABLES\_NAMES + ['Fill table "department" by random data (10 000 items)', 'Commit'],

title='Select the table to work with | command:', subtitle=msg)

selection\_menu.show()

index = selection\_menu.selected\_option

*if* index < *len*(TABLES\_NAMES):

table\_name = TABLES\_NAMES[index]

*self*.show\_entity\_menu(table\_name)

*elif* index == *len*(TABLES\_NAMES):

*self*.fill\_by\_random()

*elif* index == *len*(TABLES\_NAMES) + 1:

*self*.model.commit()

*self*.show\_init\_menu(msg='Commit success')

*else*:

*print*('Bye, have a beautiful day!')

*def* show\_entity\_menu(*self*, table\_name, msg=''):

options = ['Delete', 'Update', 'Insert']

functions = [*self*.delete, *self*.update, *self*.insert]

selection\_menu = SelectionMenu(options, f'Name of table: {table\_name}',

exit\_option\_text='Back', subtitle=msg)

selection\_menu.show()

*try*:

function = functions[selection\_menu.selected\_option]

function(table\_name)

*except IndexError*:

*self*.show\_init\_menu()

*def* insert(*self*, table\_name):

*try*:

columns, values = get\_insert\_input(

f"INSERT {table\_name}\nEnter columns divided with commas, then do the same for values in format: [value1, value2, ...]",

table\_name)

*self*.model.insert(table\_name, columns, values)

*self*.show\_entity\_menu(table\_name, 'Insert is successful!')

*except Exception as* err:

*self*.show\_entity\_menu(table\_name, *str*(err))

*def* delete(*self*, table\_name):

*try*:

condition = get\_input(

f'DELETE {table\_name}\n Enter condition (SQL):', table\_name)

*self*.model.delete(table\_name, condition)

*self*.show\_entity\_menu(table\_name, 'Delete is successful')

*except Exception as* err:

*self*.show\_entity\_menu(table\_name, *str*(err))

*def* update(*self*, table\_name):

*try*:

condition = get\_input(

f'UPDATE {table\_name}\nEnter condition (SQL):', table\_name)

statement = get\_input(

"Enter SQL statement in format [<key>=<value>]", table\_name)

*self*.model.update(table\_name, condition, statement)

*self*.show\_entity\_menu(table\_name, 'Update is successful')

*except Exception as* err:

*self*.show\_entity\_menu(table\_name, *str*(err))

*def* fill\_by\_random(*self*):

*try*:

*self*.model.fill\_department\_by\_random\_data()

*self*.show\_init\_menu('10 000 departments are generated successfully')

*except Exception as* err:

*self*.show\_init\_menu(*str*(err))

**model.py**

***from* sqlalchemy *import* create\_engine, Column, String, Integer, BigInteger, Date, ForeignKey**

*from* sqlalchemy.ext.declarative *import* declarative\_base

*from* sqlalchemy.orm *import* relationship, sessionmaker

engine = create\_engine('postgres://goroanya:goroanya99@localhost:5432/db\_lab3')

Base = declarative\_base()

*class* Department(Base):

\_\_tablename\_\_ = 'department'

id = Column(Integer, primary\_key=*True*)

name = Column(String)

manager = Column(String)

workers = relationship('Worker')

*def \_\_init\_\_*(*self*, name=*None*, number\_of\_workers=*None*, manager=*None*):

*self*.name = name

*self*.number\_of\_workers = number\_of\_workers

*self*.manager = manager

*class* Project(Base):

\_\_tablename\_\_ = 'project'

id = Column(Integer, primary\_key=*True*)

name = Column(String)

description = Column(String)

budget = Column(BigInteger)

deadline = Column(Date)

tasks = relationship('Task')

*def \_\_init\_\_*(*self*, name=*None*, budget=*None*, deadline=*None*):

*self*.name = name

*self*.budget = budget

*self*.deadline = deadline

*class* Worker(Base):

\_\_tablename\_\_ = 'worker'

id = Column(Integer, primary\_key=*True*)

full\_name = Column(String)

age = Column(Integer)

position = Column(String)

department\_id = Column(Integer, ForeignKey('department.id'))

worker\_tasks = relationship('WorkerTask')

*def \_\_init\_\_*(*self*, full\_name=*None*, age=*None*, position=*None*, department\_id=*None*):

*self*.full\_name = full\_name

*self*.age = age

*self*.position = position

*self*.department\_id = department\_id

*class* Task(Base):

\_\_tablename\_\_ = 'task'

id = Column(Integer, primary\_key=*True*)

name = Column(String)

deadline = Column(Date)

description = Column(String)

project\_id = Column(Integer, ForeignKey('project.id'))

worker\_tasks = relationship('WorkerTask')

*def \_\_init\_\_*(*self*, name=*None*, deadline=*None*, description=*None*, project\_id=*None*):

*self*.name = name

*self*.deadline = deadline

*self*.description = description

*self*.project\_id = project\_id

*class* WorkerTask(Base):

\_\_tablename\_\_ = 'worker\_task'

id = Column(Integer, primary\_key=*True*)

task\_id = Column(Integer, ForeignKey('task.id'))

worker\_id = Column(Integer, ForeignKey('worker.id'))

*def \_\_init\_\_*(*self*, task\_id=*None*, worker\_id=*None*):

*self*.task\_id = task\_id

*self*.worker\_id = worker\_id

session = sessionmaker(engine)()

Base.metadata.create\_all(engine)

TABLES = {'worker': Worker, 'task': Task, 'department': Department, 'project': Project, 'worker\_task': WorkerTask}

*class* Model:

*def* pairs\_from\_str(*self*, string):

lines = string.split(',')

pairs = {}

*for* line *in* lines:

key, value = line.split('=')

pairs[key.strip()] = value.strip()

*return* pairs

*def* filter\_by\_pairs(*self*, objects, pairs, cls):

*for* key, value *in* pairs.items():

field = *getattr*(cls, key)

objects = objects.filter(field == value)

*return* objects

*def* insert(*self*, tname, columns, values):

columns = [c.strip() *for* c *in* columns.split(',')]

values = [v.strip() *for* v *in* values.split(',')]

pairs = *dict*(*zip*(columns, values))

object\_class = TABLES[tname]

obj = object\_class(\*\*pairs)

session.add(obj)

*def* commit(*self*):

session.commit()

*def* delete(*self*, tname, condition):

pairs = *self*.pairs\_from\_str(condition)

object\_class = TABLES[tname]

objects = session.query(object\_class)

objects = *self*.filter\_by\_pairs(objects, pairs, object\_class)

objects.delete()

*def* update(*self*, tname, condition, statement):

pairs = *self*.pairs\_from\_str(condition)

new\_values = *self*.pairs\_from\_str(statement)

object\_class = TABLES[tname]

objects = session.query(object\_class)

objects = *self*.filter\_by\_pairs(objects, pairs, object\_class)

*for* obj *in* objects:

*for* field\_name, value *in* new\_values.items():

*setattr*(obj, field\_name, value)

*def* fill\_department\_by\_random\_data():

sql = """

CREATE OR REPLACE FUNCTION randomDepartments()

RETURNS void AS $$

DECLARE

step integer := 0;

BEGIN

LOOP EXIT WHEN step > 10000;

INSERT INTO department (name, manager)

VALUES (

substring(md5(random()::text), 1, 10),

substring(md5(random()::text), 1, 15)

);

step := step + 1;

END LOOP ;

END;

$$ LANGUAGE PLPGSQL;

SELECT randomDepartments();

"""

*try*:

session.execute(sql)

*finally*:

session.commit()

**Індекси**

**GIN індекс:**

*ALTER TABLE* department

*ADD COLUMN* ts\_vector *tsvector*;

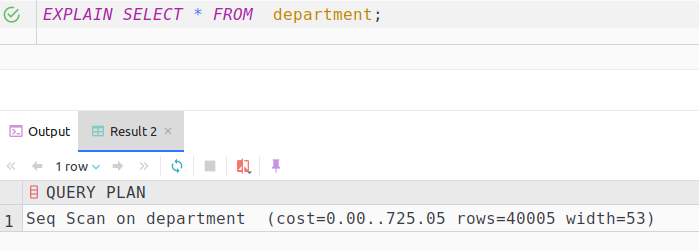
*UPDATE* department

*SET* ts\_vector = *to\_tsvector*(*name*)

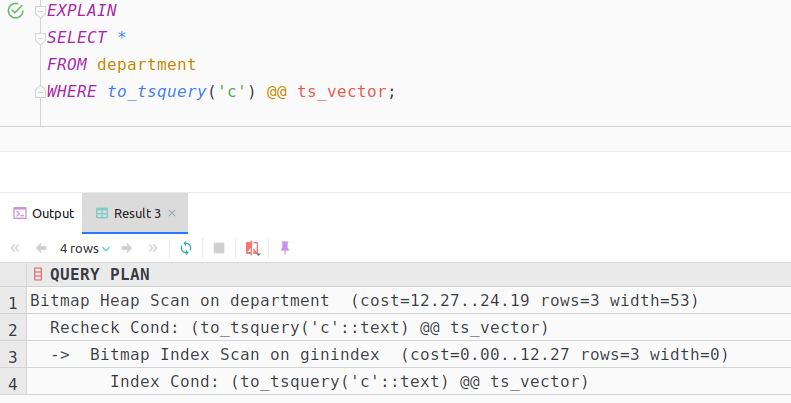
*WHERE true*;

*CREATE INDEX* ginIndex *ON* department *USING* gin (ts\_vector);

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



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



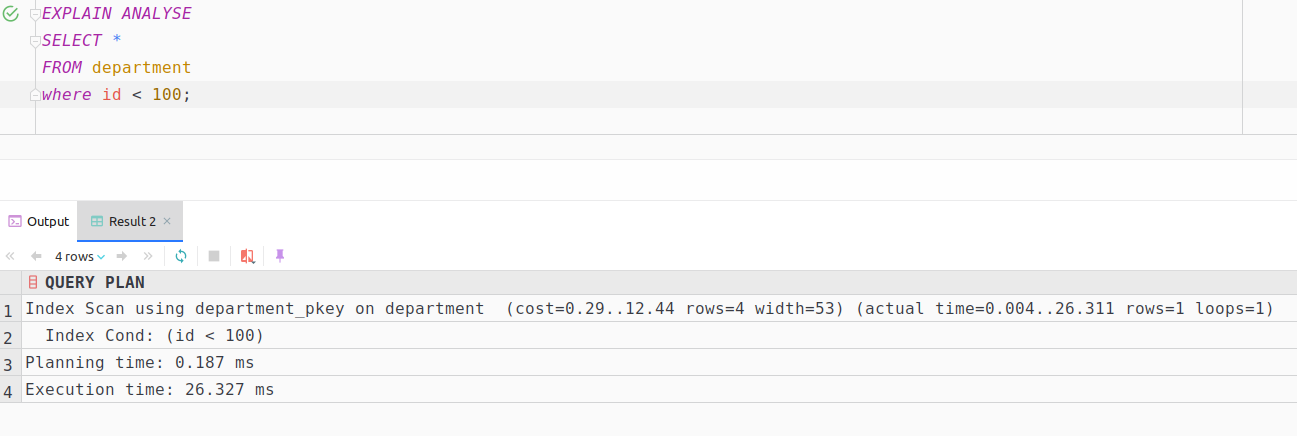
**B-tree індекс:**

*CREATE INDEX* treeIndex *ON* department *using* btree (id);

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



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



**Тригер**

**AFTER INSERT**

Якщо працівнику більше 30 років і він займає посаду “менеджер”, то до імені менеджера відділу додається вставка “old”, a якщо менше 30, то “young”.

Код:

*CREATE OR REPLACE FUNCTION afterInsertWorker*()

*returns trigger*

*language* plpgsql

*AS* $$

*BEGIN*

*IF NEW*.age > 30 *AND NEW*.position = 'manager' *THEN*

*UPDATE* department *SET* manager= *NEW*.full\_name || ' (old)' *WHERE* id= *NEW*.department\_id;

*ELSE*

*UPDATE* department *SET* manager= *NEW*.full\_name || ' (young)' *WHERE* id = *NEW*.department\_id;

*END IF*;

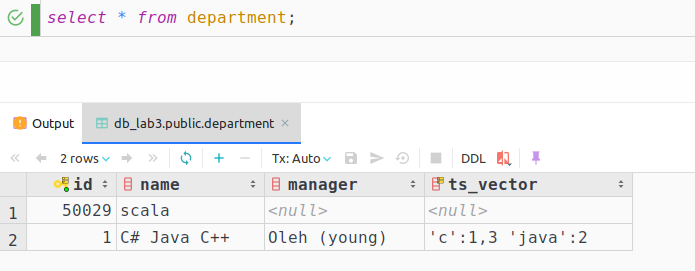
*return NEW*;

*END*;

$$;

Приклади результатів:

Відділи до вставки працівник-менеджера:

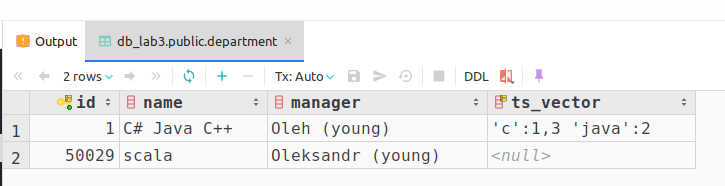


Додамо нового працівника процедуру:

*INSERT INTO* worker( full\_name, age, *position*, department\_id)

*VALUES* ('Oleksandr', 29, 'manager', 50029);

Відділи після цього:



**AFTER UPDATE**

Якщо оновлюється назва відділу, то до посади працівника додається нова назва відділу, в якому він працює.

Код:

*CREATE OR REPLACE FUNCTION afterUpdateDepartment*()

*returns trigger*

*language* plpgsql

*AS* $$

*DECLARE*

workers *cursor is select \* from* worker *where* department\_id = *NEW*.id;

*BEGIN*

*FOR* worker\_ *IN* workers *LOOP*

*UPDATE* worker

*SET position* = *position* || ' at ' || *NEW*.name || ','

*WHERE* id = worker\_.id;

*end loop*;

*return NEW*;

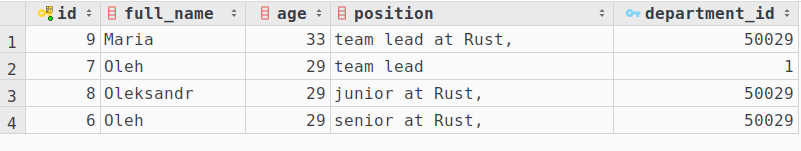
*END*;

$$;

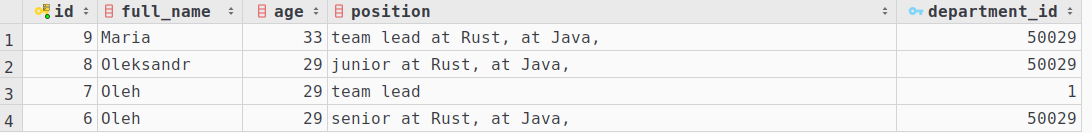
Приклади результатів:

*UPDATE* department *SET name*='Java' *WHERE* id=50029;

До оновлення :



Після оновлення :



**Дослідження рівнів ізоляції**

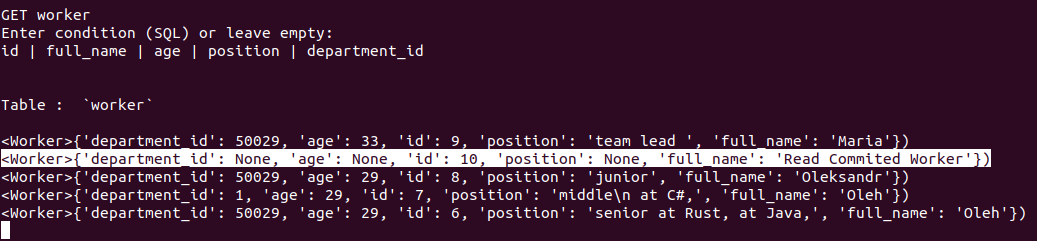
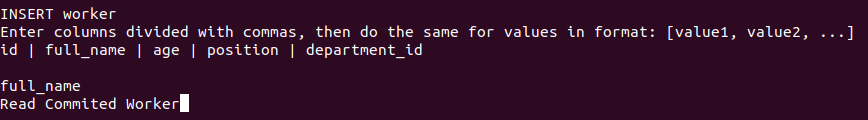
Для перевірки аномалій буде використовуватися розроблений програмний додаток, запущений у двох екземплярах паралельно.

1. **READ COMMITTED**

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

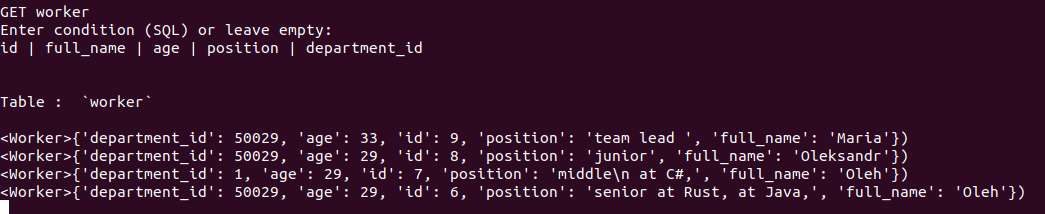
Транзакція №1

Створює нового Працівника і перевіряє його наявність для свого запиту SELECT



Транзакція №2

Отримує список всіх Працівників



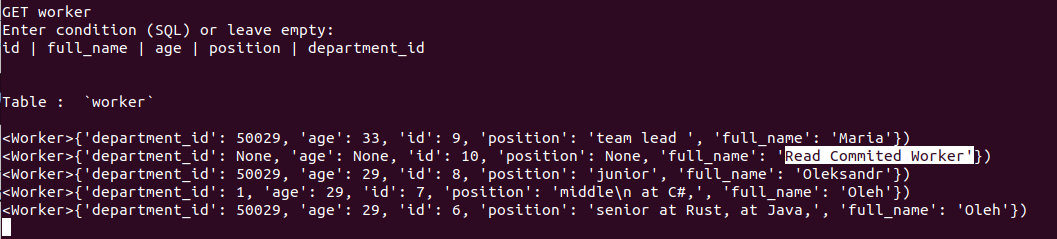
Транзакція №1

Робить коміт.



Транзакція №2

Отримує список всіх Працівників.



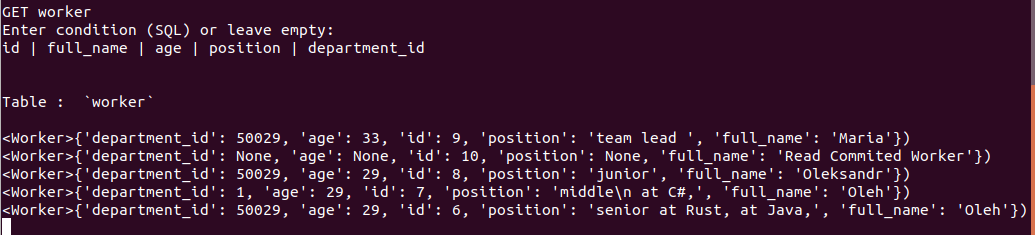
Як бачимо Транзація №2 не бачила зміни, внесені до таблиці Транзакцією №1, до ти пір поки остання не закомітила свої зміни. Отже, було доведено, що рівень ізоляції READ COMMITTED захищає від аномалії “брудного читання”.

2. **REPEATABLE READ**

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

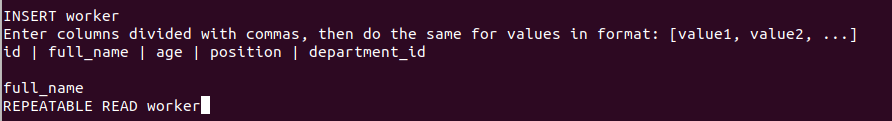
Транзакція №1

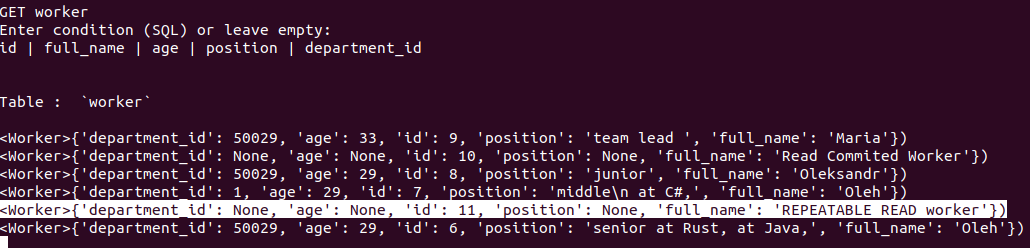
Отримує список усіх Працівників.



Транзакція №2

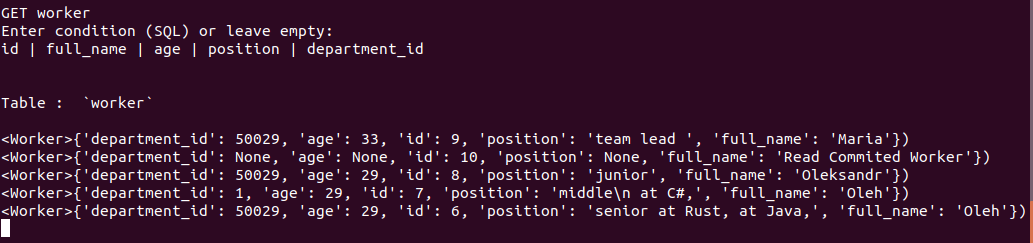
Створює нового Працівника, комітить зміни і перевіряє його наявність в списку усіх працівників.





Транзакція №1

Заново отримує список усіх клієнтів.



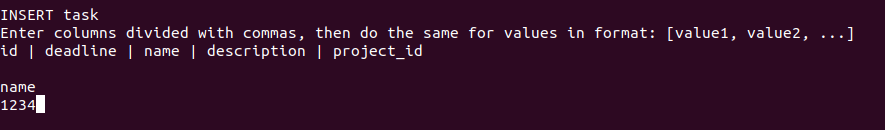
Як бачимо, Транзакція №1 так і не побачила закомічені зміни Транзакції №2, так як перша з них зчитувала дані з таблиці ще до коміту. Отже, було доведено, що рівень ізоляції REPEATBLE READ захищає від аномалії “nonrepeatable read”.

3. **SERIALIZABLE**

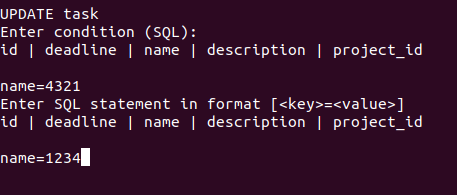
Перевіримо наявність аномалії “serialization anomaly”, коли дві паралельні транзакції хочуть закомітити свої результати і при цьому є різниця, в якому порядку виконувати команди, виконані кожною з транзакцій.

Транзакція №1

Створює нове завдання з ім’ям 1234.

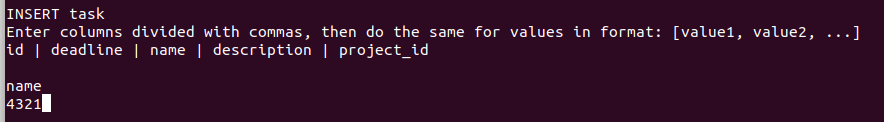


Оновлює всі завдання з ім’ям 4321.

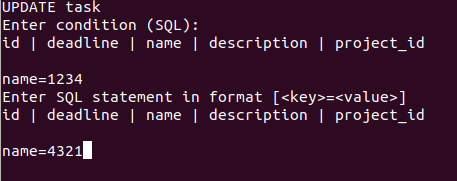


Транзакція №2

Створює нове завдання з ім’ям 4321.



Оновлює всі завдання з ім’ям 1234.



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”.