

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 ІII курсу

групи КВ-91

Неживий Б. М.

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 |
| *18* | *BTree, GIN* | *after update, insert* |

Посилання на репозиторій у GitHub з вихідним кодом програми та прикладеним звітом: [https://github.com/n0tark/lab-pp](https://github.com/vladsel/database)

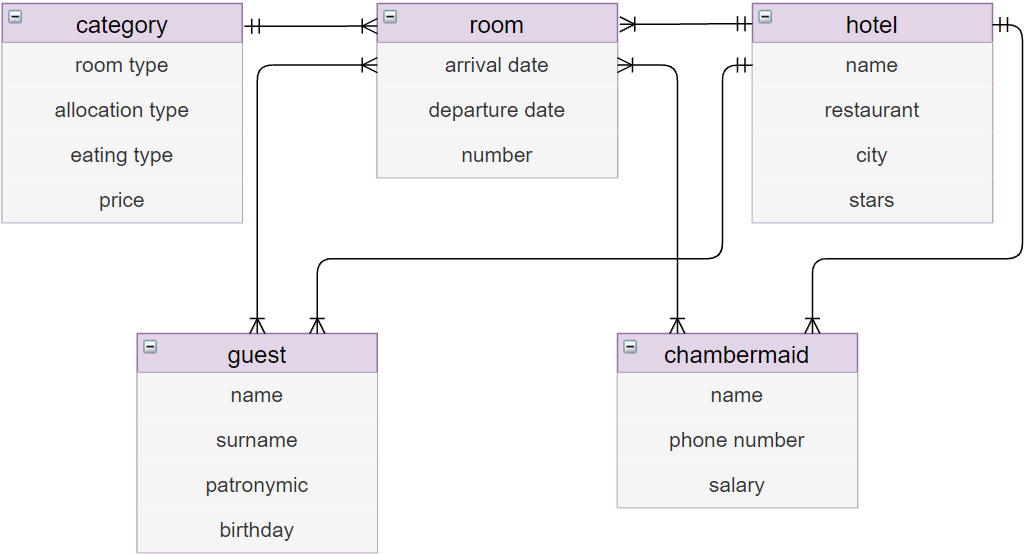
**Відомості про предметну галузь з лабораторної роботи №1**

Рисунок 1 - ER-діаграма побудована за нотацією “Пташиної лапки (Crow’s foot)”, задана ER-діаграма була побудована у додатку [draw.io](https://app.diagrams.net/)

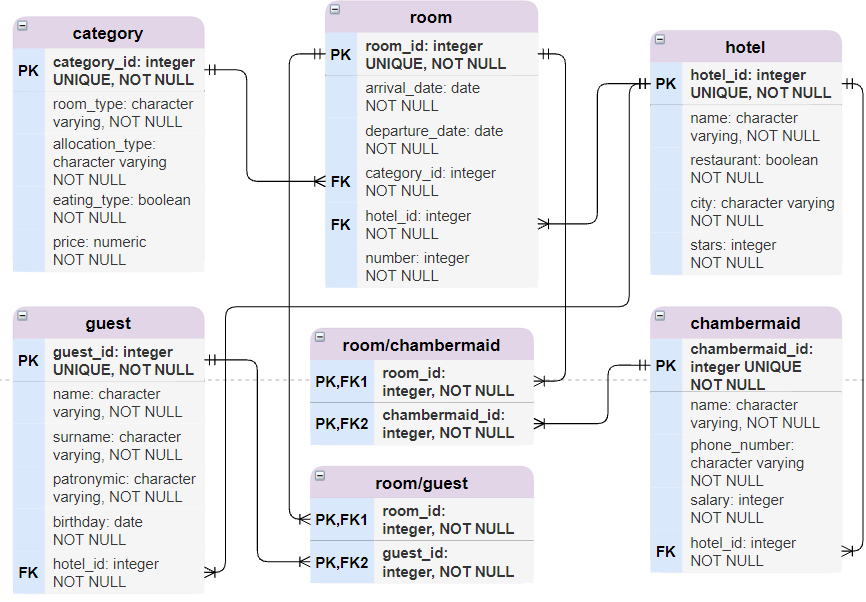
****

Рисунок 2 - Схема бази даних, побудовано у додатку [draw.io](https://app.diagrams.net/)

**Таблиця 1 - Опис структури БД.**

|  |  |  |
| --- | --- | --- |
| Відношення | Атрибут | Тип атрибуту |
| **hotel –** містить дані про готель | **hotel\_id –** унікальний ідентифікатор  **name –** назва готелю  **restaurant –** наявність ресторану  **city –** місто  **stars –** кількість зірок | **integer** (числовий)  **character varying** (рядок)  **boolean** (булевий)  **character varying** (рядок)  **integer** (числовий) |
| **category –** містить дані про категорію номеру у готелі | **category\_id** – унікальний ідентифікатор  **room\_type –** тип номеру  **allocation\_type –** тип розселення в номері  **eating\_type –** наявність харчування  **price –** ціна | **integer** (числовий)  **character varying** (рядок)  **character varying** (рядок)  **boolean** (булевий)  **numeric** (фіксований) |
| **guest –** містить дані про постояльців готелю | **guest\_id** – унікальний ідентифікатор  **name** – ім’я  **surname –** прізвище  **patronymic –** по батькові  **birthday –** день народження  **hotel\_id –** ідентифікатор готелю | **integer** (числовий)  **character varying** (рядок)  **character varying** (рядок)  **character varying** (рядок)  **date** (дата)  **integer** (числовий) |
| **room** – містить дані щодо номеру | **room\_id -** унікальний ідентифікатор  **arrival\_date –** дата заселення  **departure\_date –** дата виселення  **category\_id** – ідентифікатор категорії  **hotel\_id –** ідентифікатор готелю  **number –** номер кімнати | **integer** (числовий)  **date** (дата)  **date** (дата)  **integer** (числовий)  **integer** (числовий)  **integer** (числовий) |
| **chambermaid –** містить дані про покоївок готелю | **chambermaid\_id –** унікальний ідентифікатор  **name –** ім’я  **phone –** номер телефону  **salary –** заробітня плата  **hotel\_id –** ідентифікатор готелю | **integer** (числовий)  **character varying** (рядок)  **character varying** (рядок)  **integer** (числовий)  **integer** (числовий) |
| **room/**  **chambermaid** -  відношення покоївок до кімнат | **room\_id–** ідентифікатор номера  **chambermaid\_id –** ідентифікатор покоївки | **integer** (числовий)  **integer** (числовий) |
| **room/guest** -  відношення постояльців до кімнат | **room\_id–** ідентифікатор номера  **chambermaid\_id –** ідентифікатор постояльця | **integer** (числовий)  **integer** (числовий) |

У Обраній базі даних «Готель» можна виділити наступні таблиці: загальні відомості про готель (hotel), тип заданого номера (room), категорія номера (category), загальні відомості про постояльця (guest), інформація про покоївку (chambermaid), відношення покоївок до кімнат (room/chambermaid), відношення постояльців до кімнат (room/guest).

Стовпці заданих таблиць:

* + - 1. hotel: hotel\_id, name, restaurant, city, stars.
      2. room: room\_id, arrival date, departure date, category\_id, hotel\_id, number.
      3. category: category\_id, room type, allocation type, eating type, price.
      4. guest: guest\_id, name, surname, patronymic, birthday, hotel\_id.
      5. chambermaid: chambermaid\_id, name, phone number, salary, hotel\_id.

**Завдання №1**

**Класи ORM у реалізованому модулі Model**

class Category(base):

\_\_tablename\_\_ = 'category'

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

room\_type = Column(String(30), nullable=False)

allocation\_type = Column(String(30), nullable=False)

eating\_type = Column(Boolean, nullable=False)

def \_\_init\_\_(self, room\_type, allocation\_type, eating\_type, category\_id=-1):

self.room\_type = room\_type

self.allocation\_type = allocation\_type

self.eating\_type = eating\_type

if category\_id != -1:

self.category\_id = category\_id

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

return "{:^12}{:^15}{:^20}{:^15}".format(self.category\_id, self.room\_type, self.allocation\_type, self.eating\_type)

def \_\_str\_\_(self):

return f"{'category\_id':^12}{'room\_type':^15}{'allocation\_type':^20}{'eating\_type':^15}"

# return f"""category\_id = {self.category\_id}, room\_type = {self.room\_type}, """ \

# f"""allocation\_type = {self.allocation\_type}, eating\_type = {self.eating\_type}"""

class Hotel(base):

\_\_tablename\_\_ = 'hotel'

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

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

restaurant = Column(Boolean, nullable=False)

city = Column(String(25), nullable=False)

star = Column(Integer, nullable=False)

def \_\_init\_\_(self, name, restaurant, city, star, hotel\_id=-1):

self.name = name

self.restaurant = restaurant

self.city = city

self.star = star

if hotel\_id != -1:

self.hotel\_id = hotel\_id

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

return "{:^10}{:^15}{:^10}{:^15}{:^5}".format(self.hotel\_id, self.name, self.restaurant, self.city, self.star)

def \_\_str\_\_(self):

return f"{'hotel\_id':^10}{'name':^15}{'restaurant':^10}{'city':^15}{'star':^5}"

# return f"""hotel\_id = {self.hotel\_id}, name = {self.name}, """ \

# f"""restaurant = {self.restaurant}, city = {self.city}, star = {self.star}"""

class Chambermaid(base):

\_\_tablename\_\_ = 'chambermaid'

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

name = Column(String(100), nullable=False)

phone\_number = Column(String(20), nullable=False)

salary = Column(Integer, nullable=False)

hotel\_id = Column(Integer, ForeignKey('hotel.hotel\_id'), nullable=False)

def \_\_init\_\_(self, name, phone\_number, salary, hotel\_id, chambermaid\_id=-1):

self.name = name

self.phone\_number = phone\_number

self.salary = salary

self.hotel\_id = hotel\_id

if chambermaid\_id != -1:

self.chambermaid\_id = chambermaid\_id

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

return "{:^15}{:^15}{:^15}{:^8}{:^10}".format(self.chambermaid\_id, self.name, self.phone\_number, self.salary, self.hotel\_id)

def \_\_str\_\_(self):

return f"{'chambermaid\_id':^15}{'name':^15}{'phone\_number':^15}{'salary':^8}{'hotel\_id':^10}"

# return f"""chambermaid\_id = {self.chambermaid\_id}, name = {self.name}, """ \

# f"""phone\_number = {self.phone\_number}, salary = {self.salary}, hotel\_id = {self.hotel\_id}"""

class Guest(base):

\_\_tablename\_\_ = 'guest'

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

name = Column(String(100), nullable=False)

surname = Column(String(100), nullable=False)

patronymic = Column(String(100), nullable=False)

birthday = Column(DATETIME, nullable=False)

hotel\_id = Column(Integer, ForeignKey('hotel.hotel\_id'), nullable=False)

def \_\_init\_\_(self, name, surname, patronymic, birthday, hotel\_id, guest\_id=-1):

self.name = name

self.surname = surname

self.patronymic = patronymic

self.birthday = birthday

self.hotel\_id = hotel\_id

if guest\_id != -1:

self.guest\_id = guest\_id

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

return "{:^10}{:^15}{:^15}{:^15}\t{}{:^12}".format(self.guest\_id, self.name, self.surname, self.patronymic, self.birthday, self.hotel\_id)

def \_\_str\_\_(self):

return f"{'guest\_id':^10}{'name':^15}{'surname':^15}{'patronymic':^15}{'birthday':^12}{'hotel\_id':^10}"

# return f"""guest\_id = {self.guest\_id}, name = {self.name}, surname {self.surname}, """ \

# f"""patronymic = {self.patronymic}, birthday = {self.birthday}, hotel\_id = {self.hotel\_id}"""

class Room(base):

\_\_tablename\_\_ = 'room'

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

arrival\_date = Column(DATETIME, nullable=False)

departure\_date = Column(DATETIME, nullable=False)

category\_id = Column(Integer, ForeignKey('category.category\_id'), nullable=False)

hotel\_id = Column(Integer, ForeignKey('hotel.hotel\_id'), nullable=False)

number = Column(Integer, nullable=False)

price = Column(Float, nullable=False)

def \_\_init\_\_(self, arrival\_date, departure\_date, category\_id, hotel\_id, number, price, room\_id=-1):

self.arrival\_date = arrival\_date

self.departure\_date = departure\_date

self.category\_id = category\_id

self.hotel\_id = hotel\_id

self.number = number

self.price = price

if room\_id != -1:

self.room\_id = room\_id

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

return "{:^10}\t {}\t\t{}{:^22}{:^6}{:^8}{:^10}".format(self.room\_id, self.arrival\_date, self.departure\_date,

self.category\_id, self.hotel\_id, self.number, self.price)

def \_\_str\_\_(self):

return f"{'room\_id':^10}{'arrival\_date':^18}{'departure\_date':^18}{'category\_id':^15}{'hotel\_id':^10}{'number':^8}{'price':^8}"

# return f"""room\_id = {self.room\_id}, arrival\_date = {self.arrival\_date}, departure\_date {self.departure\_date}, """ \

# f"""category\_id = {self.category\_id}, hotel\_id = {self.hotel\_id}, hotel\_id = {self.number}, price = {self.price}"""

class RoomChambermaid(base):

\_\_tablename\_\_ = 'room/chambermaid'

room\_id = Column(Integer, ForeignKey('room.room\_id'), primary\_key=True, nullable=False)

chambermaid\_id = Column(Integer, ForeignKey('chambermaid.chambermaid\_id'), primary\_key=True, nullable=False)

def \_\_init\_\_(self, room\_id, chambermaid\_id):

self.room\_id = room\_id

self.chambermaid\_id = chambermaid\_id

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

return "{:^10}{:^18}".format(self.room\_id, self.chambermaid\_id)

def \_\_str\_\_(self):

return f"{'room\_id':^10}{'chambermaid\_id':^18}"

# return f"""room\_id = {self.room\_id}, chambermaid\_id = {self.chambermaid\_id}"""

class RoomGuest(base):

\_\_tablename\_\_ = 'room/guest'

room\_id = Column(Integer, ForeignKey('room.room\_id'), primary\_key=True, nullable=False)

guest\_id = Column(Integer, ForeignKey('guest.guest\_id'), primary\_key=True, nullable=False)

def \_\_init\_\_(self, room\_id, guest\_id):

self.room\_id = room\_id

self.guest\_id = guest\_id

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

return "{:^10}{:^10}".format(self.room\_id, self.guest\_id)

def \_\_str\_\_(self):

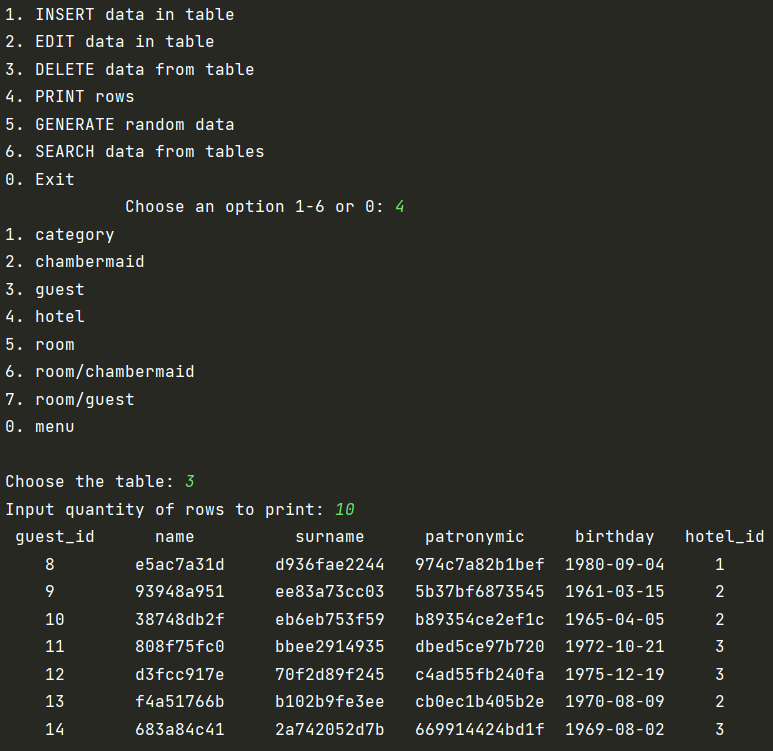
return f"{'room\_id':^10}{'guest\_id':^10}"

# return f"""room\_id = {self.room\_id}, guest\_id = {self.guest\_id}"""

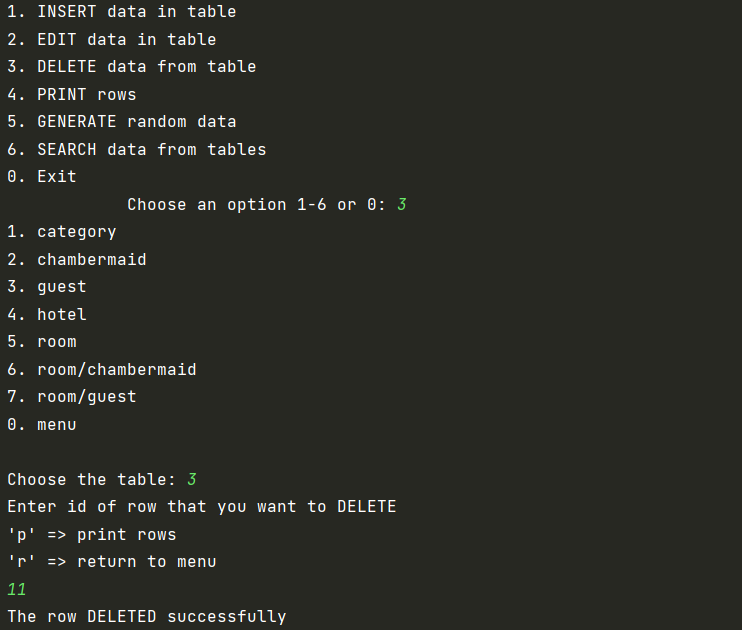
**Запити у вигляді ORM**

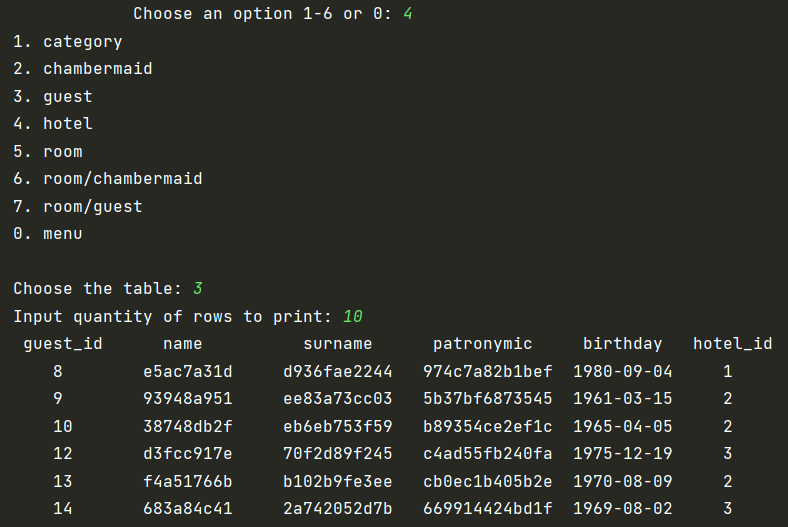
Продемонструємо вставку, вилучення, редагування даних на прикладі таблиці **guest**.

Початкові вхідні дані:

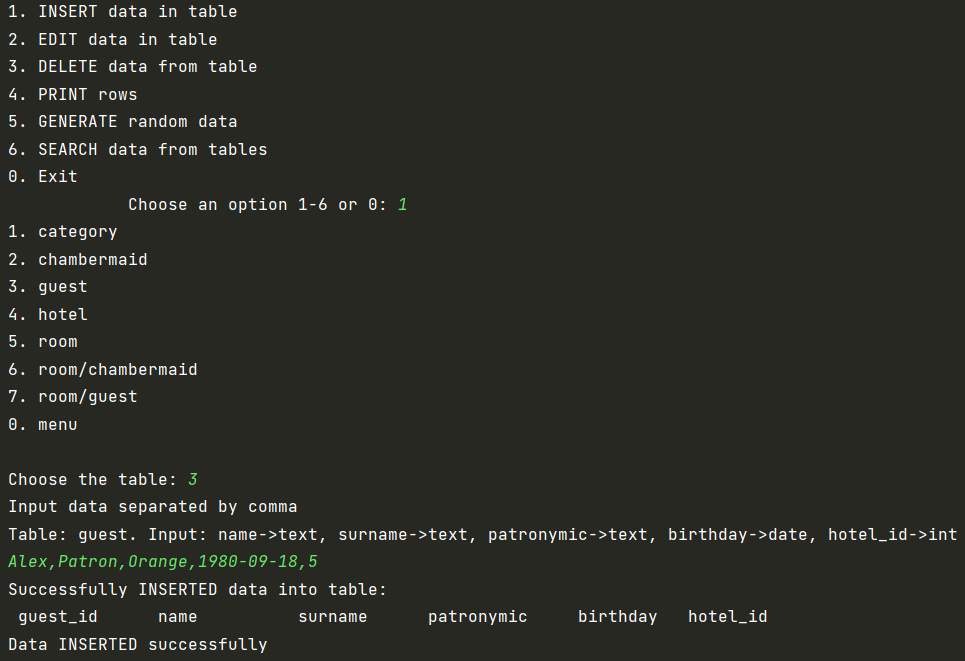


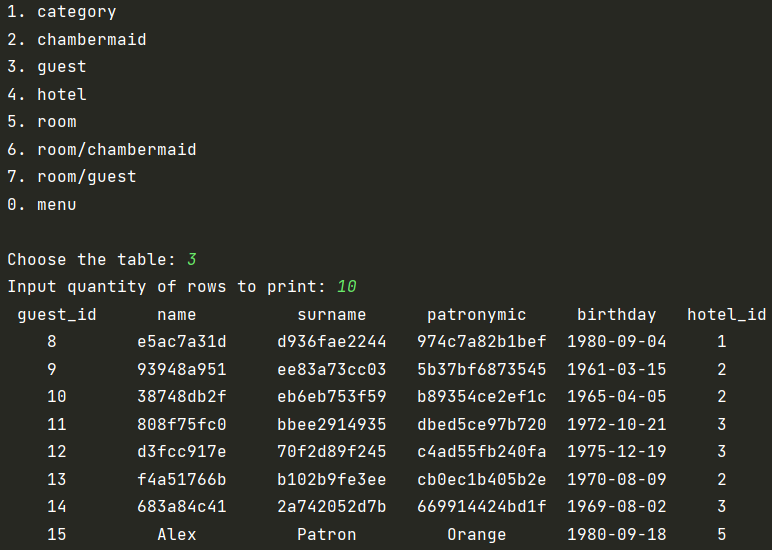
Для перевірки роботи розглянемо запити на видалення даних з даної таблиці **guest**. Спробуємо видалити рядок з **guest\_id = 11**. Нижче наведене виконанная заданих запитів.



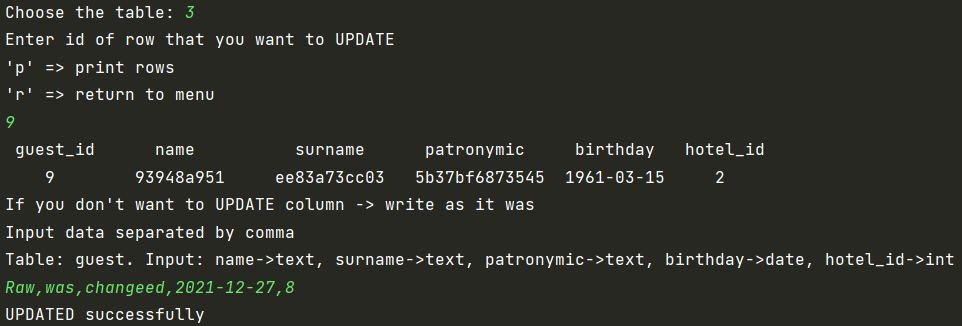


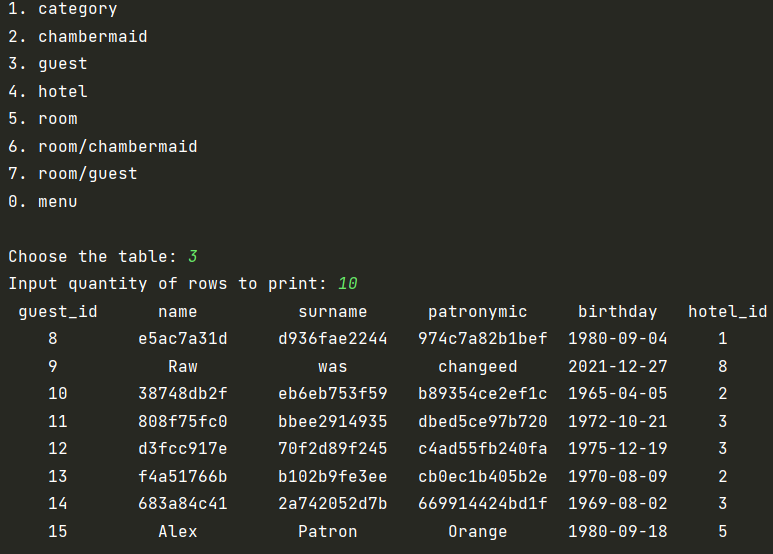
Для перевірки роботи розглянемо запити на вставку даних до даної таблиці **guest**. Спробуємо додати рядок здеякими даними, даний рядок повинен буде мати **guest\_id = 15**. Нижче наведене виконанная заданих запитів.





Для перевірки роботи розглянемо запити на редагування даних до даної таблиці **guest**. Спробуємо відредагувати рядок з **guest\_id = 9**. Нижче наведене виконанная заданих запитів.





**Завдання №2**

Для тестування індексів було створено окремі таблиці у базі даних з 100000-200000 записів.

**BTree**

1. Для початку створимо пусту таблицю з двома текстовими полями, одну з використанням індексів, одну без.

create table btree\_test(

elem varchar,

elem\_indexed varchar

);

1. Заповнимо обидва стовбця 100000 випадковими даними.

INSERT INTO "btree\_test" SELECT

md5(random()::text),

md5(random()::text) from (

SELECT \* FROM generate\_series(1,100000) AS id) AS ser;

1. Тепер за допомогою SELECT виберемо і відсортуємо всі значення за спаданням і зрівняємо час виконання кожної з команд.

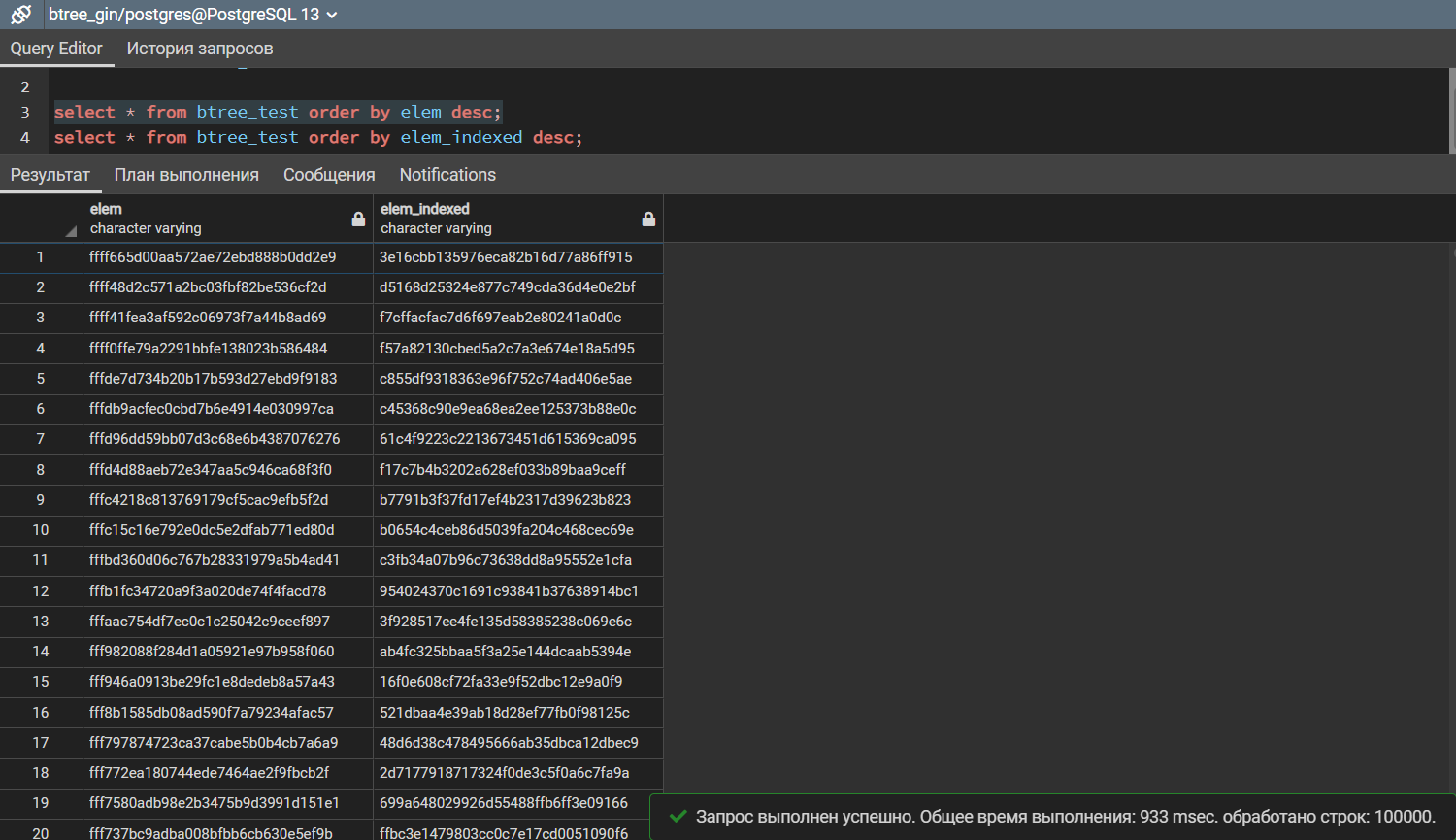
CREATE INDEX btree\_index ON

btree\_test using btree (elem\_indexed);

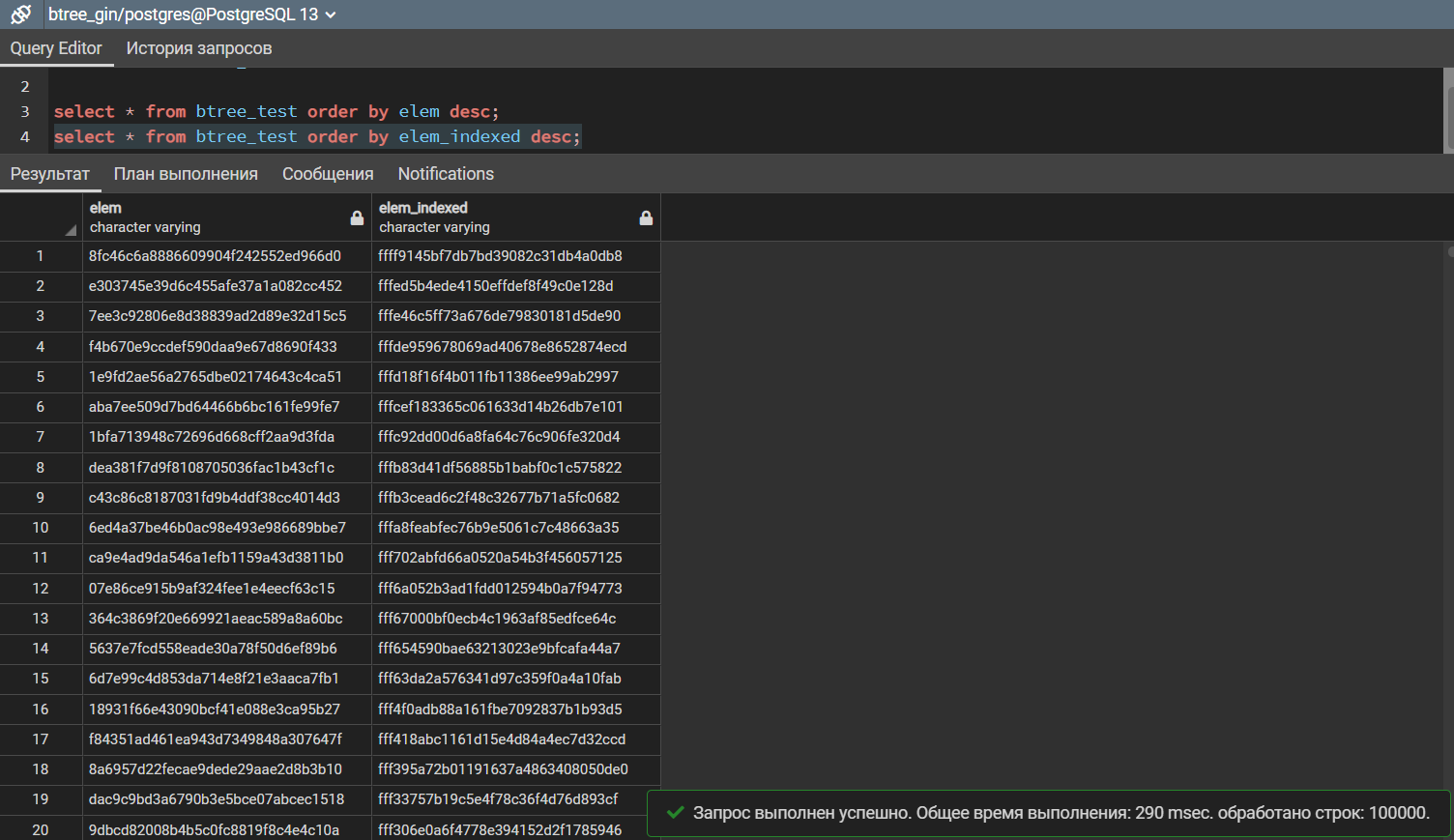
SELECT \* FROM btree\_test ORDER BY elem desc;

SELECT \* FROM btree\_test ORDER BY elem\_indexed desc;

**Без використання індексів:**



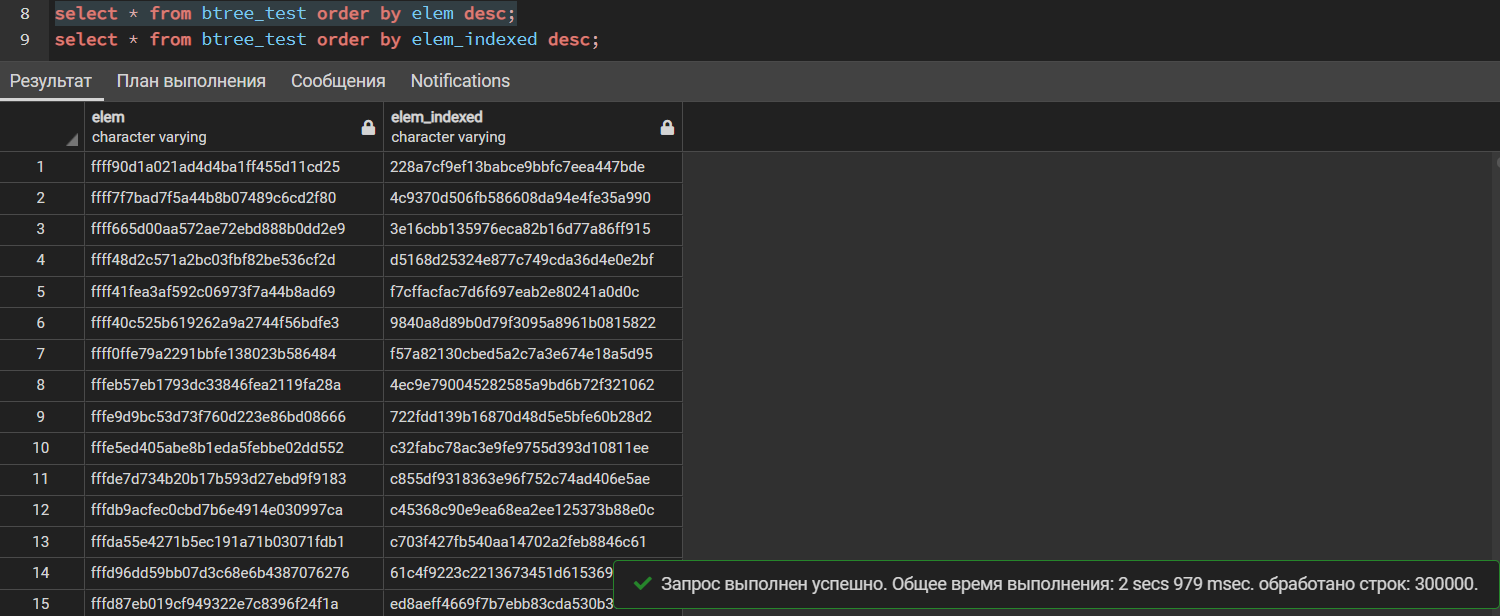
**З використанням індексів:**



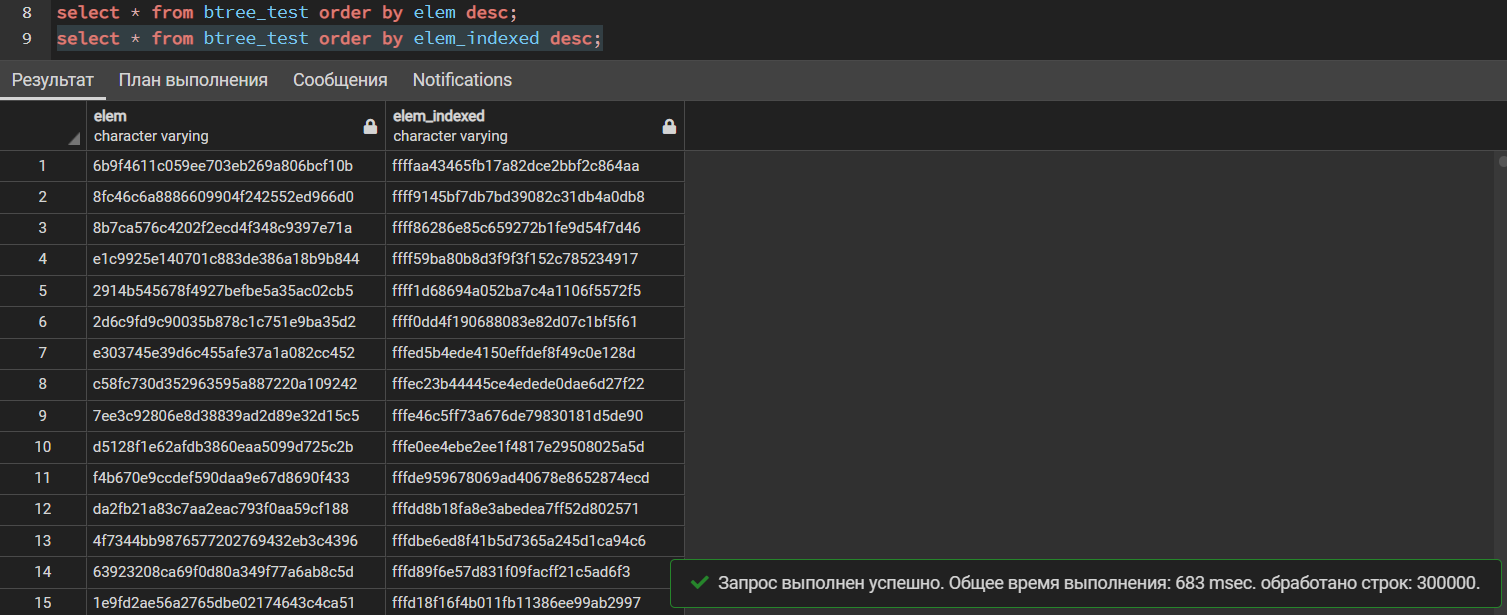
Можна помітити, що навіть при такій кількості даних, різниця в виконанні досить значна.

1. Збільшимо кількість даних на 200000 і зрівняємо знову швидкість виконання сортування.

**Без використання індексів:**



**З використанням індексів:**



Результат показує, що використання індексів значно прискорює виконання запиту, а якщо даних дуже багато, то й швидкість більша в значну кількість разів.

**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;

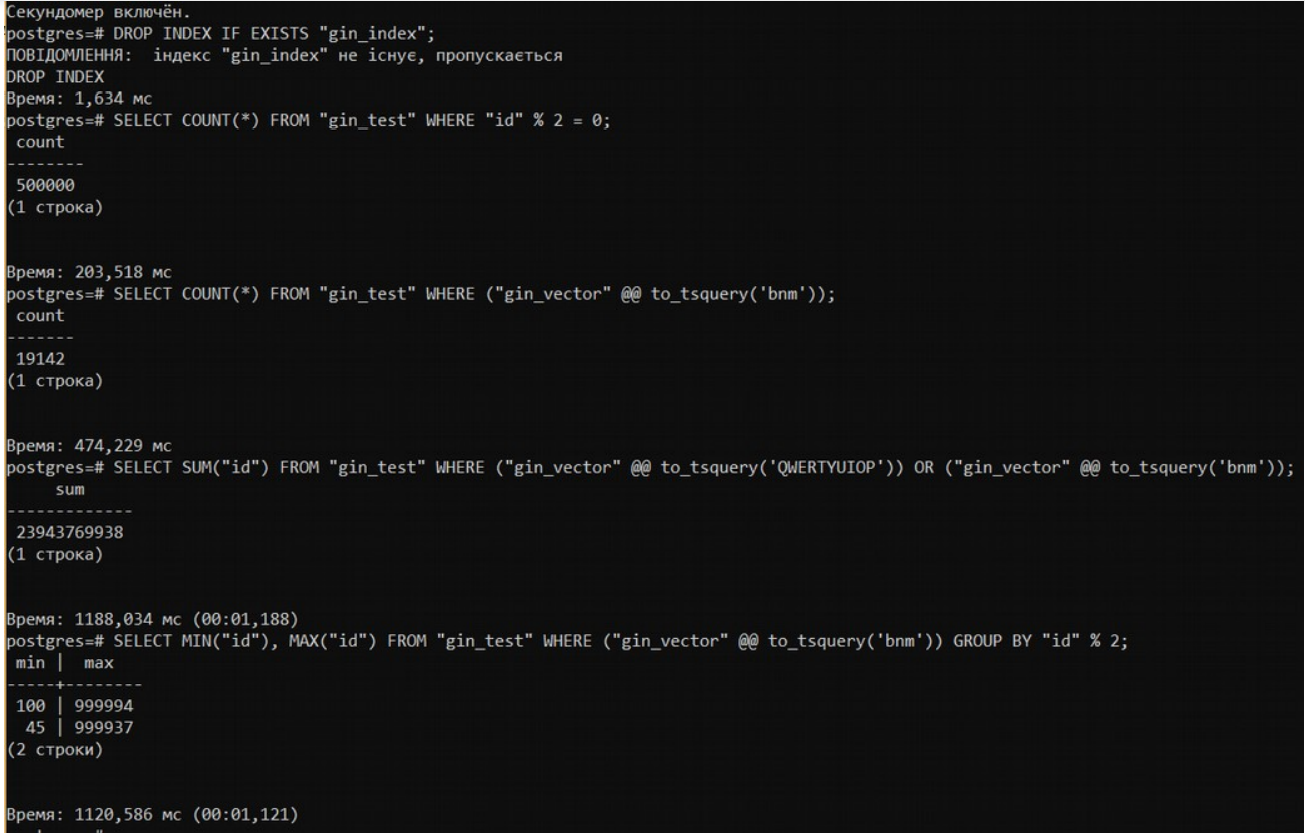
Створення індексу:

DROP INDEX IF EXISTS "gin\_index";

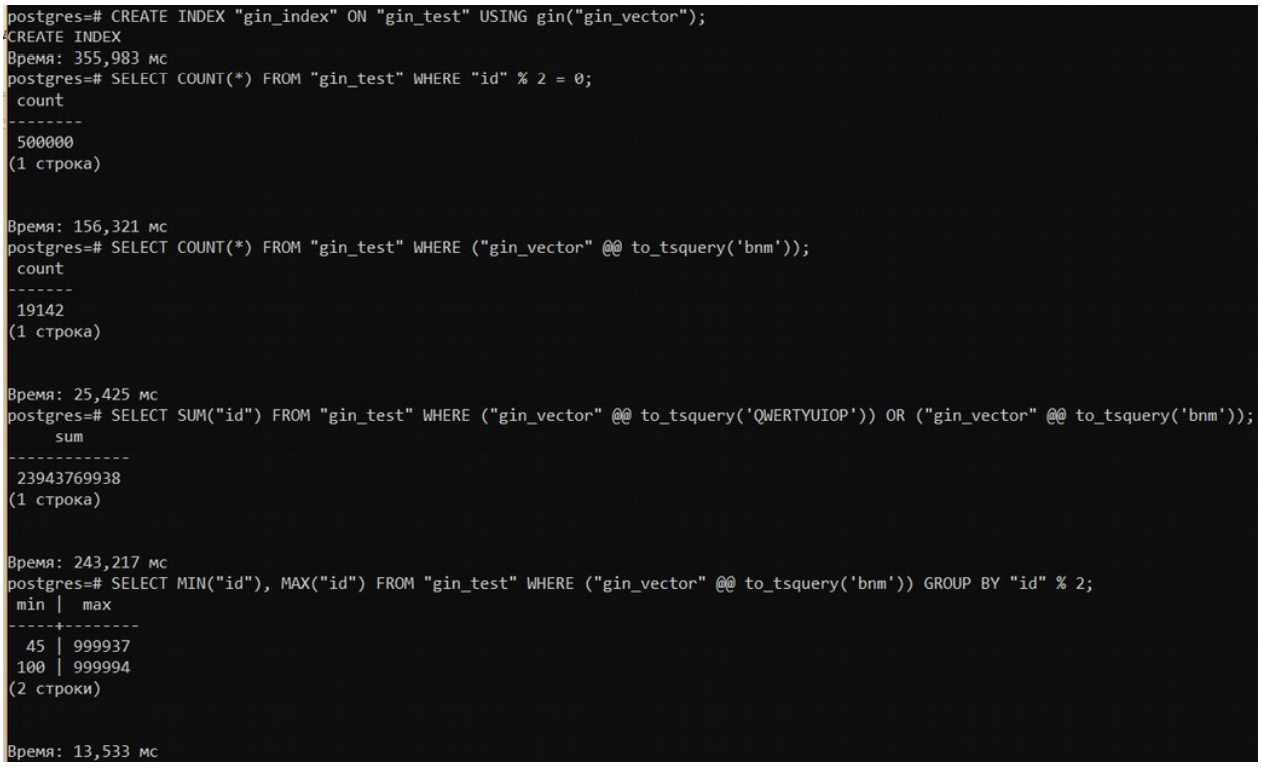
CREATE INDEX "gin\_index" ON "gin\_test" USING gin("gin\_vector");

**Результати і час виконання на скріншотах:**

**Без використання індексів:**



**З використанням індексів:**



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**

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\_delete\_update\_trigger"

BEFORE DELETE OR UPDATE ON "trigger\_test"

FOR EACH ROW EXECUTE procedure before\_delete\_update\_func();

Текст тригера:

CREATE OR REPLACE FUNCTION before\_delete\_update\_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;

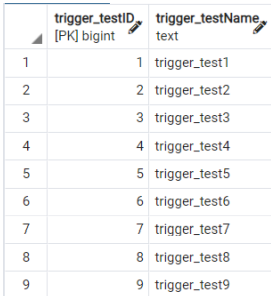
END;

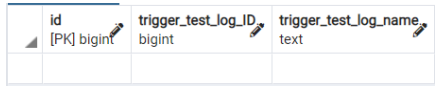
$trigger$ LANGUAGE plpgsql;

**Скріншоти зі змінами у таблицях бази даних**

Початковий стан

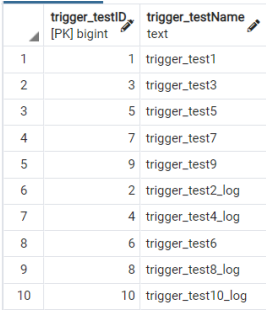
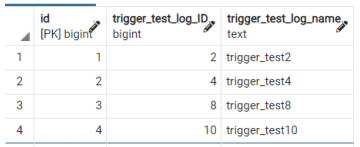
SELECT \* FROM "trigger\_test";



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

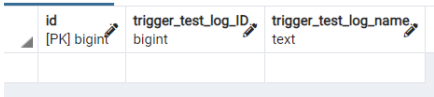
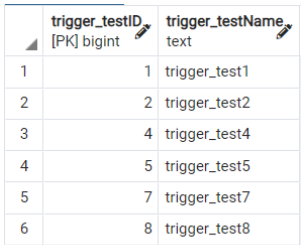
Після виконання запиту на оновлення

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



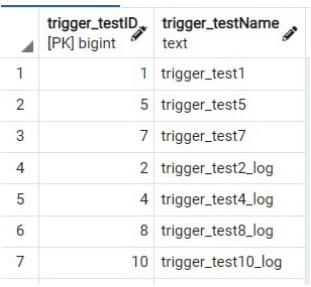
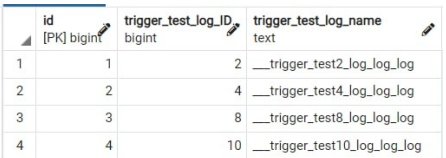
Наочно можемо переконатись, що виконалась та гілка алгоритму тригера, що відповідає за парні рядки (оскільки є умова для парних), а для 6 рядка він також виконався, але пішов іншою (вкладеною) гілкою алгоритму та повернув старий стан (OLD). При запиті на оновлення потрібно повертати новий стан, а при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

CREATE TABLE "transactions"(

"id" bigserial PRIMARY KEY,

"numeric" bigint,

"text" text

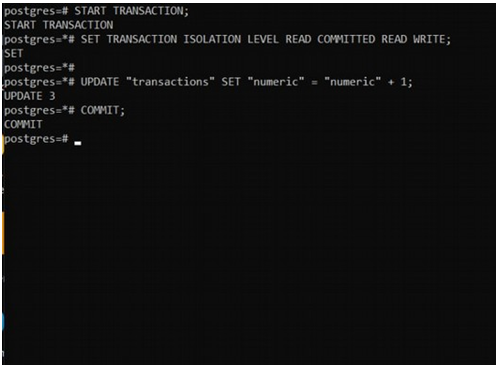
);

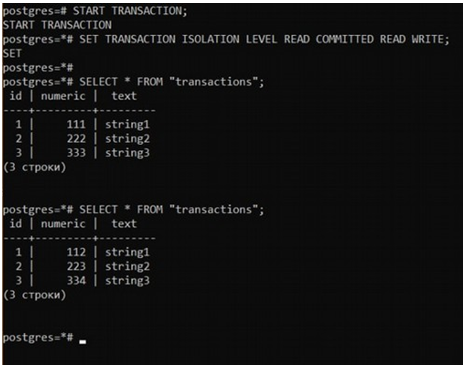
INSERT INTO "transactions"("numeric", "text")

VALUES (111, 'string1'), (222, 'string2'), (333, 'string3');

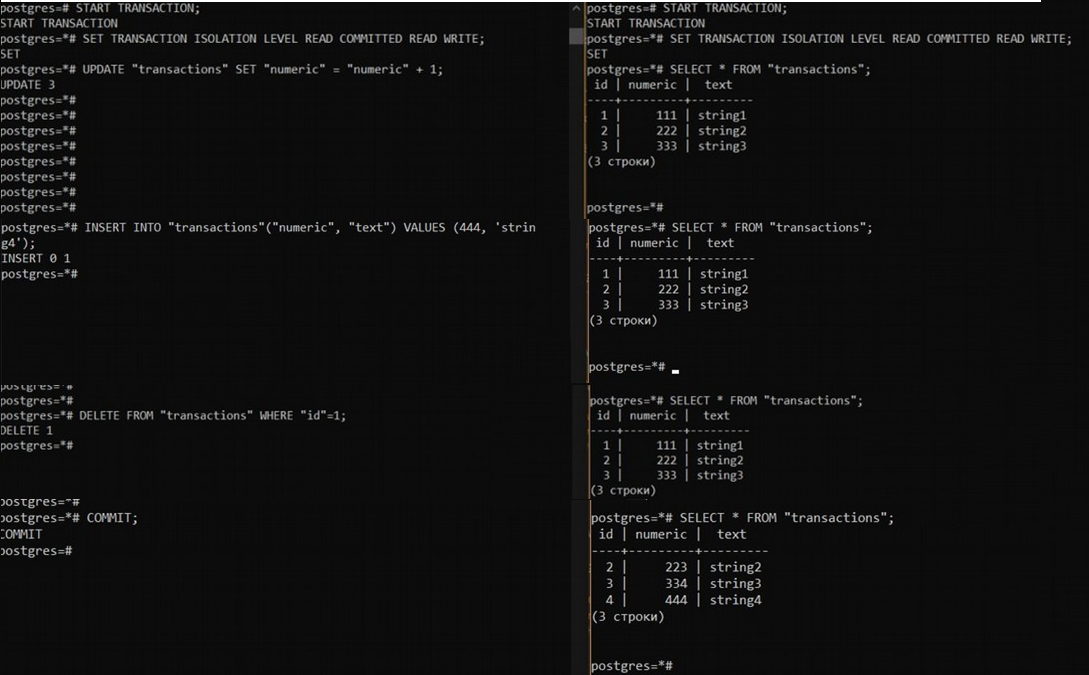
**READ COMMITTED**

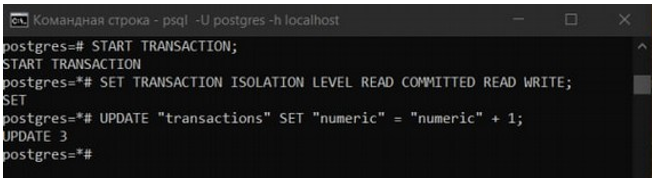
На цьому рівні ізоляції одна транзакція не бачить змін у базі даних, викликаних іншою доки та не завершить своє виконання (командою COMMIT або ROLLBACK).

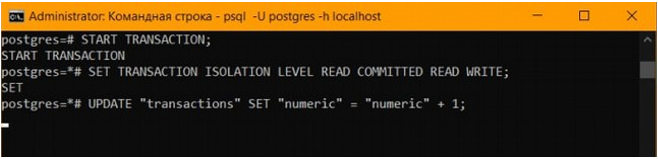




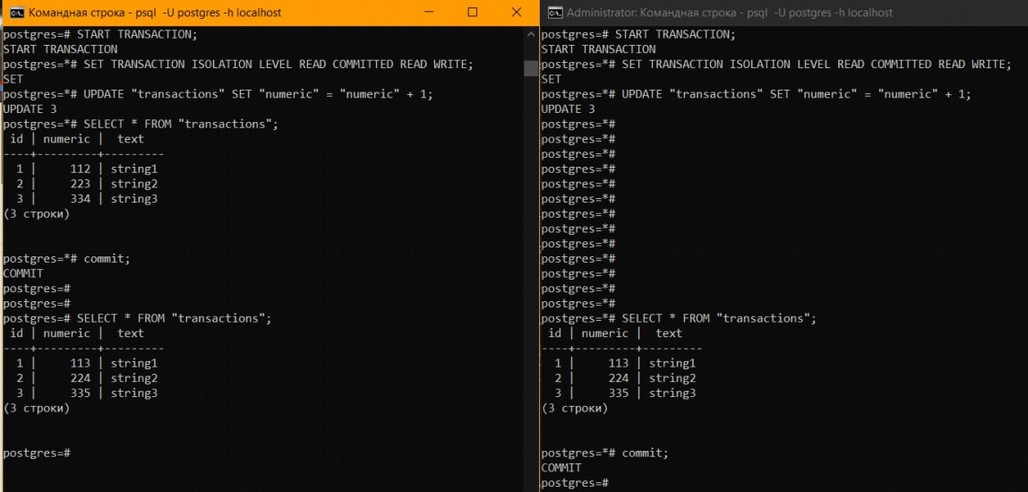
Дані після вставки та видалення так само будуть видні другій тільки після завершення першої.



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



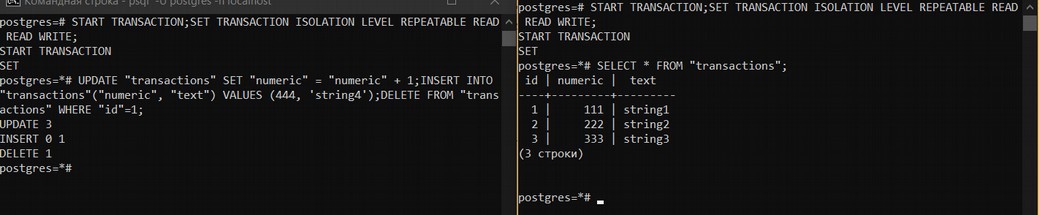
А тут бачимо, що після завершення першої, друга транзакція виконала запит, змінивши вже ті дані, що були закомічені першою транзакцією



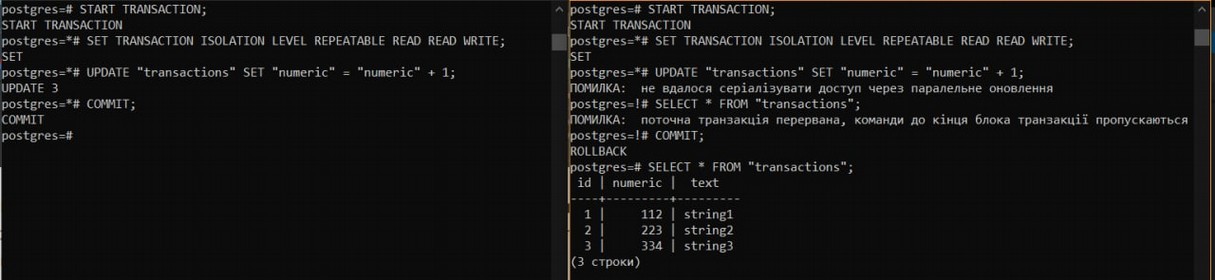
Коли Т2 бачить дані Т1 запитів UPDATE, DELETE виникає феномен повторного читаня, а коли бачить дані запиту INSERT – читання фантомів. Цей рівень ізоляції забезпечує захист від явища брудного читання.

**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:



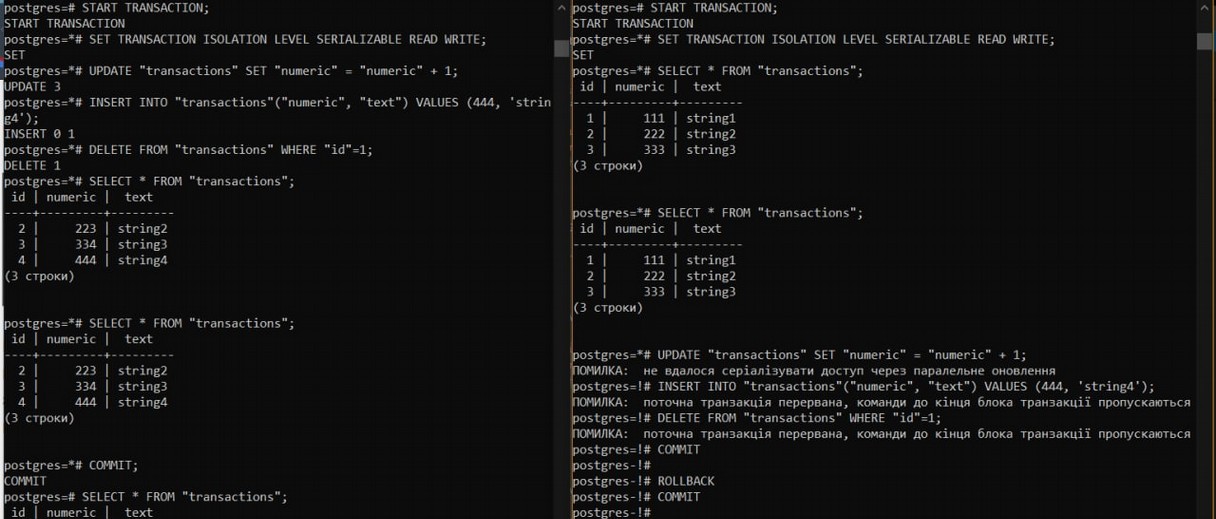
А тут, що отримуємо помилку при спробі доступу до тих самих даних:



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.