Міністерство освіти та науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп’ютерних систем

***Лабораторна Робота №2***

з дисципліни

**Бази даних та засоби управління**

**Виконав:**

студент 3-го курсу,

групи КВ-23

Литвин Максим

Телеграм: @desp1c

**Київ 2024**

**Постановка задачі:**

1. Перетворити модуль “Модель” з шаблону MVC РГР 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.

[Посилання на github](https://github.com/DeSP1C/FAM-BD.git)

Варіант 14:

|  |  |  |
| --- | --- | --- |
| *14* | *Btree, Hash* | *after insert, update* |

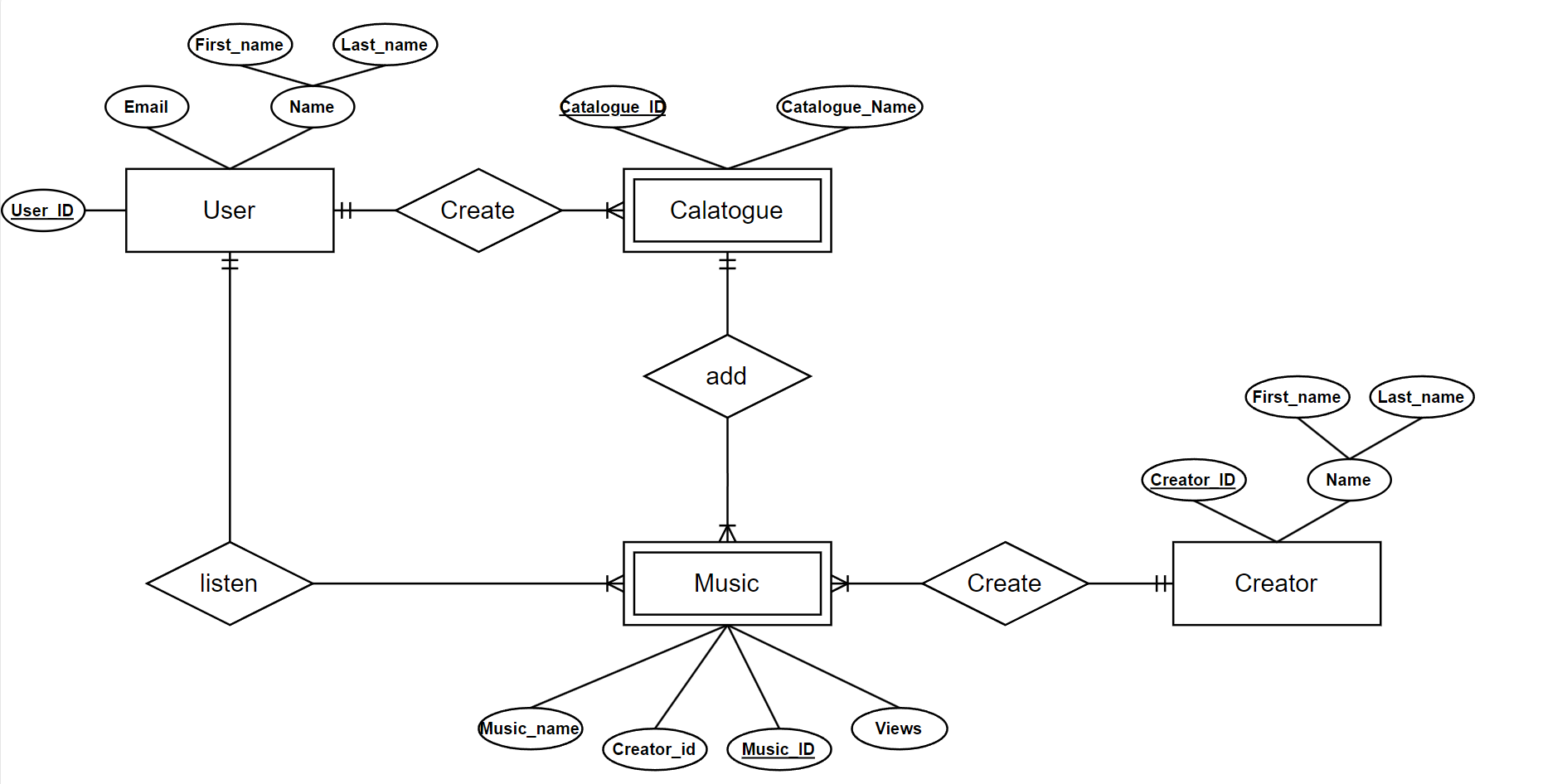
**Структура бази даних:**

User – користувач, який може створювати безліч каталогів, та має змогу додавати до кожного з них необхідну йому кількість музики до кожного каталогу.

Catalogue – каталог, який зберігає в собі посилання на довільну кількість музики, до якої через каталог має доступ користувач до прослуховування.

Music – музика, яка зберігає зв’язок між автором (творцем) та каталогом за яким вона закріплена в нашого користувача.

Creator – автор довільної кількості музики, яку може слухати наш користувач.



*ER-діаграма в стилі «нотації Чена»*

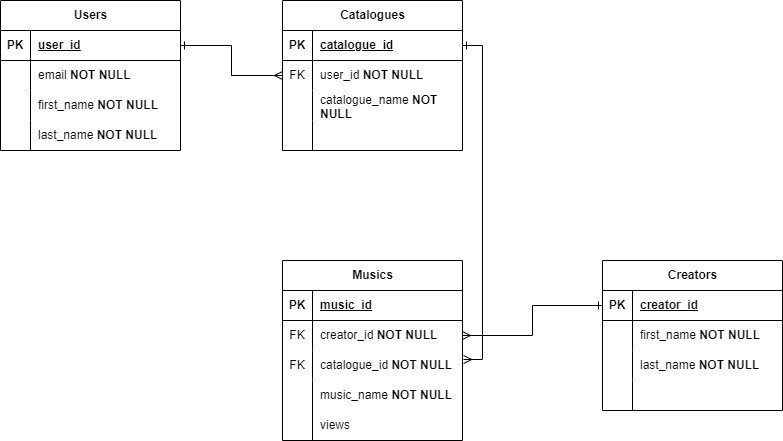
**Зв’язки:**

1:N – між user та catalogue (один користувач може створити безліч каталогів) – create(1)

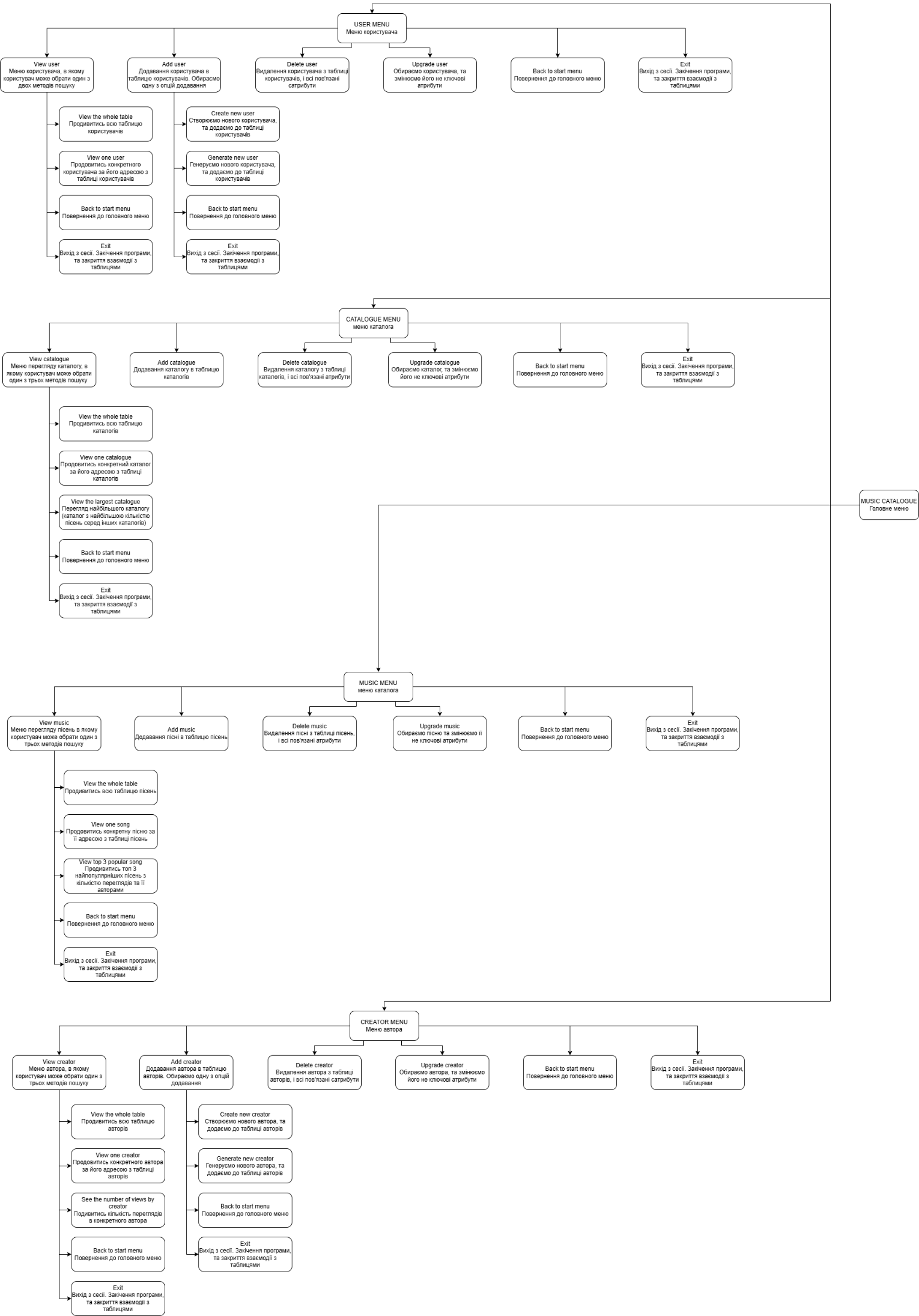
1:N – між user та music (один користувач може слухати безліч музики) – listen

1:N – між catalogue та music (один каталог може мати безліч музики) – add

1:N – між music та creator (один автор може створити безліч музики) – create(2)



*Структура бази даних*

****

*Структура меню користувача*

**Мова програмування та додаткові бібліотеки:**

Java, та базовий набір бібліотек.

Додаткові бібліотеки: [postgresql](https://jdbc.postgresql.org/download/), [hibernate](https://hibernate.org/orm/)

Етап №1:

Сутності:

User

@Entity  
@Table(name = "users")  
public class User {  
 @Id  
 @Column(name="user\_id", unique = true, nullable = false)  
 private int userId;  
  
 @Column(name="email", length = 30)  
 private String email;  
  
 @Column(name="first\_name", length = 20)  
 private String firstName;  
  
 @Column(name="last\_name", length = 20)  
 private String lastName;  
  
  
 public User(){  
  
 }  
  
 public User(int userId, String email, String firstName, String lastName){  
 this.userId = userId;  
 this.email = email;  
 this.firstName = firstName;  
 this.lastName = lastName;  
 }  
  
 public int getUserId() {  
 return userId;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }

Catalogue

@Entity  
@Table(name="catalogues")  
public class Catalogue {  
 @Id  
 @Column(name="catalogue\_id", unique=true, nullable = false)  
 int catalogueId;  
  
 @Column(name="catalogue\_name", length=20, nullable = false)  
 String catalogueName;  
  
 @ManyToOne(fetch = FetchType.EAGER)  
 @JoinColumn(name = "user\_id", referencedColumnName = "user\_id", nullable = false)  
 private User user;  
  
 @OneToMany(mappedBy = "catalogue", cascade = CascadeType.ALL, fetch = FetchType.LAZY)  
 private List<Music> musics;  
  
 public Catalogue(){  
  
 }  
  
 public Catalogue(int catalogueId, String catalogueName, User user){  
 this.catalogueId = catalogueId;  
 this.catalogueName = catalogueName;  
 this.user = user;  
 }  
  
 public void setUser(User user) {  
 this.user = user;  
 }  
  
 public int getCatalogueId() {  
 return catalogueId;  
 }  
  
 public String getCatalogueName() {  
 return catalogueName;  
 }  
  
  
 public User getUser() {  
 return user;  
 }  
}

Music

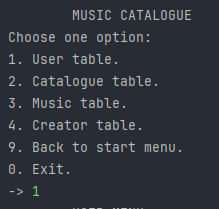
@Entity  
@Table(name="musics")  
public class Music {  
 @Id  
 @Column(name="music\_id", length = 5, unique = true, nullable = false)  
 int musicId;  
  
 @Column(name="music\_name", length = 20, nullable = false)  
 String musicName;  
  
 @Column(name="views", nullable = false)  
 int views;  
  
 @ManyToOne(fetch = FetchType.EAGER)  
 @JoinColumn(name = "catalogue\_id", referencedColumnName = "catalogue\_id", nullable = false)  
 private Catalogue catalogue;  
  
 @ManyToOne(fetch = FetchType.EAGER)  
 @JoinColumn(name = "creator\_id", referencedColumnName = "creator\_id", nullable = false)  
 private Creator creator;  
  
 public Music(){  
  
 }  
  
 public Music(int musicId, String musicName, Creator creator, Catalogue catalogue, int views){  
 this.musicId = musicId;  
 this.musicName = musicName;  
 this.creator = creator;  
 this.catalogue = catalogue;  
 this.views = views;  
 }  
  
 public int getMusicId() {  
 return musicId;  
 }  
  
 public String getMusicName() {  
 return musicName;  
 }  
  
 public Creator getCreator() {  
 return creator;  
 }  
  
 public Catalogue getCatalogue() {  
 return catalogue;  
 }  
  
 public int getViews() {  
 return views;  
 }  
  
 public void setCatalogue(Catalogue catalogue){  
 this.catalogue = catalogue;  
 }  
  
 public void setCreator(Creator creator) {  
 this.creator = creator;  
 }  
}

Creator

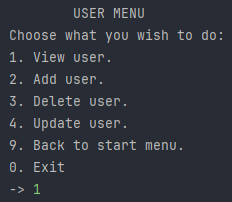
@Entity  
@Table(name="creators")  
public class Creator {  
 @Id  
 @Column(name="creator\_id", length=5, unique=true, nullable = false)  
 int creatorId;  
  
 @Column(name="first\_name", length=20, nullable = false)  
 String firstName;  
 @Column(name="last\_name", length=20, nullable = false)  
 String lastName;  
  
 public Creator(){  
  
 }  
  
 public Creator(int creatorId, String firstName, String lastName){  
 this.creatorId = creatorId;  
 this.firstName = firstName;  
 this.lastName = lastName;  
 }  
  
 public int getCreatorId() {  
 return creatorId;  
 }  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
}

Скріншоти:

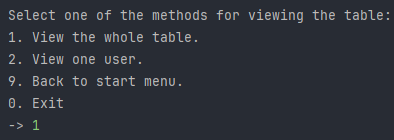
Головне меню:



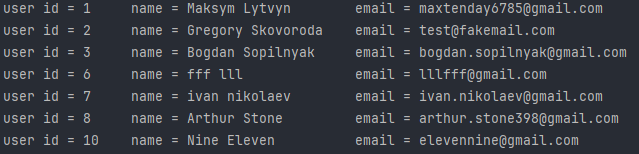
Меню користувача:



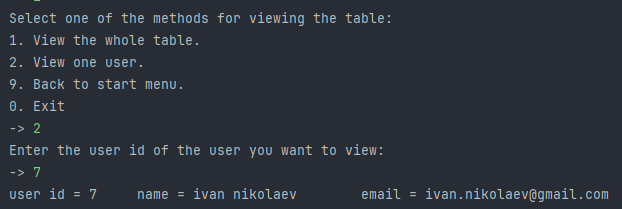
Меню перегляду:



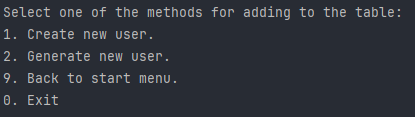
Результат перегляду всієї таблиці:



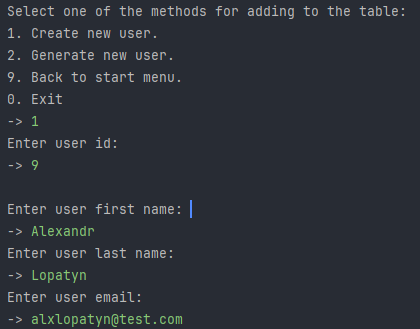
Результат перегляду одного користувача:



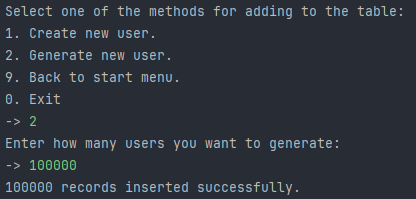
Меню додавання користувача:



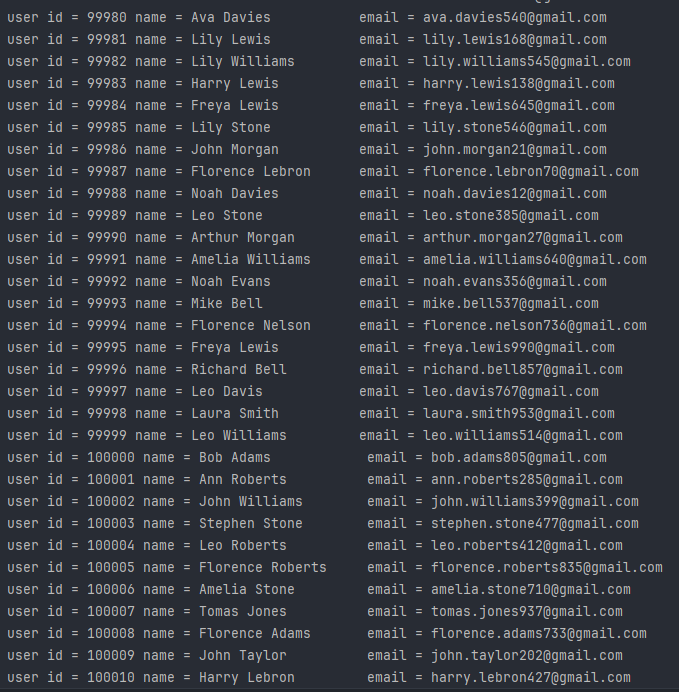
Додавання користувача власноруч:

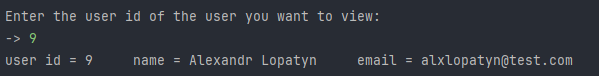


Додавання користувачів генерацією:

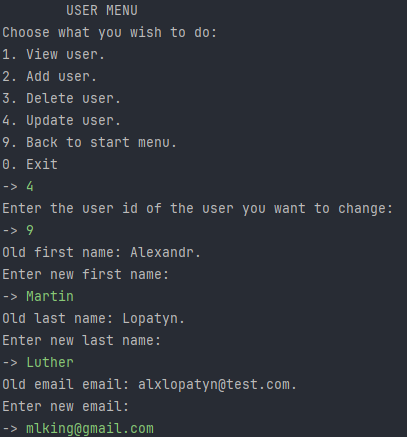


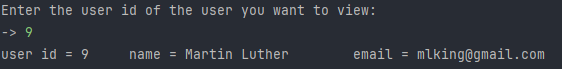
Перевіряємо результат:



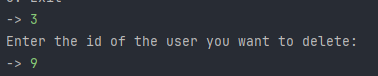
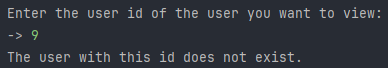


Оновлюємо користувача:

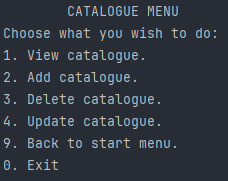




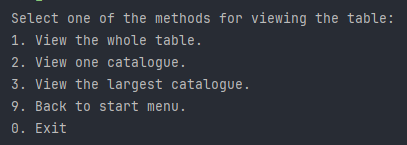
Видаляємо користувача:

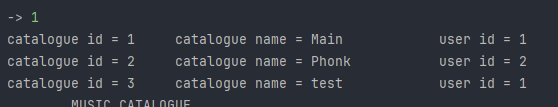
Меню каталогу:



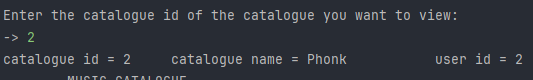
Меню перегляду:



Перегляд всієї таблиці:



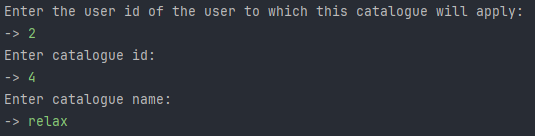
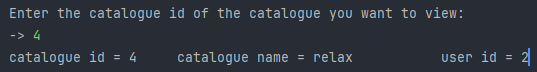
Перегляд конкретного каталогу:



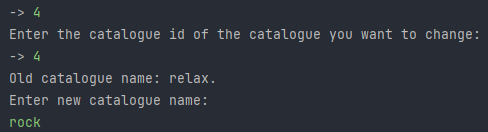
Перегляд найбільшого каталогу:

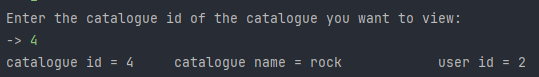


Додавання каталогу:

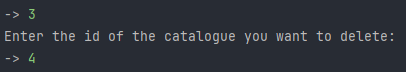
 

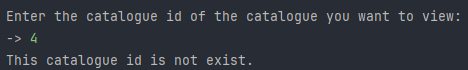
Оновлення каталогу:



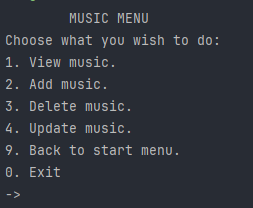


Видалення каталогу:

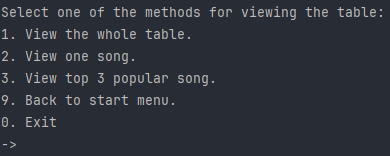




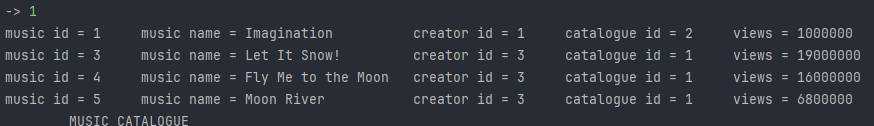
Меню музики:



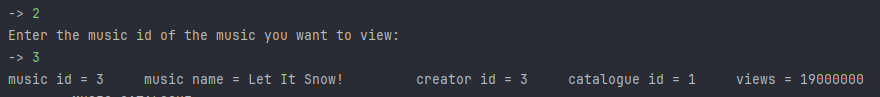
Меню перегляду:



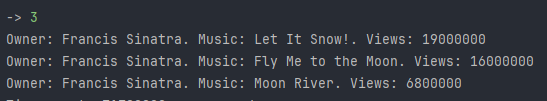
Перегляд всієї таблиці:



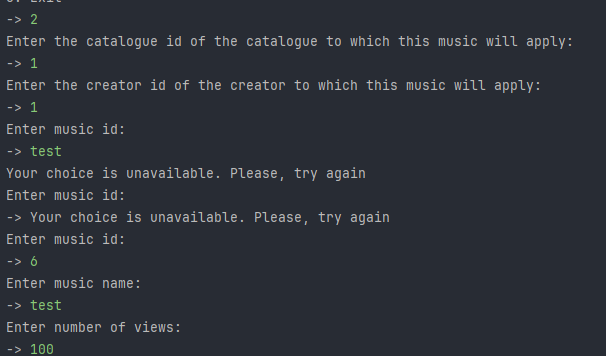
Перегляд конкретної пісні:

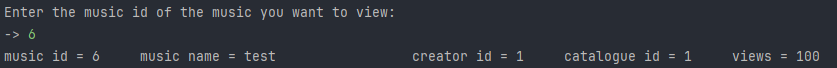


Перегляд топ 3 популярних пісень:

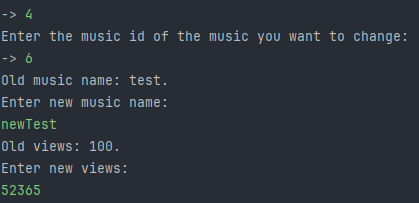


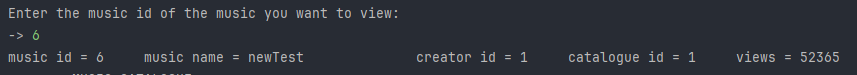
Додавання пісні:



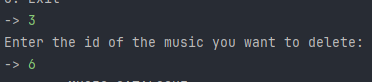


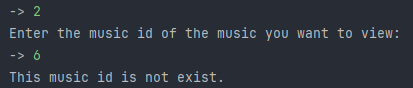
Оновлення пісні:



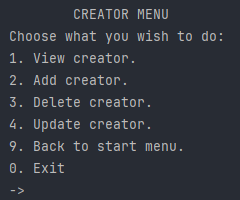


Видалення пісні:

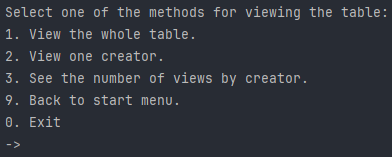




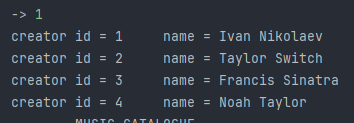
Меню творця:



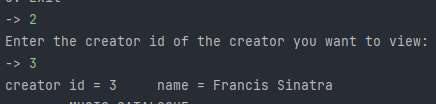
Меню перегляду:



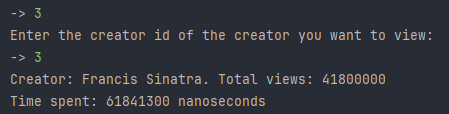
Перегляд всієї таблиці творців:



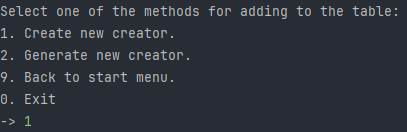
Перегляд конкретного творця:



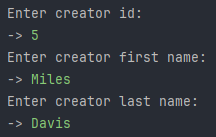
Перегляд сумарної кількості переглядів одного творця:

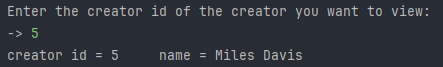


Меню додавання користувача:

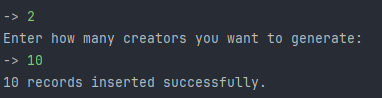


Додавання вручну:



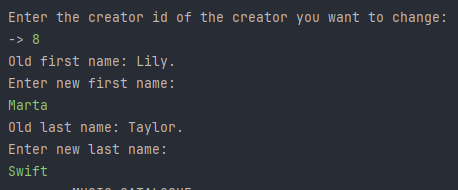


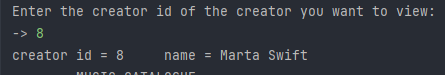
Згенеровані творці:



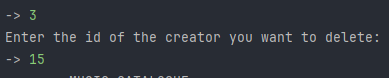


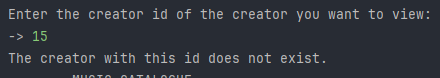
Оновлення творця:





Видалення творця:





Етап №2:

Індекси в PostgreSQL - спеціальні об'єкти бази даних, призначені в основному для прискорення доступу до даних. Це допоміжні структури: будь-який індекс можна видалити і відновити заново за інформацією в таблиці.

B-Tree – це індекс, яке використовується для організації індексів у базах даних. Воно дозволяє швидко знаходити, вставляти або видаляти дані. Ключі в B-Tree розташовані у впорядкованому вигляді, що забезпечує ефективний доступ до даних.

Hash – індекс базується на використанні хеш-функцій. Він обчислює хеш-значення для кожного ключа та використовує це значення для швидкого доступу до рядків.

B-Tree.

Плюси:

* Має швидкий пошук, та складність O(log2 n) для пошуку, вставки та видалення
* Підтримує діапазони з операціями, як: <, >, BETWEEN, LIKE..
* Підтримує впорядкованість навіть при частих змінах даних.

Мінуси:

* При правильному використанні, програє в швидкості Hash індексам, коли йде мова про пошук точних збігів
* Використовує багато пам’яті, особливо, коли йде мова про великі таблиці

Hash:

Плюси:

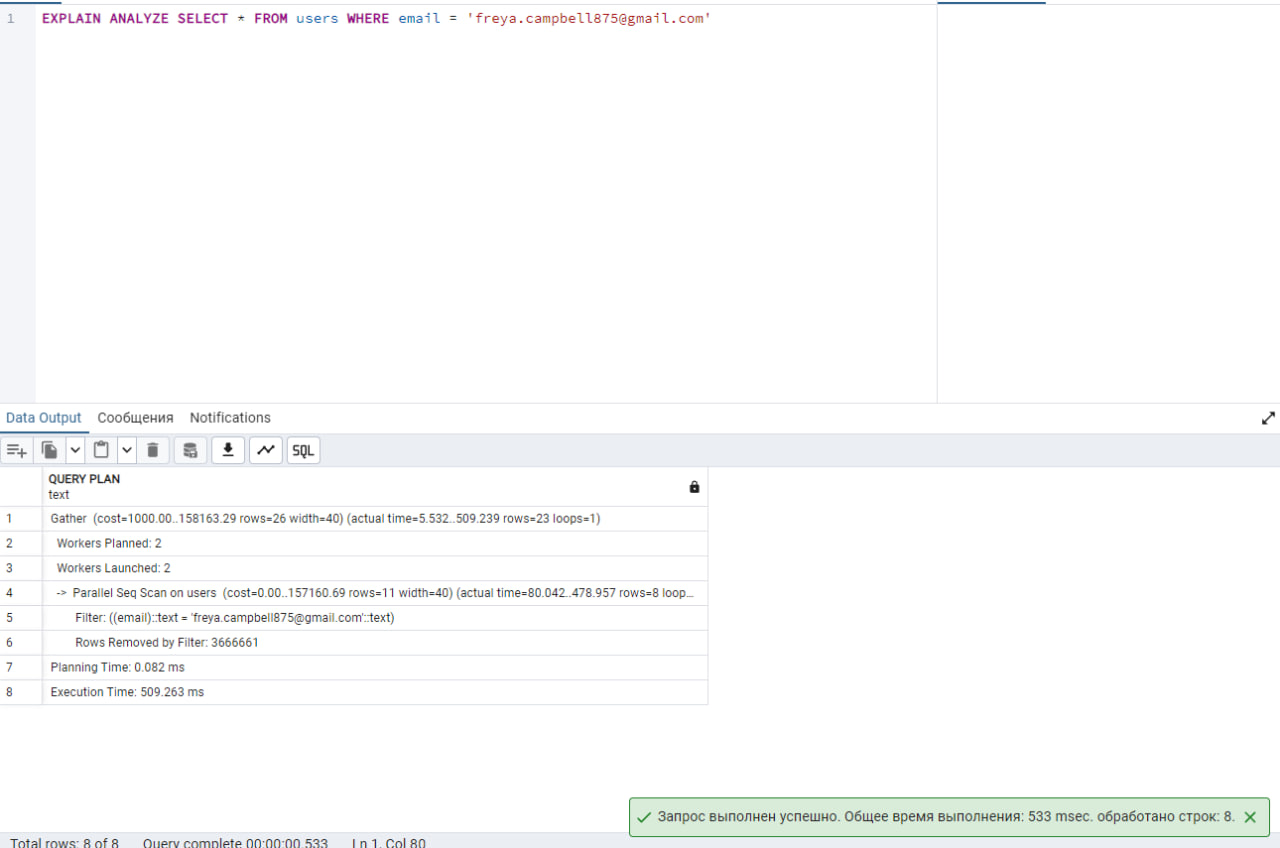
* Один з найшвидкіших індексів, коли йде мова про точний пошук
* Займає менше пам’яті, бо не зберігає порядок ключів

Мінуси:

* Не підтримує діапазони, як це робить умовний B-Tree
* Не рекомендується використовувати при великій кількості оновлень, оскільки хеш-функція може створювати більше колізій при великих обсягах даних
* Не всі системи баз даних повністю можуть підтримати Hash індекси.

Як висновок, B-Tree можна використовувати як універсальний індекс, оскільки всюди він показує непогані результати, при великих затратах пам’яті. Коли як Hash потрібен лише тоді, коли ми маємо знайти якісь конкретні рядки, не використовуючи діапазони.

Запит №1 (без використання індексів)



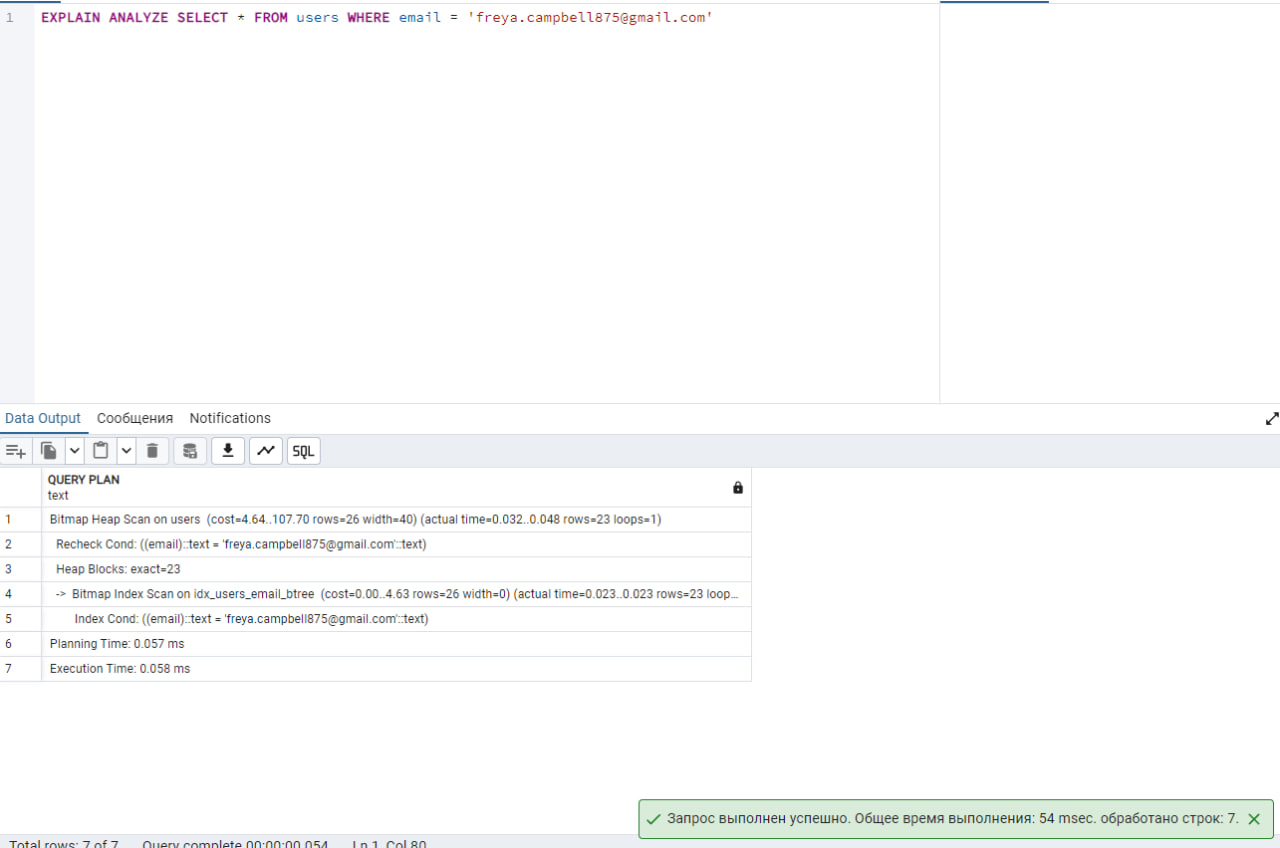
Тут можемо помітити, що при звичайному пошуку за email-ом час запиту займає пів секунди, що досить довго. Тепер, використаємо індекси.

Запит №2 (з використанням індексу B-Tree)

Індекс створюємо завдяки спеціальній команді:



Тепер, виконаємо той самий запит, але вже з використанням індексу:



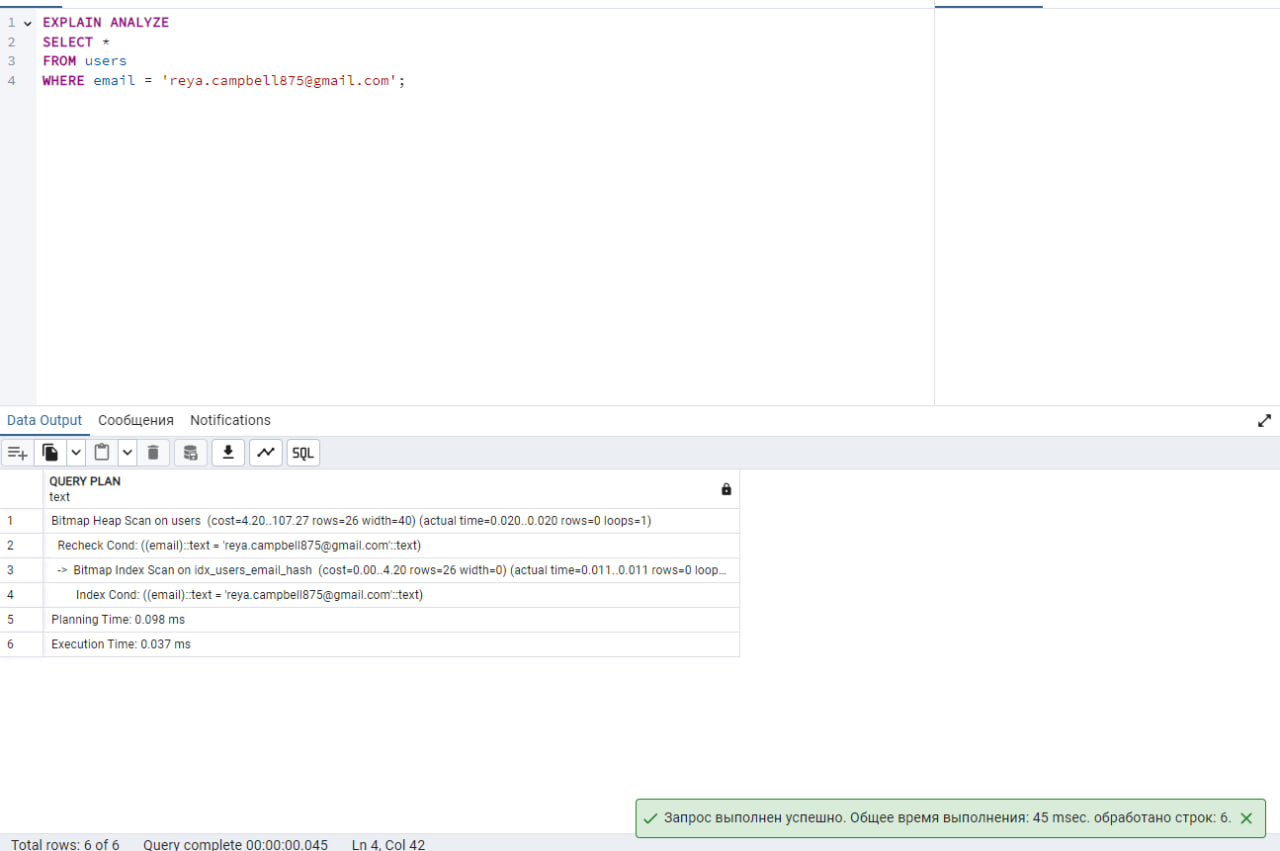
Як бачимо, прискорення відбулося майже в цілих 10 разів, що демонструє високу ефективність індексу B-Tree.

Запит №3 (з використанням Hash індексу)

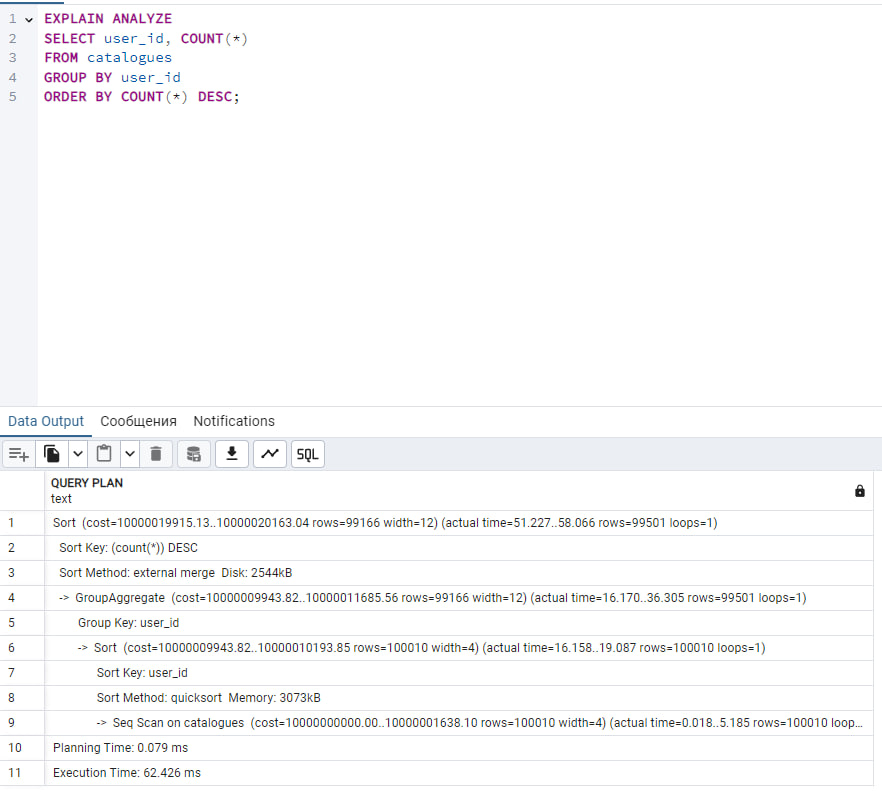
Створюємо індекс:



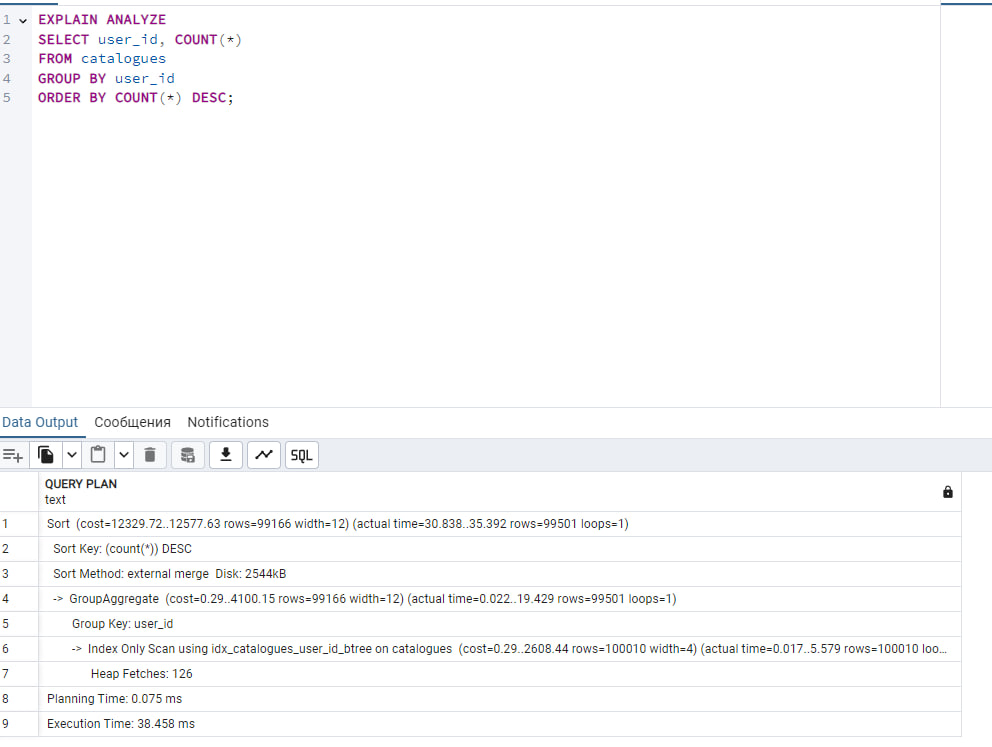
Повторюємо наш запит:



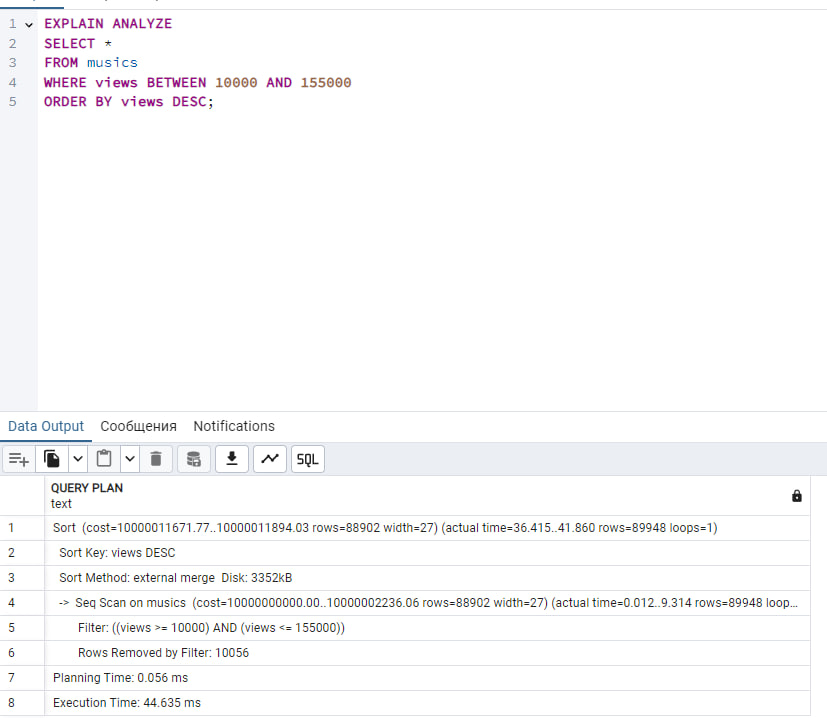
І, як можемо помітити, Hash індекс в цьому випадку спрацював навіть швидше, ніж B-Tree індекс (як це ми обговорювали раніше). А відносно звичайного виклику, то він швидше майже в 15 разів, що є неймовірно швидко!

Запит №4 (без використання індексів)  
Тут ми використали суміш агрегатної та функції сортування. І як бачимо, звичайний пошук і так займає не так багато часу, всього лише 62 мс. Давайте аподивимось, як з цим впорається B-Tree індекс.

Запит №5 (з використанням індекса B-Tree)



І бачимо, що результат з B-Tree індексом навіть за цих умов видає результат майже вдвічі швидше, що демонструє нам високу ефективність використання індексів.

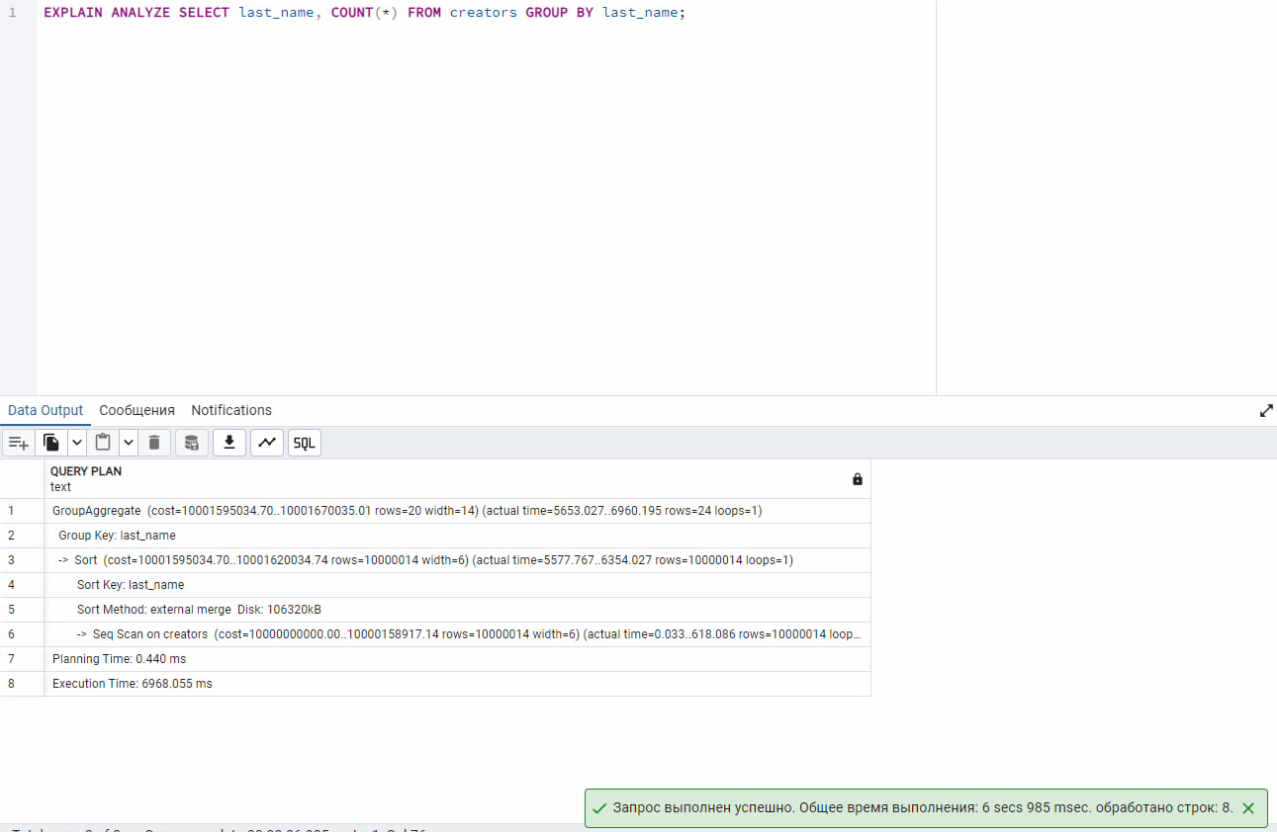
Запит №6 (без використання індекса)

Тут ми використовуємо пошук за діапазоном та сортуванням, а саме шукаємо поміж всіх пісень пісні, в яких переглядів від 10000 до 155000. Звичайним запитом без індексів вдалося знайти всі пісні всього лише за 45 мс. Подивимось, який результат буде з B-Tree індексом.

Запит №7(з B-Tree індексом)

  
І, як можемо помітити, з B-Tree індексом наш запит був у 1.5 рази швидше за звичайний. І хочу відмітити, що в цій таблиці було лише 100000 пісень. При більш високому обсязі таблиці (хоча б, як в users та creators, де були під 10 мільйонів даних), результати були б більш приємні!

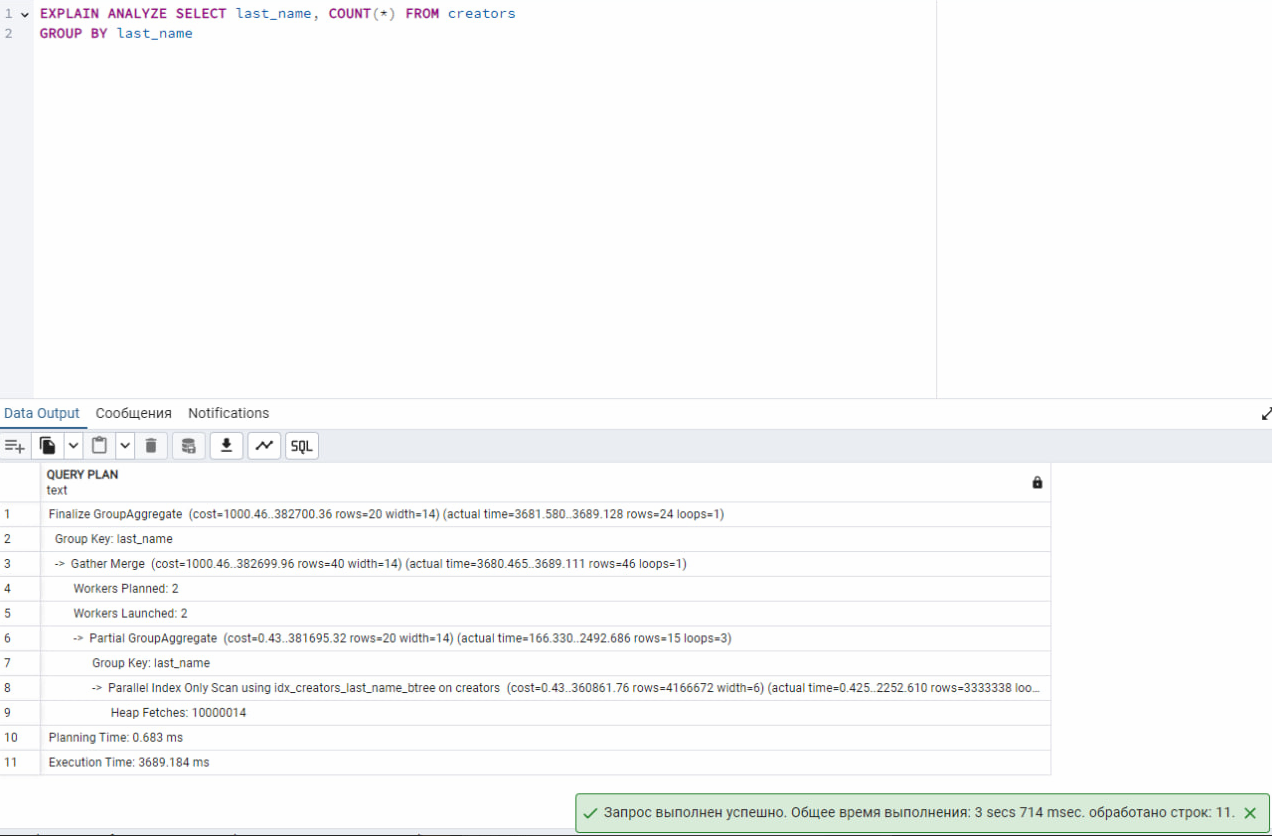
Запит №8 (без індексування)



Цей запит рахує кількість прізвищ, і виводить самі прізвища, та їх кількість у таблиці з групуванням. B-Tree індекс не дуже полюбляє працювати з купою однакових клітинок. Але, подивимось на результат.

Запит №9 (з B-Tree індексом)

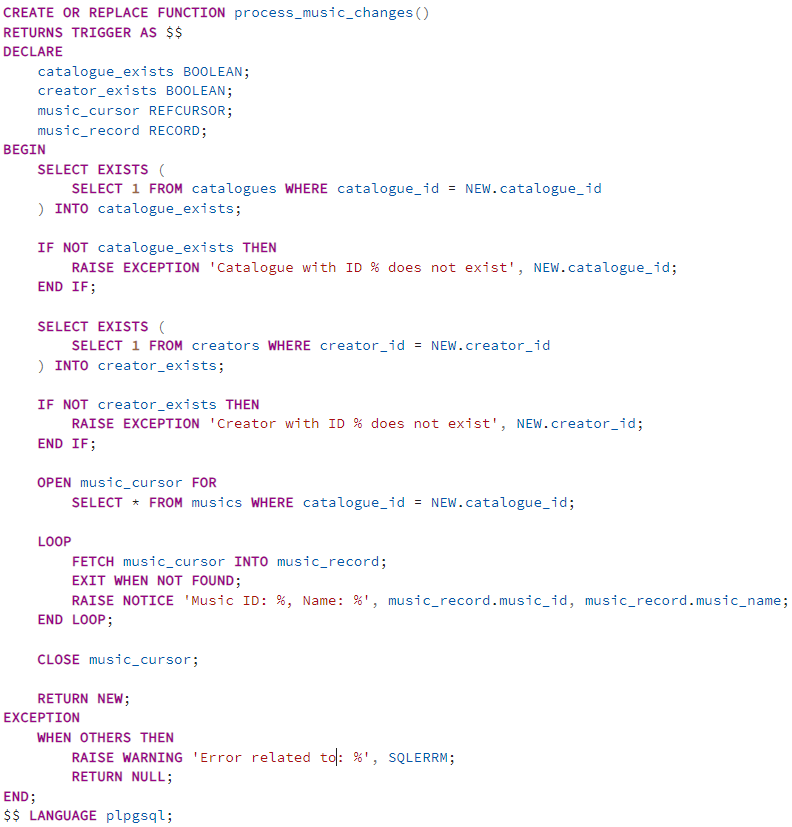




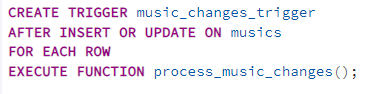
І як бачимо, B-Tree спрацював у двічі швидше, навіть не з його «ідеальним» варіантом таблиці. Тим самим, можемо побачити, наскільки ефективно можна використовувати індекси.

Етап №3:

Тригери в PostgreSQL — це спеціальні механізми, які дозволяють автоматично виконувати деякі функції у відповідь на зміни, що відбулися в таблицях бази даних. Тобто, це свого роду підключення до певних подій (вставлення, оновлення, видалення даних), які запускають наш код.

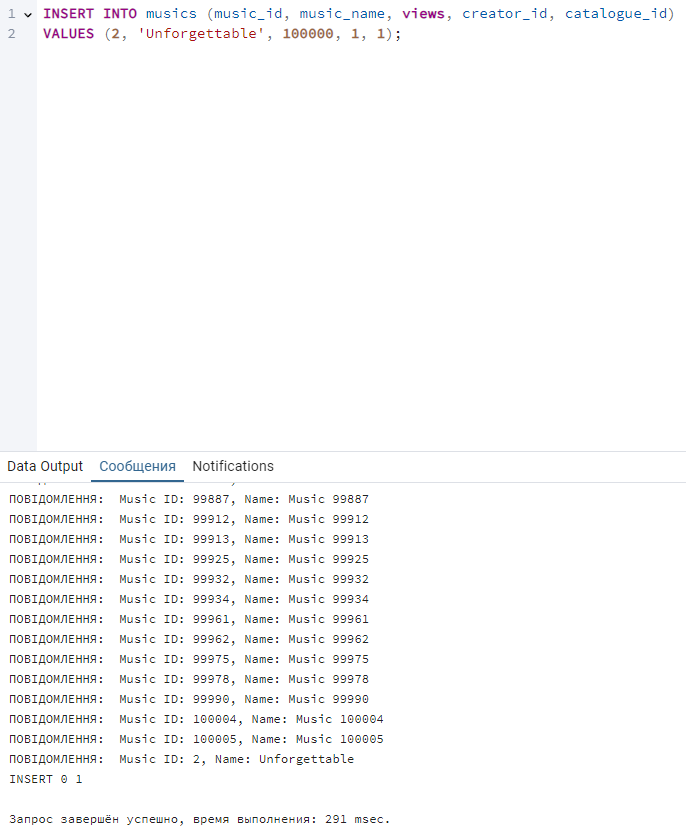
Функція тригера:  


Створення тригера:



Використання тригера:

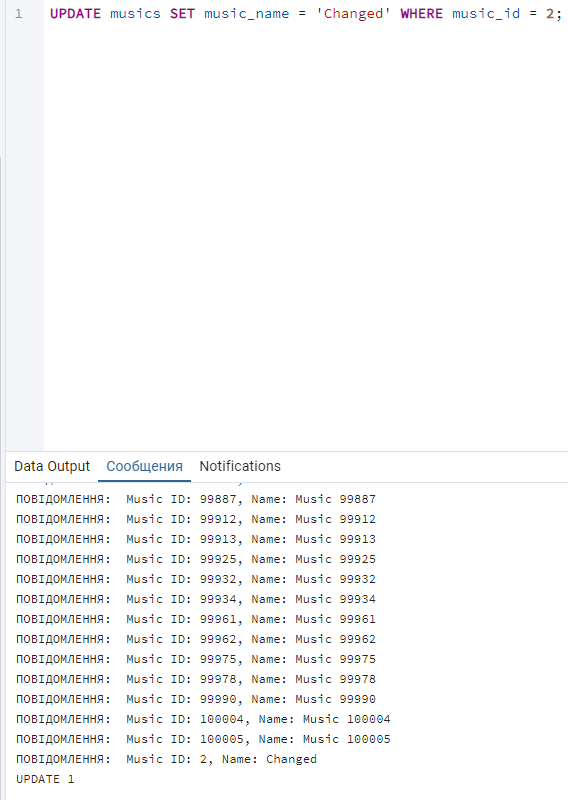
Вставка з вже існуючим каталогом та виконавцем:



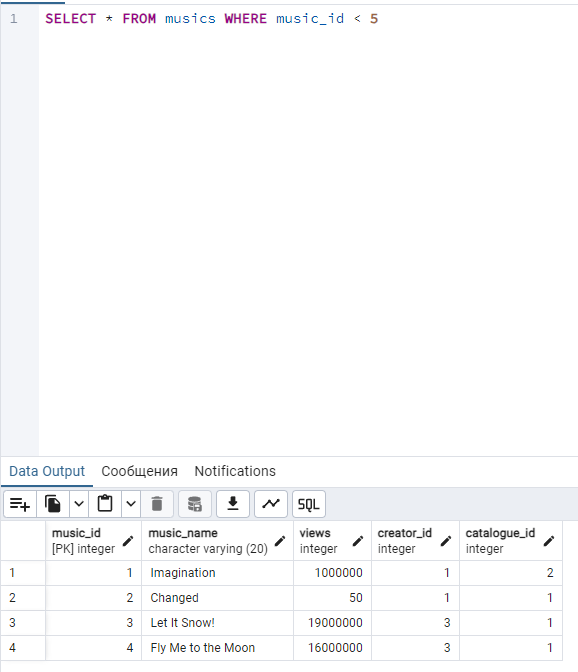
Вставка з неіснуючим каталогом:



Оновлення пісні:

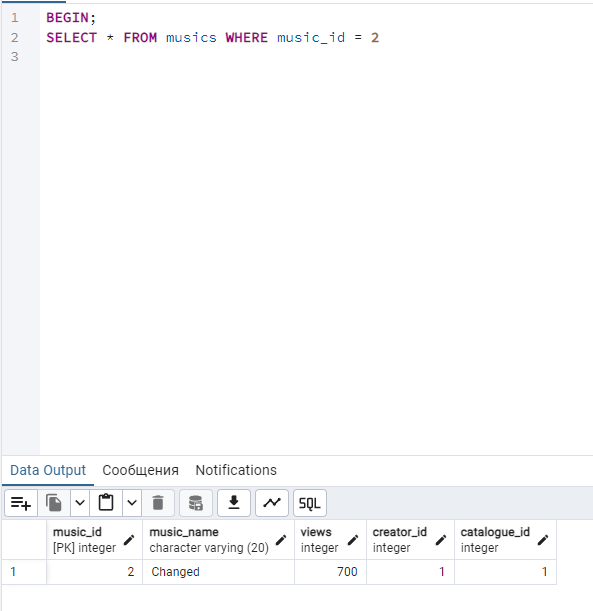


І дійсно, можемо побачити, що зміни спрацювали корректно:

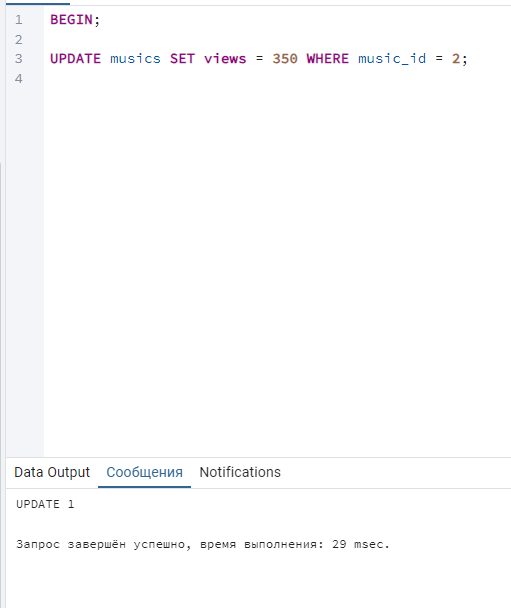


Етап №4:

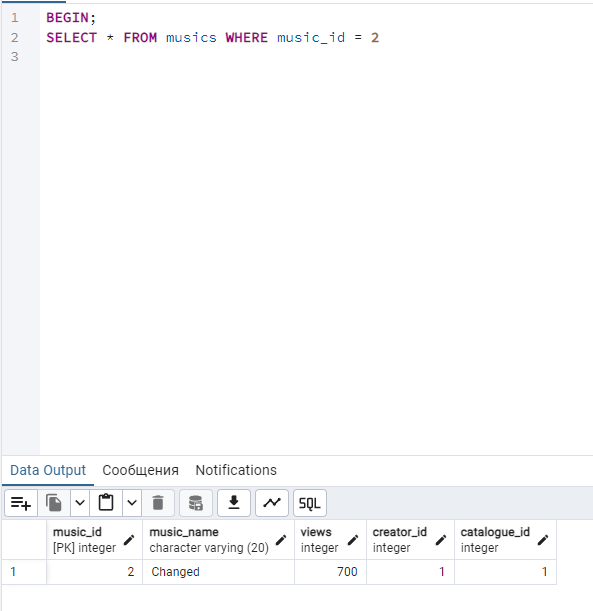
READ COMMITED – це найслабший рівень ізоляції, коли транзакція може бачити результати інших транзакцій, навіть, якщо вони ще не закомічені (не збережені)  
Транзакція 1



Транзакція 2



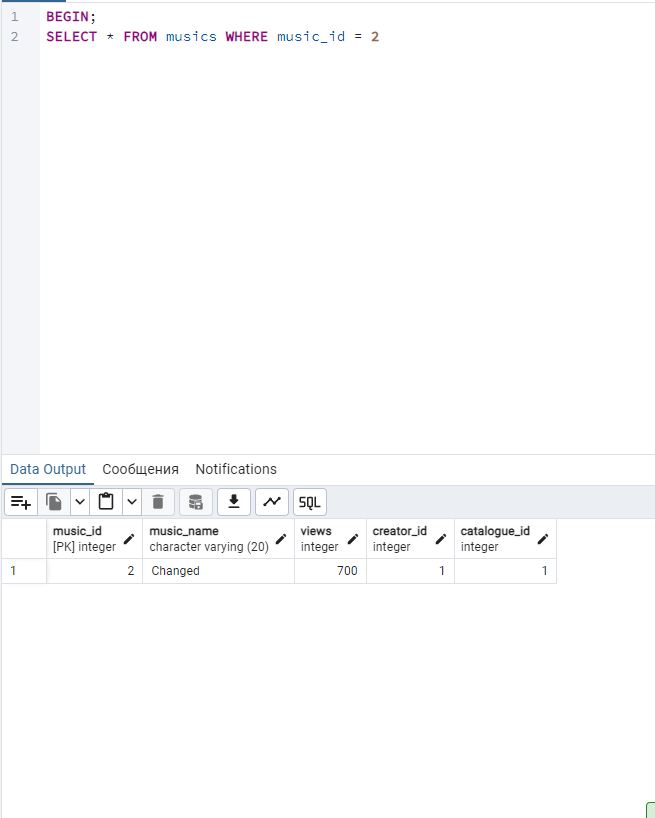
Транзакція 1



Брудне читання (dirty read) – явище, коли дані, які вже були прочитані, хтось може відкотити ще до того, як ми завершимо нашу транзакцію.  
Брудне читання погане тим, що в моменті, коли ми хочемо з таблиці взяти потрібні нам дані, та вже потім з ними працювати, через цей феномен ми можемо неправильно обробити дані, та видати не вірний результат, оскільки хтось іншим відкотив зміни в своїй транзакції, та ті дані, які ми «брудно» прочитали, вже не є актуальними.

В нашому випадку «dirty read» не відбувся. Але, подібне цілком можливо.

Транзакція №1



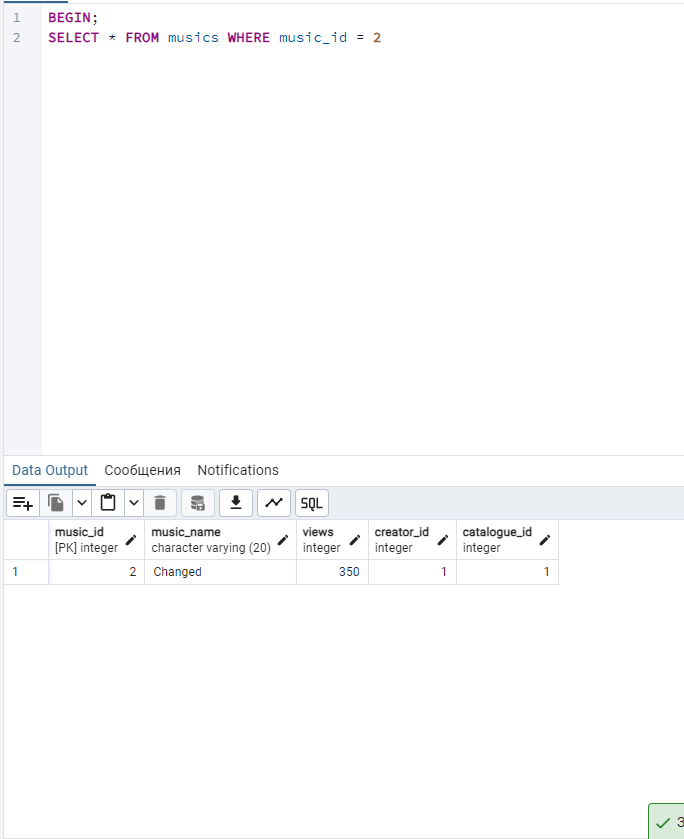
Ми викликали транзакцію, читаємо інформація з таблички, але не комітемо її.

Транзакція №2



Далі, змінюємо в другій транзакції views на 700, та комітемо. І зараз, в нас висить **транзакція 1**, яка ще не закомічена, але вже читає застарілі дані. І в разі коміту, вона може повторно прочитати таблицю, і ми матимемо в результаті два результати, які між собою є різними.

Транзакція 1

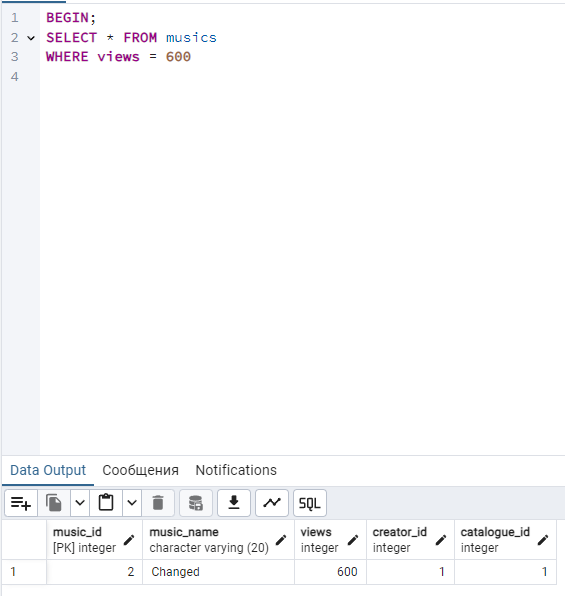


Ця проблема називається «Non-repeatable Read».

Неповторюване читання (Non-repeatable Read) – це коли дані, котрі ми прочитали, хтось може змінити ще до того, як ми завершимо нашу транзакцію. Тобто, значення одного й того ж рядка може змінюється не в межах транзакції між двома читаннями.

Транзакція 1

Ми читаємо дані з таблиці:



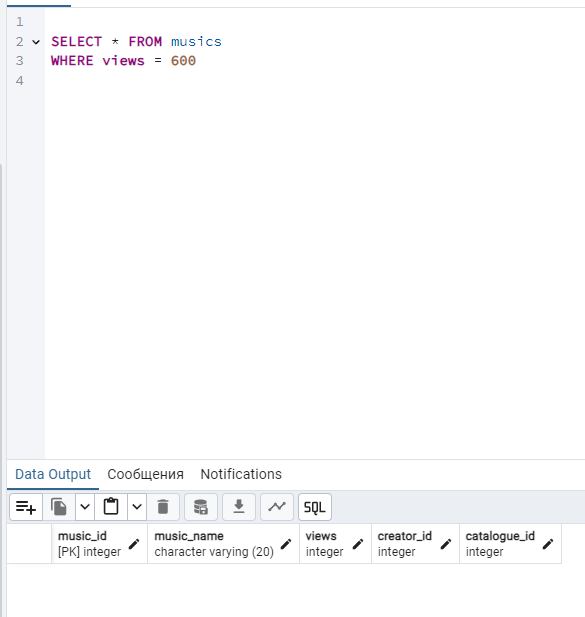
І наприклад, під час нашої транзакції, хтось у своїй (транзакція 2) транзакції поновлює дані таблиці:

Транзакція 2



І вже в транзації один, коли ми ще не завершили нашу транзакцію виникає проблема, і помічаємо що на час коміту, таких пісень, де views = 600 вже нема.

Транзакція 1



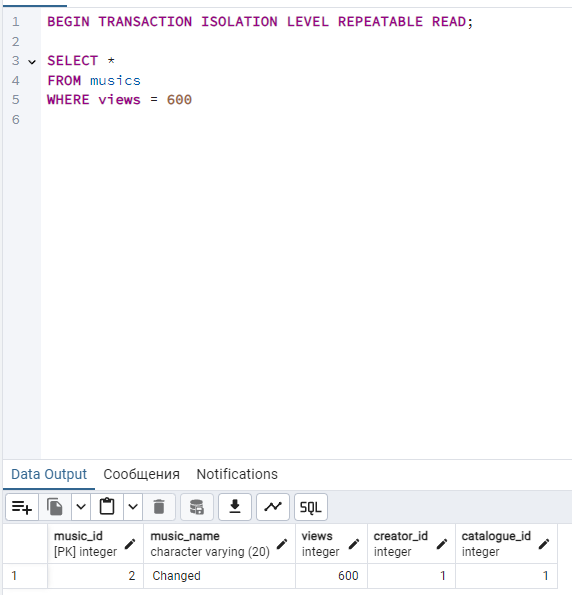
Цей феномен називається «Phantom Read»

Фантомне читання (Phantom Read) – це коли певні дані, які ми прочитали, хтось може видалити або додати ще до того, як ми завершимо власну транзакцію. Тобто, під час повторного виконання запиту з’являються або зникають нові рядки, яких раніше не було

REPEATABLE READ – цей рівень означає, що поки транзакція не буде завершена, ніхто не зможе змінювати або видаляти рядки, які транзакція вже прочитала.

Цей рівень ізоляції рятує нас від таких проблем, як «dirty read» та «non-repetable read». Але, все ще залишається проблема «Фантомного читання».  
  
Причина? Тому що при даній ізоляції хоч нам і не дають змогу видаляти, або змінювати строки, але ми все ще можемо їх додавати паралельно.  
  
І давайте перевіримо, чи це так:

Транзакція 1

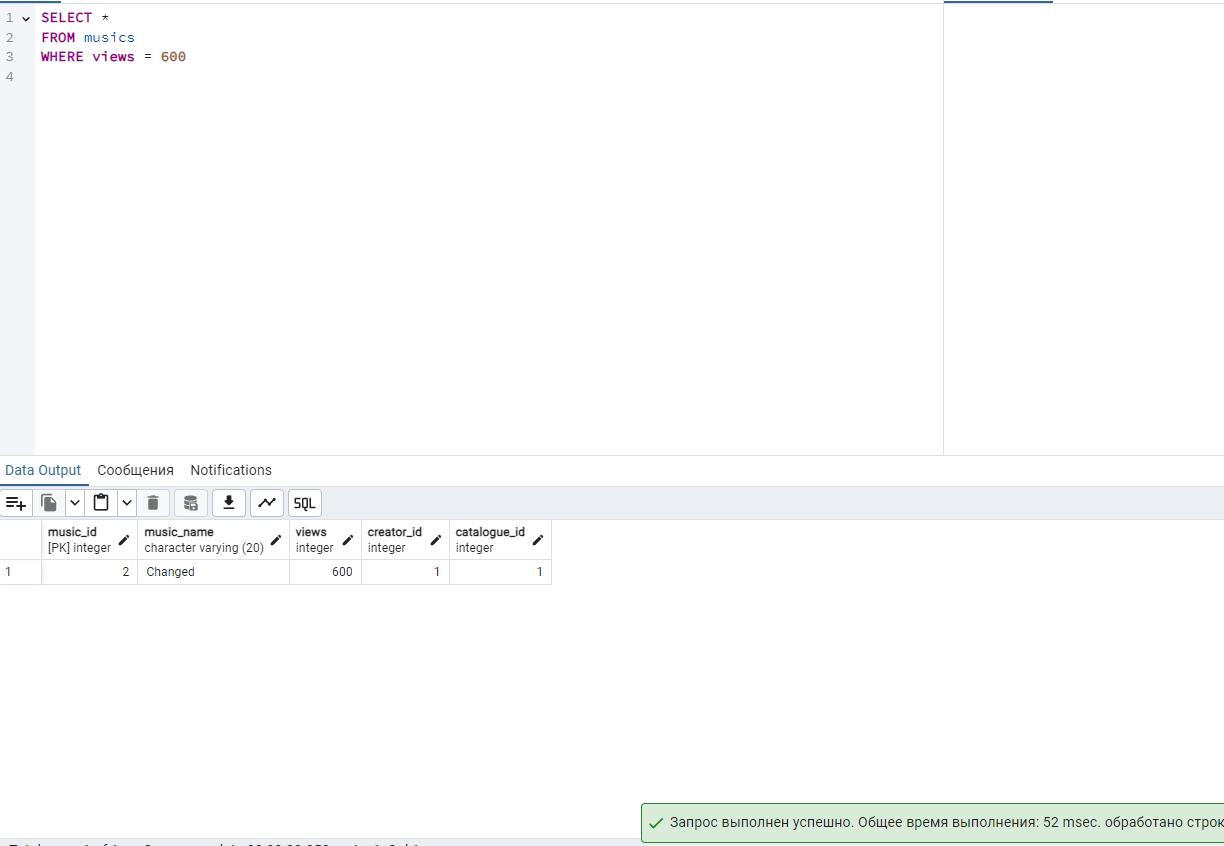


Транзакція 2



Оновлюємо дані. Тепер, якщо «брудне читання» працює, ми маємо побачити результат 0:

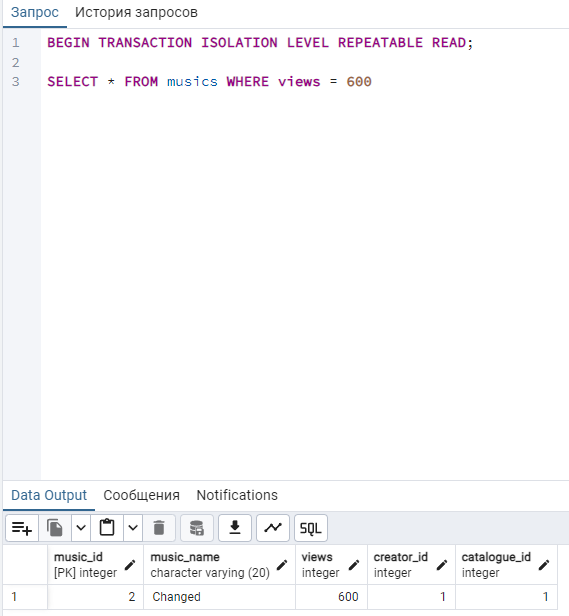
Транзакція 1



Але, як ми бачимо, все працює корректно.

Тепер, подивимось на феномен фантомного читання:

Транзакція 1



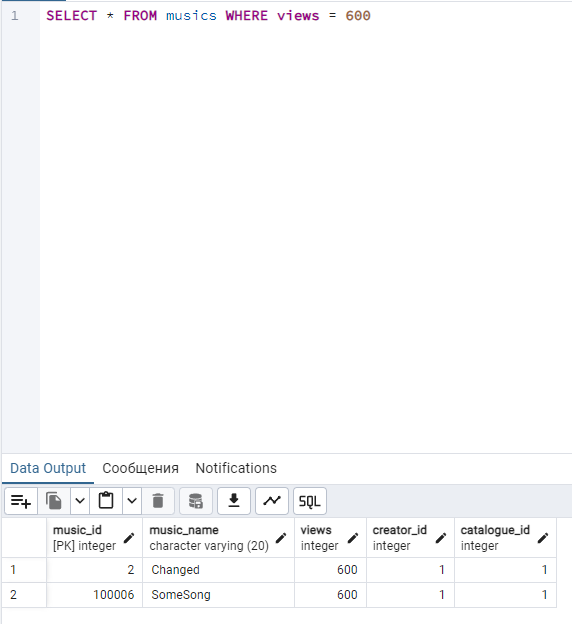
Як бачимо, лише одна пісня має 600 переглядів.

Тепер, додамо в іншій транзакції новий рядок (пісню), в якої буде 600 переглядів:

Транзакція 2



Транзакція 1



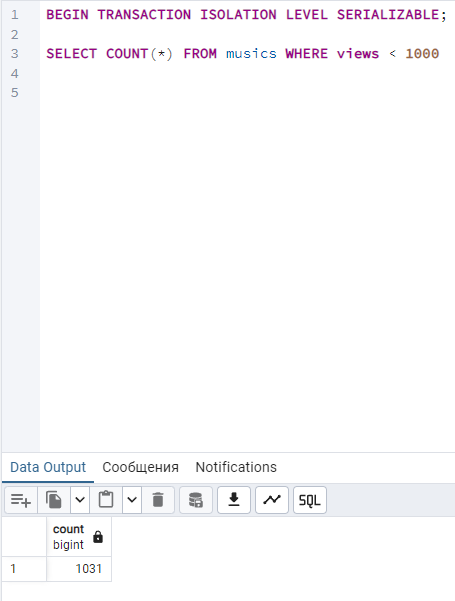
Ось таким чином мало б спрацювати фантомне читання.

SERIALIZABLE – найбезпечніший, але найтяжчий для баз даних та повільний для обробки запитів. Він блокує будь-які дії, доки працює транзакція – транзакції йдуть одна за одною і максимально ізолюються одне від одного. Це трапляється завдяки блокуванню всієї таблиці від будь-яких взаємодій з нею. Але, деякі СУБД роблять її менш радикальною, а саме – блокують тільки ті строки, котрі задіюють нинішню транзакцію або діапазон рядків.

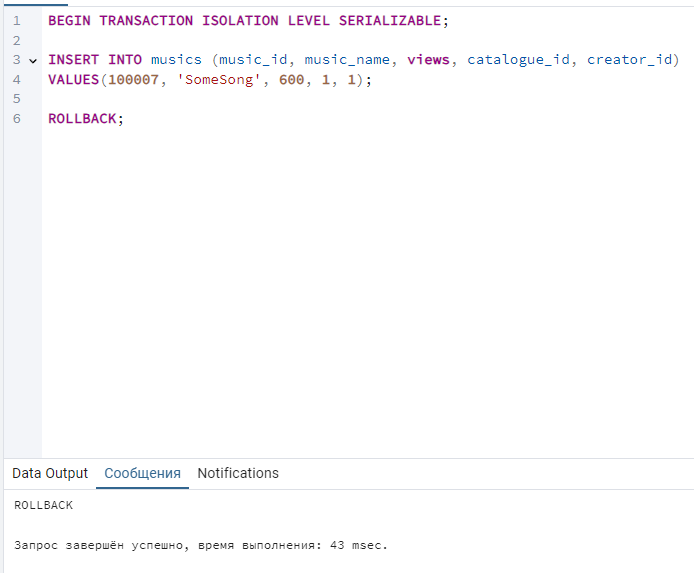
Перевіримо всі три випадки (dirty read, non-repeatable read, phantom read):

dirty read:

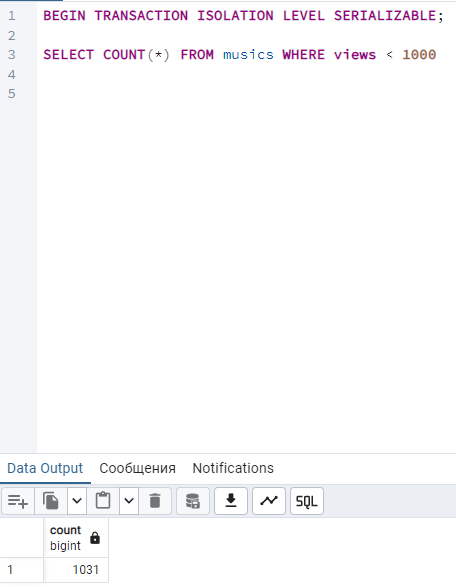
Транзакція 1



Транзакція 2



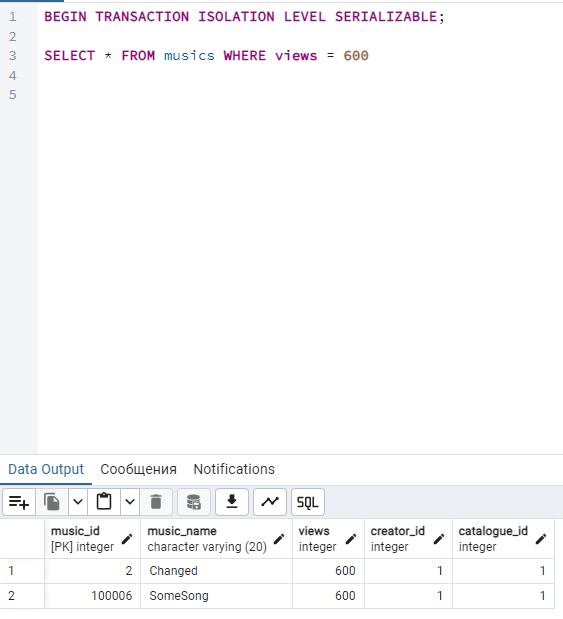
Транзакція 1



Як бачимо, ми не перетинались з транзакцією 2, і наші дані залишились не змінними.

non-repeatable read:

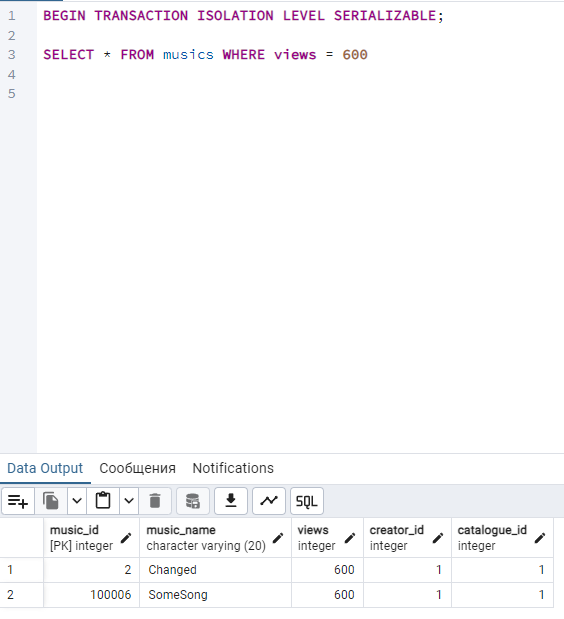
Транзакція 1



Транзакція 2



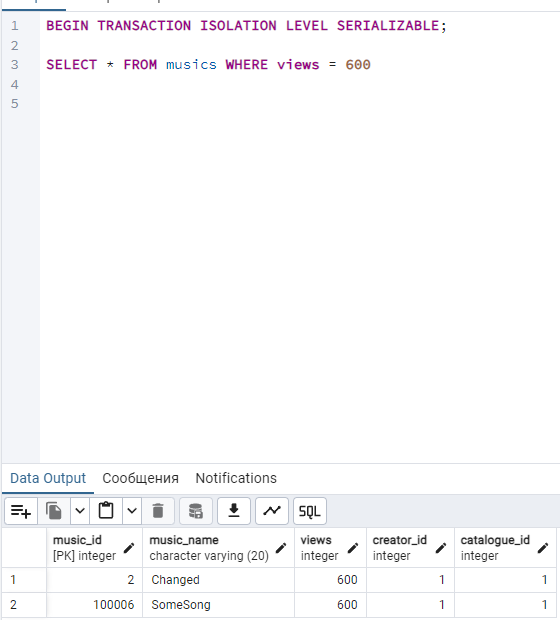
Транзакція 1



Як бачимо, змін не відбулось, а отже в нашій транзакції під час роботи нічого не змінилось. А отже, все працює вірно.

Phantom read:

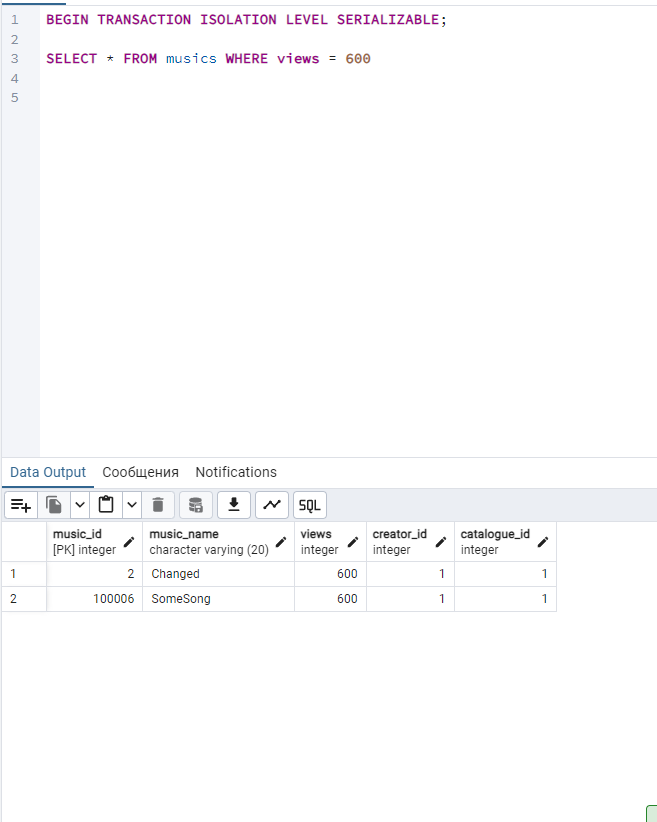
Транзакція 1



Транзакція 2



Транзакція 1



Як бачимо, видалення сюди не дійшло, і це єдиний рівень ізоляції, в якому уникається phantom read. А отже, все вийшло правильно!

ПРИМІТКА: Транзакції відбуваються послідовно, а саме так, як завантажені