



МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

Факультет прикладної математики
Кафедра програмного забезпечення комп’ютерних систем

Лабораторна робота № 3
з дисципліни “ Бази даних ”
тема “ **Засоби оптимізації роботи СУБД PostgreSQL** ”
Варіант №27

Виконав
студент II курсу
групи КП-92

Яковлєв Денис Сергійович
(прізвище, ім'я, по батькові)

Перевірів
“ ____ ” “ _____ ” 20__ р.
викладач

Петрашенко А.В.
(прізвище, ім'я, по батькові)

Київ 2020

Метою роботи є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

Завдання роботи полягає у наступному:

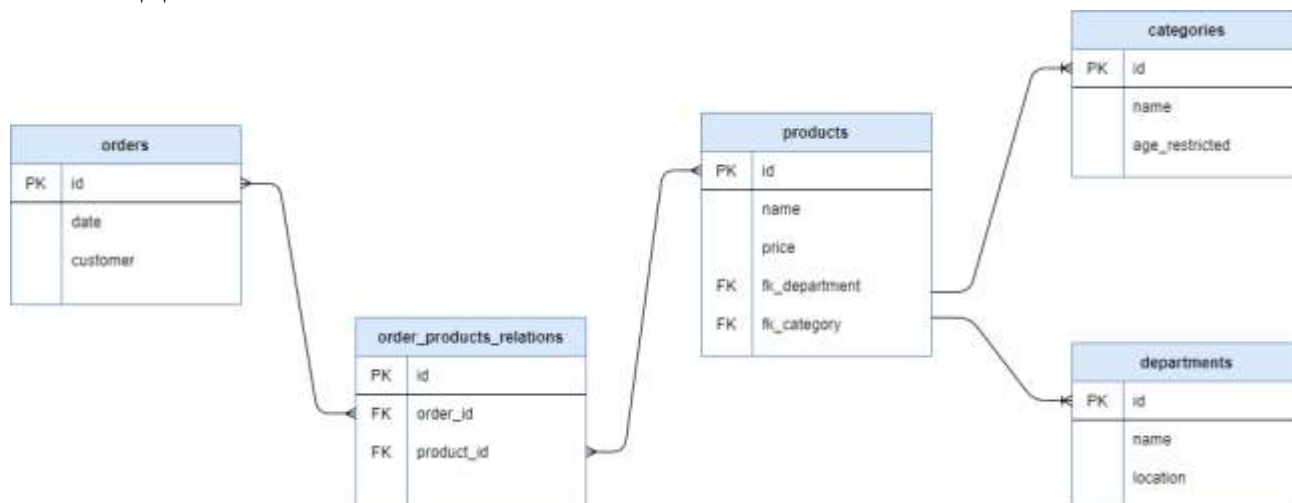
1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL.
3. Розробити тригер бази даних PostgreSQL.

27	<i>GIN, BRIN</i>	<i>after update, insert</i>
----	------------------	-----------------------------

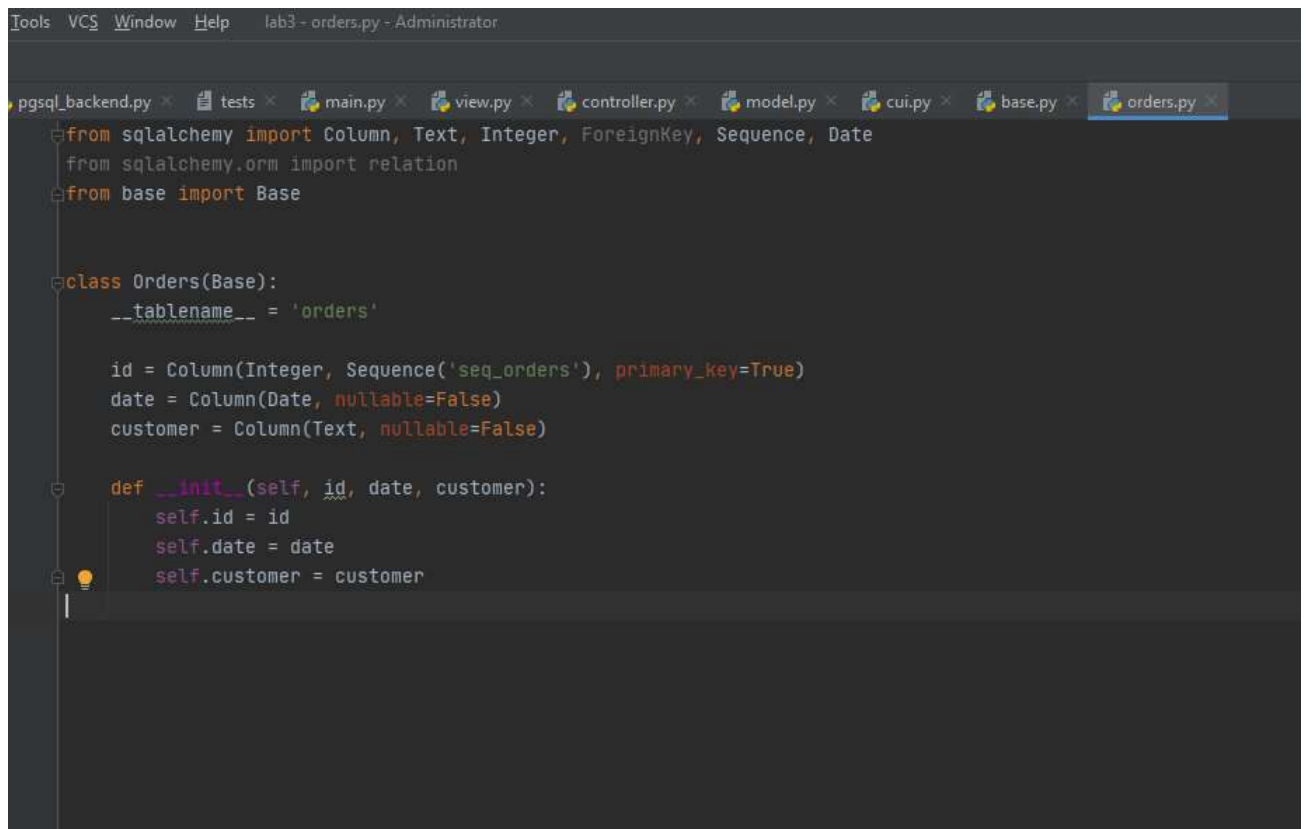
Завдання 1

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

Схема БД:



ORM класи:



```
Tools  VCS  Window  Help  lab3 - orders.py - Administrator

pgsql_backend.py × tests × main.py × view.py × controller.py × model.py × cui.py × base.py × orders.py ×

from sqlalchemy import Column, Text, Integer, ForeignKey, Sequence, Date
from sqlalchemy.orm import relation
from base import Base

class Orders(Base):
    __tablename__ = 'orders'

    id = Column(Integer, Sequence('seq_orders'), primary_key=True)
    date = Column(Date, nullable=False)
    customer = Column(Text, nullable=False)

    def __init__(self, id, date, customer):
        self.id = id
        self.date = date
        self.customer = customer
```

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy_repr import PrettyRepresentableBase

Base = declarative_base(cls=PrettyRepresentableBase)
```

```
from sqlalchemy import Column, Text, Integer, ForeignKey, Sequence, Date
from sqlalchemy.orm import relation
from base import Base
from categories import Categories
from departments import Departments

class Products(Base):
    __tablename__ = 'products'

    id = Column(Integer, Sequence('seq_products'), primary_key=True)
    name = Column(Text, nullable=False)
    price = Column(Integer, nullable=False)
    fk_department = Column(Integer, ForeignKey('departments.id'))
    fk_category = Column(Integer, ForeignKey('categories.id'))
    department = relation(Departments, backref='products')
    category = relation(Categories, backref='products')

    def __init__(self, id, name, price, fk_department, fk_category):
        self.id = id
        self.name = name
        self.price = price
        self.fk_department = fk_department
        self.fk_category = fk_category
```

```
lab3 - departments.py - Administrator

from sqlalchemy import Column, Text, Integer, ForeignKey, Sequence, Date, Boolean
from sqlalchemy.orm import relation
from base import Base

class Departments(Base):
    __tablename__ = 'departments'

    id = Column(Integer, Sequence('seq_departments'), primary_key=True)
    name = Column(Text, nullable=False)
    location = Column(Text, nullable=False)

    def __init__(self, id, name, location):
        self.id = id
        self.name = name
        self.location = location
```

```
lab3 - categories.py - Administrator

from sqlalchemy import Column, Text, Integer, ForeignKey, Sequence, Date, Boolean
from sqlalchemy.orm import relation
from base import Base

class Categories(Base):
    __tablename__ = 'categories'

    id = Column(Integer, Sequence('seq_categories'), primary_key=True)
    name = Column(Text, nullable=False)
    age_restricted = Column(Boolean, nullable=False)

    def __init__(self, id, name, age_restricted):
        self.id = id
        self.name = name
        self.age_restricted = age_restricted
```

```

from sqlalchemy import Column, Text, Integer, ForeignKey, Sequence, Date
from sqlalchemy.orm import relation
from base import Base
from orders import Orders
from products import Products

class OP_relations(Base):
    __tablename__ = 'order_products_relations'

    id = Column(Integer, Sequence('seq_op_relation'), primary_key=True)
    order_id = Column(Integer, ForeignKey('orders.id'))
    product_id = Column(Integer, ForeignKey('products.id'))
    order = relation(Orders, backref='order_products_relations')
    product = relation(Products, backref='order_products_relations')

    def __init__(self, id, order_id, product_id):
        self.id = id
        self.order_id = order_id
        self.product_id = product_id

```

Приклади запитів:

```
def update(self, table_instance, id, new_state: dict):
    try:
        self.session.query(table_instance).filter(table_instance.id == id).update(new_state)
        self.session.commit()

    except Exception as err:
        raise Exception('DataBase update error\n', str(err))

def select_order_products(self, order_id):
    try:
        items = self.session.query(Products, OP_relations).filter(Products.id == OP_relations.product_id). \
            filter(OP_relations.order_id == order_id).all()

        return items
    except Exception as err:
        raise Exception('DataBase select order products error\n', str(err))

def select_orders_between(self, start, end):
    try:
        items = self.session.query(Orders).filter(Orders.date.between(start, end))
        return items

    except Exception as err:
        raise Exception('DataBase select orders between error\n', str(err))

def select_products_between(self, start, end):
    try:
        items = self.select_orders_between(start, end)
        id_list = [item.id for item in items]

        items = self.session.query(OP_relations).filter(OP_relations.order_id.in_(id_list)).all()
        return self.session.query(Products).filter(Products.id.in_([item.id for item in items]))
    except Exception as err:
        raise Exception('DataBase select products between error\n', str(err))

def select_product_by_restriction(self, restricted: bool):
    try:
        items = self.session.query(Products, Categories).filter(Products.fk_category == Categories.id). \
            filter(Categories.age_restricted == restricted)

        return items
    except Exception as err:
        raise Exception('DataBase select product by restriction error\n', str(err))
```

DataBase > select_order_products() > try

models.py > queries.py > queries.py > products.py > ipdb.locked.py > departments.py > categories.py > orders.py

```
def select_product_by_restriction(self, restricted: bool):
    try:
        items = self.session.query(Products, Categories).filter(Products.fk_category == Categories.id). \
            filter(Categories.age_restricted == restricted)

        return items
    except Exception as err:
        raise Exception('Database select products by restriction error\n', str(err))

def select_table_columns_name(self, table_instance):
    try:
        items = table_instance.__table__.columns.keys()
        return items
    except Exception as err:
        raise Exception('Database wrong table\n', str(err))

def get_class_by_tablename(self, table_name):
    for c in Base._decl_class_registry.values():
        if hasattr(c, '__tablename__') and c.__tablename__ == table_name:
            return c

def generate(self):
    try:
        self.engine.execute(text('INSERT INTO orders(date, customer)
            (SELECT timestamp "2020-01-01 20:00:00" + random() *
            (timestamp "2020-12-31 23:59:59" - timestamp "2020-01-01 20:00:00"),
            md5(random()::text) FROM generate_series(1,50000));

            INSERT INTO products(name, price, fk_department, fk_category)
            (SELECT md5(random()::text), floor(random() * 1000 + 1)::int,
            floor(random() * (SELECT COUNT(*) FROM departments) + 1)::int, floor(random() * (SELECT COUNT(*) FROM categories) + 1)::int
            FROM generate_series(1,50000));

            INSERT INTO order_products_relations(order_id, product_id)
            (SELECT floor(random() * (SELECT COUNT(*) FROM orders) + 1)::int, floor(random() * (SELECT COUNT(*) FROM products) + 1)::int
            FROM generate_series(1,50000));'))).execution_options(isolation_level='serializable')

    except Exception as err:
        raise Exception('Database generate error\n', str(err))
```

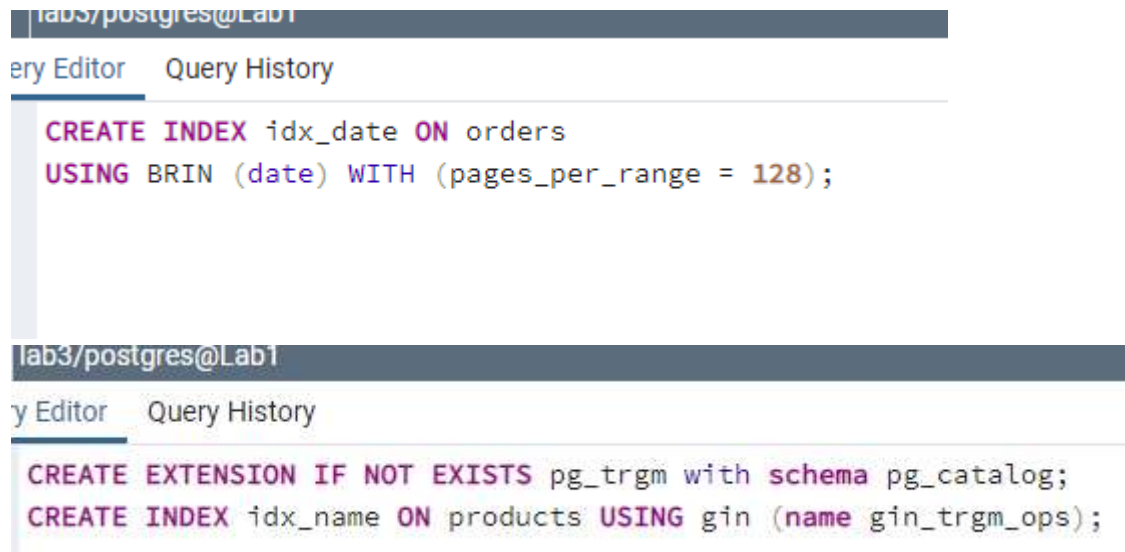
Database > select_order_products() > ip

Завдання 2

Створити та проаналізувати різні типи індексів у PostgreSQL.

В даному прикладі використовується GIN та BRIN

Створення:



The image contains two screenshots of a PostgreSQL query editor interface. The top screenshot shows the 'Query Editor' tab with the following SQL command: `CREATE INDEX idx_date ON orders USING BRIN (date) WITH (pages_per_range = 128);`. The bottom screenshot shows the same interface with two SQL commands: `CREATE EXTENSION IF NOT EXISTS pg_trgm with schema pg_catalog;` and `CREATE INDEX idx_name ON products USING gin (name gin_trgm_ops);`. Both screenshots show a dark-themed header bar with the text 'lab3/postgres@Lab1' and a light-colored sidebar on the left.

```
lab3/postgres@Lab1
Query Editor  Query History

CREATE INDEX idx_date ON orders
USING BRIN (date) WITH (pages_per_range = 128);

lab3/postgres@Lab1
Query Editor  Query History

CREATE EXTENSION IF NOT EXISTS pg_trgm with schema pg_catalog;
CREATE INDEX idx_name ON products USING gin (name gin_trgm_ops);
```

Тести:

```

1 EXPLAIN ANALYZE SELECT * FROM orders
2 WHERE date BETWEEN '2020-08-18' AND '2020-12-18'

```

Data Output Explain Messages Notifications

QUERY PLAN

1. Seq Scan on orders (cost=0.00..12170.00 rows=25826 width=41) (actual rows=5)

2. Filter: (date >= 2020-08-18'00:00:00) AND (date <= 2020-12-18'00:00:00)

3. Rows Removed by Filter: 24793

4. Planning Time: 0.098 ms

5. Execution Time: 81.315 ms

✓ Successfully run. Total query runtime: 84 msec. 5 rows affected.

```

1 EXPLAIN ANALYZE SELECT date,COUNT(*) FROM orders
2 GROUP BY date
3 ORDER BY date ASC;

```

Data Output Explain Messages Notifications

QUERY PLAN

1. Finalize GroupAggregate (cost=6817.33..6910.38 rows=367 width=12) (actual rows=1)

2. Group Key: date

3. → Gather Merge (cost=5817.33..6902.36 rows=738 width=12) (actual rows=1)

4. Workers Planned: 2

5. Workers Launched: 2

6. → Sort (cost=7617.33..7618.22 rows=367 width=12) (actual time=1.00 ms)

7. Sort Key: date

8. Sort Method: quicksort Memory: 42kB

9. Worker 0: Sort Method: quicksort Memory: 42kB

10. Worker 1: Sort Method: quicksort Memory: 42kB

11. → Partial HashAggregate (cost=7598.00..7601.67 rows=367 width=12)

✓ Successfully run. Total query runtime: 109 msec. 18 rows affected.

The image displays two screenshots of a database query interface, likely PostgreSQL, showing SQL queries and their execution plans.

Top Screenshot:

```
1. EXPLAIN ANALYZE SELECT * FROM products;
2. WHERE name LIKE 'a%';
```

The execution plan shows a sequential scan on the 'products' table, which has 100,000 rows. The plan includes details about the scan, the filter applied, and the execution time.

Bottom Screenshot:

```
1. EXPLAIN ANALYZE SELECT name, COUNT(*) FROM products;
2. WHERE name LIKE 'a%';
3. GROUP BY name;
4. ORDER BY name ASC;
```

The execution plan for this query shows a group aggregate operation. It details the scan of the 'products' table, the filter applied, the grouping operation, and the execution time.

В данному випадку, GIN індекс не буде дуже ефективним через випадку генерацію даних.

BRIN індексування може показати себе більш ефективно при більших розмірів даних(у тесті використовується таблиця з 500000 рядків).

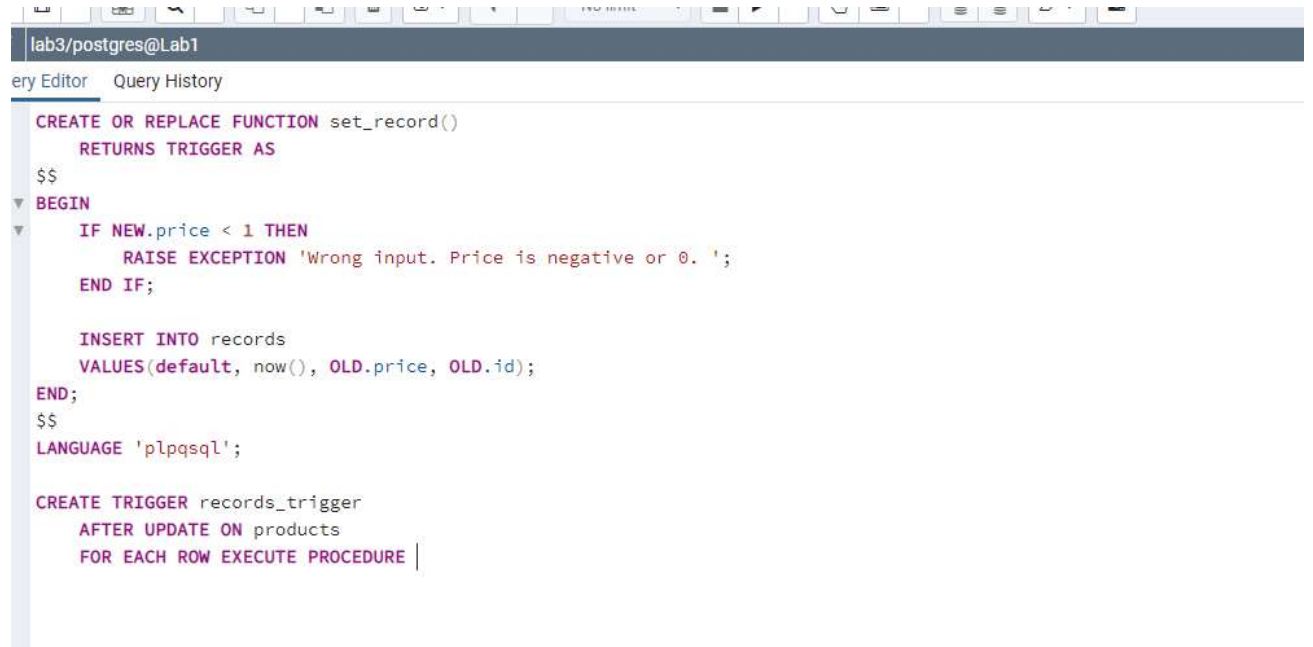
BTree індекс більш підходить до цього завдання до поки розмір таблиці не занадто гігантський, адже при великих розмірах він буде займати велику кількість пам'яті, навідміну від BRIN.

Завдання 3

Розробити тригер бази даних PostgreSQL.

Розроблений тригер відслідковує зміну ціни товару та записує її стару ціну у іншу таблицю. Також відбувається валідація ціни.

Текст тригера:



```
lab3/postgres@Lab1
Query Editor  Query History

CREATE OR REPLACE FUNCTION set_record()
    RETURNS TRIGGER AS
$$
BEGIN
    IF NEW.price < 1 THEN
        RAISE EXCEPTION 'Wrong input. Price is negative or 0. ';
    END IF;

    INSERT INTO records
        VALUES(default, now(), OLD.price, OLD.id);
END;
$$
LANGUAGE 'plpgsql';

CREATE TRIGGER records_trigger
    AFTER UPDATE ON products
    FOR EACH ROW EXECUTE PROCEDURE
```

Тести:

Оновлення рядка:

Commands:

1. Read items in table
2. Insert item to table
3. Update item in table
4. Read all products in order
5. Read all orders in dates
6. Read all products in dates
7. Read products by restriction
8. Generate some data
9. Exit

command: 3

Enter item id: 20

Enter table name: products

id|name|price|fk_department|fk_category

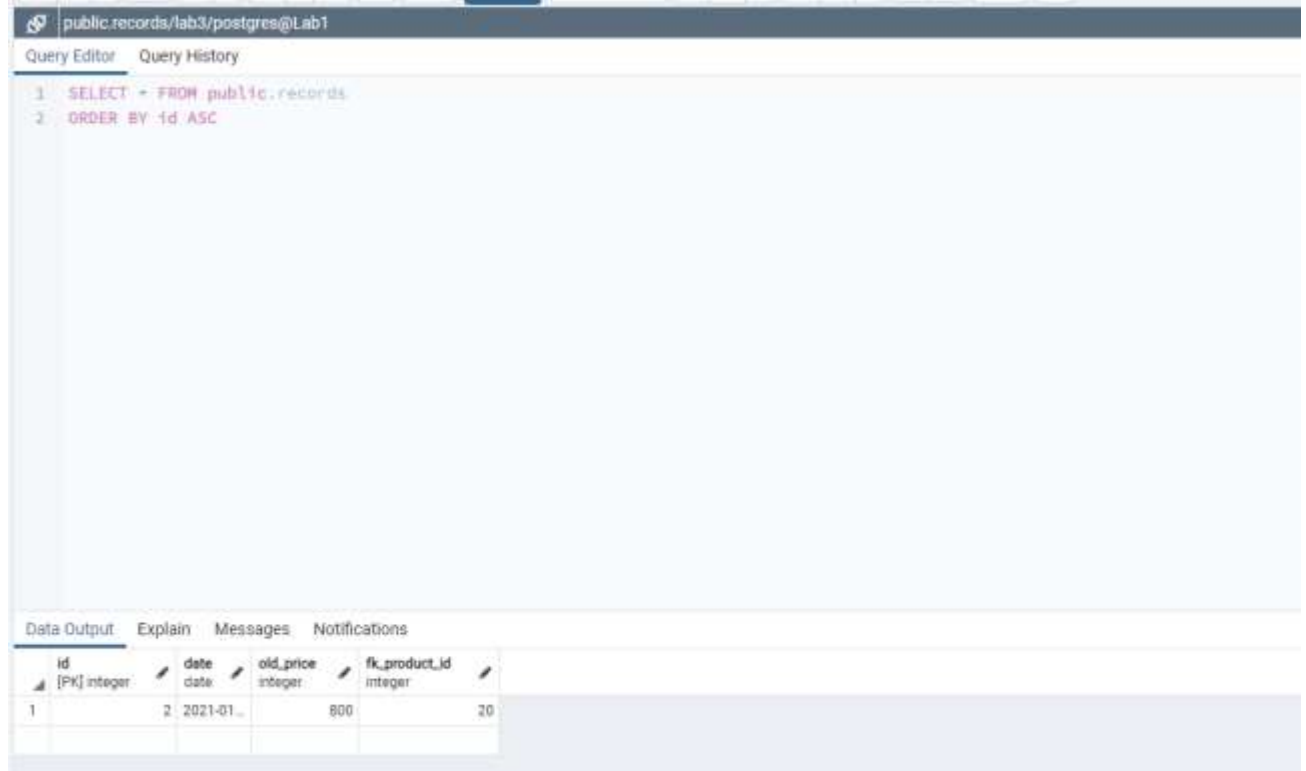
Enter data(coma split, dict): price=59

Just updated item with id#20!

0.04845452308654785 ms

Press any key to continue|

Відповідний запис у іншій таблиці:



The screenshot shows a PostgreSQL Query Editor window with the following content:

Query Editor Query History

```
1 SELECT * FROM public.records
2 ORDER BY id ASC
```

Data Output Explain Messages Notifications

id	date	old_price	fk_product_id
[PK] integer	date	integer	integer
1	2021-01-01	800	20

Валідація:



The screenshot shows a terminal window with the following content:

```
1. Read items in table
2. Insert item to table
3. Update item in table
4. Read all products in table
5. Read all items in table
6. Read all products in table
7. Read products by restriction
8. Generate new data
9. Exit
```

```
Insert item (id=1, date='2021-01-01', old_price=800, fk_product_id=20)
ERROR: insert or update on table 'records' violates foreign key constraint 'fk_records_fk_product_id'
DETAIL:  (key=1, value='2021-01-01', old_price=800, fk_product_id=20) does not have a matching row in table 'products'.
```

The error message indicates a foreign key constraint violation. The details show that the record being inserted has a date of '2021-01-01', an old price of 800, and a foreign key value of 20, which does not match any row in the 'products' table.