НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування та спеціалізованих комп'ютерних систем

Лабораторна робота №3

з дисципліни **Бази даних і засоби управління**

на тему: "Засоби оптимізації роботи СУБД PostgreSQL"

Виконала:
студентка III курсу
групи КВ-92
Корж А. А.
Перевірив:
Петрашенко А. В.

Київ – 2021

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.

№ варіанта	Види індексів	Умови для тригера
11	*GIN, Hash*	*before update, delete*

Посилання на репозиторій у GitHub з вихідним кодом програми та звітом: https://github.com/narcissichka/DataBase

Завдання №1

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

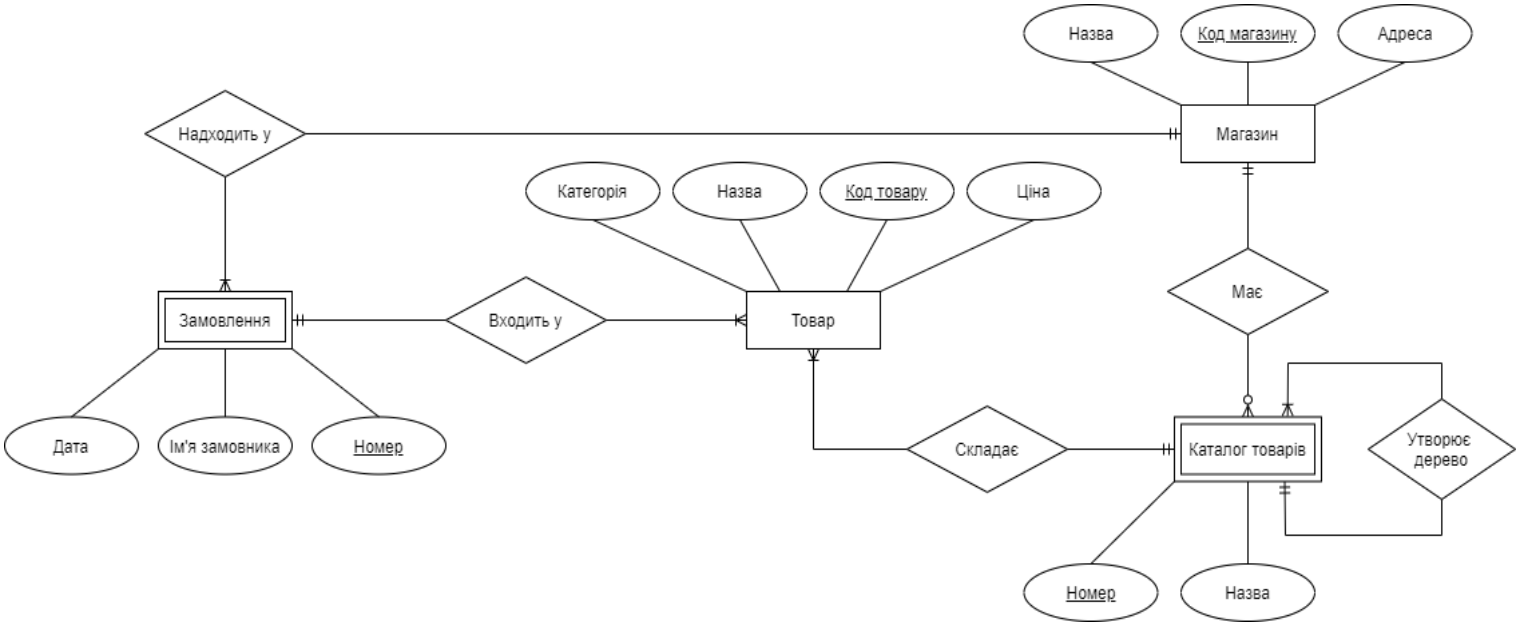


Рисунок 1. ER-діаграма, побудована за нотацією Чена

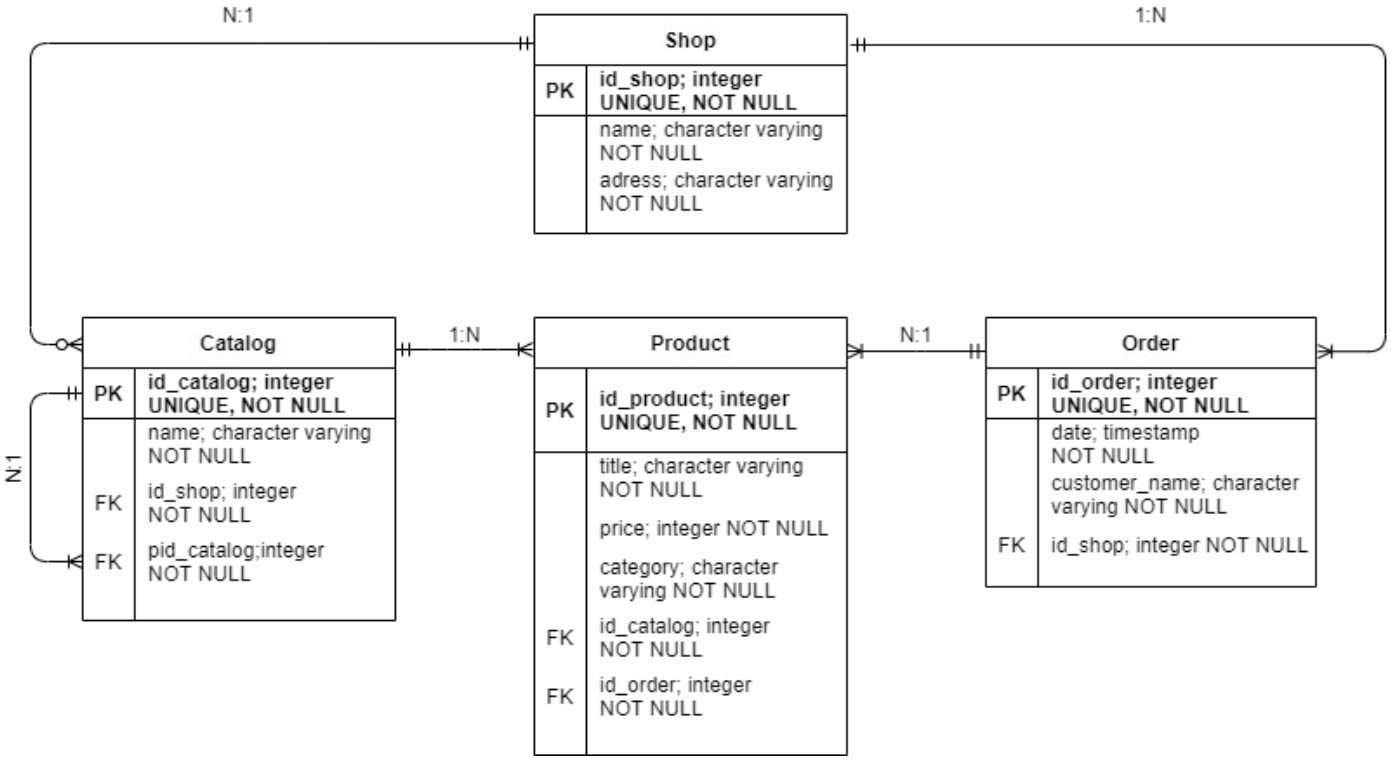


Рисунок 2. Схема бази даних

Таблиці бази даних у середовищі PgAdmin4

```
-- Table: public.Product

-- DROP TABLE public."Product";

CREATE TABLE IF NOT EXISTS public."Product"
(
    id_product integer NOT NULL,
    title character varying COLLATE pg_catalog."default" NOT NULL,
    price integer NOT NULL,
    category character varying COLLATE pg_catalog."default" NOT NULL,
    id_catalog integer NOT NULL,
    id_order integer NOT NULL,
    CONSTRAINT "Product_pkey" PRIMARY KEY (id_product),
    CONSTRAINT "Product_id_catalog_fkey" FOREIGN KEY (id_catalog)
        REFERENCES public."Catalog" (id_catalog) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID,
    CONSTRAINT "Product_id_order_fkey" FOREIGN KEY (id_order)
        REFERENCES public."Order" (id_order) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID
)

TABLESPACE pg_default;

ALTER TABLE public."Product"
    OWNER to postgres;

COMMENT ON COLUMN public."Product".id_order
    IS '(UNIQUE)';
```

```
-- Table: public.Order

-- DROP TABLE public."Order";

CREATE TABLE IF NOT EXISTS public."Order"
(
    id_order integer NOT NULL,
    customer_name character varying COLLATE pg_catalog."default" NOT NULL,
    id_shop integer NOT NULL,
    date timestamp without time zone NOT NULL,
    CONSTRAINT "Order_pkey" PRIMARY KEY (id_order),
    CONSTRAINT "Order_id_shop_fkey" FOREIGN KEY (id_shop)
        REFERENCES public."Shop" (id_shop) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID
)

TABLESPACE pg_default;

ALTER TABLE public."Order"
```

```
    OWNER to postgres;
```

```
-- Table: public.Catalog

-- DROP TABLE public."Catalog";

CREATE TABLE IF NOT EXISTS public."Catalog"
(
    id_catalog integer NOT NULL,
    name character varying COLLATE pg_catalog."default" NOT NULL,
    id_shop integer NOT NULL,
    pid_catalog integer NOT NULL,
    CONSTRAINT "Catalog_pkey" PRIMARY KEY (id_catalog),
    CONSTRAINT "Catalog_id_shop_fkey" FOREIGN KEY (id_shop)
        REFERENCES public."Shop" (id_shop) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID,
    CONSTRAINT pid_catalog_fkey FOREIGN KEY (pid_catalog)
        REFERENCES public."Catalog" (id_catalog) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID
)

TABLESPACE pg_default;

ALTER TABLE public."Catalog"
    OWNER to postgres;
```

```
-- Table: public.Shop

-- DROP TABLE public."Shop";

CREATE TABLE IF NOT EXISTS public."Shop"
(
    id_shop integer NOT NULL,
    address character varying COLLATE pg_catalog."default" NOT NULL,
    name character varying COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT "Shop_pkey" PRIMARY KEY (id_shop)
)

TABLESPACE pg_default;

ALTER TABLE public."Shop"
    OWNER to postgres;
```

Класи ORM у реалізованому модулі Model

```
class Shop(Orders):
    __tablename__ = 'Shop'
    id_shop = Column(Integer, primary_key=True)
    address = Column(String)
    name = Column(String)
    catalogs = relationship("Catalog")
    orders = relationship("Order")

    def __init__(self, id_shop, address, name):
        self.id_shop = id_shop
        self.address = address
        self.name = name
    def __repr__(self):
        return "{:>10}{:>35}{:>15}" \
            .format(self.id_shop, self.address, self.name)
```

```
class Catalog(Orders):
    __tablename__ = 'Catalog'
    id_catalog = Column(Integer, primary_key=True)
    name = Column(String)
    id_shop = Column(Integer, ForeignKey('Shop.id_shop'))
    pid_catalog = Column(Integer, ForeignKey('Catalog.id_catalog'))
    products = relationship("Product")
    parent_catalogs = relationship("Catalog")

    def __init__(self, id_catalog, name, id_shop, pid_catalog):
        self.id_catalog = id_catalog
        self.name = name
        self.id_shop = id_shop
        self.pid_catalog = pid_catalog
    def __repr__(self):
        return "{:>10}{:>15}{:>10}{:>10}" \
            .format(self.id_catalog, self.name, self.id_shop, self.pid_catalog)
```

```
class Order(Orders):
    __tablename__ = 'Order'
    id_order = Column(Integer, primary_key=True)
    customer_name = Column(String)
    id_shop = Column(Integer, ForeignKey('Shop.id_shop'))
    date = Column(Date)
    products = relationship("Product")

    def __init__(self, id_order, customer_name, id_shop, date):
        self.id_order = id_order
        self.customer_name = customer_name
        self.id_shop = id_shop
        self.date = date
    def __repr__(self):
        return "{:>10}{:>25}{:>10}{:>25}" \
            .format(self.id_order, self.customer_name, self.id_shop, self.date)
```

```
class Product(Orders):
    __tablename__ = 'Product'
    id_product = Column(Integer, primary_key=True)
    title = Column(String)
```

```
    price = Column(Float)
    category = Column(String)
    id_catalog = Column(Integer, ForeignKey('Catalog.id_catalog'))
    id_order = Column(Integer, ForeignKey('Order.id_order'))

    def __init__(self, id_product, title, price, category, id_catalog, id_order):
        self.id_product = id_product
        self.title = title
        self.price = price
        self.category = category
        self.id_catalog = id_catalog
        self.id_order = id_order
    def __repr__(self):
        return "{:>10}{:>30}{:>10}{:>15}{:>10}{:>10}" \
            .format(self.id_product, self.title, self.price, self.category,
self.id_catalog, self.id_order)
```

Запити у вигляді ORM

Продемонструємо вставку, виучення, редагування даних на прикладі таблиці Product

Початковий стан:

```
PS D:\3course\БД3У\Lab3_> python main.py print_table Product
Product table:
id_product                         title     price       category  id_catalog    id_order
         1                   Dior handbag   10000.0           bags         4      1200
         2             Hogo Boss trousers   20000.0         unders         7      1201
         3                  Armani jacket   15000.0           tops         9      1201
         4                 Moscino T-Shirt    500.0           tops         6      1202
         5            Dolce&Gabbana skirt    300.0         unders         6      1202
         6              Louis Vuitton bag  100500.0           bags         4      1202
         7                          Jeans    457.0         unders         9      1200
         8                         lsjxcu   32381.0      gqgnqluho         5      1201
         9                           cvcz    2347.0          fqbuy         8      1202
        10                           gdui   20645.0         qouphh         6      1203
        12                        CoolHat    144.0           hats         3      1203
        13                              t   56185.0    accessories         6      1208
        14                        shwhtetj    266.0       plwgjdcov        10      1203
        15                           mhxal   16360.0        lfykfqlh         5      1204
        16                            hrso   61132.0       nzhjlxaec         1      1203
PS D:\3course\БД3У\Lab3_>
```

Видалення запису:

```
PS D:\3course\БД3У\Lab3_> python main.py delete_record Product 16
PS D:\3course\БД3У\Lab3_> python main.py print_table Product
Product table:
id_product                        title     price       category  id_catalog    id_order
         1                 Dior handbag   10000.0           bags         4      1200
         2           Hogo Boss trousers   20000.0         unders         7      1201
         3                Armani jacket   15000.0           tops         9      1201
         4              Moscino T-Shirt     500.0           tops         6      1202
         5          Dolce&Gabbana skirt     300.0         unders         6      1202
         6            Louis Vuitton bag  100500.0           bags         4      1202
         7                        Jeans     457.0         unders         9      1200
         8                       lsjxcu   32381.0      gqgnqluho         5      1201
         9                         cvcz    2347.0           fqbuy         8      1202
        10                         gdui   20645.0         qouphh         6      1203
        12                      CoolHat     144.0           hats         3      1203
        13                            t   56185.0    accessories         6      1208
        14                      shwhtetj     266.0       plwgjdcov        10      1203
        15                         mhxal   16360.0        lfykfqlh         5      1204
PS D:\3course\БД3У\Lab3_>
```

Вставка запису:

```
PS D:\3course\БД3У\Lab3_> python main.py insert_record Product 11 test 5690 unisex 8 1206
PS D:\3course\БД3У\Lab3_> python main.py print_table Product
Product table:
id_product                        title     price       category  id_catalog    id_order
         1                 Dior handbag   10000.0           bags         4      1200
         2           Hogo Boss trousers   20000.0         unders         7      1201
         3                Armani jacket   15000.0           tops         9      1201
         4              Moscino T-Shirt     500.0           tops         6      1202
         5          Dolce&Gabbana skirt     300.0         unders         6      1202
         6            Louis Vuitton bag  100500.0           bags         4      1202
         7                        Jeans     457.0         unders         9      1200
         8                       lsjxcu   32381.0      gqgnqluho         5      1201
         9                         cvcz    2347.0           fqbuy         8      1202
        10                         gdui   20645.0         qouphh         6      1203
        11                         test    5690.0         unisex         8      1206
        12                      CoolHat     144.0           hats         3      1203
        13                            t   56185.0    accessories         6      1208
        14                      shwhtetj     266.0       plwgjdcov        10      1203
        15                         mhxal   16360.0        lfykfqlh         5      1204
PS D:\3course\БД3У\Lab3_>
```

Редагування запису:

```
PS D:\3course\БД3У\Lab3_> python main.py update_record Product 13 parampampam 0 blabla 10 1203
PS D:\3course\БД3У\Lab3_> python main.py print_table Product
Product table:
id_product                        title     price       category  id_catalog    id_order
         1                  Dior handbag   10000.0           bags         4      1200
         2            Hogo Boss trousers   20000.0         unders         7      1201
         3                 Armani jacket   15000.0           tops         9      1201
         4               Moscino T-Shirt     500.0           tops         6      1202
         5           Dolce&Gabbana skirt     300.0         unders         6      1202
         6             Louis Vuitton bag  100500.0           bags         4      1202
         7                         Jeans     457.0         unders         9      1200
         8                        lsjxcu   32381.0      gqgnqluho         5      1201
         9                          cvcz    2347.0           fqbuy         8      1202
        10                          gdui   20645.0         qouphh         6      1203
        11                          test    5690.0         unisex         8      1206
        12                       CoolHat     144.0           hats         3      1203
        13                    parampampam       0.0         blabla        10      1203
        14                       shwhtetj     266.0      plwgjdcov        10      1203
        15                          mhxal   16360.0       lfykfqlh         5      1204
```

Запити пошуку та генерації рандомізованих даних також було реалізовано, логіку пошуку було змінено у порівнянні з лабораторною роботою №2 (усі дані для пошуку передвизначено, тепер вони не вводяться з клавіатури). Запити на пошук ті самі, що і л.р. №2.

Запит на генерацію даних продемонструємо на прикладі таблиці Order.

Початковий стан:

```
PS D:\3course\БД3У\Lab3_> python main.py print_table Order
Order table:
  id_order            customer_name   id_shop                        date
      1200               Jane Olsen      9657           2021-09-15 21:25:00
      1201             Victor Crump      9657           2021-09-15 22:03:30
      1202         Elizabeth Taylor      9657           2021-09-16 09:15:24
      1203                     Anna      9658           2020-12-31 00:00:01
      1204                   pfuufcr      9658           2020-01-24 05:13:07.997146
      1205                  ktvkuceq      9658           2019-03-10 00:33:02.442442
      1206                     hmbvy      9657           2020-06-29 12:28:35.424963
      1208                   hrxlzbp      9658           2019-04-18 02:37:39.638047
      1209                       ykm      9659           2019-12-30 12:14:24.794337
      1210                   fsmcdrq      9661           2020-02-26 14:57:52.981811
PS D:\3course\БД3У\Lab3_>
```

Вставка 3-х випадково згенерованих записів:

```
PS D:\3course\БД3У\Lab3_> python main.py generate_randomly Order 3
PS D:\3course\БД3У\Lab3_> python main.py print_table Order
Order table:
  id_order            customer_name   id_shop                    date
      1200               Jane Olsen      9657          2021-09-15 21:25:00
      1201             Victor Crump      9657          2021-09-15 22:03:30
      1202         Elizabeth Taylor      9657          2021-09-16 09:15:24
      1203                     Anna      9658          2020-12-31 00:00:01
      1204                   pfuufcr     9658          2020-01-24 05:13:07.997146
      1205                  ktvkuceq     9658          2019-03-10 00:33:02.442442
      1206                     hmbvy     9657          2020-06-29 12:28:35.424963
      1208                   hrxlzbp     9658          2019-04-18 02:37:39.638047
      1209                       ykm     9659          2019-12-30 12:14:24.794337
      1210                   fsmcdrq     9661          2020-02-26 14:57:52.981811
      1211                 gheletrxj     9662          2019-02-12 18:41:17.781046
      1212                    bpizgc     9659          2019-08-11 14:30:20.814744
      1213                     jtofr     9662          2019-08-22 03:47:07.736142
```

Пошук за трьома атрибутами у двох таблицях, за трьома атрибутами у трьох таблицях, за чотирма атрибутами у чотирьох таблицях (виводяться відповідні записи з таблиці Product):

```
PS D:\3course\БД3У\Lab3_> python main.py search_records
specify the number of tables you`d like to search in: 2
search result:
         4               Moscino T-Shirt     500.0          tops         6      1202
         5           Dolce&Gabbana skirt     300.0         unders        6      1202
        10                          gdui   20645.0         qouphh        6      1203
PS D:\3course\БД3У\Lab3_> python main.py search_records
specify the number of tables you`d like to search in: 3
search result:
         3                 Armani jacket   15000.0          tops         9      1201
         7                         Jeans     457.0         unders        9      1200
PS D:\3course\БД3У\Lab3_> python main.py search_records
specify the number of tables you`d like to search in: 4
search result:
         6             Louis Vuitton bag  100500.0          bags         4      1202
         1                  Dior handbag   10000.0          bags         4      1200
```

Завдання №2

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

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:

```
DROP TABLE IF EXISTS "gin_test";
CREATE TABLE "gin_test"("id" bigserial PRIMARY KEY, "string" text, "gin_vector"
tsvector);
INSERT INTO "gin_test"("string") SELECT substr(characters, (random() *
length(characters) + 1)::integer, 10) FROM
(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as
symbols(characters), generate_series(1, 1000000) as q;
UPDATE "gin_test" set "gin_vector" = to_tsvector("string");
```

Запити для тестування:

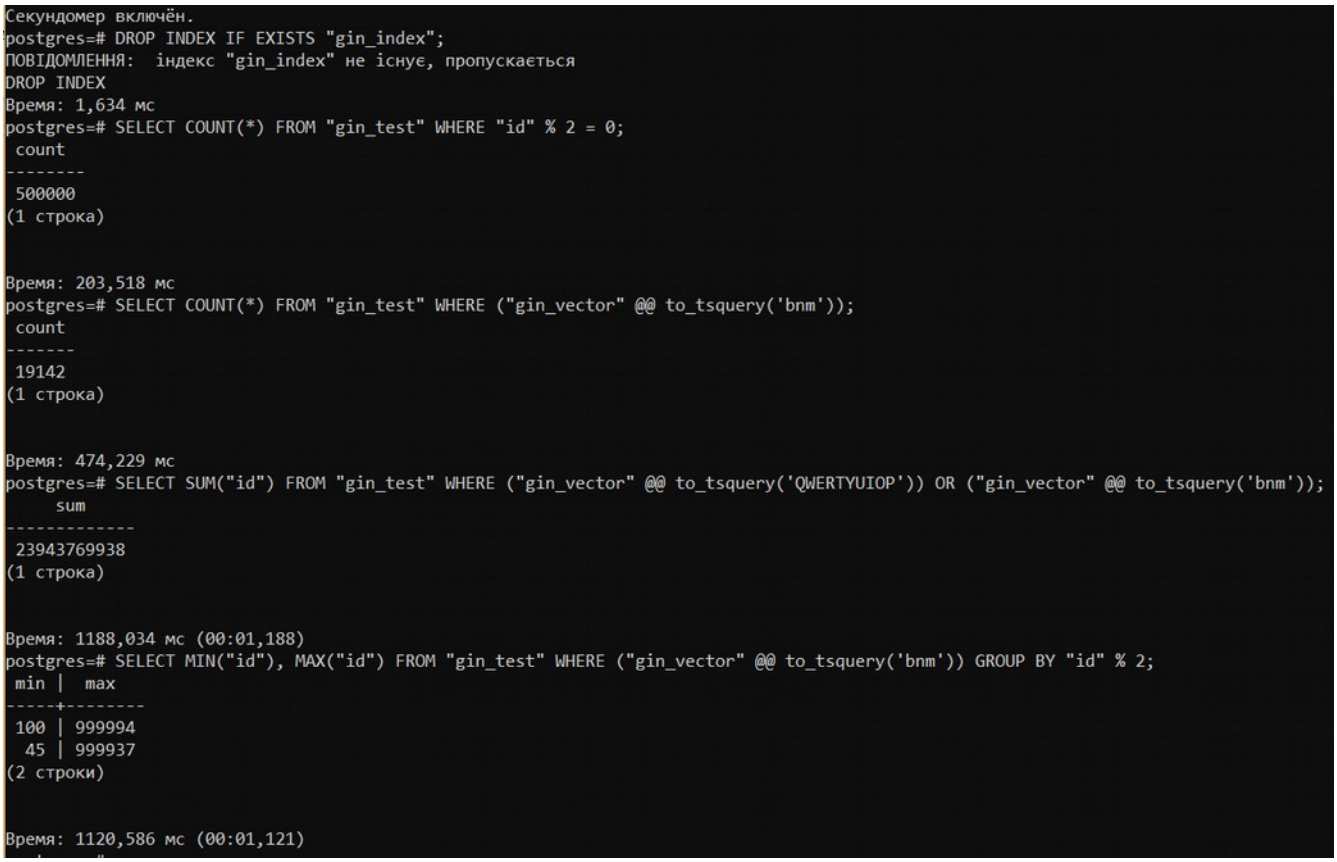
```
SELECT COUNT(*) FROM "gin_test" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm'));
SELECT SUM("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('QWERTYUIOP')) OR
("gin_vector" @@ to_tsquery('bnm'));
SELECT MIN("id"), MAX("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm'))
GROUP BY "id" % 2;
```

Створення індексу:

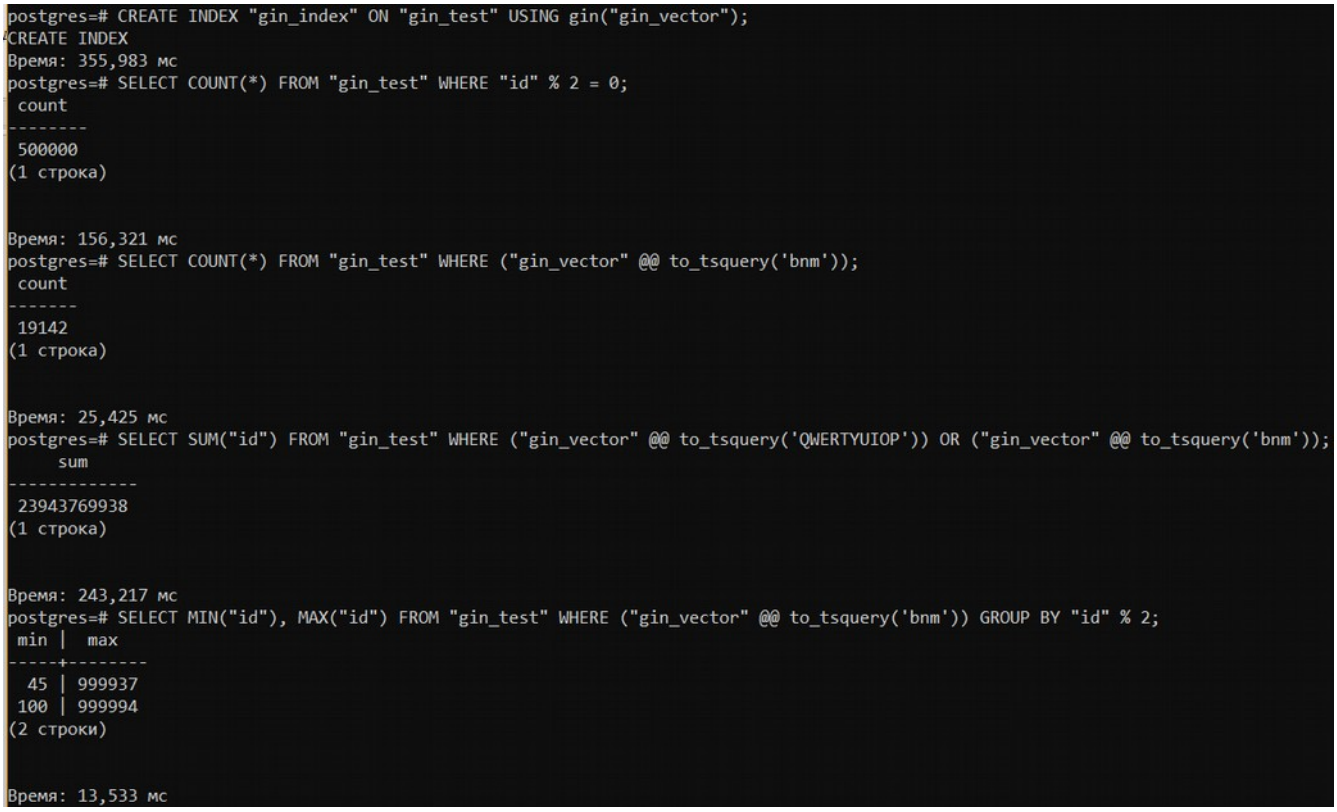
```
DROP INDEX IF EXISTS "gin_index";
CREATE INDEX "gin_index" ON "gin_test" USING gin("gin_vector");
```

Результати і час виконання на скріншотах з psql.exe

Запити без індексування:



Запити з індексуванням:



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.

Hash

Хеш-індекси в PostgreSQL використовують форму структури даних хеш-таблиці (використовують хеш-функцію). Хеш-коди поділені на обмежену кількість комірок. Коли до індексу додається нове значення, PostgreSQL застосовує хеш-функцію до значення і поміщає хеш-код і вказівник на кортеж у відповідну комірку. Коли відбувається запит за допомогою індексу хешування, PostgreSQL бере значення індексу і застосовує хеш-функцію, щоб визначити, яка комірка може містити потрібні дані.

Створення таблиці БД:

```
DROP TABLE IF EXISTS "hash_test";
CREATE TABLE "hash_test"("id" bigserial PRIMARY KEY, "time" timestamp);
INSERT INTO "hash_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;
```

Запити для тестування:

```
SELECT COUNT(*) FROM "hash_test" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "hash_test" WHERE "time" >= '20191001';
SELECT AVG("id") FROM "hash_test" WHERE "time" >= '20191001' AND "time" <= '20211207';
SELECT SUM("id"), MAX("id") FROM "hash_test" WHERE "time" >= '20200505' AND "time" <=
'20210505' GROUP BY "id" % 2;
```

Створення індексу:

```
DROP TABLE IF EXISTS "hash_test";
CREATE INDEX "time_hash_index" ON "hash_test" USING hash("id");
```

Результати і час виконання на скріншотах з psql.exe

Запити без індексування:

```
Секундомер включён.
postgres=# DROP INDEX IF EXISTS "time_hash_index";
ПОВІДОМЛЕННЯ:  індекс "time_hash_index" не існує, пропускається
DROP INDEX
Время: 2,492 мс
postgres=# SELECT COUNT(*) FROM "hash_test" WHERE "id" % 2 = 0;
 count
--------
 500000
(1 строка)


Время: 111,332 мс
postgres=# SELECT COUNT(*) FROM "hash_test" WHERE "time" >= '20191001';
 count
--------
 626919
(1 строка)


Время: 163,147 мс
postgres=# SELECT AVG("id") FROM "hash_test" WHERE "time" >= '20191001' AND "time" <= '20211207';
         avg
---------------------
 500254.786621557171
(1 строка)


Время: 176,904 мс
postgres=# SELECT SUM("id"), MAX("id") FROM "hash_test" WHERE "time" >= '20200505' AND "time" <= '20210505' GROUP BY "id" % 2;
     sum     |  max
-------------+--------
 82418773060 | 999999
 82457311990 | 999994
(2 строки)


Время: 120,725 мс
```

Запити з індексуванням:

```
postgres=# CREATE INDEX "time_hash_index" ON "hash_test" USING hash("id");
CREATE INDEX
Время: 3745,561 мс (00:03,746)
postgres=# SELECT COUNT(*) FROM "hash_test" WHERE "id" % 2 = 0;
 count
--------
 500000
(1 строка)


Время: 95,619 мс
postgres=# SELECT COUNT(*) FROM "hash_test" WHERE "time" >= '20191001';
 count
--------
 626336
(1 строка)


Время: 100,444 мс
postgres=# SELECT AVG("id") FROM "hash_test" WHERE "time" >= '20191001' AND "time" <= '20211207';
         avg
---------------------
 500044.170913056251
(1 строка)


Время: 155,144 мс
postgres=# SELECT SUM("id"), MAX("id") FROM "hash_test" WHERE "time" >= '20200505' AND "time" <= '20210505' GROUP BY "id" % 2;
     sum     |  max
-------------+--------
 82587011394 | 999994
 82348391481 | 999989
(2 строки)


Время: 155,314 мс
```

Очевидно, що індексування за допомогою hash не значно пришвидшує пошук даних у табиці, а іноді навіть показує гірші результати, ніж запити без ідексування. Це випливає 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.

Завдання №3

Для тестування тригера було створено дві таблиці:

```
DROP TABLE IF EXISTS "trigger_test";
CREATE TABLE "trigger_test"(
  "trigger_testID" bigserial PRIMARY KEY,
  "trigger_testName" text
);
DROP TABLE IF EXISTS "trigger_test_log";
CREATE TABLE "trigger_test_log"(
  "id" bigserial PRIMARY KEY,
  "trigger_test_log_ID" bigint,
  "trigger_test_log_name" text
);
```

Початкові дані у таблицях:

```
INSERT INTO "trigger_test"("trigger_testName")
VALUES ('trigger_test1'), ('trigger_test2'), ('trigger_test3'), ('trigger_test4'),
('trigger_test5'), ('trigger_test6'), ('trigger_test7'), ('trigger_test8'),
('trigger_test9'), ('trigger_test10');
```

Команди, що ініціюють виконання тригера:

```
CREATE TRIGGER "before_update_delete_trigger"
BEFORE UPDATE OR DELETE ON "trigger_test"
FOR EACH ROW
EXECUTE procedure before_update_delete_func();
```

Текст тригера:

```
CREATE OR REPLACE FUNCTION before_update_delete_func() RETURNS TRIGGER as $trigger$
DECLARE
  CURSOR_LOG CURSOR FOR SELECT * FROM "trigger_test_log";
  row_ "trigger_test_log"%ROWTYPE;

BEGIN
  IF old."trigger_testID" % 2 = 0 THEN
    IF old."trigger_testID" % 3 = 0 THEN
      RAISE NOTICE 'trigger_testID is multiple of 2 and 3';
      FOR row_ IN CURSOR_LOG LOOP
        UPDATE "trigger_test_log" SET "trigger_test_log_name" = '_' ||
row_."trigger_test_log_name" || '_log' WHERE "id" = row_."id";
      END LOOP;
      RETURN OLD;
    ELSE
      RAISE NOTICE 'trigger_testID is even';
      INSERT INTO "trigger_test_log"("trigger_test_log_ID", "trigger_test_log_name")
VALUES (old."trigger_testID", old."trigger_testName");
      UPDATE "trigger_test_log" SET "trigger_test_log_name" = trim(BOTH '_log' FROM
"trigger_test_log_name");
      RETURN NEW;
    END IF;
```

```
  ELSE
    RAISE NOTICE 'trigger_testID is odd';
    FOR row_ IN CURSOR_LOG LOOP
      UPDATE "trigger_test_log" SET "trigger_test_log_name" = '_' ||
row_."trigger_test_log_name" || '_log' WHERE "id" = row_."id";
    END LOOP;
    RETURN OLD;
  END IF;
END;
$trigger$ LANGUAGE plpgsql;
```

Скріншоти зі змінами у таблицях бази даних

Початковий стан

```
SELECT * FROM "trigger_test";
```

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	2	trigger_test2
3	3	trigger_test3
4	4	trigger_test4
5	5	trigger_test5
6	6	trigger_test6
7	7	trigger_test7
8	8	trigger_test8
9	9	trigger_test9
10	10	trigger_test10

```
SELECT * FROM "trigger_test_log";
```

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text

Після виконання запиту на оновлення

```
UPDATE "trigger_test" SET "trigger_testName" = "trigger_testName" || '_log' WHERE
"trigger_testID" % 2 = 0;
```

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	3	trigger_test3
3	5	trigger_test5
4	7	trigger_test7
5	9	trigger_test9
6	2	trigger_test2_log
7	4	trigger_test4_log
8	6	trigger_test6
9	8	trigger_test8_log
10	10	trigger_test10_log

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text
1	1	2	trigger_test2
2	2	4	trigger_test4
3	3	8	trigger_test8
4	4	10	trigger_test10

Наочно можемо переконатись, що виконалась та гілка алгоритму тригера, що відповідає за парні рядки (оскільки є умова для парних), а для 6 рядка він також виконався, але пішов іншою (вкладеною) гілкою алгоритму та повернув старий стан (OLD). При запиті на оновлення потрібно повертати новий стан, а при запиті а видалення старий.

Після виконання запиту на видалення

```
DELETE FROM "trigger_test" WHERE "trigger_testID" % 3 = 0;
```

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	2	trigger_test2
3	4	trigger_test4
4	5	trigger_test5
5	7	trigger_test7
6	8	trigger_test8
7	10	trigger_test10

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text

Якщо виконувати ці запити окремо одне від одного, то у таблиці trigger_test видаляються кратні трьом рядки, але таблиця trigger_test_log виявляється пустою. Так відбувається тому, що у гілці алгоритму для чисел кратних трьом у trigger_test_log лише модифікуються існуючі записи, але нові не додаються. Оскільки до цього не було виконано оновлення, ця таблиця пуста і модифікувати нема чого.

Якщо зробити вищезгадані запити підряд побачимо наступне:

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	5	trigger_test5
3	7	trigger_test7
4	2	trigger_test2_log
5	4	trigger_test4_log
6	8	trigger_test8_log
7	10	trigger_test10_log

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text
1	1	2	___trigger_test2_log_log_log
2	2	4	___trigger_test4_log_log_log
3	3	8	___trigger_test8_log_log_log
4	4	10	___trigger_test10_log_log_log

Бачимо, що записи кратні трьом видалились з trigger_test, а до текстових полів цих записів у кінці додалось "_log".

До текстових полів trigger_test_log на початку додались два вимволи "_", а в кінці три "_log". Один "_log" в кінці додався завдяки виконанню запиту update для всіх парних рядків. А інші два "_log" та два символи "_" на початку додались тому, що запит на видалення для записів 3 та 9 виконались за тією самою гілкою алгоритму (кратні трьом), а запит на видалення запису 6 виконався за іншою гілкою (кратність 2 та 3).

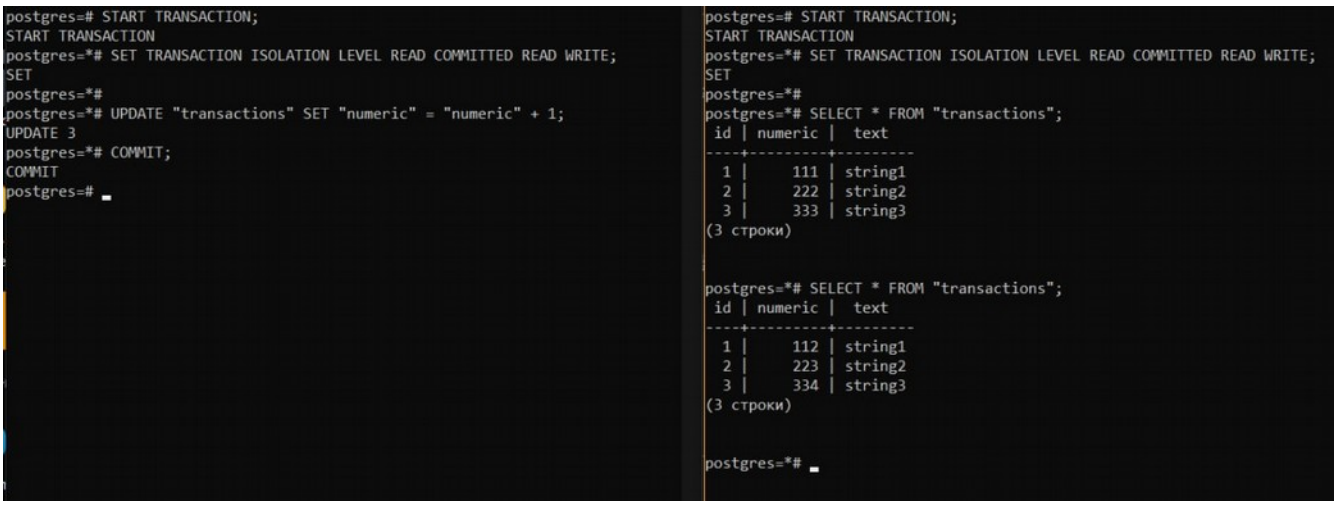
Завдання №4

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

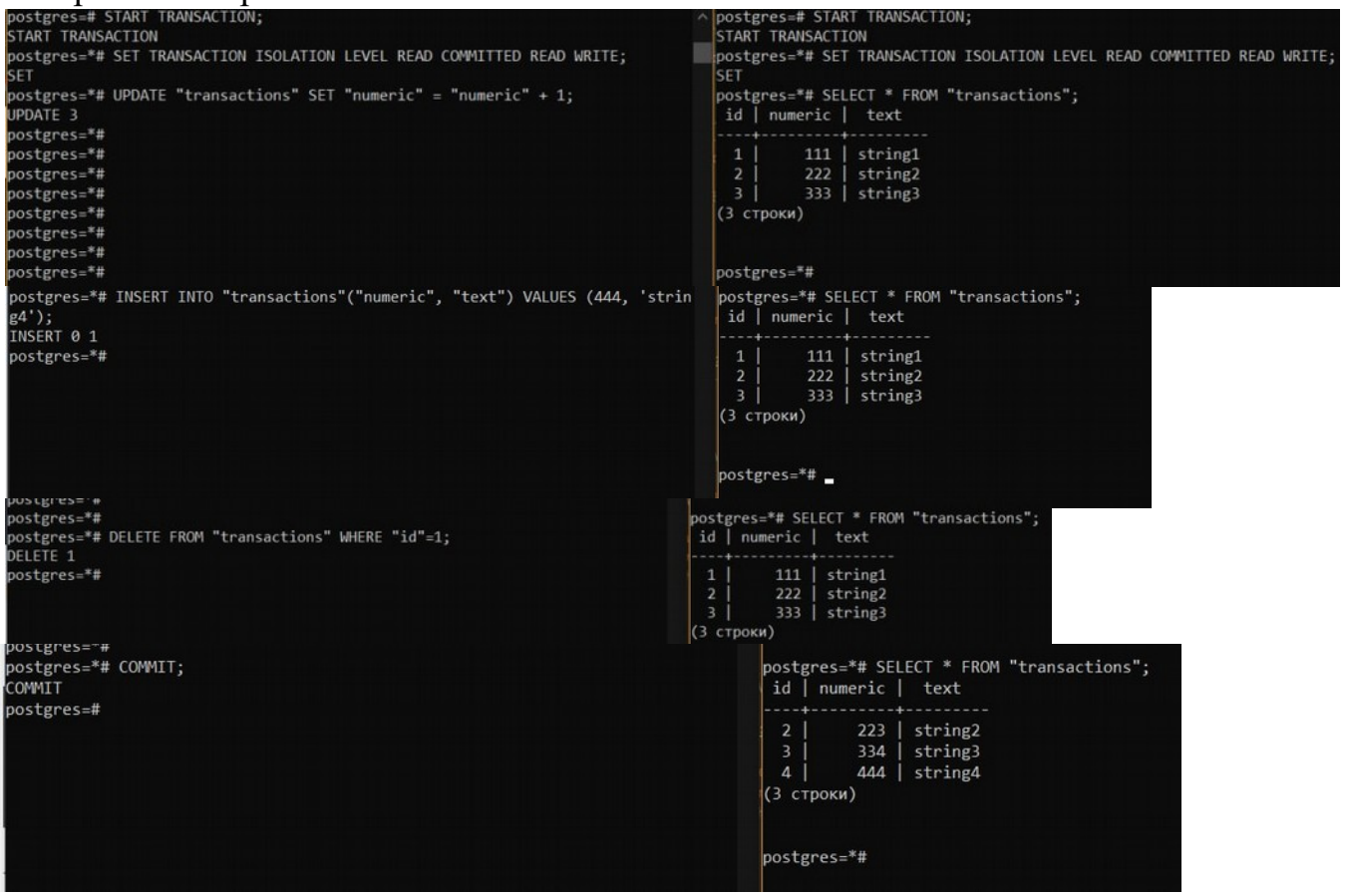
```
DROP TABLE IF EXISTS "transactions";
CREATE TABLE "transactions"(
  "id" bigserial PRIMARY KEY,
  "numeric" bigint,
  "text" text
);

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

READ COMMITTED

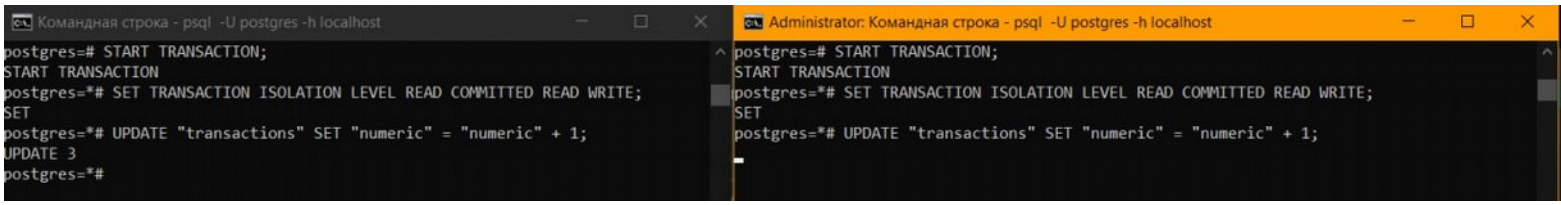
На цьому рівні ізоляції одна транзакція не бачить змін у базі даних, викликаних іншою доки та не завершить своє виконання (командою COMMIT або ROLLBACK).



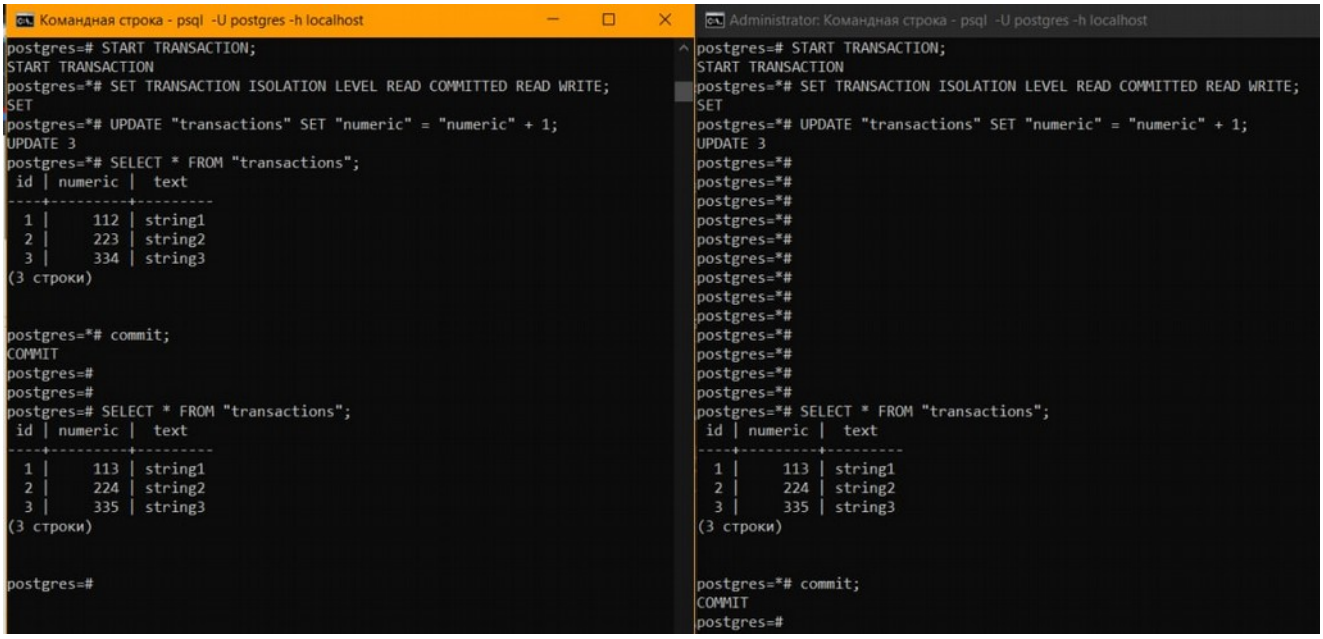
Дані після вставки та видалення так само будуть видні другій тільки після завершення першої.



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



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

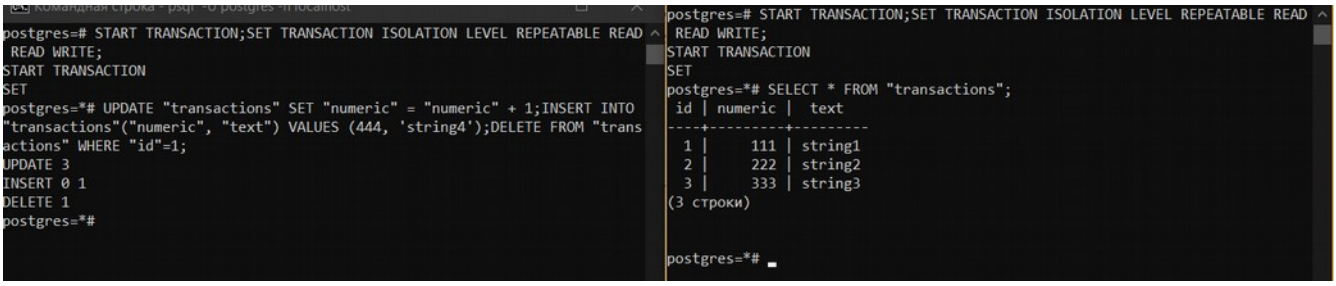


Коли T2 бачить дані T1 запитів UPDATE, DELETE виникає феномен повторного читаня, а коли бачить дані запиту INSERT – читання фантомів. Цей рівень ізоляції забезпечує захист від явища брудного читання.

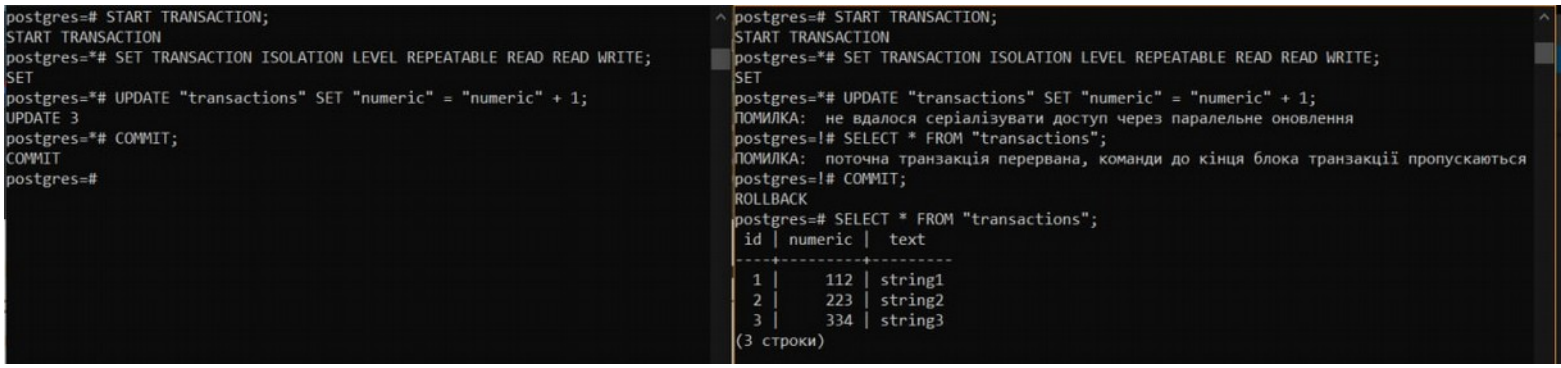
REPEATABLE READ

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:



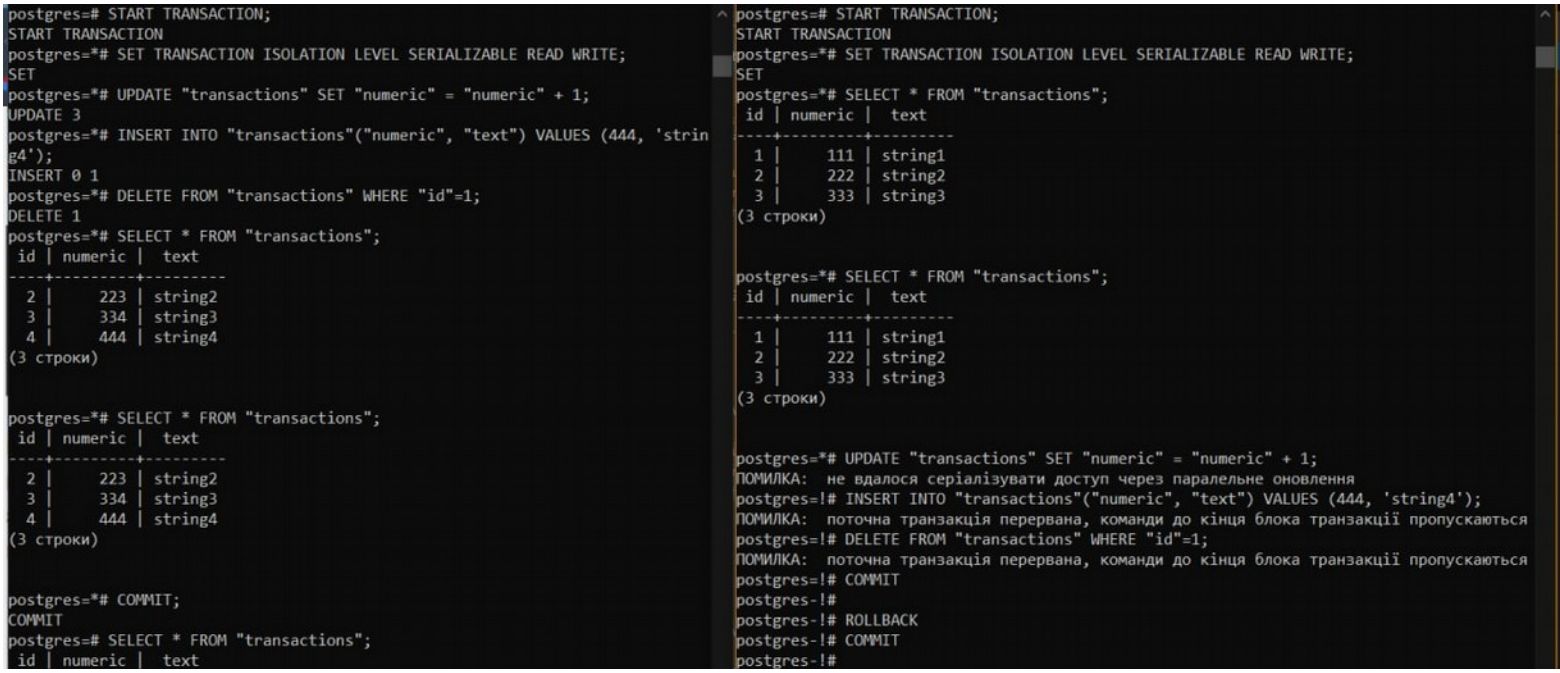
А тут, що отримуємо помилку при спробі доступу до тих самих даних:



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 транзакіції поводять 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, 'strin
g4');
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
```

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

У попередньому випадку вдалось “відкатити” другу тразакцію 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.