

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”*

Виконав: студент 3 курсу

групи КВ-91 Зиков А.С. Перевірив: Павловський В. І.

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* |

**Інформація про програму**

Посилання на репозиторій у GitHub з вихідним кодом програми та звітом: <https://github.com/Luritron/Data-Base-KV-91-Zykov-Andrii>

Використана мова програмування: Python 3.10 (IDE PyCharm)

Використані бібліотеки: psycopg2, time, sqlalchemy

# Завдання №1

**Модель «сутність-зв’язок» для галузі сервіс “питання-відповіді”**

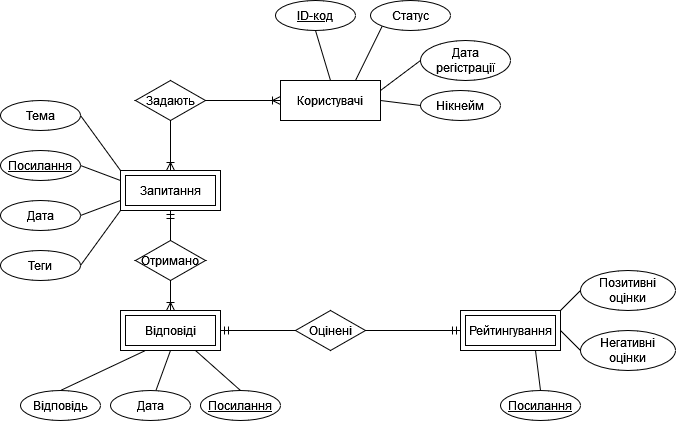


Рисунок 1 - ER-діаграма побудована за нотацією Чена

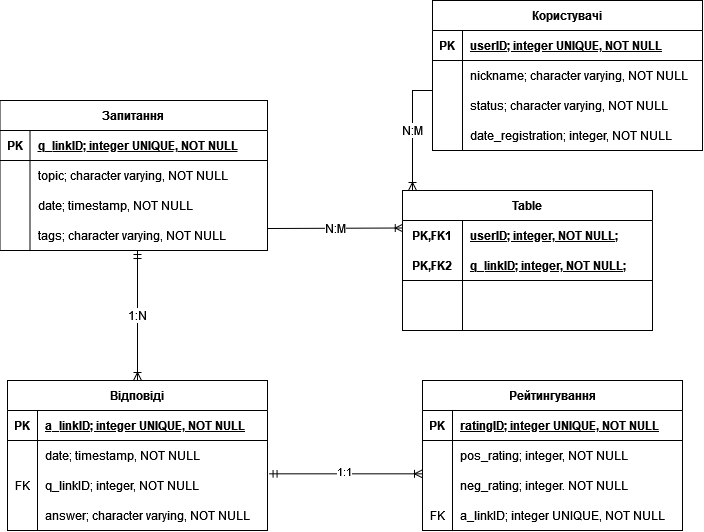


Рисунок 2 - Схема бази даних

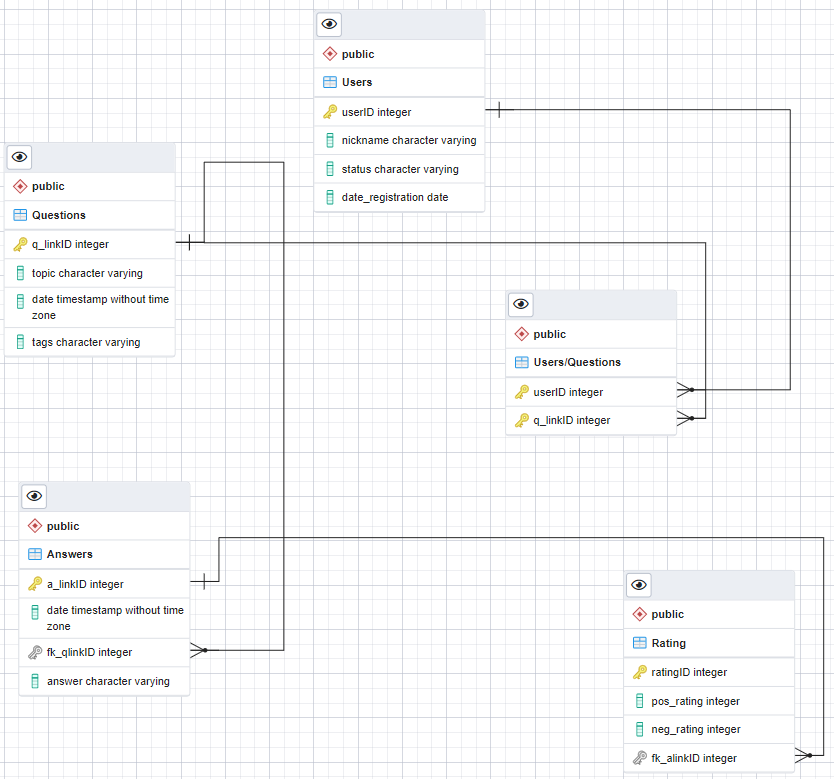
****

Рисунок 3 - Схема бази даних у pgAdmin4

Таблиця 1 - Опис структури бази даних

|  |  |  |
| --- | --- | --- |
| Сутність | Атрибут | Тип атрибуту |
| **Users -** *містить у собі основні дані про користувача* | **userID -** *унікальний ідентифікатор користувача*  **nickname -** *містить дані про найменування користувача*  **status -** *містить дані, чи перебуває зараз користувач на сайті*  **date\_registration -** *містить дані про дату народження користувача* | **integer -** *числовий*  **character varying -** *рядок*  **character varying -** *рядок*  **integer -** *числовий* |
| **Users/Questions -** *містить у собі проміжні дані про ключі кожної з відповідних таблиць (використовується для реалізації N:M)* | **userID -** *ідентифікатор користувача, що відсилає нас на відповідну таблицю Users*  **q\_linkID -** *ідентифікатор запитання, що відсилає нас на відповідну таблицю Questions* | **integer -** *числовий*  **integer -** *числовий* |
| **Questions -** *містить у собі основні дані про створене запитання* | **q\_linkID -** *унікальний ідентифікатор запитання*  **topic -** *містить дані про тему запитання*  **date -** *містить дані про час коли було створено запитання*  **tags -** *містить короткі дані за якими простіше знаходити спорідненні запитання* | **integer -** *числовий*  **character varying -** *рядок*  **timestamp without time zone** - *мітка часу*  **character varying -** *рядок* |
| **Answers -** *містить у собі основні дані про відповідь дану на запитання* | **a\_linkID -** *унікальний ідентифікатор відповіді*  **date -** *містить дані про час коли було створено відповідь*  **fk\_qlinkID -** *ідентифікатор запитання, на яке дано відповідь*  **answer -** *містить дані про надану на запитання відповідь* | **integer -** *числовий*  **timestamp without time zone** - *мітка часу*  **integer -** *числовий*  **character varying -** *рядок* |
| **Rating -** *містить у собі дані про оцінене запитання* | **ratingID -** *унікальний ідентифікатор оцінки*  **pos\_rating -** *позитивна оцінка*  **neg\_rating -** *негативна оцінка*  **fk\_alinkID -** *ідентифікатор відповіді, яка була оцінена* | **integer -** *числовий*  **integer -** *числовий*  **integer -** *числовий*  **integer -** *числовий* |

Схема відповідає 1НФ, тому що:

1) Кожна таблиця має основний ключ (Primary key, PK) та мінімальний набір атрибутів, які ідентифікують запис;

2) Кожен атрибут повинен має лише одне значення, а не множину значень.

Схема відповідає 2НФ, тому що:

1) Вона відповідає вимогам 1НФ;

2) Кожен не ключовий атрибут функціонально залежить від основного атрибуту.

Схема відповідає 3НФ, а саме:

1) Вона відповідає вимогам 2НФ;

2) Дані в таблиці залежать винятково від основного ключа

Класи ORM у реалізованому модулі Model

class Users(base):

\_\_tablename\_\_ = "Users"

userID = Column(Integer, primary\_key=True, nullable=False)

nickname = Column(String(50), nullable=False)

status = Column(String(50), nullable=False)

date\_registration = Column(Date, nullable=False)

def \_\_init\_\_(self, nickname, status, date\_registration):

self.nickname = nickname

self.status = status

self.date\_registration = date\_registration

format\_str = "{:^12}{:^40}{:^40}{:^30}"

def \_\_repr\_\_(self):

return self.format\_str.format(self.userID, self.nickname, self.status, str(self.date\_registration))

def \_\_attributes\_print\_\_(self):

return self.format\_str.format("userID", "nickname", "status", "date\_registration")

class Questions(base):

\_\_tablename\_\_ = "Questions"

q\_linkID = Column(Integer, primary\_key=True, nullable=False)

topic = Column(String(500), nullable=False)

date = Column(Date, nullable=False)

tags = Column(String(50), nullable=False)

def \_\_init\_\_(self, topic, date, tags):

self.tags = tags

self.topic = topic

self.date = date

format\_str = "{:^15}{:^100}{:^30}{:^40}"

def \_\_repr\_\_(self):

return self.format\_str.format(self.q\_linkID, self.topic, str(self.date), self.tags)

def \_\_attributes\_print\_\_(self):

return self.format\_str.format("q\_linkID", "topic", "date", "tags")

class Users\_Questions(base):

\_\_tablename\_\_ = "Users/Questions"

userID = Column(Integer, ForeignKey("Users.userID"), primary\_key=True, nullable=False)

q\_linkID = Column(Integer, ForeignKey("Questions.q\_linkID"), primary\_key=True, nullable=False)

Users = relationship("Users")

Questions = relationship("Questions")

format\_str = "{:^12}{:^12}"

def \_\_repr\_\_(self):

return self.format\_str.format(self.userID, self.q\_linkID)

def \_\_attributes\_print\_\_(self):

return self.format\_str.format("userID", "q\_linkID")

class Answers(base):

\_\_tablename\_\_ = "Answers"

a\_linkID = Column(Integer, primary\_key=True, nullable=False)

date = Column(Date, nullable=False)

fk\_qlinkID = Column(Integer, ForeignKey("Questions.q\_linkID"), nullable=False)

answer = Column(String(500), nullable=False)

pos\_rating = Column(Integer, nullable=False)

neg\_rating = Column(Integer, nullable=False)

Questions = relationship("Questions")

def \_\_init\_\_(self, date, fk\_qlinkID, answer, pos\_rating, neg\_rating):

self.date = date

self.fk\_qlinkID = fk\_qlinkID

self.answer = answer

self.pos\_rating = pos\_rating

self.neg\_rating = neg\_rating

format\_str = "{:^12}{:^30}{:^15}{:^60}{:^12}{:^12}"

def \_\_repr\_\_(self):

return self.format\_str.format(self.a\_linkID, str(self.date), self.fk\_qlinkID, self.answer, self.pos\_rating, self.neg\_rating)

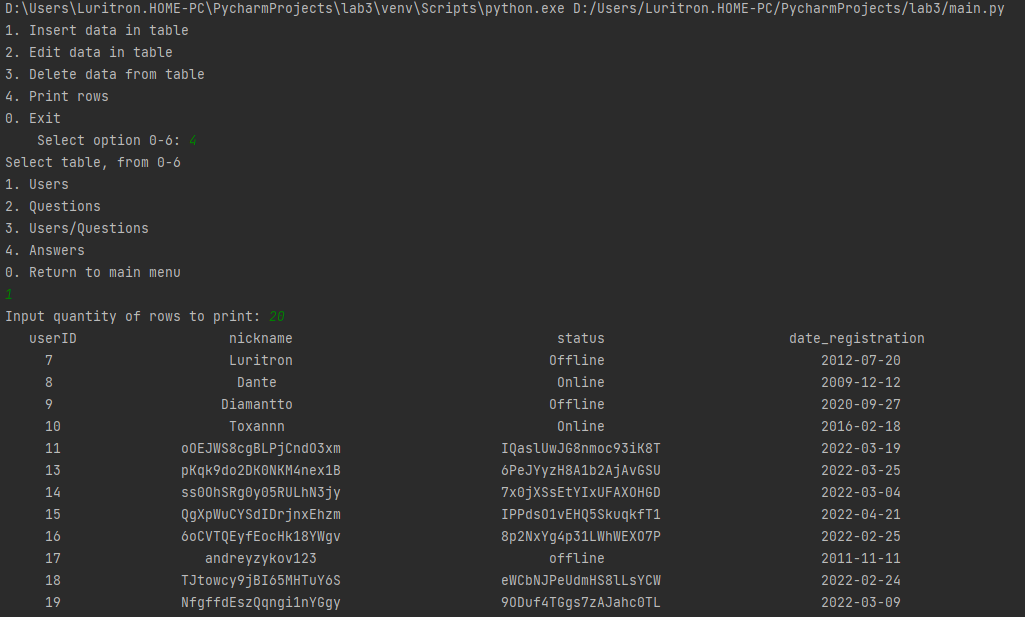
def \_\_attributes\_print\_\_(self):

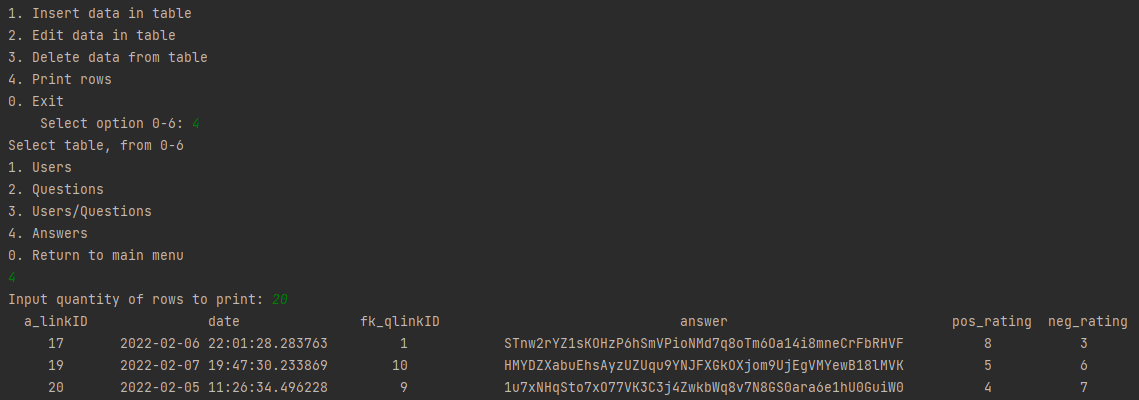
return self.format\_str.format("a\_linkID", "date", "fk\_qlinkID", "answer", "pos\_rating", "neg\_rating")

Запити у вигляді ORM

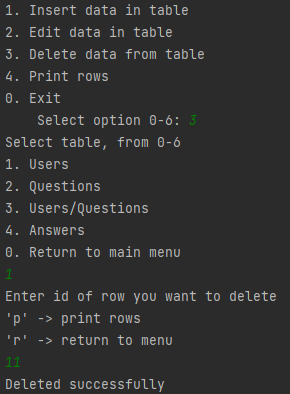
Продемонструємо вставку на прикладі Users, вилучення на прикладі Questions, редагування на прикладі Answers.

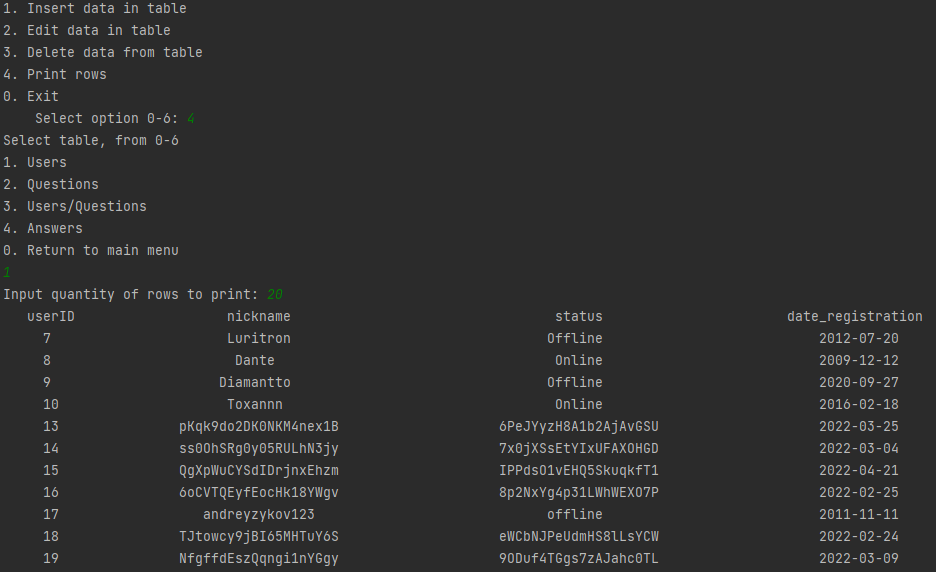
Початковий стан Users:

 Початковий стан Answers:

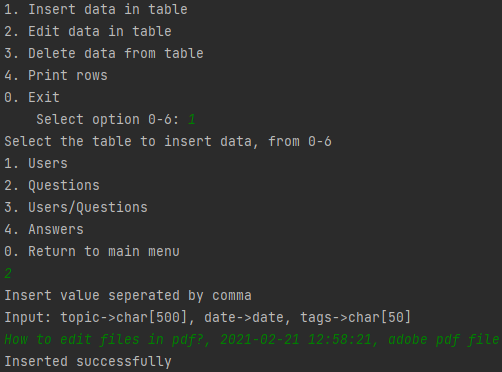
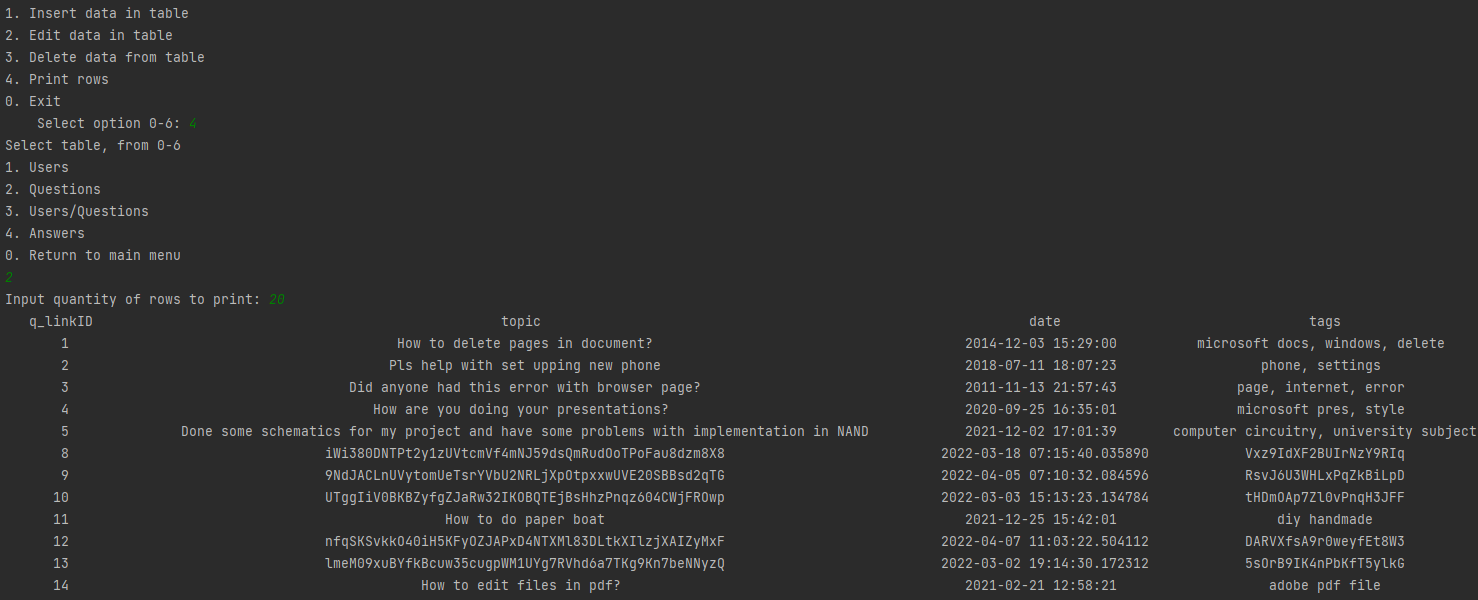


Видалення запису з Users:

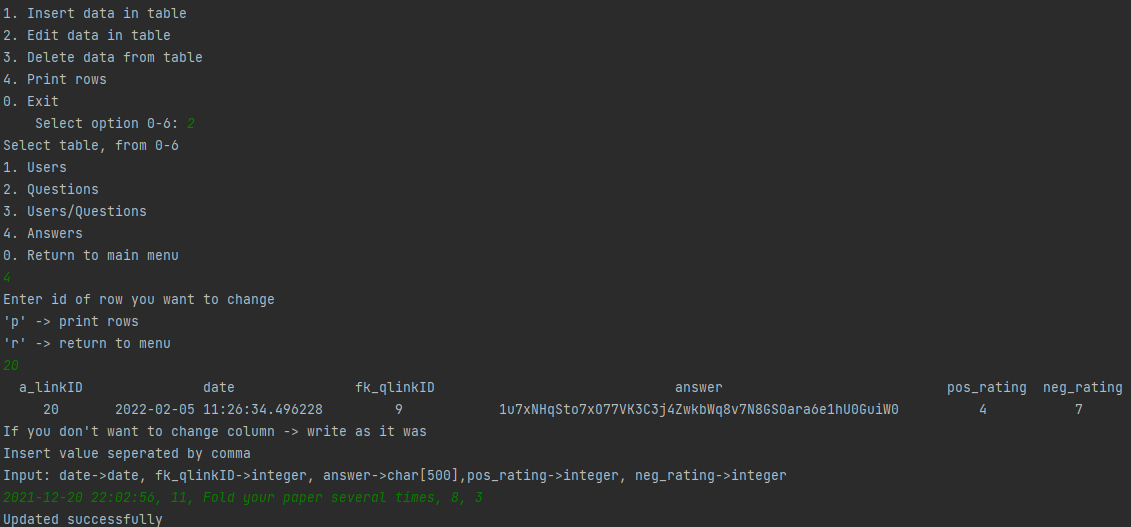


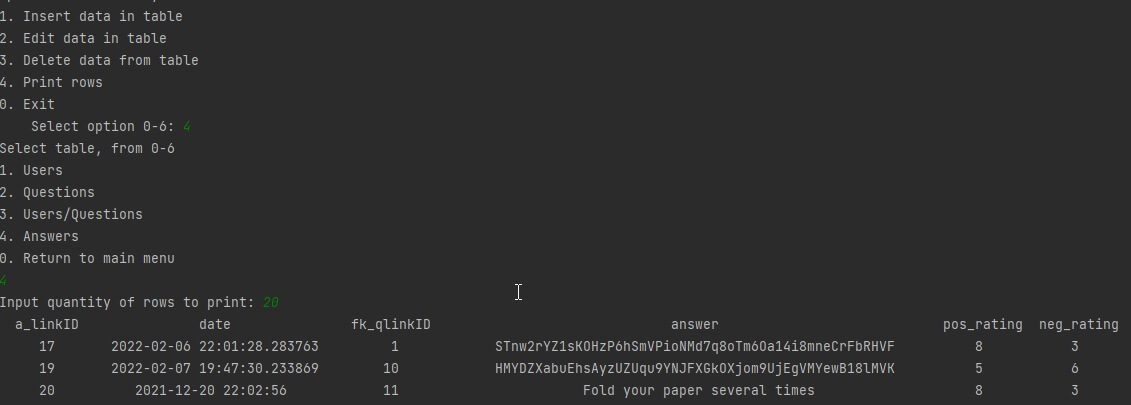


Вставка запису в Questions:



Редагування запису в Answers:





# 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.

Стверення 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;

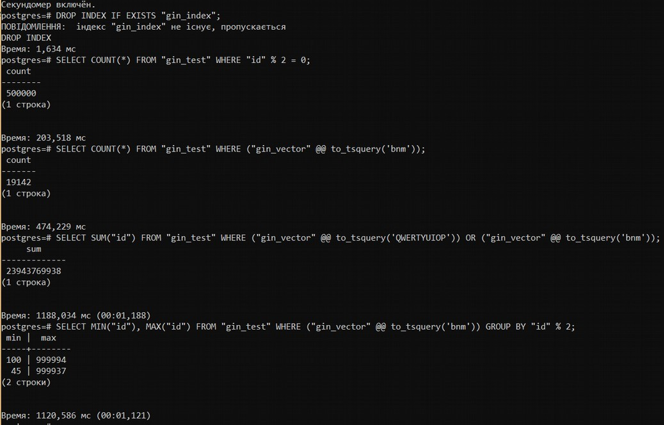
Створення індексу:

DROP INDEX IF EXISTS "gin\_index";

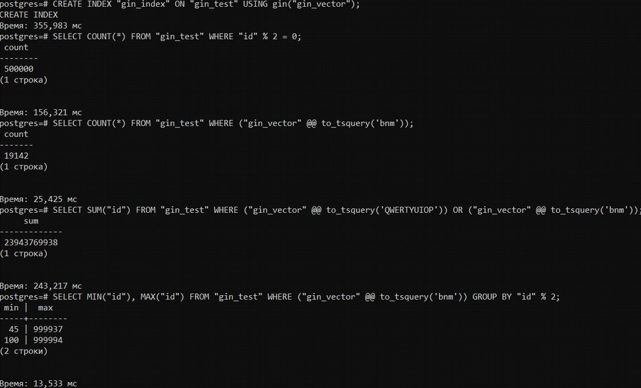
CREATE INDEX "gin\_index" ON "gin\_test" USING gin("gin\_vector");

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.

Стверення 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;

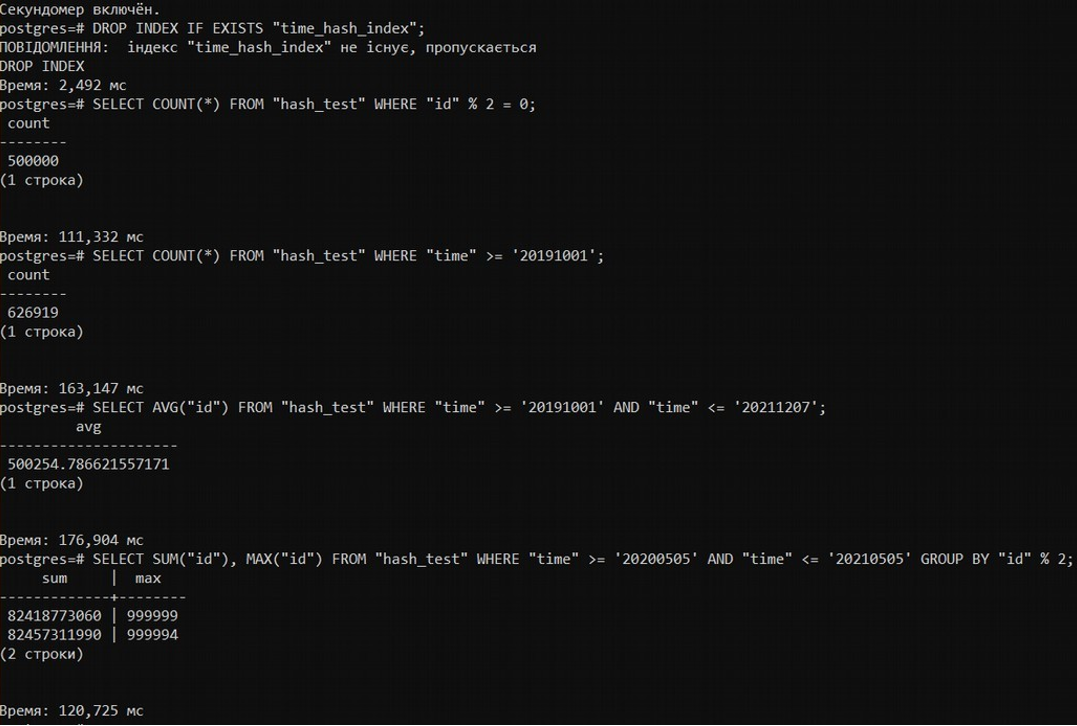
Створення індексу:

DROP TABLE IF EXISTS "hash\_test";

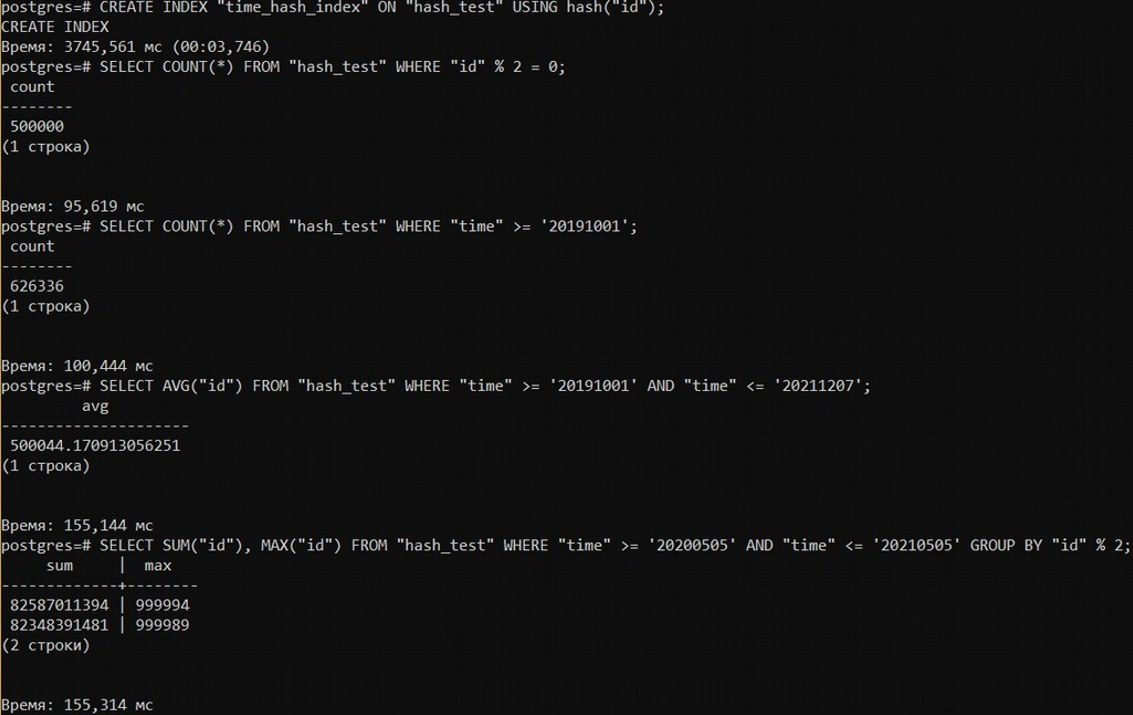
CREATE INDEX "time\_hash\_index" ON "hash\_test" USING hash("id");

Результати і час виконання на скріншотах з psql.exe

Запити без індексування:



Запити з індексуванням:



Очевидно, що індексування за допомогою hash не значно пришвидшує пошук даних у таблиці, а іноді навіть показує гірші результати, ніж запити без індексування. Це випливає з того, що це один із найпримітившіних 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 EXIST"trigger\_test";

CREATE TABLE "trigger\_test"(

"trigger\_testID" bigserial PRIMARY KEY, "trigger\_testName" text

);

DROP TABLE IF EXIST"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;

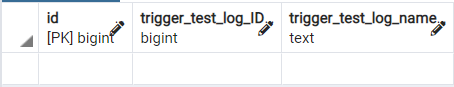
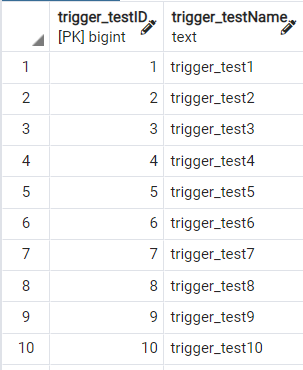
END IF;

END;

$trigger$ LANGUAGE plpgsql;

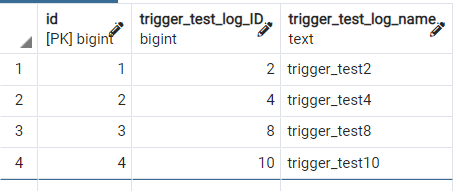
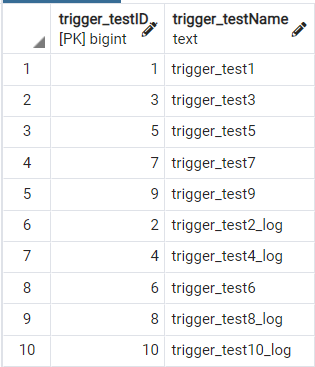
Скріншоти зі змінами у таблицях бази даних

Початковий стан

SELECT \* FROM "trigger\_test"; SELECT \* FROM "trigger\_test\_log";

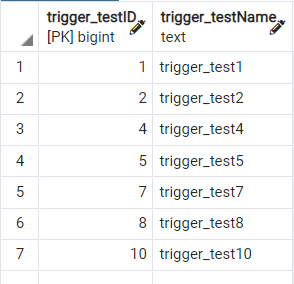
Після виконання запиту на оновлення

UPDATE "trigger\_test" SET "trigger\_testName" = "trigger\_testName" || '\_log' WHERE "trigger\_testID" % 2 = 0;

 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.

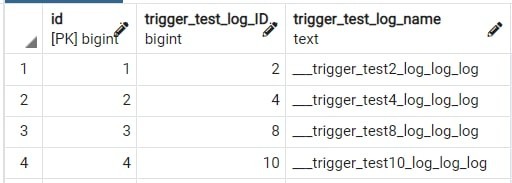
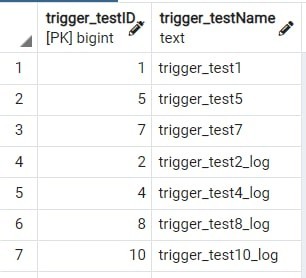
Після виконання запиту на видалення

DELETE FROM "trigger\_test" WHERE "trigger\_testID" % 3 = 0;

 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:



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 EXIST "transactions";

CREATE TABLE "transactions"(

"id" bigserial, PRIMARY KEY, "numeric" bigint,

"text" text

);

INSERT INTO "transactions"("numeric", "text")

VALUES

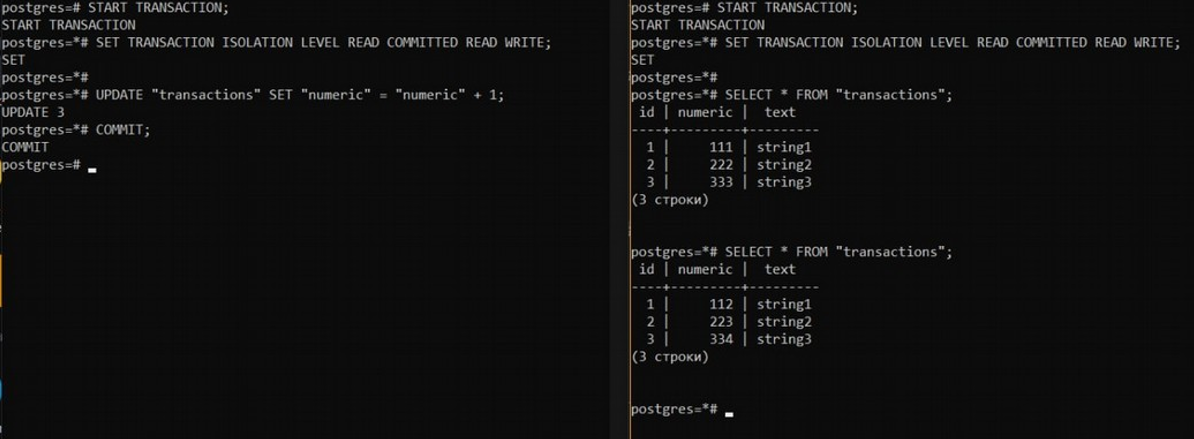
(111, 'string1'),

(222, 'string2'),

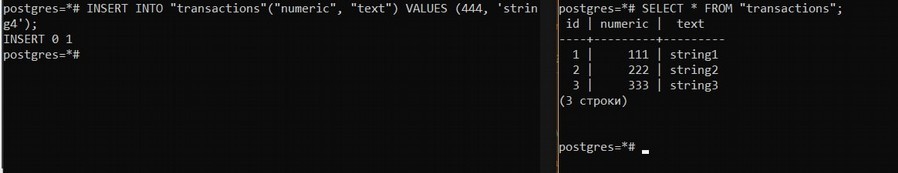
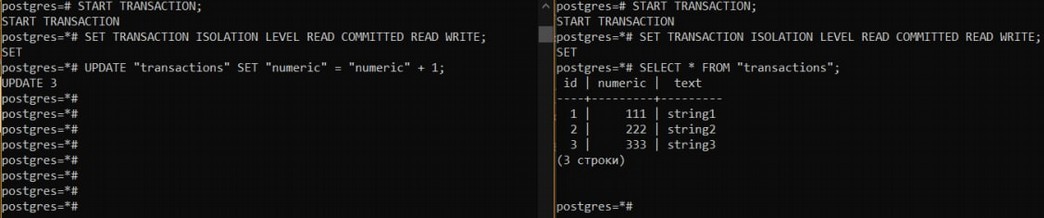
(333, 'string3');

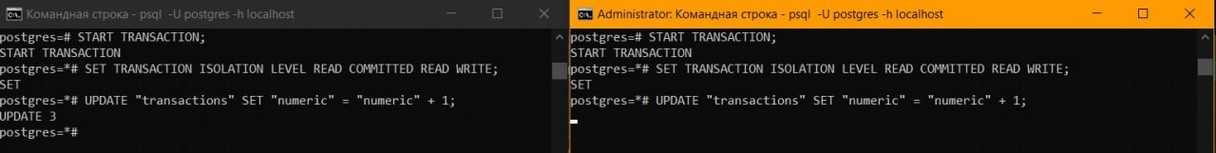
**READ COMMITED**

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

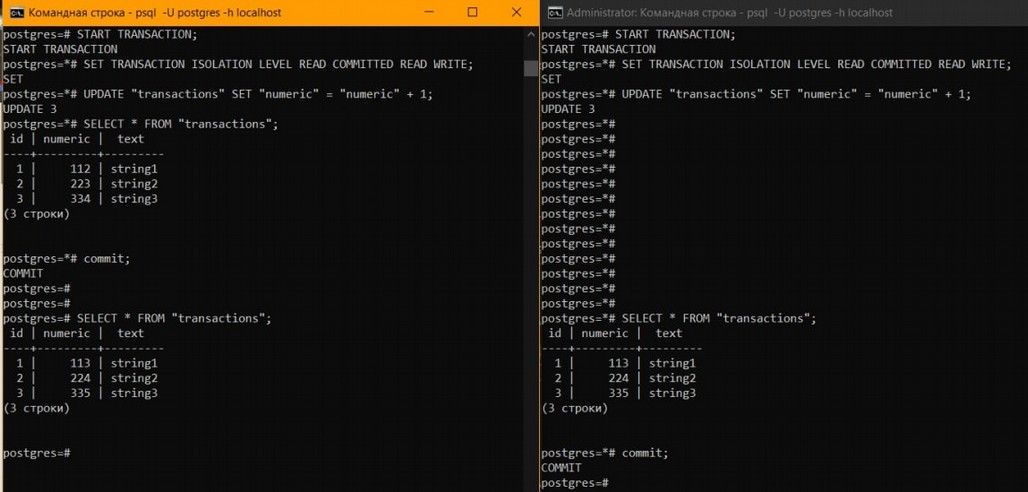


Дані після вставки та видалення так само будуть видні другій тільки після завершення першої.



 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.

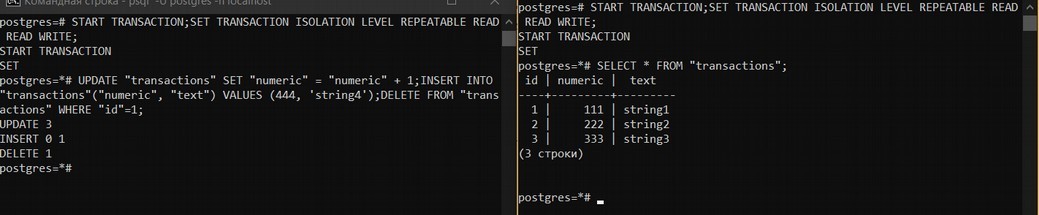


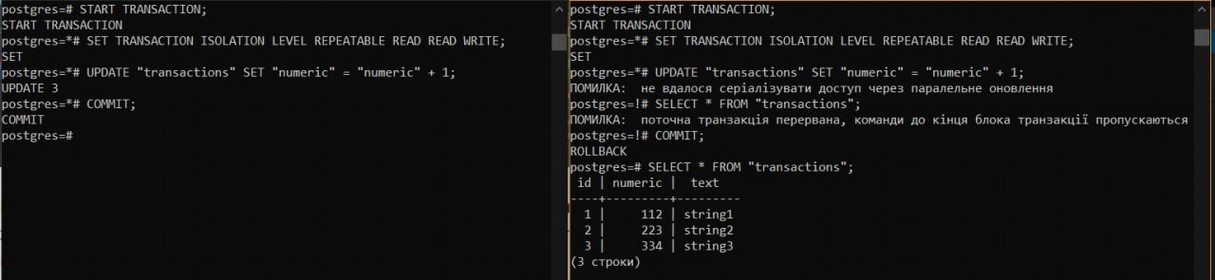
Коли Т2 бачить дані Т1 запитів UPDATE, DELETE виникає феномен повторного читаня, а коли бачить дані запиту INSERT – читання фантомів. Цей рівень ізоляції забезпечує захист від явища брудного читання.

**REPEATABLE READ**

На цьому рівні ізоляції Т2 не бачитиме змінені дані транзакцією Т1, але також не зможе отримати доступ до тих самих даних.

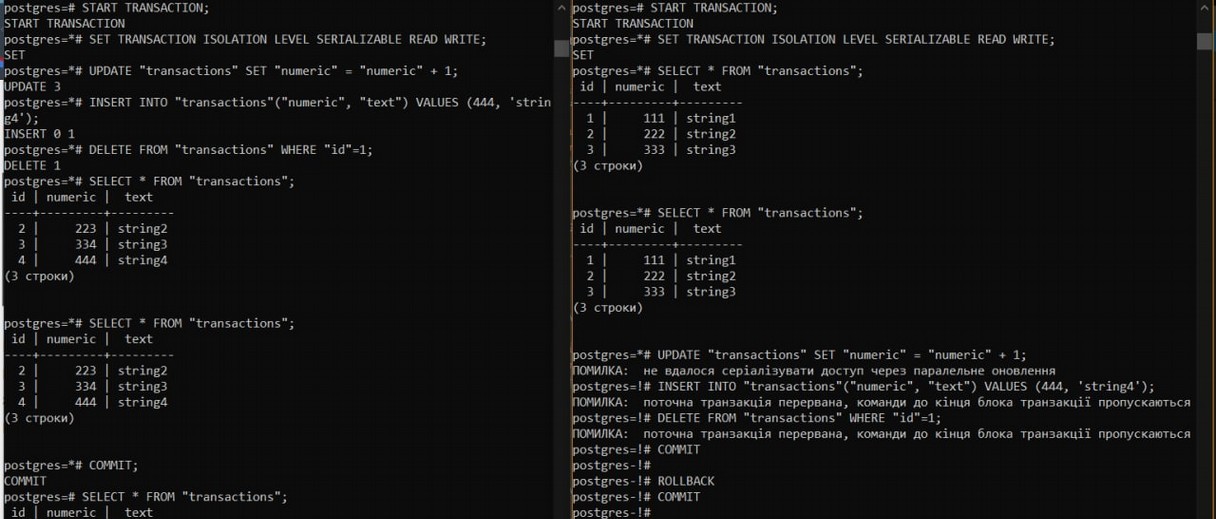
Тут видно, що друга не бачить змін з першої:



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.

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.