



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
ім. Ігоря СІКОРСЬКОГО»  
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ  
**Кафедра системного програмування і спеціалізованих комп'ютерних систем**

**Лабораторна робота №3**  
з дисципліни  
**«Бази даних і засоби управління»**  
*Тема: «Засоби оптимізації роботи СУБД PostgreSQL»*

Виконав: студент III курсу  
ФПМ групи КВ-94  
Колесніков Є. О.  
Перевірів: Петрашенко А.В.

## Постановка задачі

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

### Варіант 12

12	BTree, GIN	after update, insert
----	------------	----------------------

Посилання на репозиторій у GitHub з вихідним кодом програми та звітом:

[https://github.com/kolesnikov-dev/DB\\_lab3](https://github.com/kolesnikov-dev/DB_lab3)

## Завдання №1

Обрана предметна галузь передбачає отримання і обробку замовлень з різних інтернет-магазинів.

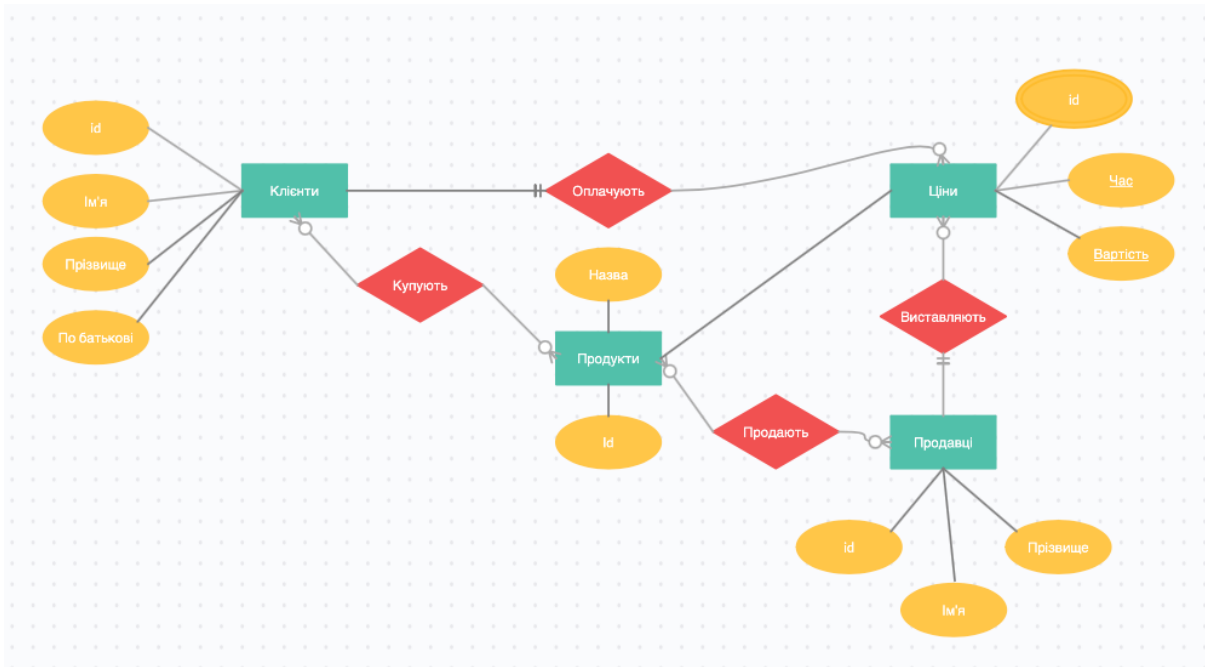


Рисунок 1. ER-діаграма, побудована за нотацією Чена

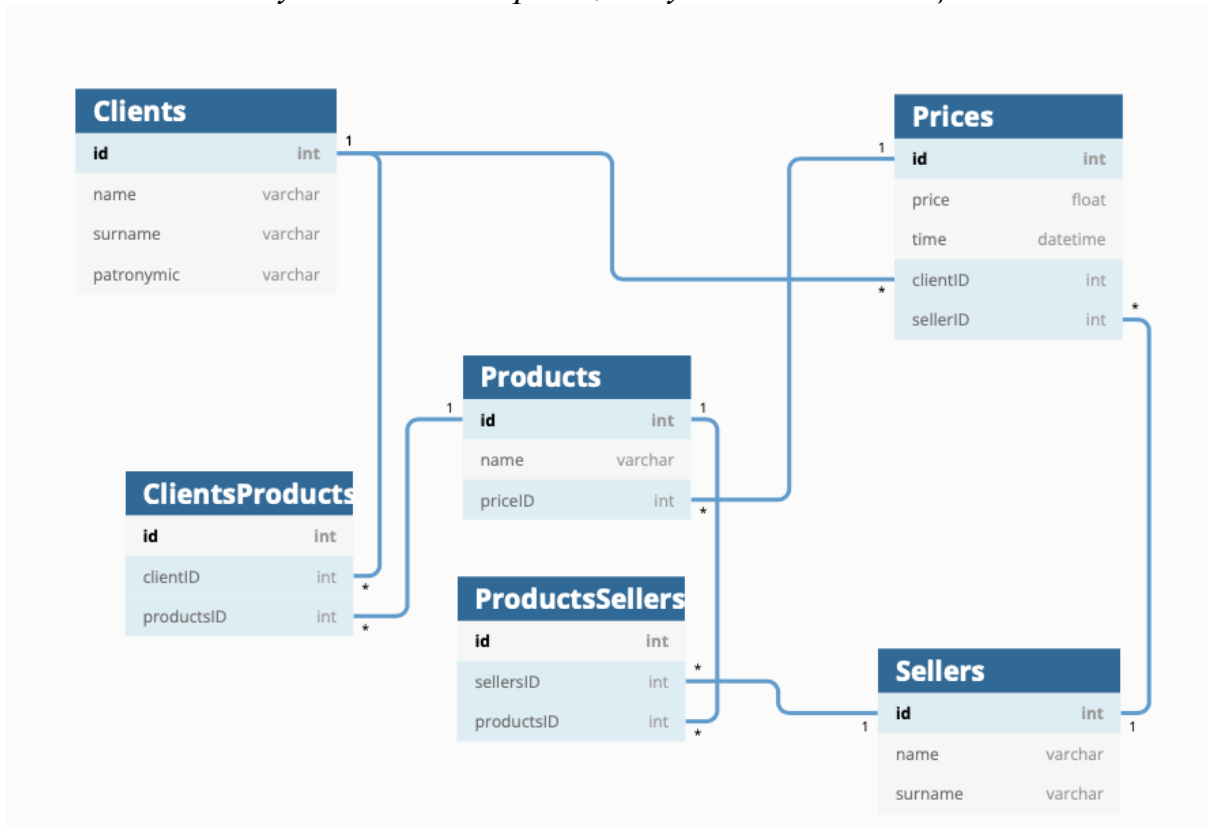


Рисунок 2. Схема бази даних

## Класи ORM у реалізованому модулі Model

```
from sqlalchemy import BigInteger, Column, DateTime, Float, ForeignKey, Index, Numeric, String, Table,
Text, text

from sqlalchemy.dialects.postgresql import OID

from sqlalchemy.orm import relationship

from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

metadata = Base.metadata


class Client(Base):
    __tablename__ = 'Clients'
    __table_args__ = (
        Index('PupilsBtree', 'Surname', 'Patronymic', 'Name'),
    )

    Id = Column(BigInteger, primary_key=True, server_default=text("nextval('\\"Pupils_Id_seq\\"'::regclass)"))
    Name = Column(String(20), nullable=False)
    Patronymic = Column(String(20), nullable=False)
    Surname = Column(String(20), nullable=False, index=True)


class Seller(Base):
    __tablename__ = 'Sellers'

    id = Column(BigInteger, primary_key=True, unique=True,
server_default=text("nextval('\\"Teachers_id_seq\\"'::regclass)"))
    name = Column(String(20), nullable=False)
    surname = Column(String(20), nullable=False)


class Price(Base):
    __tablename__ = 'Prices'

    id = Column(BigInteger, primary_key=True, unique=True,
server_default=text("nextval('\\"Marks_id_seq\\"'::regclass)"))
    time = Column(DateTime, nullable=False)
    clientsid = Column(ForeignKey('Clients.Id'), nullable=False)
```

```
sellersid = Column(ForeignKey('Sellers.id'), nullable=False)
price = Column(Float, nullable=False)
```

```
Client = relationship('Client')
```

```
Seller = relationship('Seller')
```

```
class Product(Base):
```

```
    __tablename__ = 'Products'
```

```
id = Column(BigInteger, primary_key=True, unique=True,
server_default=text("nextval('"Subjects_id_seq'"::regclass)))
```

```
name = Column(String(20), nullable=False)
```

```
pricesid = Column(ForeignKey('Prices.id'), nullable=False)
```

```
Price = relationship('Price')
```

```
class ClientsProduct(Base):
```

```
    __tablename__ = 'ClientsProducts'
```

```
id = Column(BigInteger, primary_key=True, unique=True,
server_default=text("nextval('"PupilsSubjects_id_seq'"::regclass)))
```

```
clientsid = Column(ForeignKey('Clients.Id'), nullable=False)
```

```
productsid = Column(ForeignKey('Products.id'), nullable=False)
```

```
Client = relationship('Client')
```

```
Product = relationship('Product')
```

```
class SellersProduct(Base):
```

```
    __tablename__ = 'SellersProducts'
```

```
id = Column(BigInteger, primary_key=True, unique=True,
server_default=text("nextval('"TeachersSubjects_id_seq'"::regclass)))
```

```
sellersid = Column(ForeignKey('Sellers.id'), nullable=False)
```

```
productsid = Column(ForeignKey('Products.id'), nullable=False)
```

```
Product = relationship('Product')
```

```
Seller = relationship('Seller')
```

## Запити у вигляді ORM

Продемонструємо вставку, виучення, редагування даних на прикладі таблиці Products.  
Початковий стан:

```
Choose your table: 2
*****

id      Name      Surname
8472    OB        AF
8474    WV        GM
8475    QG        JD
8476    HC        CF
8477    SC        QQ
8478    FO        EF
8479    GQ        ST
8480    KS        NV
8482    LS        QR
8483    FS        XK
8484    LZ        UR
8485    XA        MO
8486    OO        XG
8487    OX        PM
8489    KP        ZF
8490    QB        MD
8491    YY        JY
8496    New       Seller
*****
```

Додання запису:

```
Your choice is: 3

1          => Products
2          => Sellers
3          => Clients
4          => Prices
5          => ClientsProducts
6          => SellersProducts

Choose your table: 2
Name = Pupa
Surname = Lupa
'added'
```

Видалення запису:

```
Your choice is: 4

1          => Products
2          => Sellers
3          => Clients
4          => Prices
5          => ClientsProducts
6          => SellersProducts

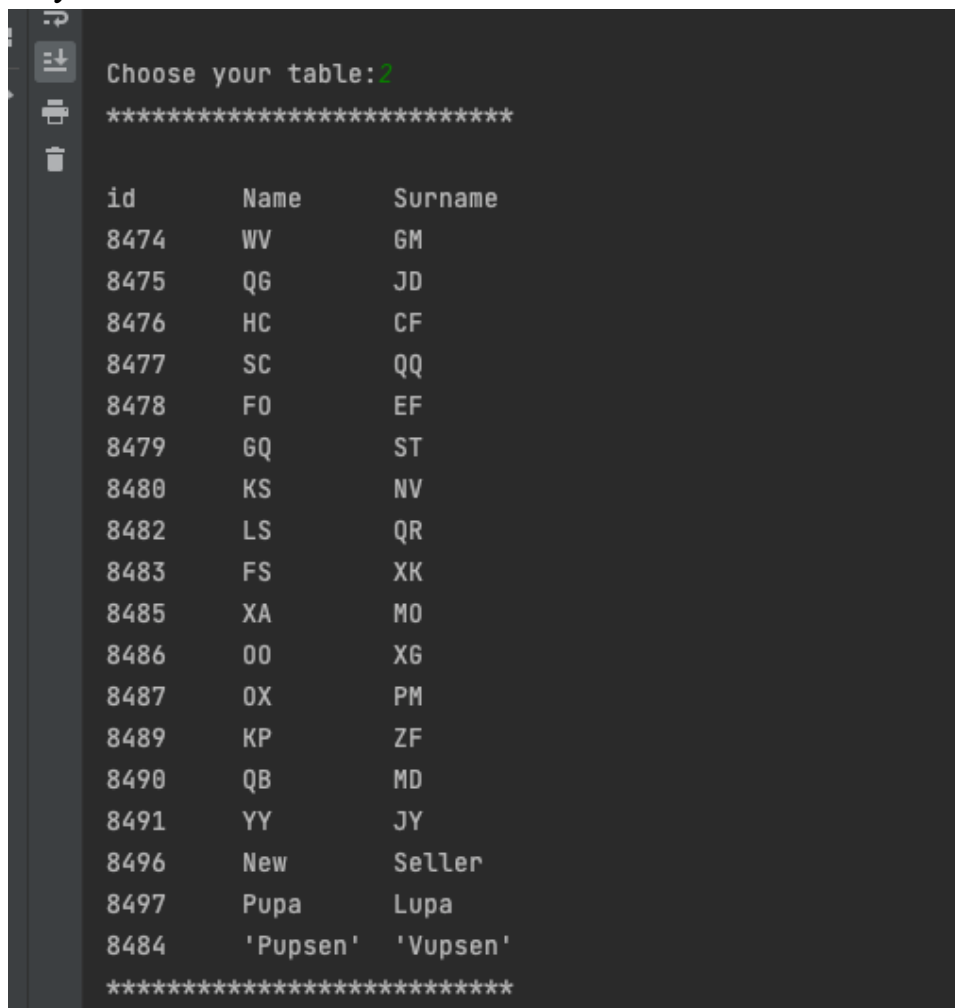
Choose your table: 2
Attribute to delete ID = 8472
'deleted'
```

Редагування запису:

```
1          => Products
2          => Sellers
3          => Clients
4          => Prices
5          => ClientsProducts
6          => SellersProducts

Choose your table: 2
Row to update where id = 8484
Name = Pupsen
Surname = Vupsen
'updated'
```

Стан таблиці після усіх змін:



```
Choose your table: 2
*****
id      Name      Surname
8474    WV         GM
8475    QG         JD
8476    HC         CF
8477    SC         QQ
8478    FO         EF
8479    GQ         ST
8480    KS         NV
8482    LS         QR
8483    FS         XK
8485    XA         MO
8486    OO         XG
8487    OX         PM
8489    KP         ZF
8490    QB         MD
8491    YY         JY
8496    New       Seller
8497    Pupa      Lupa
8484    'Pupsen'  'Vupsen'
*****
```

## Завдання №2

Для тестування індексів було створено окремі таблиці у базі даних з 1000000 записів.

### *BTree*

Індекс btree, він же В-дерево, придатний для даних, які можна відсортувати. Іншими словами, для типу даних повинні бути визначені оператори «більше», «більше або одно», «менше», «менше або одно» та «рівно». Зауважте, що ті самі дані іноді можна сортувати різними способами, що повертає нас до концепції сімейства операторів.

Як завжди, індексні записи В-дерева упаковані у сторінки. У листових сторінках ці записи містять індексовані дані (ключі) та посилання на рядки таблиці (TID-и); у внутрішніх сторінках кожен запис посилається на дочірню сторінку індексу та містить мінімальне значення ключа у цій сторінці.

*В-дерева мають кілька важливих властивостей:*

- Вони збалансовані, тобто будь-яку листову сторінку відокремлює від кореня те саме число внутрішніх сторінок. Тому пошук будь-якого значення займає однаковий час.
- Вони дуже гіллясті, тобто кожна сторінка (зазвичай, 8 КБ) містить відразу багато (сотні) TID-ів. За рахунок цього глибина В-дерев виходить невеликою; практично до 4–5 для



великих таблиць.

- Дані в індексі впорядковані через незменшення (як між сторінками, так і всередині кожної сторінки), а сторінки одного рівня пов'язані між собою двонаправленим списком. Тому отримати впорядкований набір даних ми можемо просто проходячи по списку в одну або в іншу сторону, не повертаючись щоразу до кореня.

Створення таблиці БД:

```
DROP TABLE IF EXISTS "test_btree";  
CREATE TABLE "test_btree"(  
    "id" bigserial PRIMARY KEY,  
    "test_text" varchar(255)  
);
```

Запити для тестування:

```
SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0;  
SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0 OR "test_text" LIKE 'b%';  
SELECT COUNT(*), SUM("id") FROM "test_btree" WHERE "test_text" LIKE 'b%' GROUP BY "id" % 2;
```

Створення індексу:

```
DROP INDEX IF EXISTS "test_btree_test_text_index";  
CREATE INDEX "test_btree_test_text_index" ON "test_btree" USING btree ("test_text");
```

## Результати і час виконання psql

```
psql (14.1, server 13.4 (Ubuntu 13.4-4.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

itqsairb=> \i /Users/deadpixel/Downloads/SimpleMVC/testIndex.sql
Timing is on.
DROP TABLE
Time: 66.749 ms
CREATE TABLE
Time: 60.458 ms
INSERT 0 1000000
Time: 4558.799 ms (00:04.559)
 count
-----
500000
(1 row)

Time: 609.408 ms
 count
-----
509621
(1 row)

Time: 309.230 ms
 count |      sum
-----+-----
 9550 | 4787055590
 9621 | 4794412865
(2 rows)

Time: 359.660 ms
psql:/Users/deadpixel/Downloads/SimpleMVC/testIndex.sql:19: NOTICE:  index "test_btree_test_text_index" does not exist, skipping
DROP INDEX
Time: 36.164 ms
CREATE INDEX
Time: 2672.624 ms (00:02.673)
 count
-----
500000
(1 row)

Time: 332.404 ms
 count
-----
509621
(1 row)

Time: 299.006 ms
 count |      sum
-----+-----
 9621 | 4794412865
 9550 | 4787055590
(2 rows)

Time: 290.205 ms
```

Очевидно, що індексування за допомогою BTree не значно пришвидшує пошук даних у таблиці, а іноді навіть показує гірші результати, ніж запити без індексування. Це впливає з того, що це один із найпримітивніших методів індексування і для пошуку потрібних даних алгоритм все одно проходить через усі записи у таблиці (на відміну від GIN). Він ефективний при застосуванні до поля числового типу.

### *GIN*

GIN призначений для обробки випадків, коли елементи, що підлягають індексації, є складеними значеннями (наприклад - реченнями), а запити, які обробляються індексом, мають шукати значення елементів, які з'являються в складених елементах (повторювані частини слів або речень). Індекс GIN зберігає набір пар (ключ, список появи ключа), де список появи — це набір ідентифікаторів рядків, у яких міститься ключ. Один і той самий ідентифікатор рядка може знаходитись у кількох списках, оскільки елемент може містити більше одного ключа. Кожне значення ключа зберігається лише один раз, тому індекс GIN дуже швидкий для випадків, коли один і той же ключ з'являється багато разів. Цей індекс може взаємодіяти тільки з полем типу tsvector.

Створення таблиці БД:

```
CREATE TABLE "test_gin"(  
  "id" bigserial PRIMARY KEY,  
  "test_time" timestamp  
);
```

Запити для тестування:

```
INSERT INTO "test_gin"("test_time")  
SELECT  
  (timestamp '2021-01-01' + random() * (timestamp '2020-01-01' - timestamp '2022-01-01'))  
FROM  
  (VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as symbols(characters),  
  generate_series(1, 1000000) as q;
```

Створення індексу:

```
DROP INDEX IF EXISTS "test_gin_test_time_index";  
CREATE INDEX "test_gin_test_time_index" ON "test_gin" USING gin ("test_time");
```

## Результати і час виконання psql

```
Time: 290.205 ms
psql:/Users/deadpixel/Downloads/SimpleMVC/testIndex.sql:28: NOTICE:  table "test_gin" does not exist, skipping
DROP TABLE
Time: 34.352 ms
CREATE TABLE
Time: 44.561 ms
INSERT 0 1000000
Time: 4529.278 ms (00:04.529)
 count
-----
500000
(1 row)

Time: 217.694 ms
 count
-----
329038
(1 row)

Time: 396.568 ms
 count |      sum
-----+-----
164333 | 82298627417
164705 | 82307300058
(2 rows)

Time: 314.203 ms
psql:/Users/deadpixel/Downloads/SimpleMVC/testIndex.sql:45: NOTICE:  index "test_gin_test_time_index" does not exist, skipping
DROP INDEX
Time: 41.183 ms
CREATE INDEX
Time: 4868.611 ms (00:04.869)
 count
-----
500000
(1 row)

Time: 253.141 ms
 count
-----
329038
(1 row)

Time: 666.471 ms
 count |      sum
-----+-----
164333 | 82298627417
164705 | 82307300058
(2 rows)

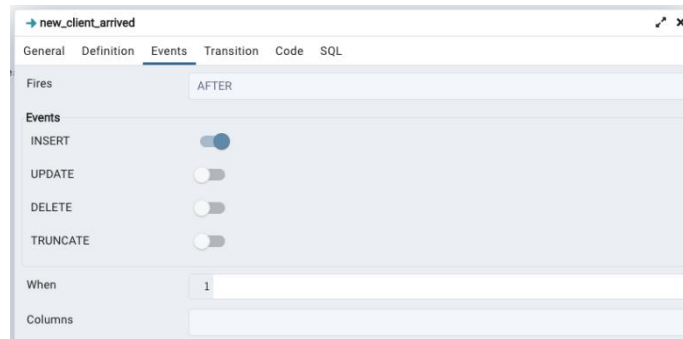
Time: 463.638 ms
itqsairb=>
```

З отриманих результатів бачимо, що в усіх заданих випадках пошук з індексацією відбувається значно швидше, ніж пошук без індексації (окрім першого, оскільки на перший запит дана індексація не впливає). Це відбувається завдяки головній особливості індексування GIN: кожне значення шуканого ключа зберігається один раз і запит іде не по всій таблиці, а лише по тим даним, що містяться у списку появи цього ключа. Для даних типу `numeric` даний тип індексування використовувати недоцільно і неможливо.

## Завдання №3

Тригер створений для таблиці Clients. Під час додання нового учня він прив'язується до продавця та продукта.

Команди, що ініціюють виконання тригера:



The screenshot shows the configuration window for a trigger named 'new\_client\_arrived'. The 'Events' tab is selected, showing that the trigger fires 'AFTER' the 'INSERT' event. The 'When' field is set to '1', indicating it fires on the first row of the data set. The 'Columns' field is empty.

Текст тригера:

```
declare
    sellerID bigint;
    productID bigint;
BEGIN
    select id into sellerID from public."Sellers" order by random() limit 1;
    select id into productID from public."Products" order by random() limit 1;
    insert into public."ClientsProducts"(clientsid, productsid) values (NEW."Id", productID);
    insert into public."SellersProducts"(sellerID, productsid) values (sellerID, productID);
    RETURN NULL;
END
```

Результат роботи тригера:

itqsairb/itqsairb@ElephantDB

Query EditorQuery History

1

insert into public."Clients"("Name", "Patronymic", "Surname") values ('Test', 'DB', 'Trigger');

2

select \* from public."Clients"

3

join public."ClientsProducts" on public."ClientsProducts".clientsid = public."Clients"."Id"

4

join public."Products" on public."Products".id = public."ClientsProducts".productsid

5

where public."Clients"."Surname" = 'Trigger';

Data Output

Explain

Messages

Notifications

	id bigint	Name character varying (20)	Patronymic character varying (20)	Surname character varying (20)	id bigint	clientsid bigint	productsid bigint	id bigint	name character varying (20)	pricesid bigint
1	59	Test	DB	Trigger	24	59	37	37	'pupa'	8

За результатами видно що після додання нового студента з’явилися відповідні записи у Products та ClientsProducts.

## Завдання №4

Для цього завдання також створювалась окрема таблиця з деякими початковими даними:

```
DROP TABLE IF EXISTS "transactions";  
CREATE TABLE "transactions"(  
    "id" bigserial PRIMARY KEY,  
    "numeric" bigint,  
    "text" text  
);
```

```
INSERT INTO "transactions"("numeric", "text") VALUES (111, 'string1'), (222,  
'string2'), (333, 'string3');
```



## REPEATABLE READ

На цьому рівні ізоляції T2 не бачитиме змінені дані транзакцією T1, але також не зможе отримати доступ до тих самих даних.

Тут видно, що друга не бачить змін з першої:

```
postgres=# START TRANSACTION;SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ^
READ WRITE;
START TRANSACTION
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;INSERT INTO
"transactions"("numeric", "text") VALUES (444, 'string4');DELETE FROM "trans
actions" WHERE "id"=1;
UPDATE 3
INSERT 0 1
DELETE 1
postgres=#
```

```
postgres=# START TRANSACTION;SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ^
READ WRITE;
START TRANSACTION
SET
postgres=# SELECT * FROM "transactions";
id | numeric | text
-----+-----+-----
1  |    111  | string1
2  |    222  | string2
3  |    333  | string3
(3 строки)

postgres=#
```

А тут, що отримуємо помилку при спробі доступу до тих самих даних:

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=# COMMIT;
COMMIT
postgres=#
```

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
ПОМИЛКА: не вдалося серіалізувати доступ через паралельне оновлення
postgres=# SELECT * FROM "transactions";
ПОМИЛКА: поточна транзакція перервана, команди до кінця блока транзакції пропускаються
postgres=# COMMIT;
ROLLBACK
postgres=# SELECT * FROM "transactions";
id | numeric | text
-----+-----+-----
1  |    112  | string1
2  |    223  | string2
3  |    334  | string3
(3 строки)
```

Бачимо, що не виникає читання фантомів та повторного читання, а також заборонено одночасний доступ до незбережених даних. Хоча класично цей рівень ізоляції призначений для попередження повторного читання.

## SERIALIZABLE

На цьому рівні транзакції поведуть себе так, ніби вони не знають одна про одну. Вони не можуть вплинути одна на одну і одночасний доступ строго заборонений.

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=# INSERT INTO "transactions"("numeric", "text") VALUES (444, 'string4');
INSERT 0 1
postgres=# DELETE FROM "transactions" WHERE "id"=1;
DELETE 1
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  2 |    223 | string2
  3 |    334 | string3
  4 |    444 | string4
(3 строки)

postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  2 |    223 | string2
  3 |    334 | string3
  4 |    444 | string4
(3 строки)

postgres=# COMMIT;
COMMIT
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  2 |    223 | string2
  3 |    334 | string3
  4 |    444 | string4
(3 строки)

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE;
SET
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
ПОВИЛКА: не вдалося серіалізувати доступ через паралельне оновлення
postgres=# INSERT INTO "transactions"("numeric", "text") VALUES (444, 'string4');
ПОВИЛКА: поточна транзакція перервана, команди до кінця блока транзакції пропускаються
postgres=# DELETE FROM "transactions" WHERE "id"=1;
ПОВИЛКА: поточна транзакція перервана, команди до кінця блока транзакції пропускаються
postgres=# COMMIT
postgres=# ROLLBACK
postgres=# COMMIT
postgres=#
```

У попередньому випадку вдалось “відкатити” другу транзакцію і це не вплинуло на подальшу можливість роботи в терміналі. На цьому ж рівні навіть після завершення першої не вдалося зробити ні COMMIT ні ROLLBACK для другої транзакції. Взагалі, в класичному представленні цей рівень призначений для недопущення явища читання фантомів. На цьому рівні ізоляції ми отримуємо максимальну узгодженість даних і можемо бути впевнені, що зайві дані не будуть зафіксовані.

## READ COMMITTED

На цьому рівні ізоляції одна транзакція не бачить змін у базі даних, викликаних іншою доки та не завершить своє виконання (командою COMMIT або ROLLBACK).

Дані після вставки та видалення так само будуть видні другій тільки після завершення першої.

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=# INSERT INTO "transactions"("numeric", "text") VALUES (444, 'string4');
INSERT 0 1
postgres=#
```

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=#
```

```
postgres=#
postgres=#
postgres=# DELETE FROM "transactions" WHERE "id"=1;
DELETE 1
postgres=#

postgres=#
postgres=# COMMIT;
COMMIT
postgres=#
```

```
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=#
```

```
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  2 |    223 | string2
  3 |    334 | string3
  4 |    444 | string4
(3 строки)

postgres=#
```

На цьому знімку також бачимо, що друга транзакція (справа) не може внести дані у базу, доки не завершилась попередня.

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=#
```

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
-
```

А тут бачимо, що після завершення першої, друга транзакція виконала запит, змінивши вже ті дані, що були закомічені першою транзакцією

```

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    112 | string1
  2 |    223 | string2
  3 |    334 | string3
(3 строки)

postgres=# commit;
COMMIT
postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    113 | string1
  2 |    224 | string2
  3 |    335 | string3
(3 строки)

postgres=#

```

```

^ postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    113 | string1
  2 |    224 | string2
  3 |    335 | string3
(3 строки)

postgres=# commit;
COMMIT
postgres=#

```

Коли T2 бачить дані T1 запитів UPDATE, DELETE виникає феномен повторного читання, а коли бачить дані запиту INSERT – читання фантомів. Цей рівень ізоляції забезпечує захист від явища брудного читання.