

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 ІII курсу

групи КВ-91 Кузьмич А. А. 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 |
| *12* | *BTree, GIN* | *after update, insert* |

Посилання на репозиторій у GitHub з вихідним кодом програми та звітом: <https://github.com/Annakuzmych/>

# Завдання №1

# Відомості про обрану предметну галузь з лабораторної роботи №1

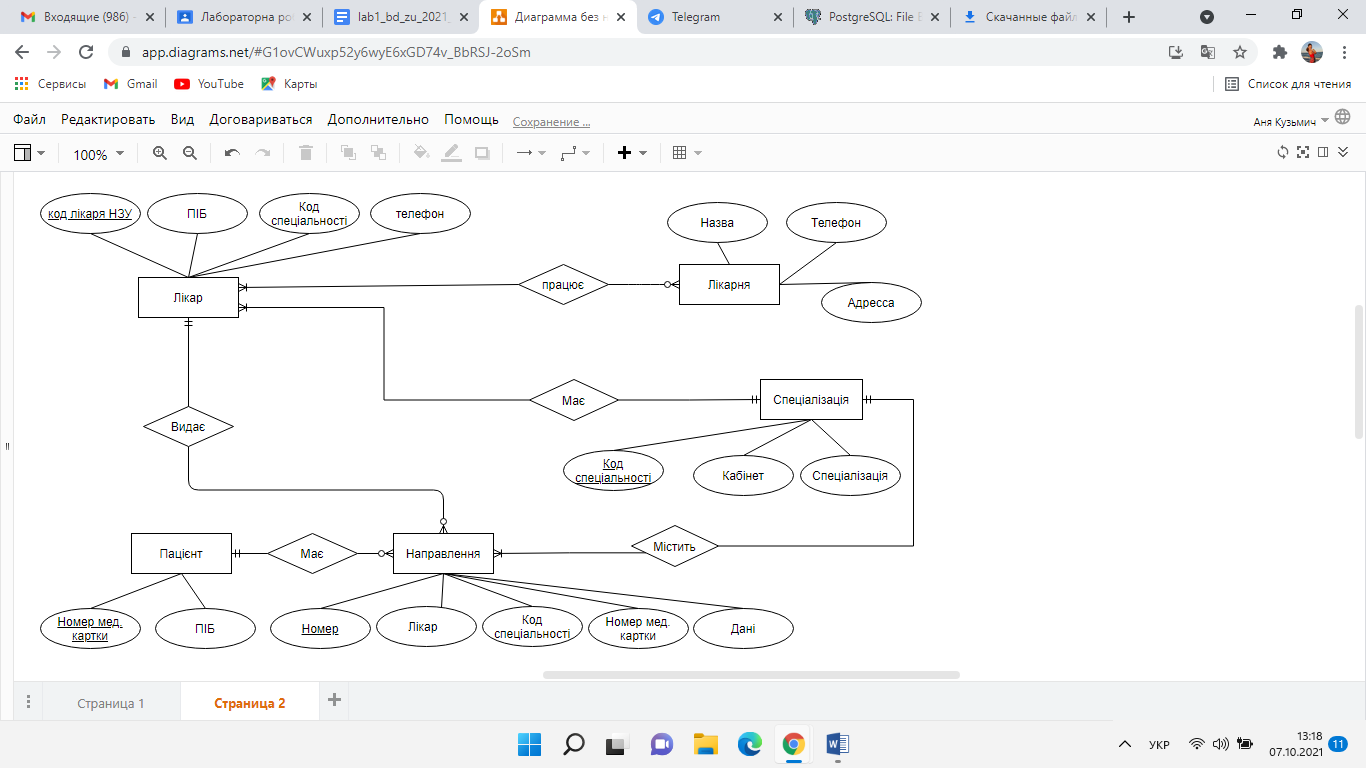


Рисунок 1 - ER-діаграма, побудована за нотацією Чена

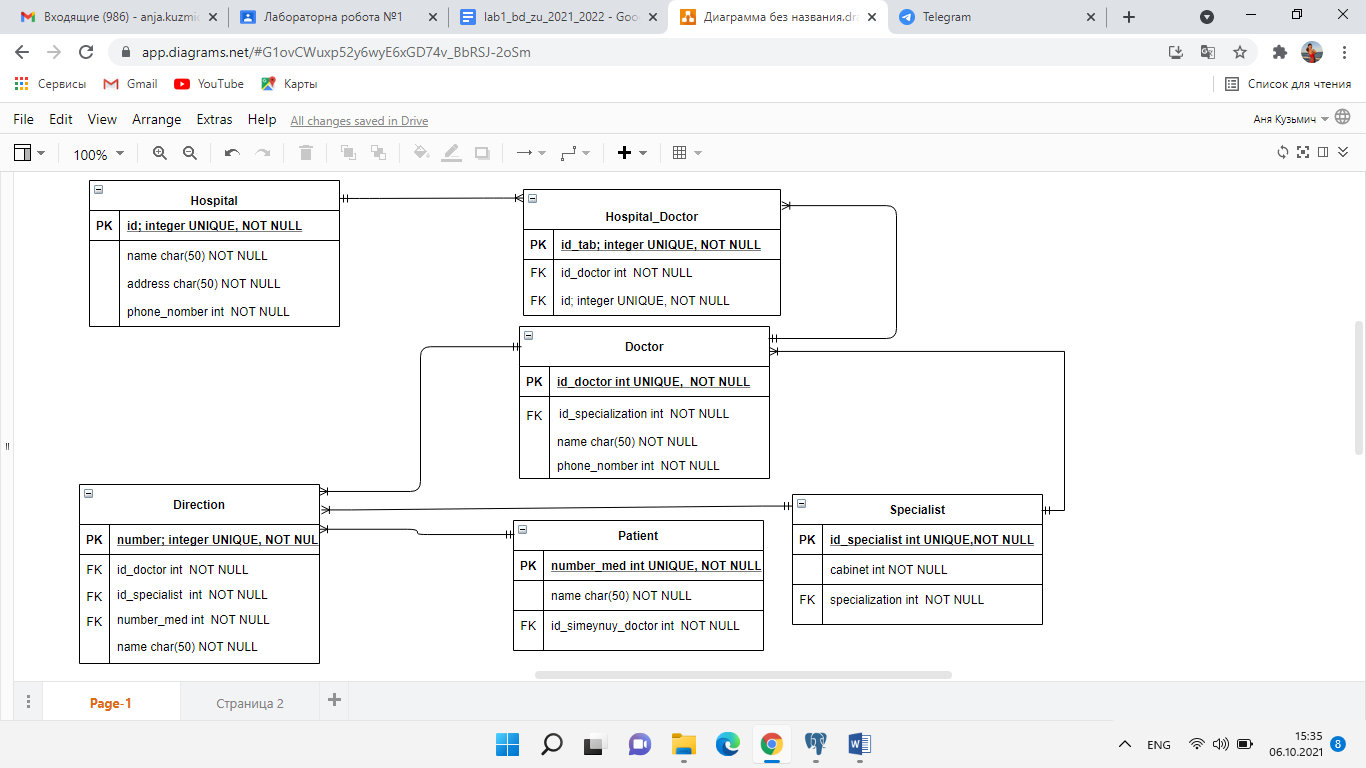


Рисунок 2. Схема бази даних

Таблиця 1 - Опис структури БД.

|  |  |  |
| --- | --- | --- |
| Сутність | Атрибут | Тип атрибуту |
| **Hospital** - *містить дані про лікарню* | **id** *- унікальний ідентифікатор лікарні*  **name** *- назва лікарні*  **phone** *– телефонний номер лікарні*  **address** *- фізична адреса лікарні* | **integer** *(числовий)*  **character varying** *(рядок)*  **integer** *(числовий)*  **character varying** *(рядок)* |
| **Doctor** - *містить у собі всі дані лікарів* | **id\_doctor** *- унікальний ідентифікатор лікаря*  **name** *– ПІБ лікаря*  **id\_specialist** *- ідентифікатор спеціальності*  **phone** *- телефонний номер лікаря* | **integer** *(числовий)*  **character varying** *(рядок)*  **integer** *(числовий)*  **integer** *(числовий)* |
| **Patient** - *зберігає дані про пацієнтів* | **Number\_med** *- унікальний номер медичної картки*  **Name\_p** *– ПІБ пацієнта* | **integer** *(числовий)*  **character varying** *(рядок)* |
| **Direction** - *містить інформацію про обстеження пацієнтів.* | **number** *- унікальний ідентифікатор направлення*  **Number\_med** *- унікальний номер медичної картки*  **id\_doctor** *- унікальний ідентифікатор лікаря*  **id\_specialist -** *унікальний ідентифікатор спеціальності*  **data** *– дані обстеження* | **integer** *(числовий)*  **integer** *(числовий)*  **integer** *(числовий)*  **integer** *(числовий)*  **character varying** *(рядок* |
| **Specialist** - *містить інформацію про спеціальності лікарів* | **id\_specialist -** *унікальний ідентифікатор спеціальності*  **cabinet –** *номер кабінету*  **specialization –** *назва спеціальності* | **integer** *(числовий)*  **integer** *(числовий)*  **character varying** *(рядок* |
| **Hospital\_ Doctor**  *містить інформацію про відповідність лікаря і лікарні, де він працює.* | **id\_tab - -** *унікальний ідентифікатор таблиці відповідності*  **id\_doctor***- унікальний ідентифікатор лікаря*  **id***- унікальний ідентифікатор лікарні* | **integer** *(числовий)*  **integer** *(числовий)*  **integer** *(числовий)* |

Обрана предметна галузь передбачає отримання і обробку потрібної інформації щодо лікарні. Згідно цієї області для побудови бази даних було виділено наступні сутності:

Лікарня з атрибутами : назва лікарні, телефон приймального покою, адреса лікарні. Призначення: збереження даних, що стосуються тих лікарень, які цікавлять користувача.

Лікар з атрибутами: код лікаря НЗУ, лікарня, де працює лікар, код спеціальності лікаря, ПІБ лікаря та телефон. Призначення: збереження даних щодо лікарів.

Пацієнт з атрибутами: номер медичної картки, ПІБ. Призначення: збереження даних щодо пацієнтів.

Направлення з атрибутами: номер направлення, код лікаря що видав направлення, код спеціальності, номер медичної картки пацієнта, дані обстеження. Призначення: збереження інформації стосовно обстежень пацієнтів та роботи лікарів.

**Абзаци**

Спеціалізація з атрибутами: код спеціальності, кабінет, що закріплений за спеціальністю, спеціальність. Призначення: : збереження інформації стосовно спеціальностей лікарів, що представлені в лікарні і їх зв'язок з кабінетами.

Класи ORM у реалізованому модулі Model

**class Direction(base):**

**\_\_tablename\_\_ = 'Direction'**

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

**number\_med = Column(Integer, ForeignKey('Patient.number\_med'), nullable=False)**

**id\_doctor = Column(Integer, ForeignKey('Doctor.id\_doctor'), nullable=False)**

**id\_specialist = Column(Integer, ForeignKey('Specialist.id\_specialist'), nullable=False)**

**data = Column(String(50), nullable=False)**

**Doctor = relationship('Doctor')**

**Specialist = relationship('Specialist')**

**Patient = relationship('Patient')**

**def \_\_init\_\_(self, number\_med, id\_doctor, id\_specialist, data, number=-1):**

**self.number\_med = number\_med**

**self.id\_doctor = id\_doctor**

**self.id\_specialist = id\_specialist**

**self.data = data**

**if id != -1:**

**self.number = number**

**format\_str = '{:^8}{:^12}{:^12}{:^12}{:^20}'**

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

**return self.format\_str.format(self.number, self.number\_med, self.id\_doctor, self.id\_specialist, self.data)**

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

**return self.format\_str.format('number', 'number\_med', 'id\_doctor', 'id\_specialist', 'data')**

**class Doctor(base):**

**\_\_tablename\_\_ = 'Doctor'**

**id\_doctor = Column(Integer, primary\_key=True, nullable=False)**

**id\_specialist = Column(Integer, ForeignKey('Specialist.id\_specialist'), nullable=False)**

**name\_doc = Column(String(50), nullable=False)**

**phone\_num = Column(Integer, nullable=False)**

**Specialist = relationship('Specialist')**

**def \_\_init\_\_(self, id\_specialist, name\_doc, phone\_num, id\_doctor=-1):**

**self.phone\_num = phone\_num**

**self.id\_specialist = id\_specialist**

**self.name\_doc = name\_doc**

**if id\_doctor != -1:**

**self.id\_doctor = id\_doctor**

**format\_str = '{:^10}{:^12}{:^12}{:^30}'**

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

**return self.format\_str.format(self.id\_doctor, self.id\_specialist, self.name\_doc, self.phone\_num)**

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

**return self.format\_str.format('id\_doctor', 'id\_specialist', 'name\_doc', 'phone\_num')**

**class Hospital(base):**

**\_\_tablename\_\_ = 'Hospital'**

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

**name = Column(String(50), nullable=False)**

**address = Column(String(50), nullable=False)**

**phone = Column(Integer, nullable=False)**

**def \_\_init\_\_(self, name, address, phone, id=-1):**

**self.phone = phone**

**self.name = name**

**self.address = address**

**if id != -1:**

**self.id = id**

**format\_str = '{:^8}{:^30}{:^40}{:^20}'**

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

**return self.format\_str.format(self.id, self.name, self.address, self.phone)**

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

**return self.format\_str.format('id', 'name', 'address', 'phone')**

**class Hospital\_Doctor(base):**

**\_\_tablename\_\_ = 'Hospital\_Doctor'**

**id\_tab = Column(Integer, primary\_key=True, nullable=False)**

**id = Column(Integer, ForeignKey('Hospital.id'), nullable=False)**

**id\_doctor = Column(Integer, ForeignKey('Doctor.id\_doctor'), nullable=False)**

**Hospital = relationship('Hospital')**

**Doctor = relationship('Doctor')**

**def \_\_init\_\_(self, id, id\_doctor, id\_tab=-1):**

**self.id\_doctor = id\_doctor**

**self.id = id**

**if id\_tab != -1:**

**self.id\_tab = id\_tab**

**format\_str = '{:^8}{:^12}{:^12}'**

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

**return self.format\_str.format(self.id\_tab, self.id, self.id\_doctor)**

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

**return self.format\_str.format('id\_tab', 'id', 'id\_doctor')**

**class Patient(base):**

**\_\_tablename\_\_ = 'Patient'**

**number\_med = Column(Integer, primary\_key=True, nullable=False)**

**name = Column(String(50), nullable=False)**

**def \_\_init\_\_(self, name, number\_med=-1):**

**self.name = name**

**if number\_med != -1:**

**self.number\_med = number\_med**

**format\_str = '{:^10}{:^30}'**

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

**return self.format\_str.format(self.number\_med, self.name)**

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

**return self.format\_str.format('number\_med', 'name')**

**class Specialist(base):**

**\_\_tablename\_\_ = 'Specialist'**

**id\_specialist = Column(Integer, primary\_key=True, nullable=False)**

**cabinet = Column(Integer, nullable=False)**

**specialization = Column(String(50), nullable=False)**

**def \_\_init\_\_(self, cabinet, specialization, phone, id\_specialist=-1):**

**self.phone = phone**

**self.name = name**

**self.specialization = specialization**

**if id\_specialist != -1:**

**self.id\_specialist = id\_specialist**

**format\_str = '{:^12}{:^12}{:^40}'**

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

**return self.format\_str.format(self.id\_specialist, self.cabinet, self.specialization)**

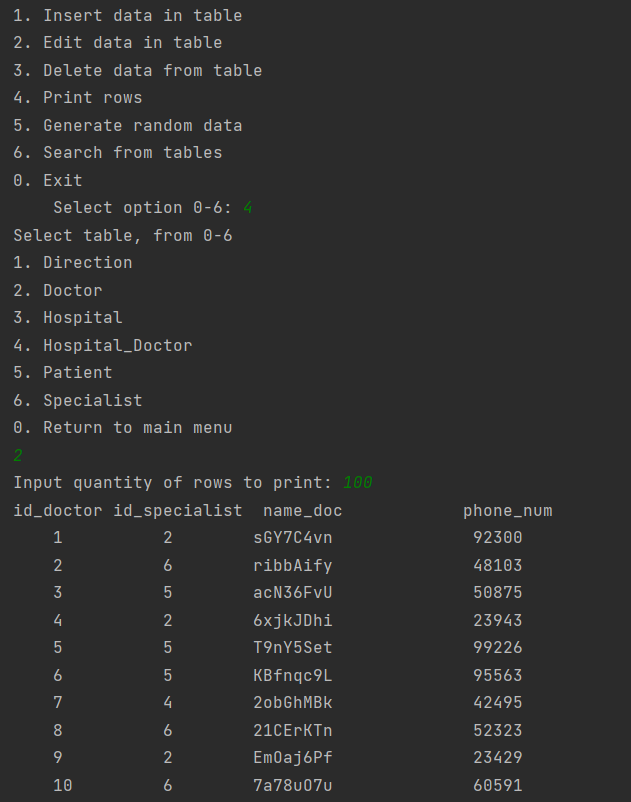
**def \_\_attributes\_print\_\_(self):**

**return self.format\_str.format('id\_specialist', 'cabinet', 'specialization')**

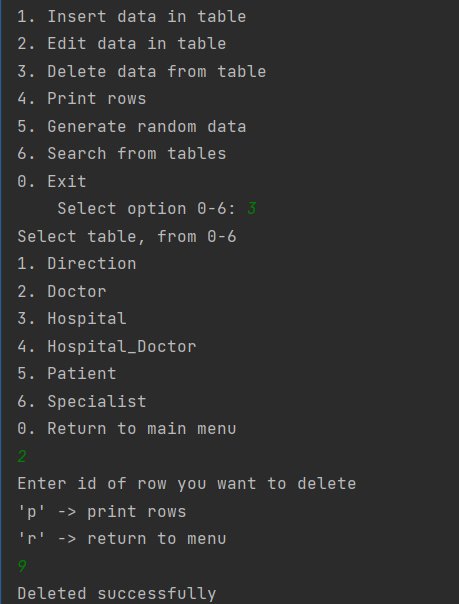
Запити у вигляді ORM

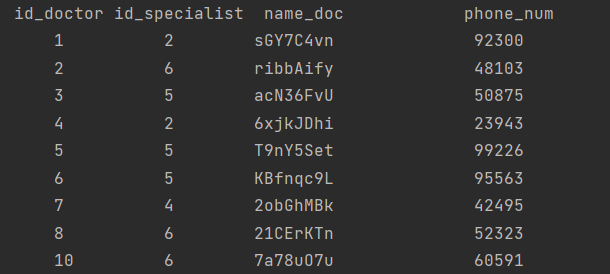
Продемонструємо вставку, виучення, редагування даних на прикладі таблиці Doctor

Початковий стан:

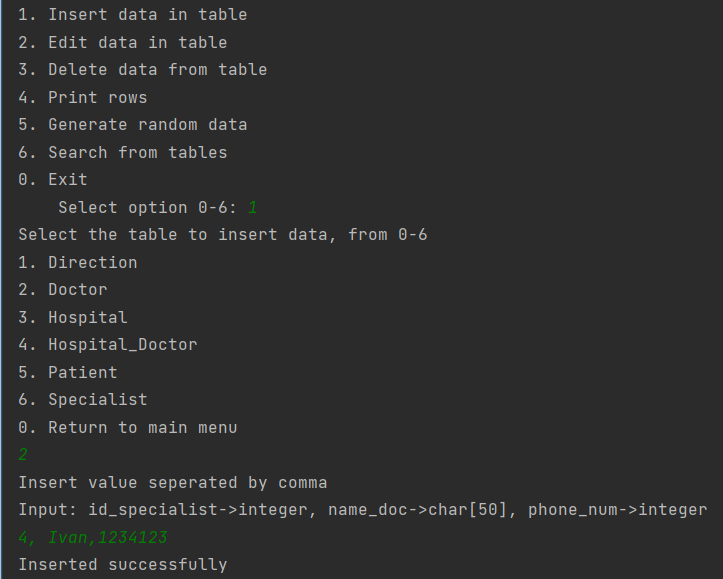


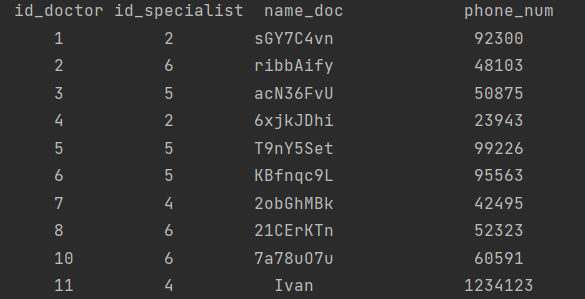
Видалення запису:



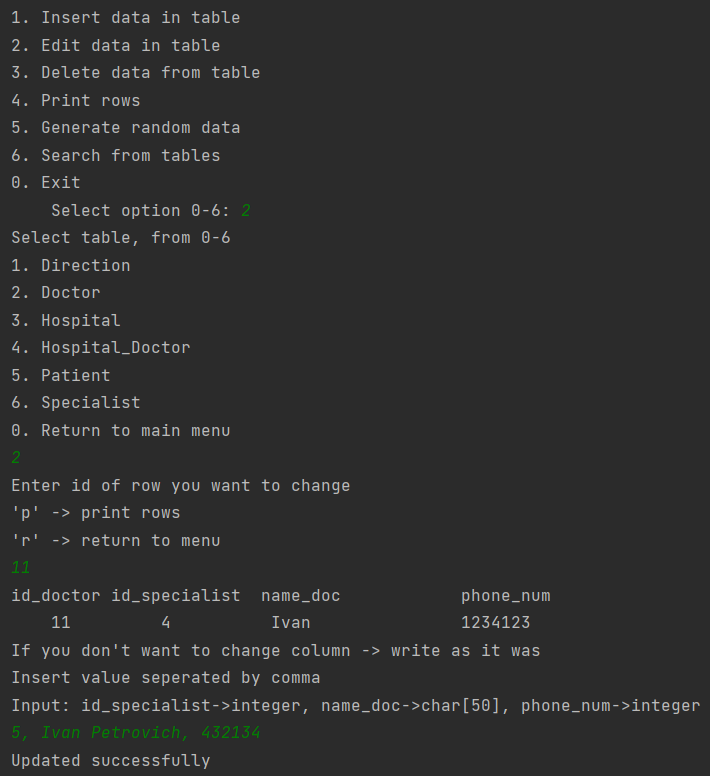


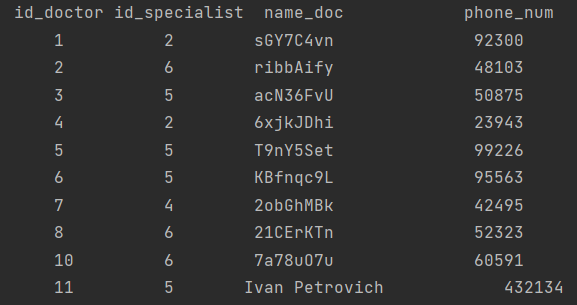
Вставка запису:





Редагування запису:





# 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.

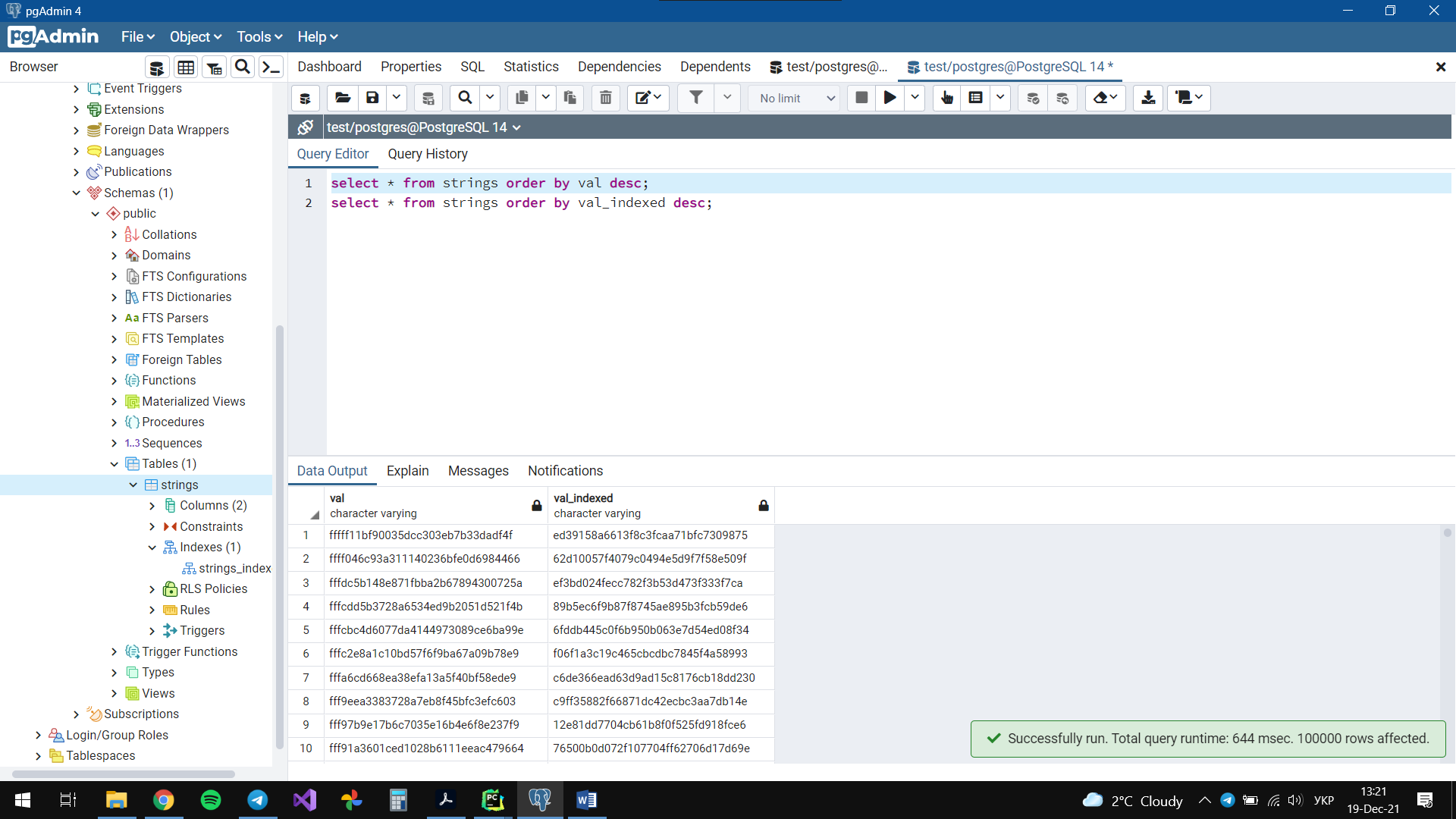
create table strings(  
 val varchar,  
 val\_indexed varchar  
);  
CREATE INDEX strings\_indexed ON strings using btree (val\_indexed);

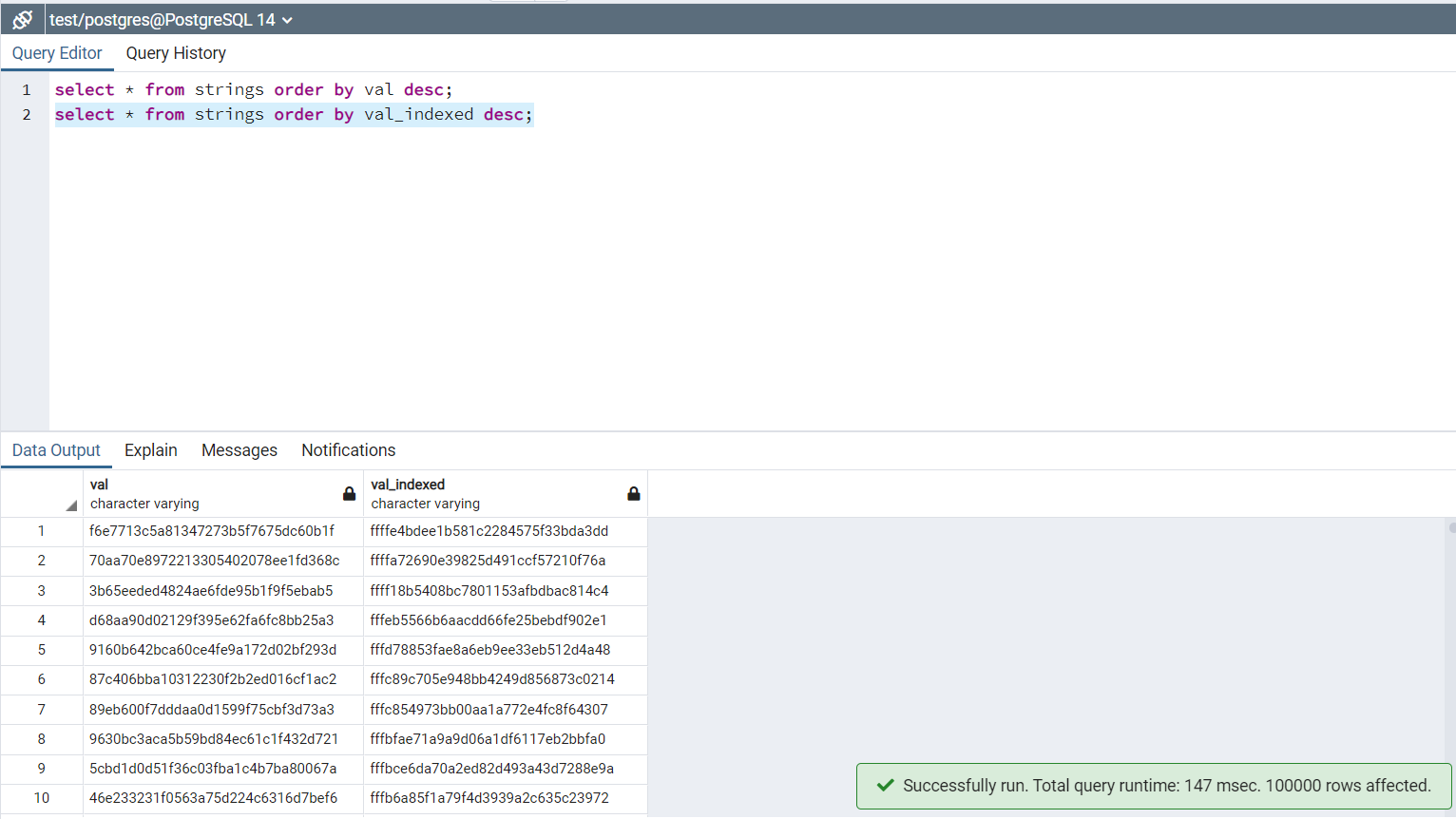
1. Заповнимо обидва стовбця 100000 випадковими даними.

**insert** **into** strings  
**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.

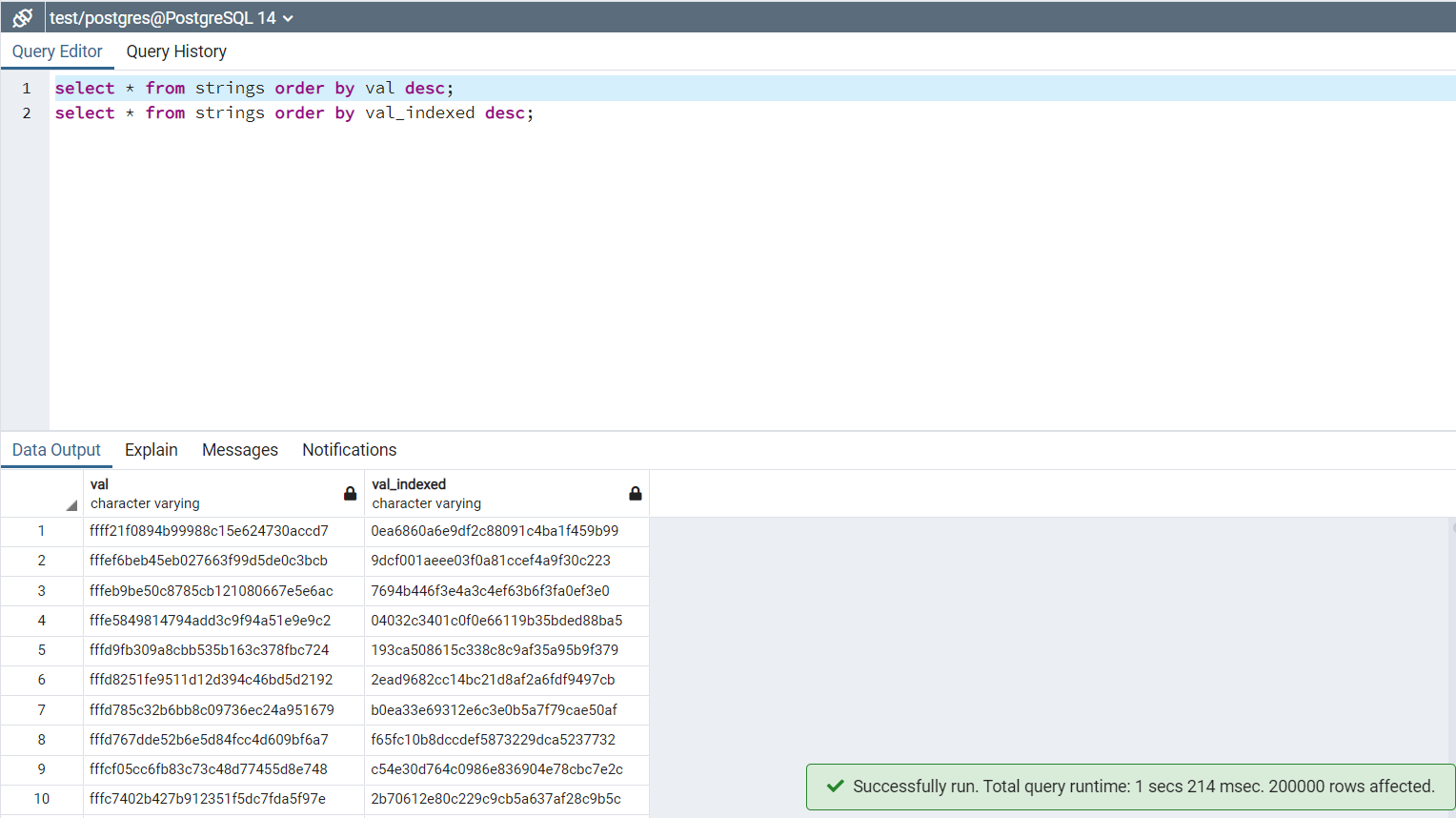
**select** \* **from** strings **order** **by** val **desc**;  
**select** \* **from** strings **order** **by** val\_indexed **desc**;

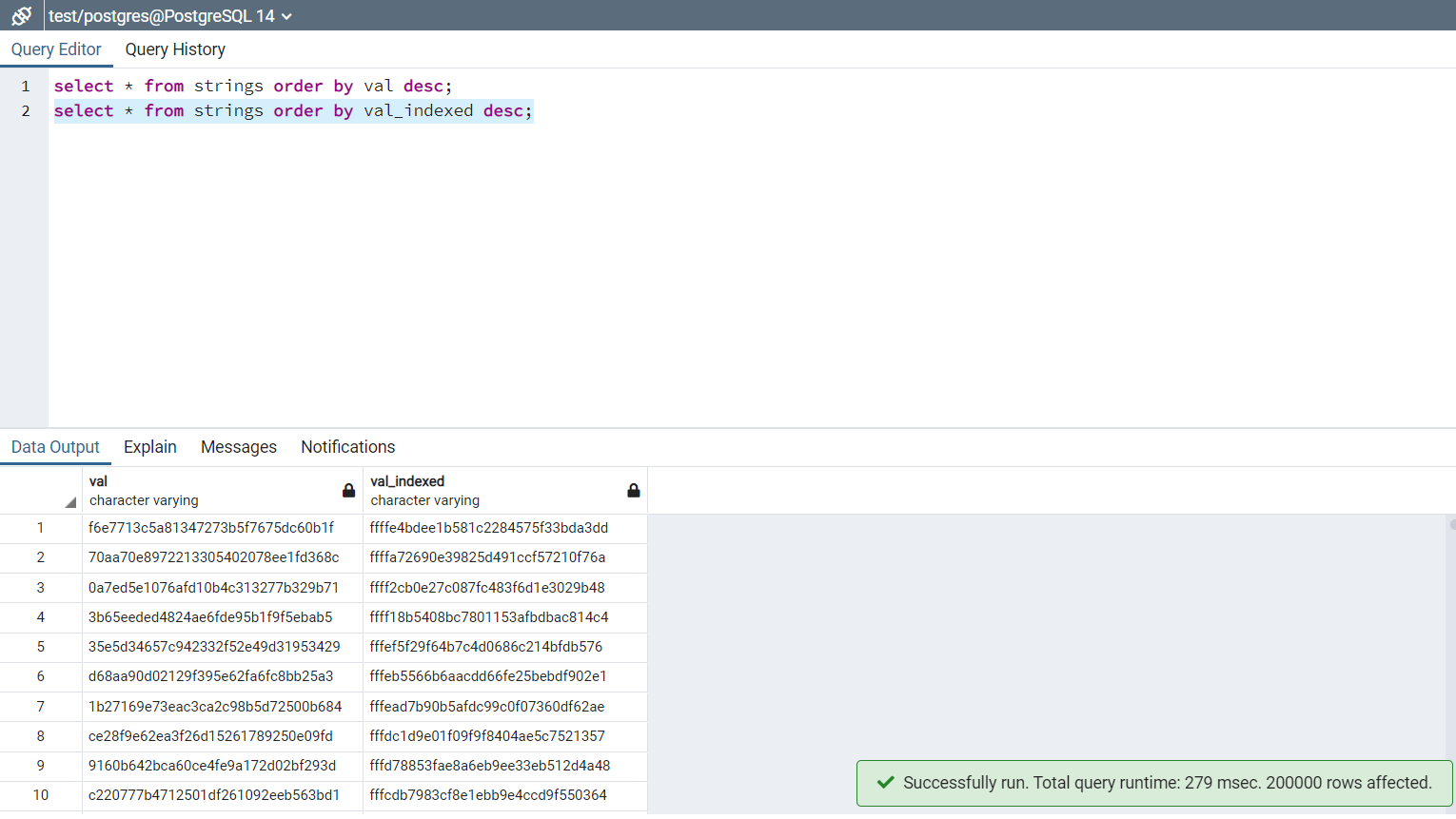




Можна побачити що навіть при такій кількості даних, різниця в виконанні досить значна.

1. Збільшимо кількість даних на 100000 і зрівняємо знову швидкість виконання сортування.





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:

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'));

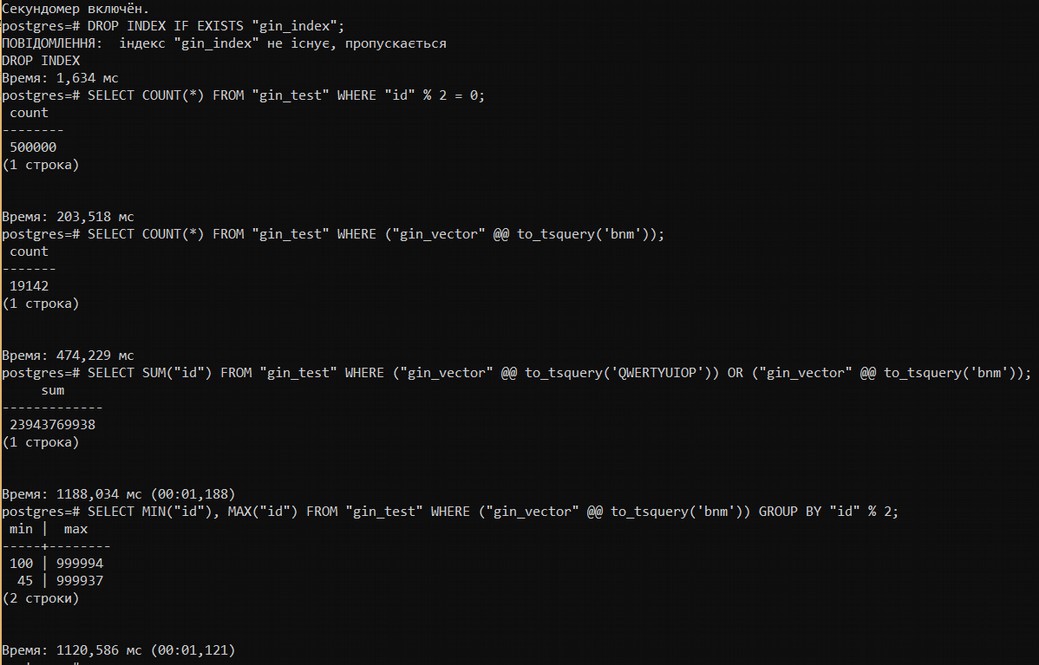
SELECT MIN("id"), MAX("id") FROM "gin\_test" WHERE ("gin\_vector" @@ to\_tsquery('bnm')) GROUP BY "id" % 2;

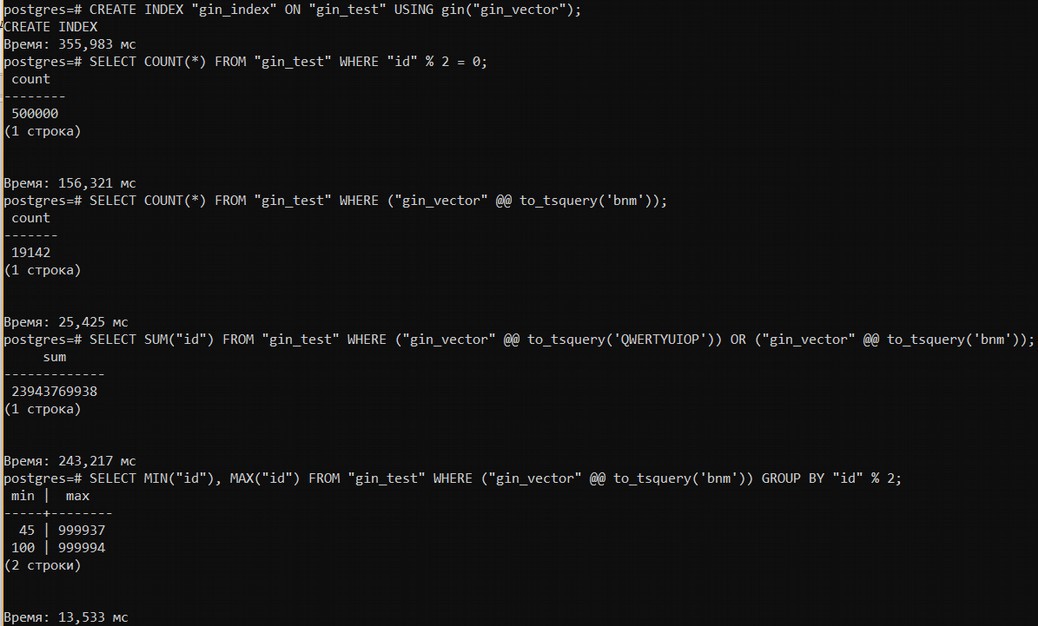
Створення індексу:

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.

# 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 "after\_update\_insert\_trigger" BEFORE DELETE OR UPDATE ON "trigger\_test" FOR EACH ROW

EXECUTE procedure after\_update\_insert\_func();

Текст тригера:

CREATE OR REPLACE FUNCTION after\_update\_insert\_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";

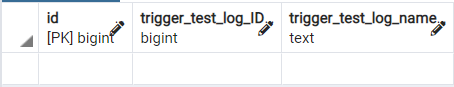
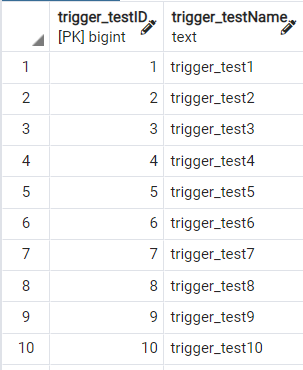
END LOOP; RETURN OLD;

END IF; END;

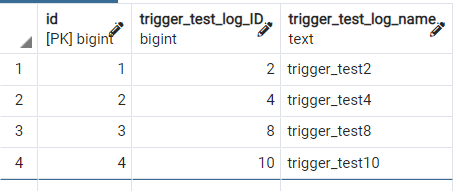
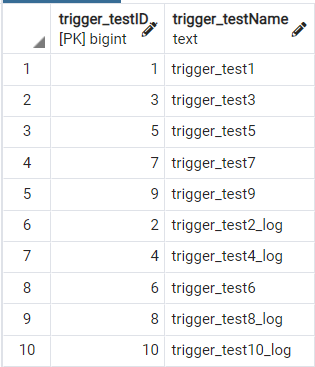
$trigger$ LANGUAGE plpgsql;

Скріншоти зі змінами у таблицях бази даних

Початковий стан

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

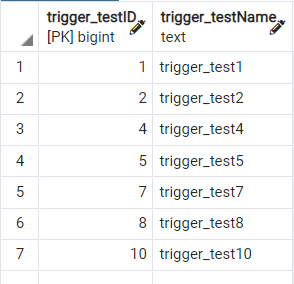
Після виконання запиту на оновлення

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 видалення потрібно повертати новий стан, а при запиті а оновлення старий.

Після виконання запиту на видалення

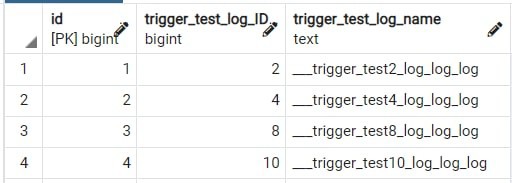
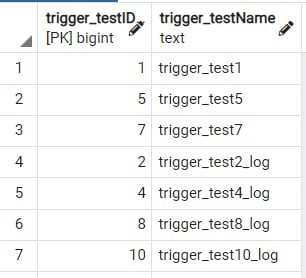
DELETE FROM "trigger\_test" WHERE "trigger\_testID" % 3 = 0;



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,

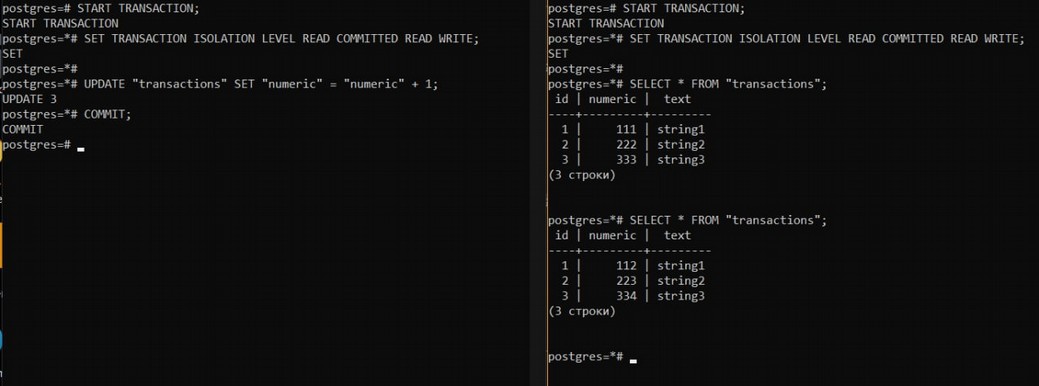
"text" text

);

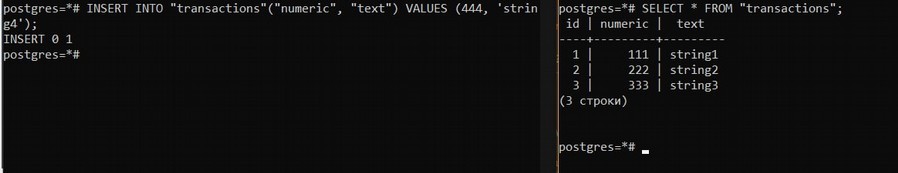
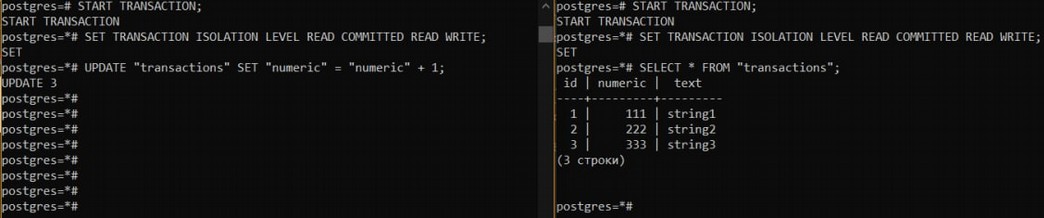
INSERT INTO "transactions"("numeric", "text") VALUES (111, 'string1'), (222, 'string2'), (333, 'string3');

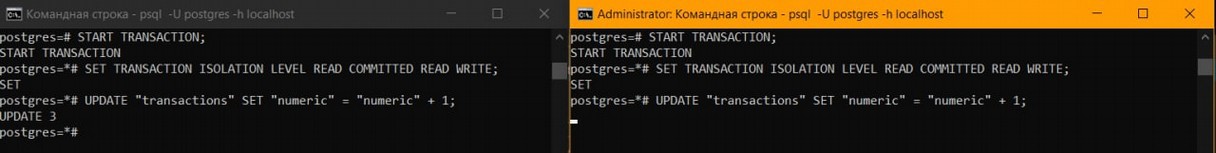
READ COMMITTED

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

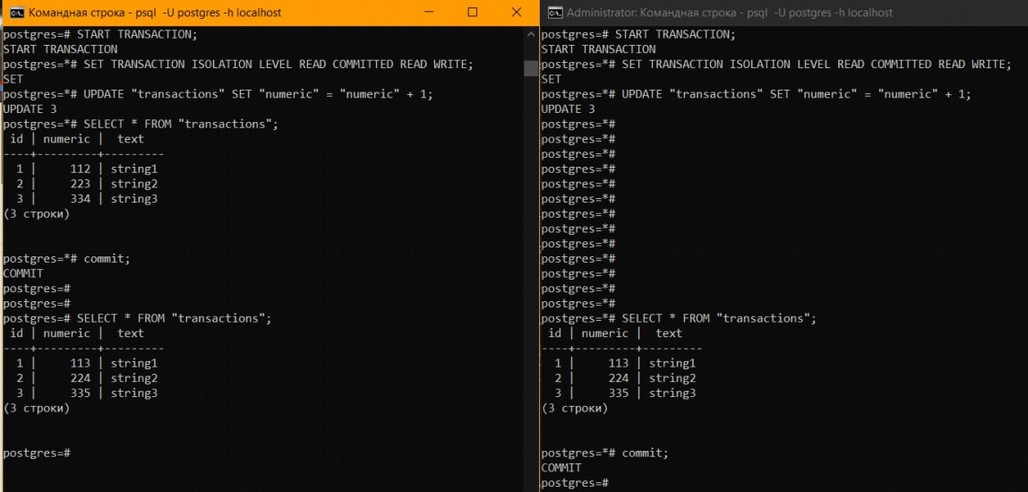


Дані після вставки та видалення так само будуть видні другій тільки після завершення першої.



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

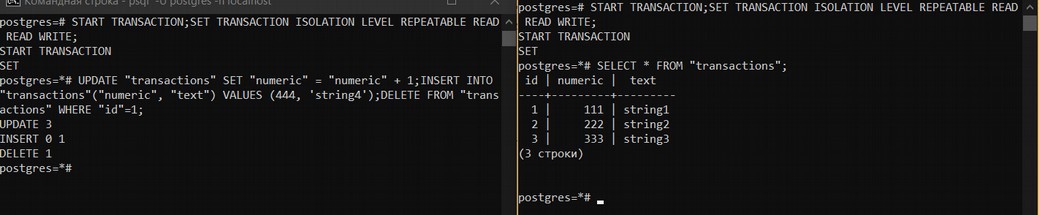


Коли Т2 бачить дані Т1 запитів UPDATE, DELETE виникає феномен повторного читаня, а коли бачить дані запиту INSERT – читання фантомів. Цей рівень ізоляції забезпечує захист від явища брудного читання.

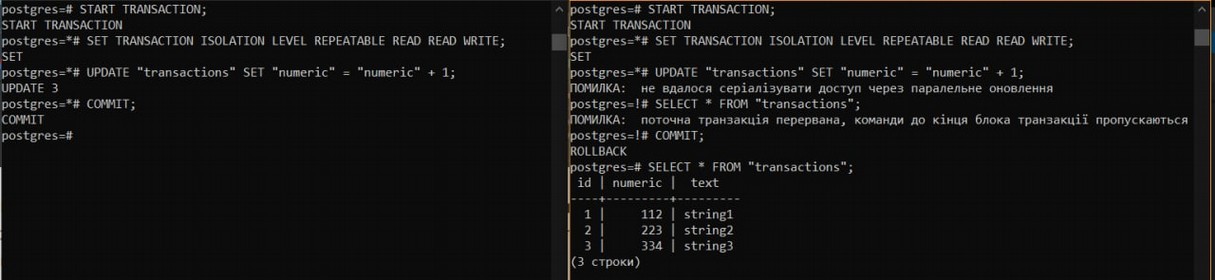
REPEATABLE READ

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

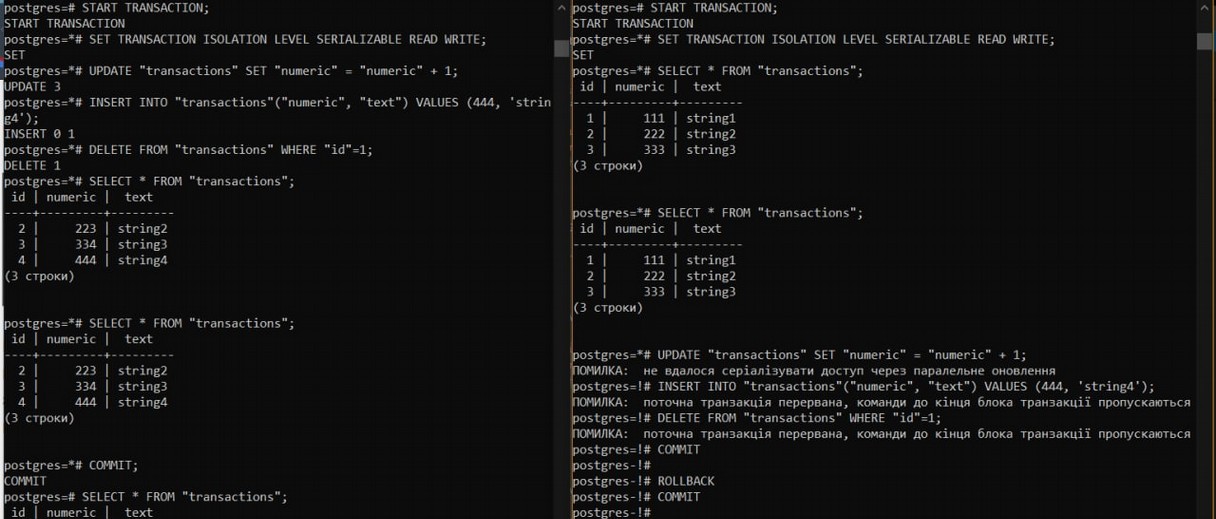
Тут видно, що друга не бачить змін з першої:



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.