**Національний технічний університет України**

**“Київський політехнічний інститут імені Ігоря Сікорського”**

**Факультет прикладної математики**

**Кафедра системного програмування і спеціалізованих комп’ютерних систем**

**ЛАБОРАТОРНА РОБОТА №3**

***з дисципліни “Бази даних та засоби управління”***

**ТЕМА: “Засоби оптимізації роботи СУБД PostgreSQL”**

**Група: КВ-03**

**Виконав: Колесник В.В**

**Київ – 2023**

**Посилання на репозиторій Github з вихідним кодом програми:**

<https://github.com/k4rnageee/databases_lab3>

**Варіант 11**

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

Завдання роботи полягає у наступному:

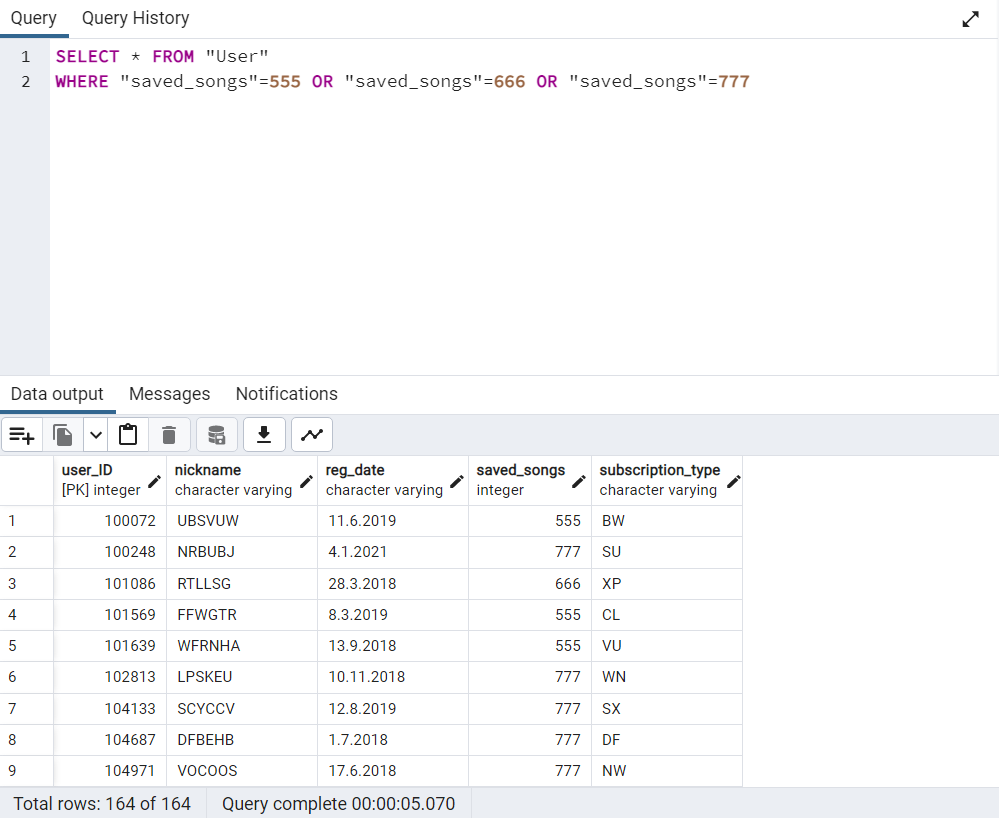
1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL.
3. Розробити тригер бази даних PostgreSQL.
4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

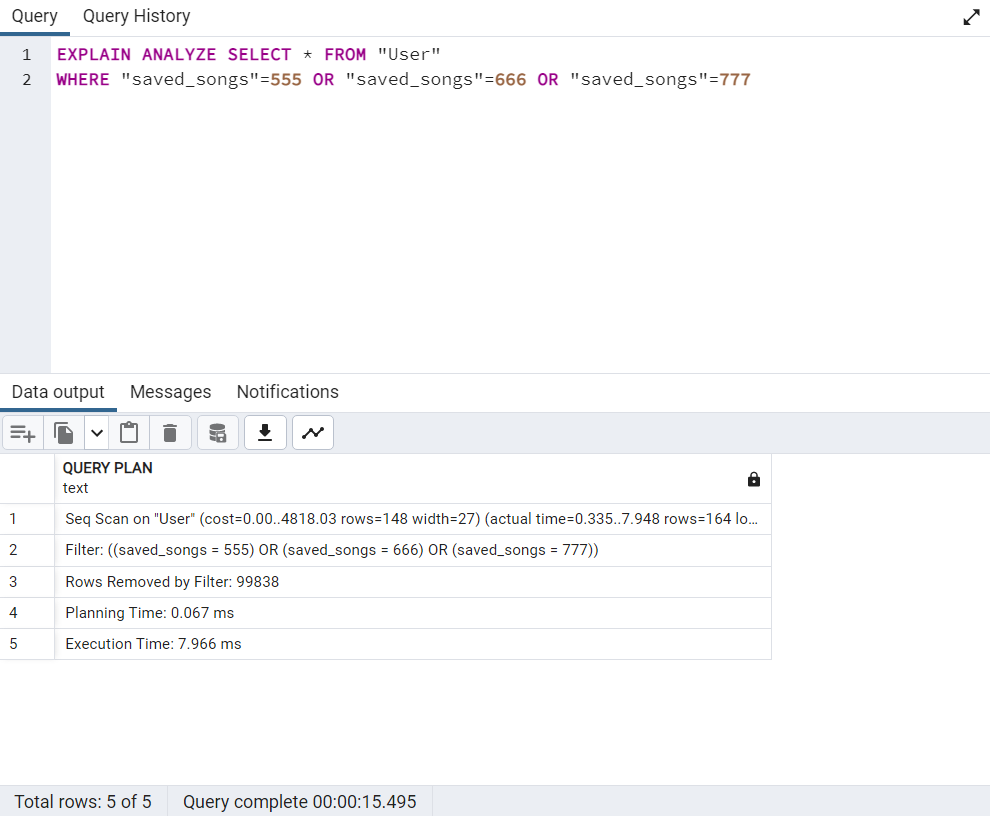
**Завдання 2**

**Індекс Hash**

Для початку перед створенням індексу hash, зробимо запити без нього та заміряємо їх швидкодію (не забуваємо, що ***hash працює лише з оператором =***):

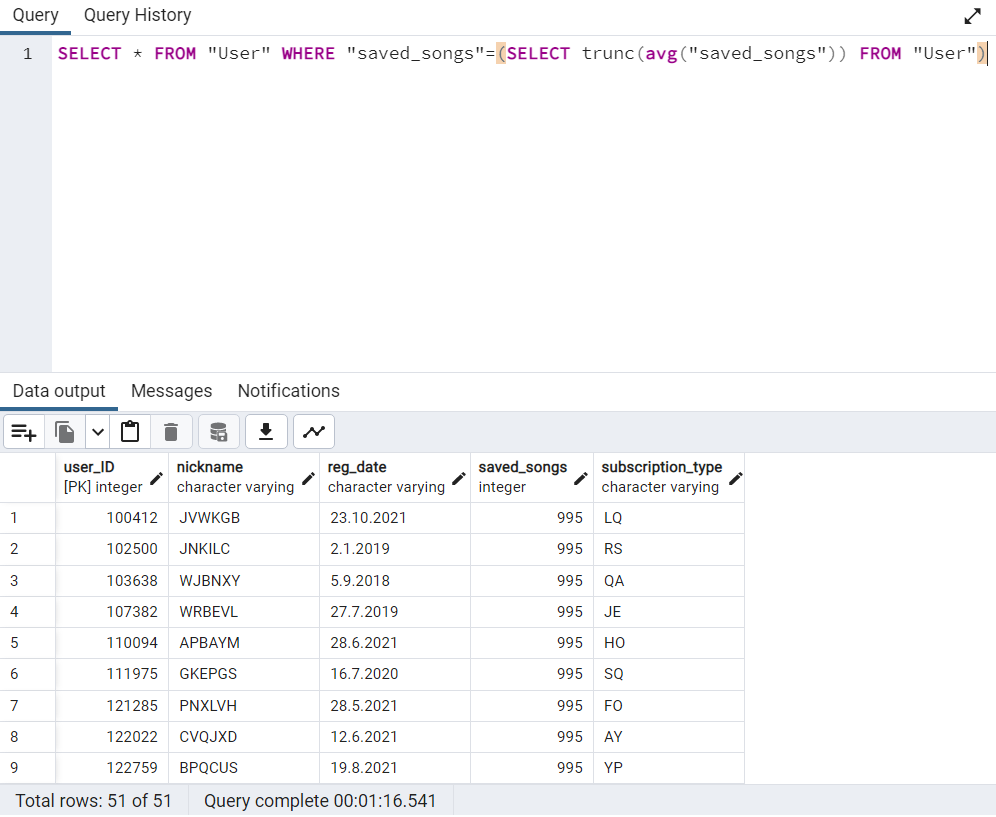
***Перший запит***

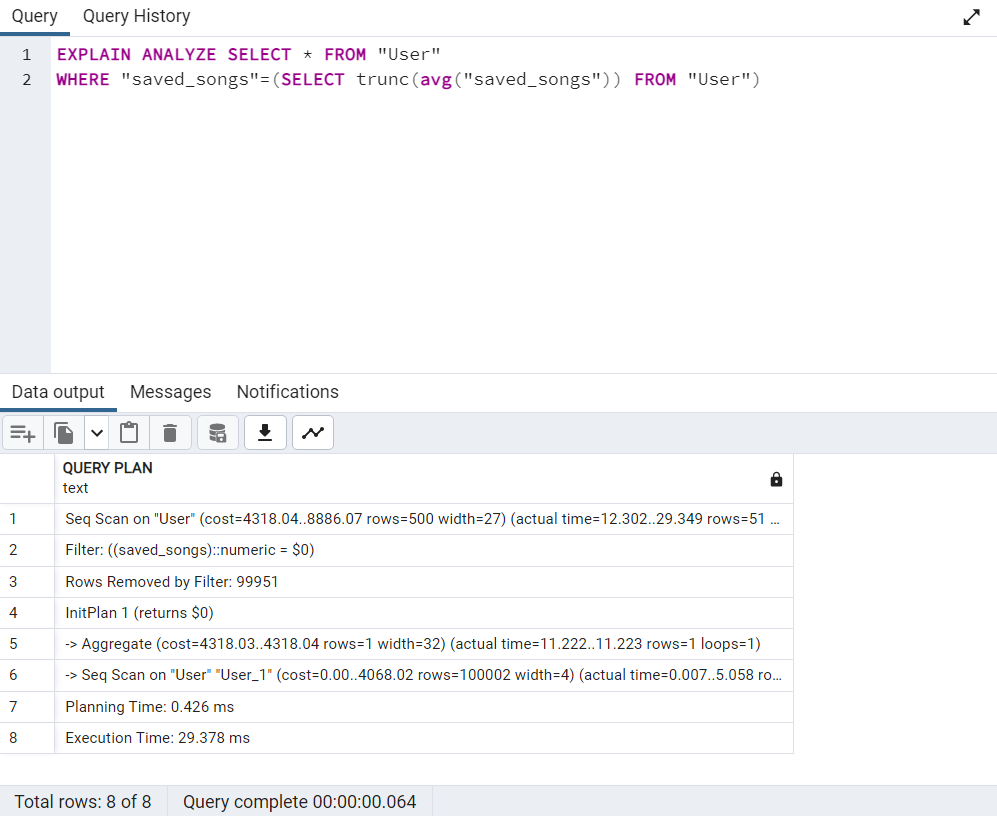




Результат – Execution Time: **7.966 ms**; seq scan.

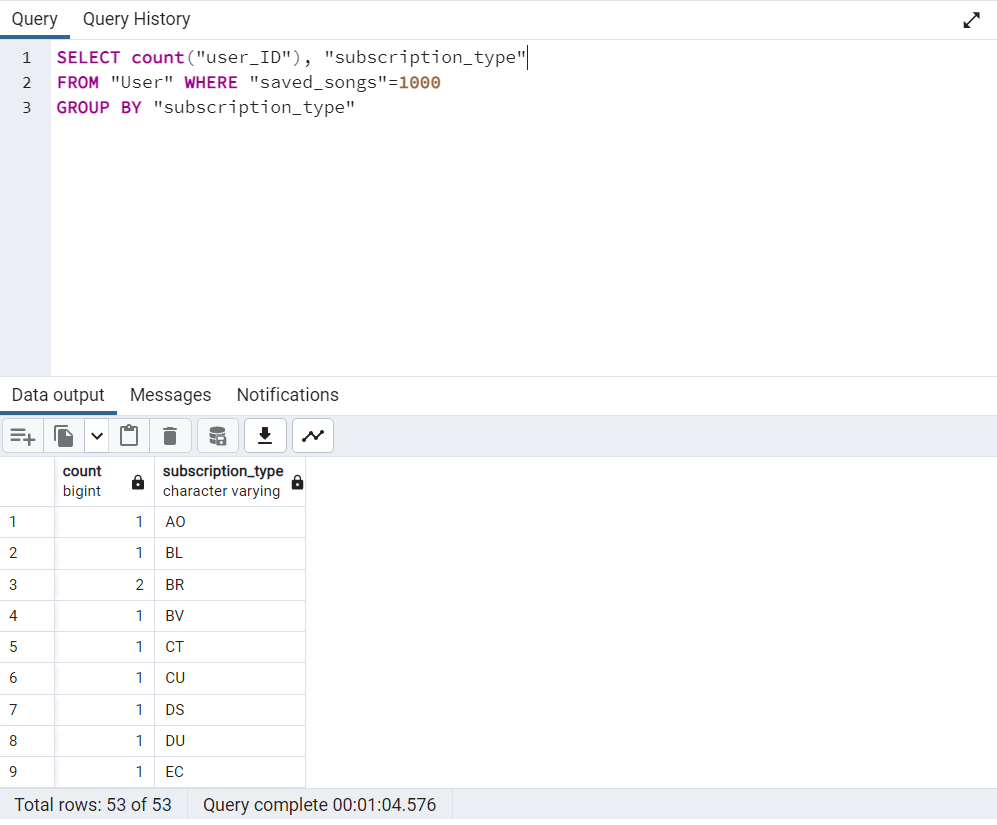
***Другий запит***

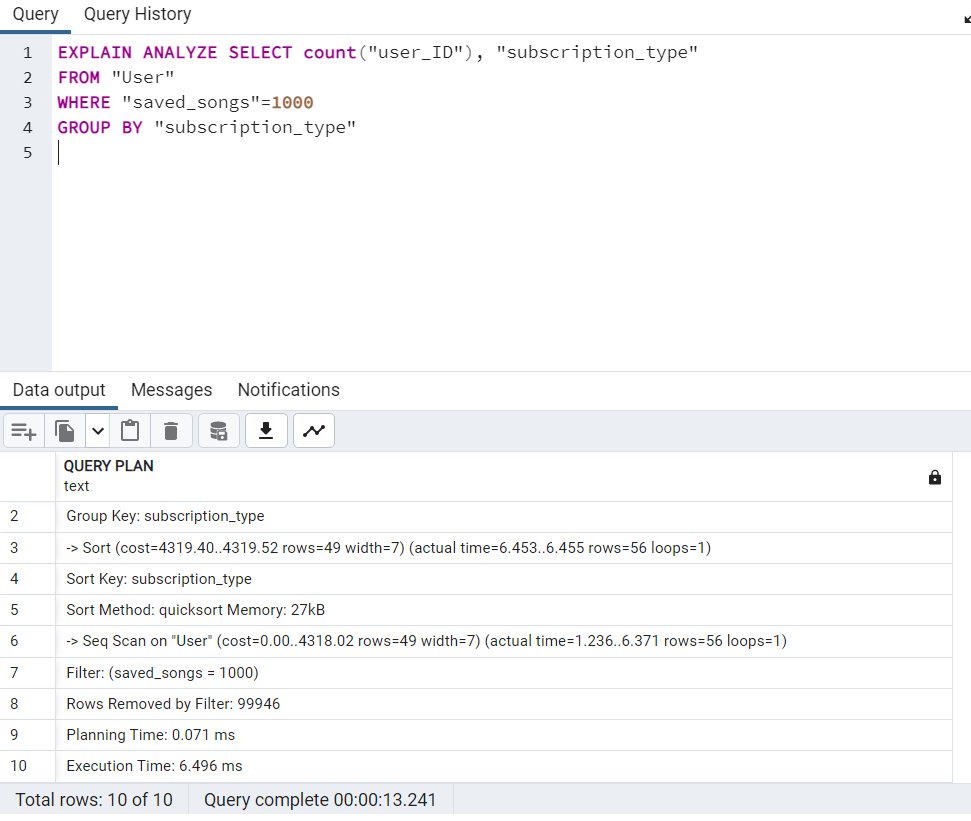




Результат – Execution Time: **29.378 ms**; seq scan.

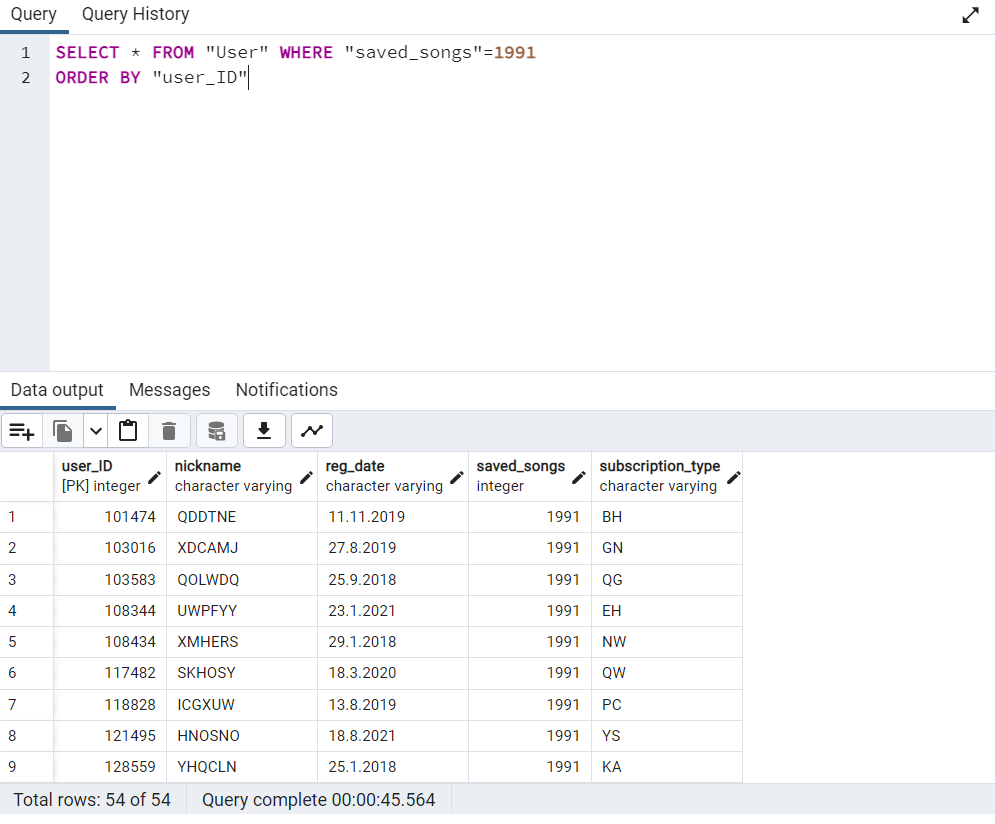
***Третій запит***

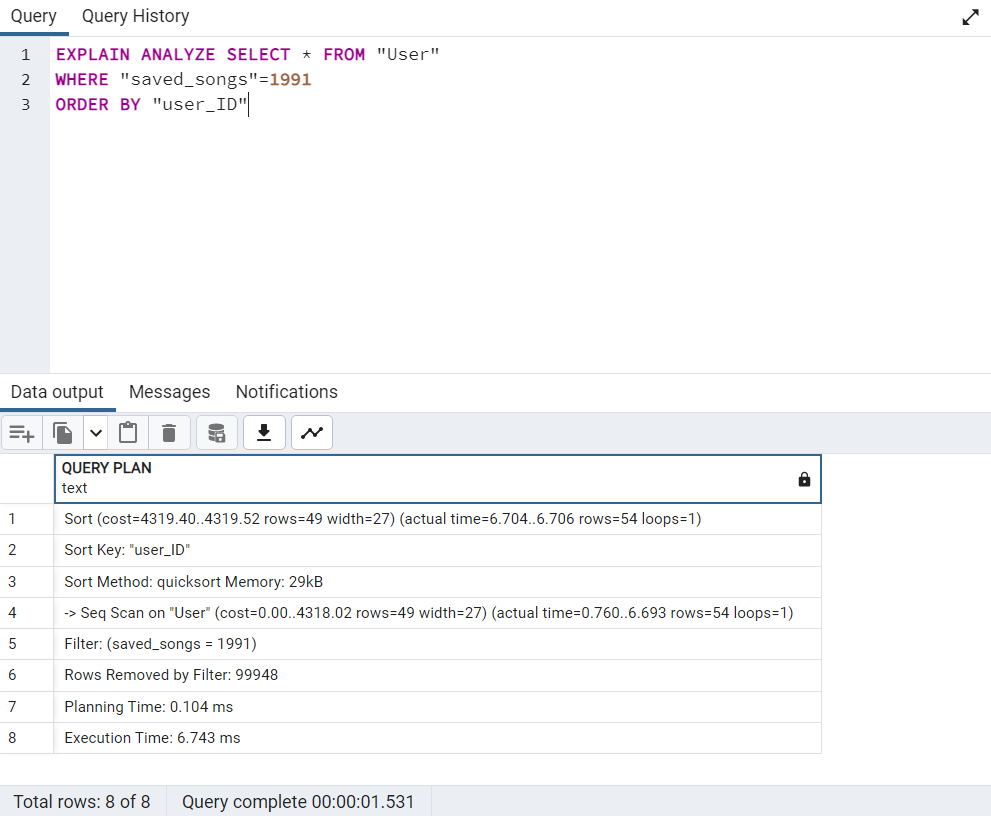




Результат – Execution Time: **6.496 ms**; seq scan.

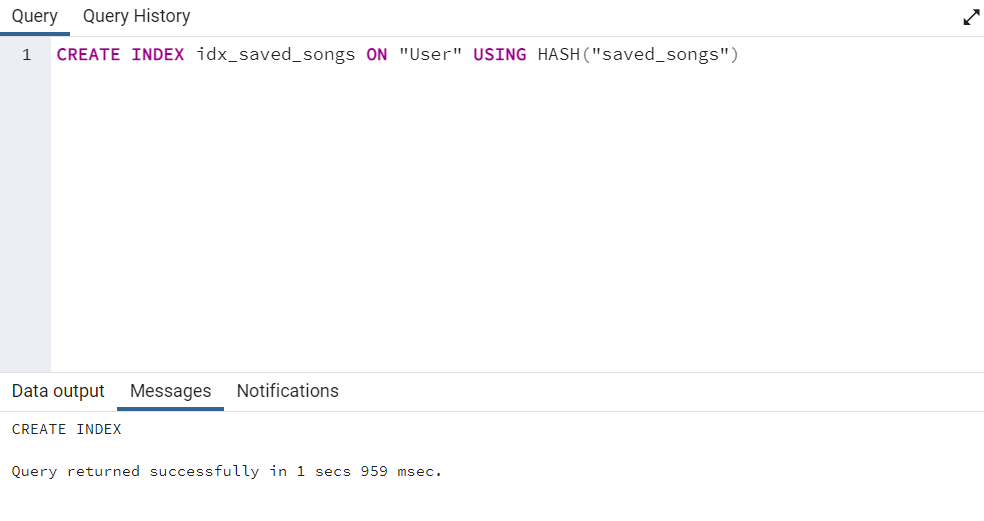
***Четвертий запит***





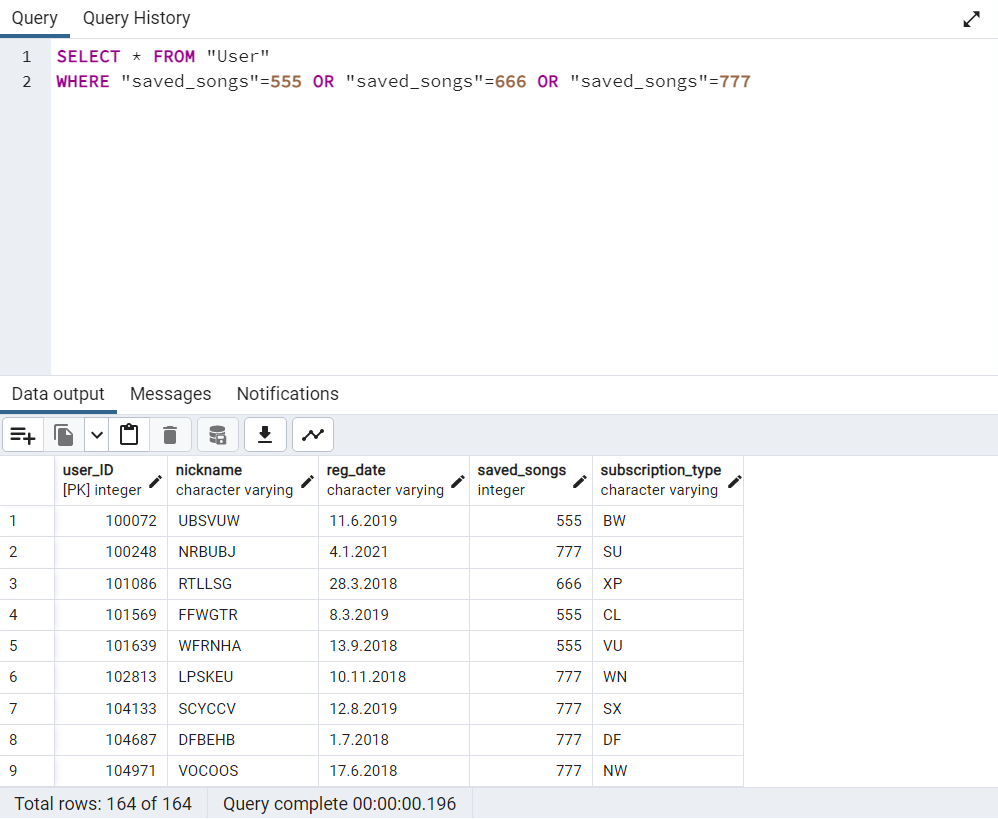
Результат – Execution Time: **6.743 ms**; seq scan.

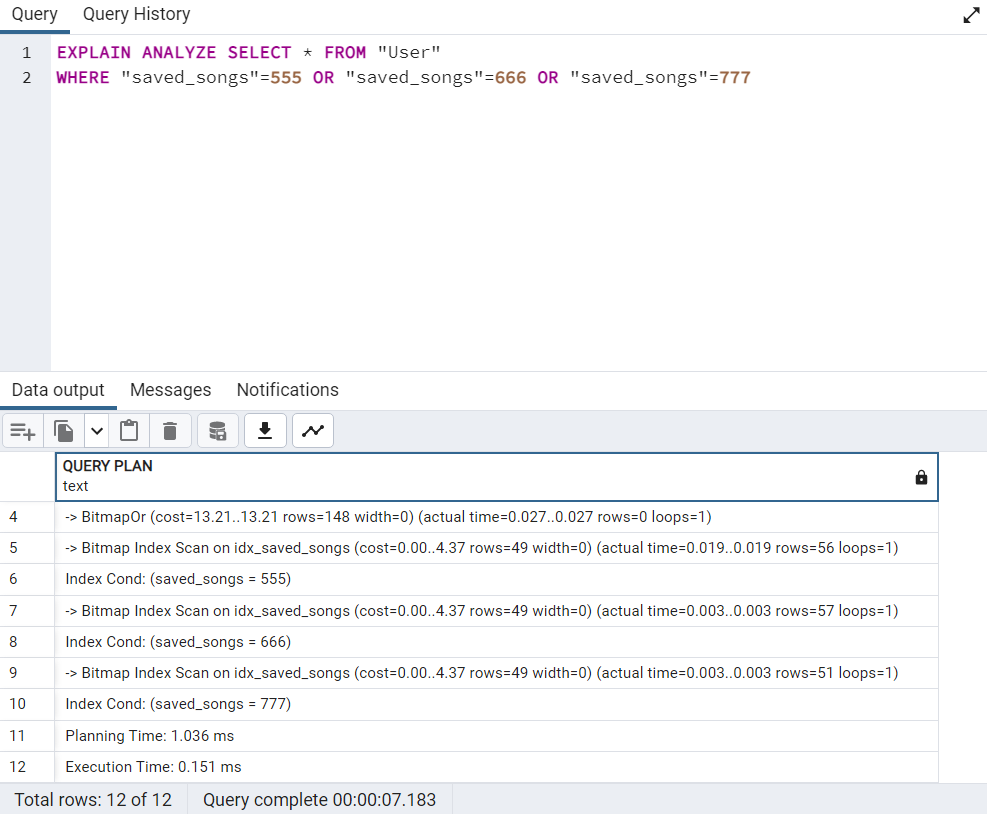
Тепер створимо hash-індекс для стовпця saved\_songs:



Після цього повторимо відповідні запити, подивимось результати та порівняємо з результатами без індексу.

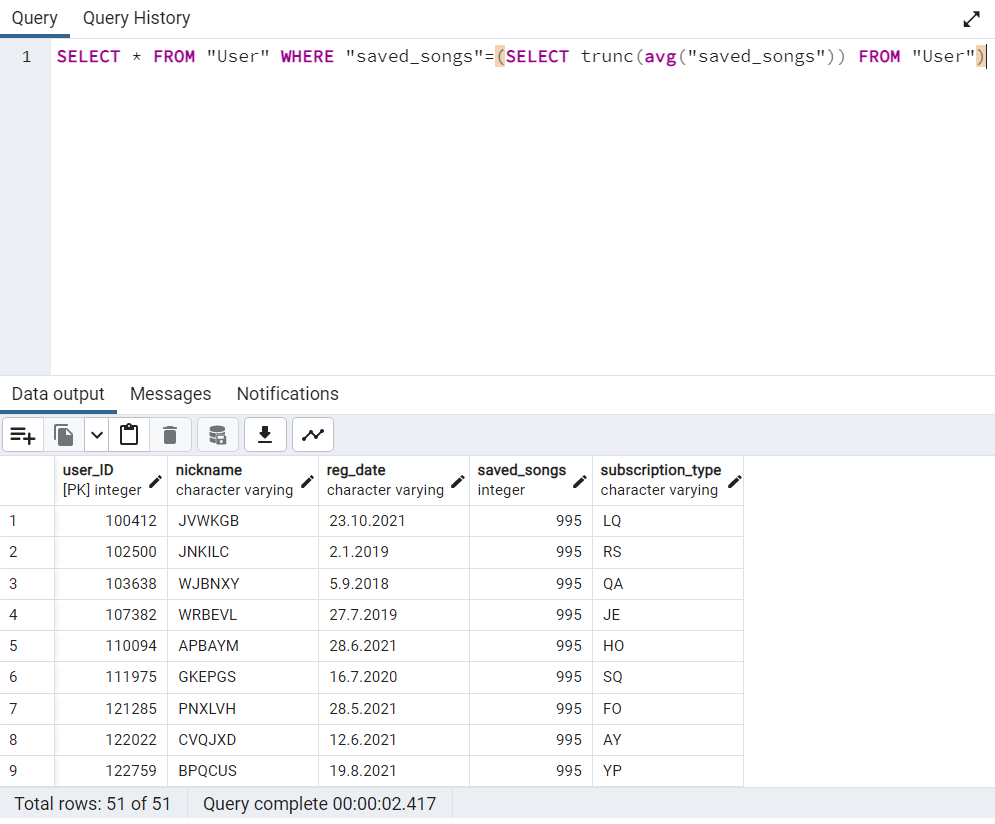
***Перший запит***

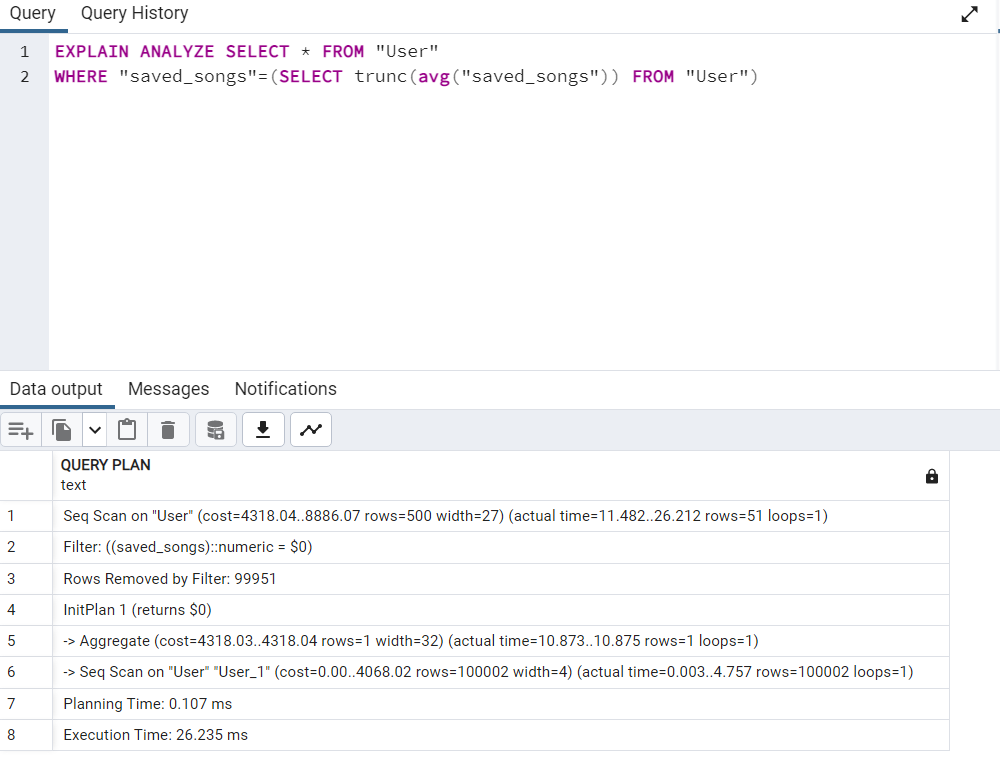




Результат – Execution Time: **0.151 ms**; index scan.

***Другий запит***

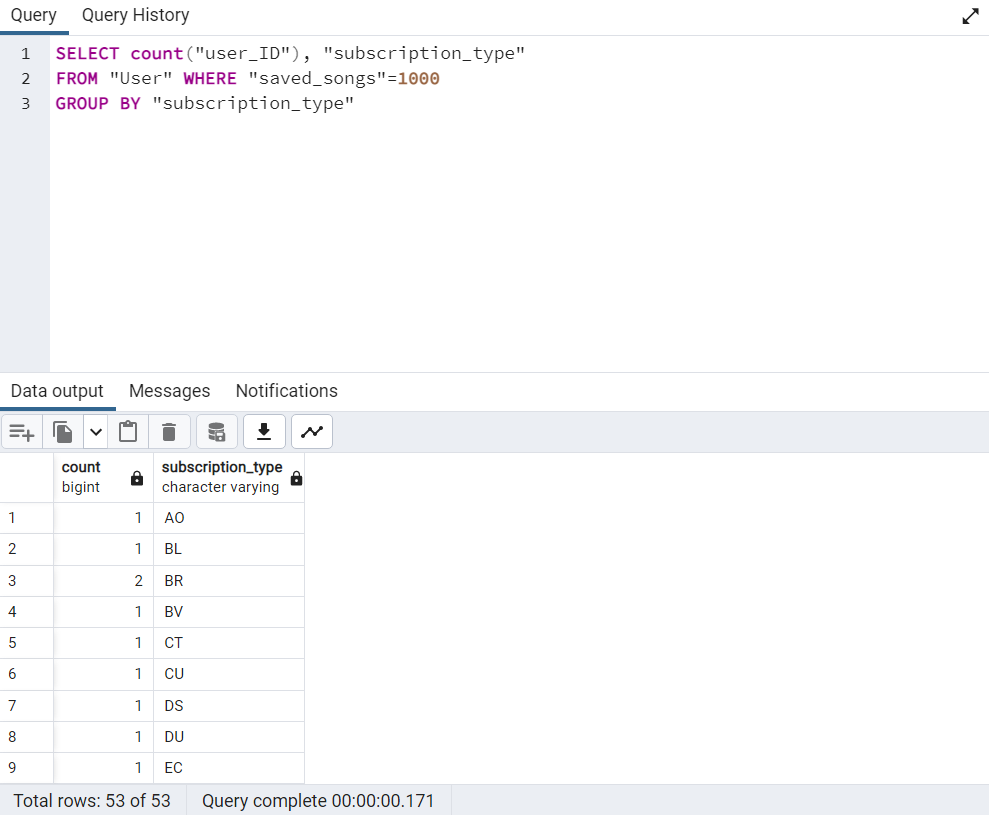


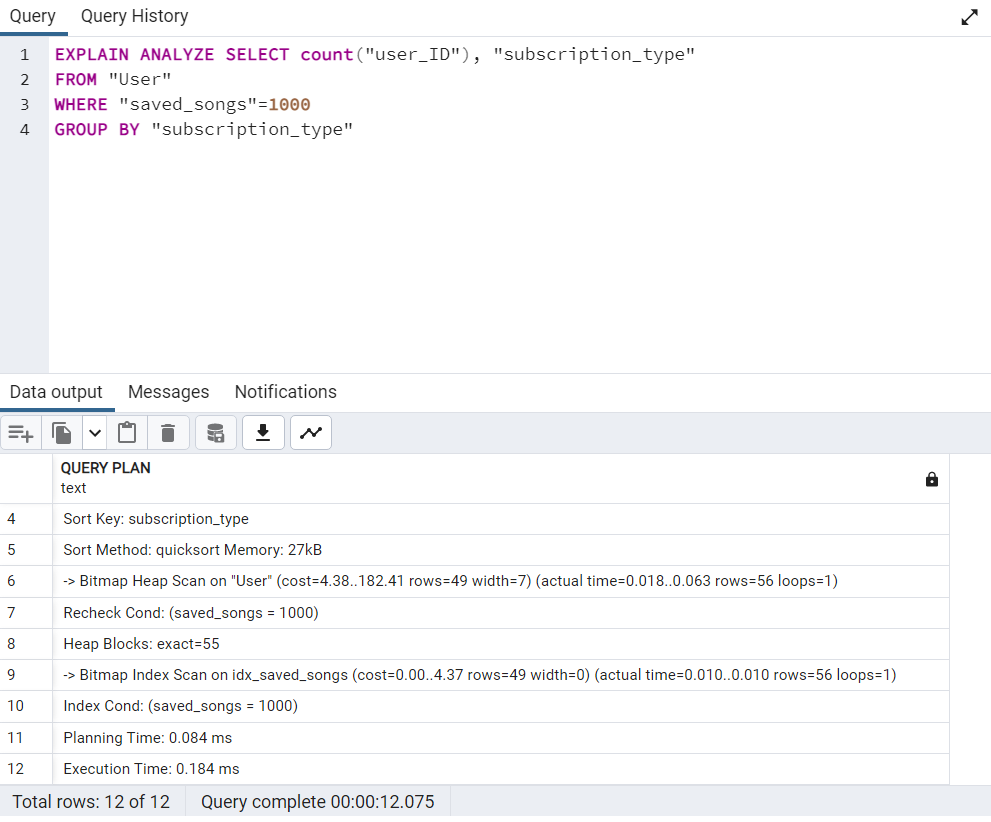


Результат – Execution Time: **26.235 ms**; seq scan.

Тут маємо цікавий результат. Як бачимо, даний запит не використовує індекс, а використовує seq scan, як і при запиті, коли індексу ще не було створено. Можливо, це пояснюється тим, що в даному запиті у черзі спочатку стоїть запит, де відбувається звичайний seq scan. Отже робимо висновок, що всі умови мають містити оператор = для коректної роботи даного індексу.

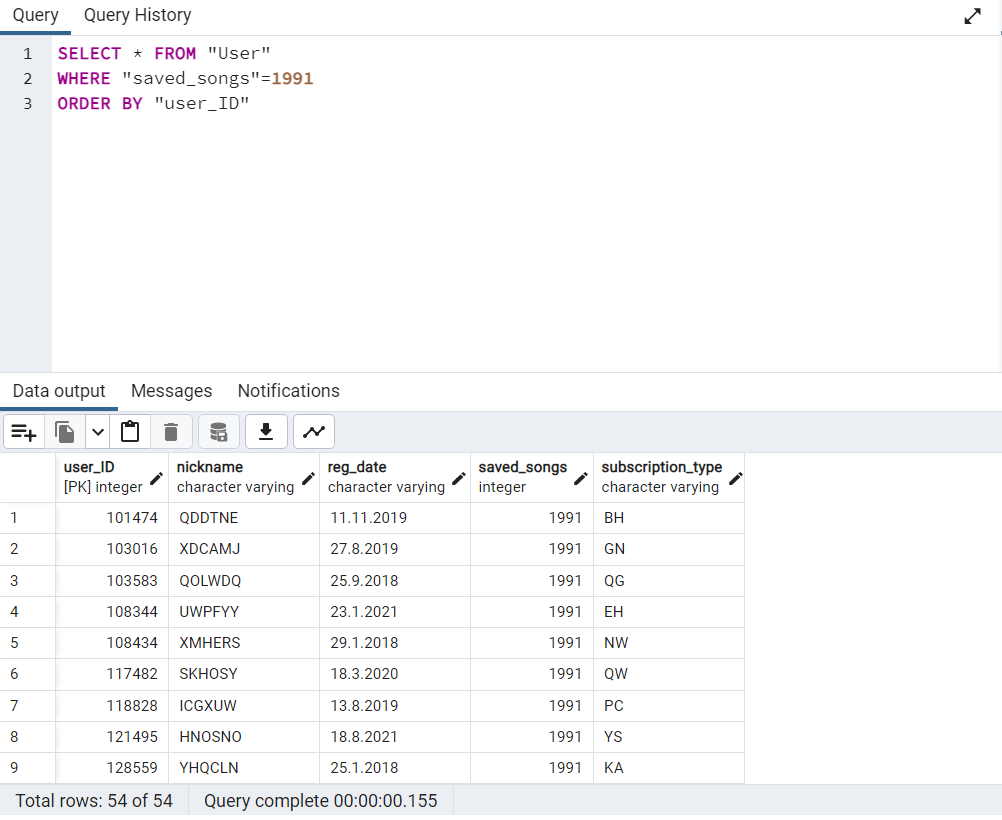
***Третій запит***

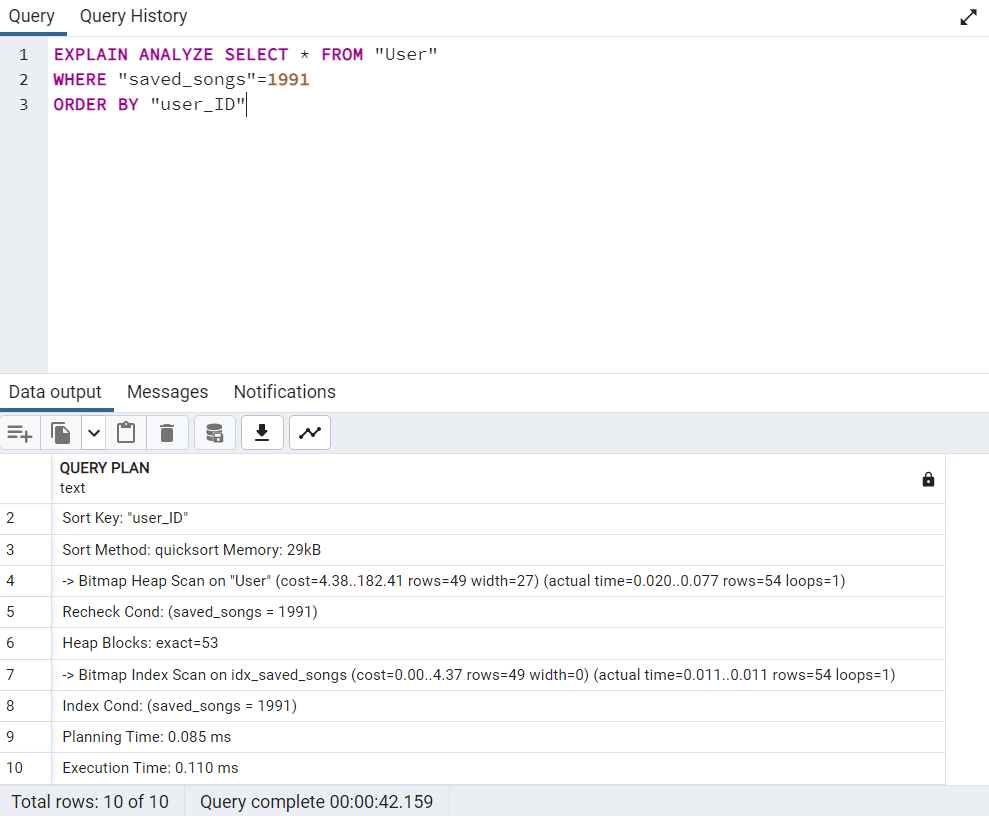




Результат – Execution Time: **0.184 ms**; index scan.

***Четвертий запит***





Результат – Execution Time: **0.110 ms**; index scan.

**Підсумок по hash-індексу**

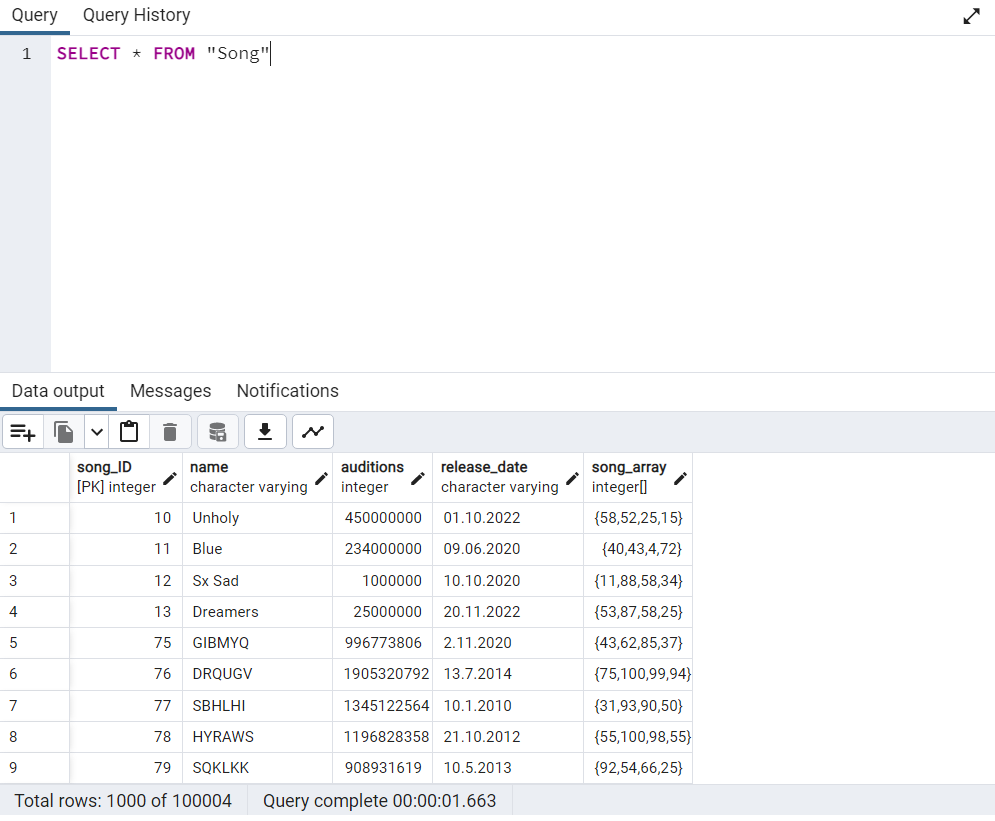
Розглянувши теорію та виконавши дані тестові запити, можна зробити певні висновки по індексу hash:

1. індекс hash, як і будь-який інший індекс, доцільно використовувати, коли нам потрібна вибірка з невеликої кількості даних, адже якщо ми завжди при запиті будемо повертати майже всю таблицю, то втрачається сенс індексів. При цьому сама база даних має містити великий обсяг даних, недоцільно використовувати індекси при малій кількості записів.
2. окрім цього, індекс hash підтримує лише операцію =, на відміну від інших індексів, тому він має вузький спектр використання, але звичайно, деколи він буде доцільний, якщо майже всі наші запити будуються на основі оператора =, адже швидкість даного індексу – O(1), тобто константа;
3. так як цей індекс підтримує лише операцію =, то варто контролювати, щоб кожен підзапит також містив оператор =;
4. як бачимо зі скрінів роботи запитів, після створення індексу hash, 3 з 4 запитів використали саме його, і одразу можна побачити, що в усіх набагато зменшились cost та execution time. Отже даний індекс набагато пришвидшує роботу запиту, але знову ж таки, має вузький спектр застосування.

**Індекс GIN**

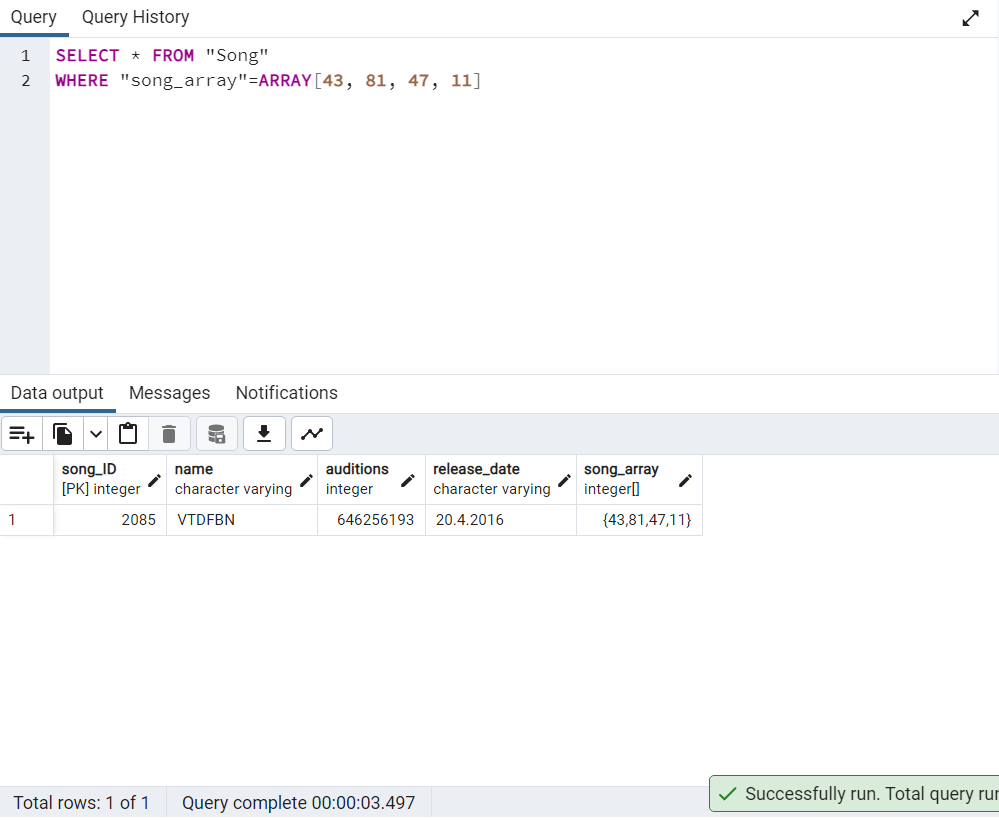
Індекс GIN найкраще працює з колекціями, тобто з форматами ключ-значення, JSON, масиви тощо. Тому створимо в одній з таблиць додатковий стовпець з масивом, і заповнимо його випадковими даними:

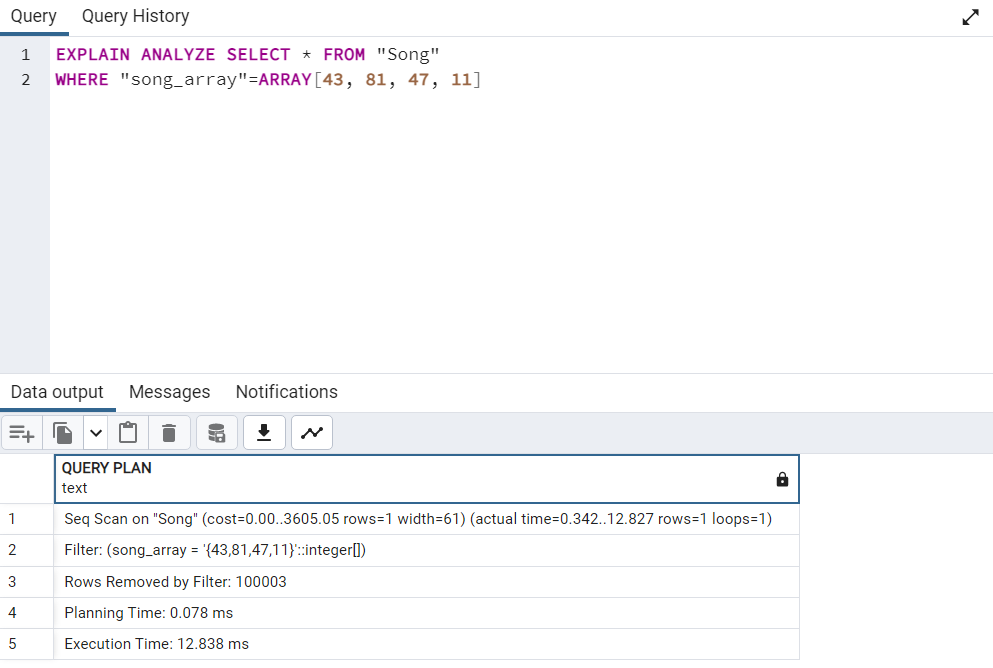




Для початку перед створенням індексу GIN, зробимо запити без нього та заміряємо їх швидкодію:

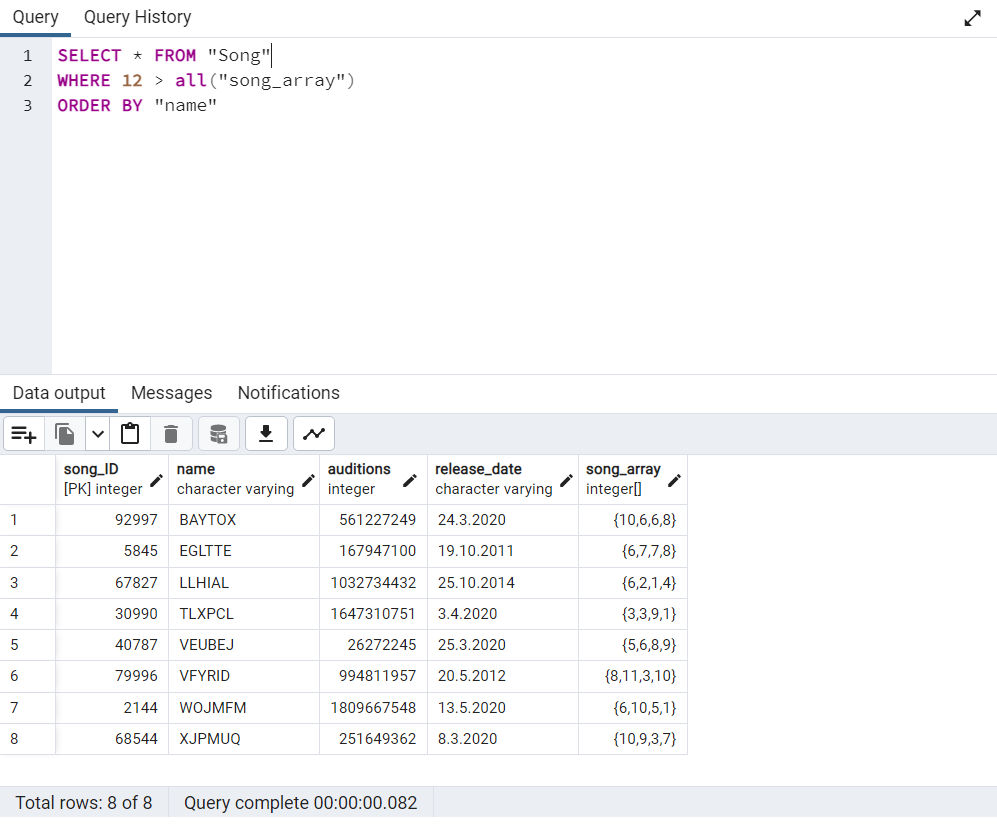
***Перший запит***

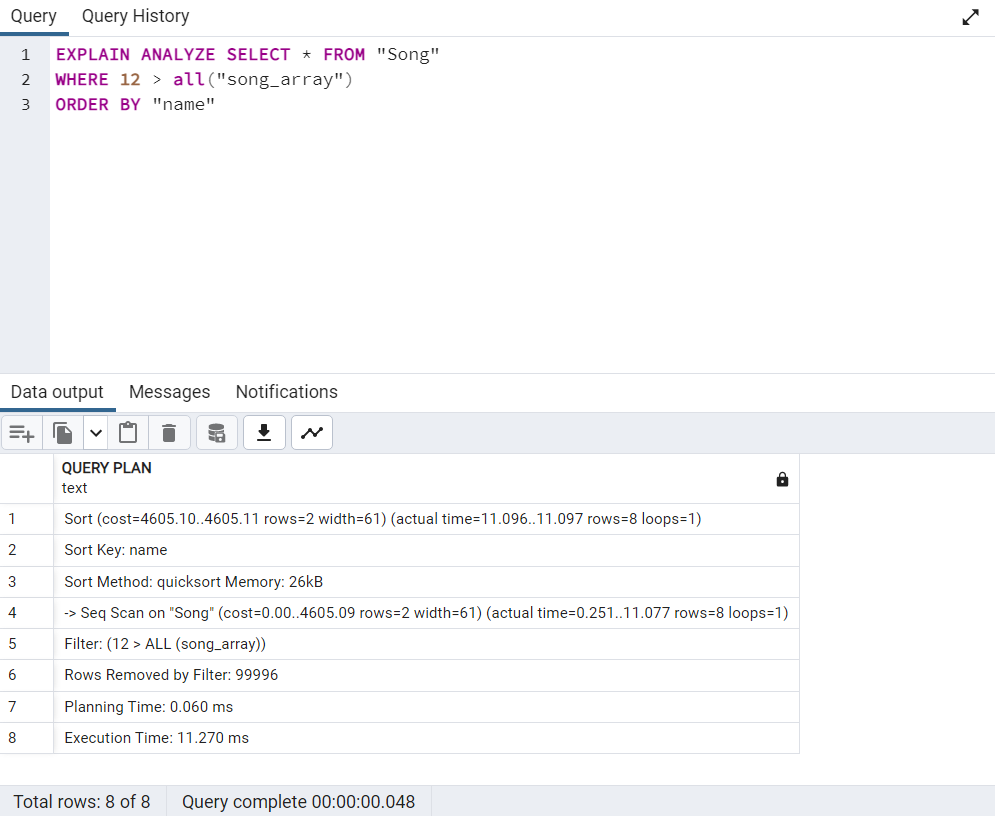




Результат – Execution Time: **12.838 ms**; seq scan.

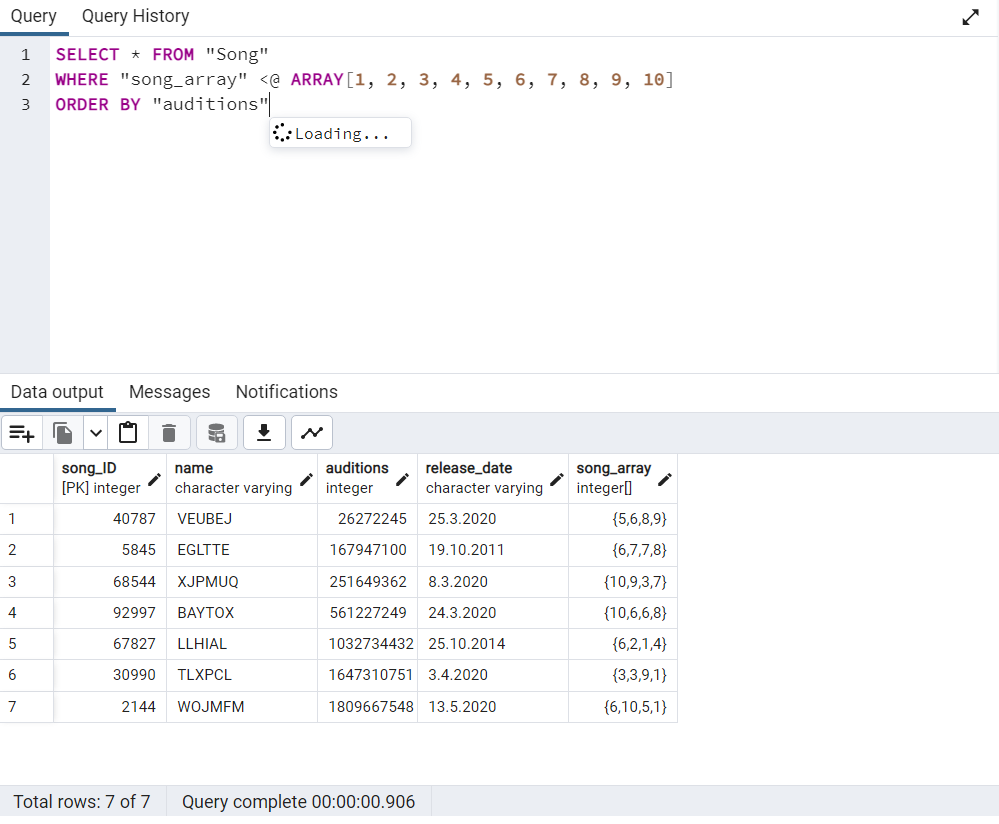
***Другий запит***

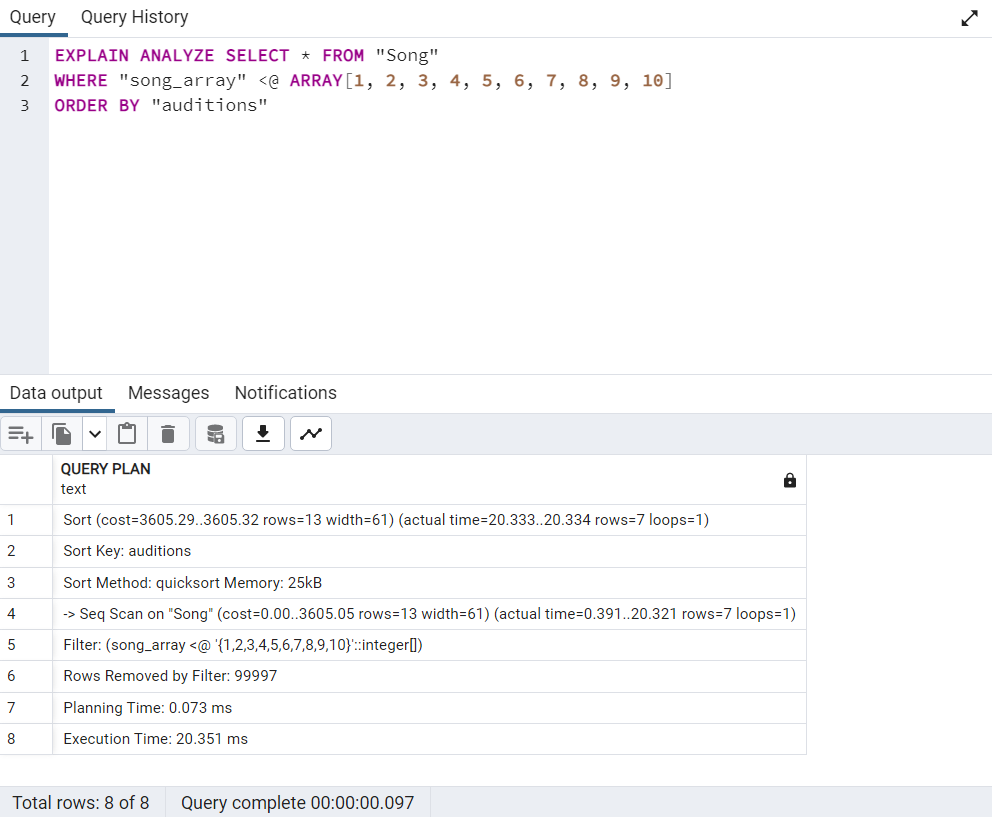




Результат – Execution Time: **11.270** **ms**; seq scan.

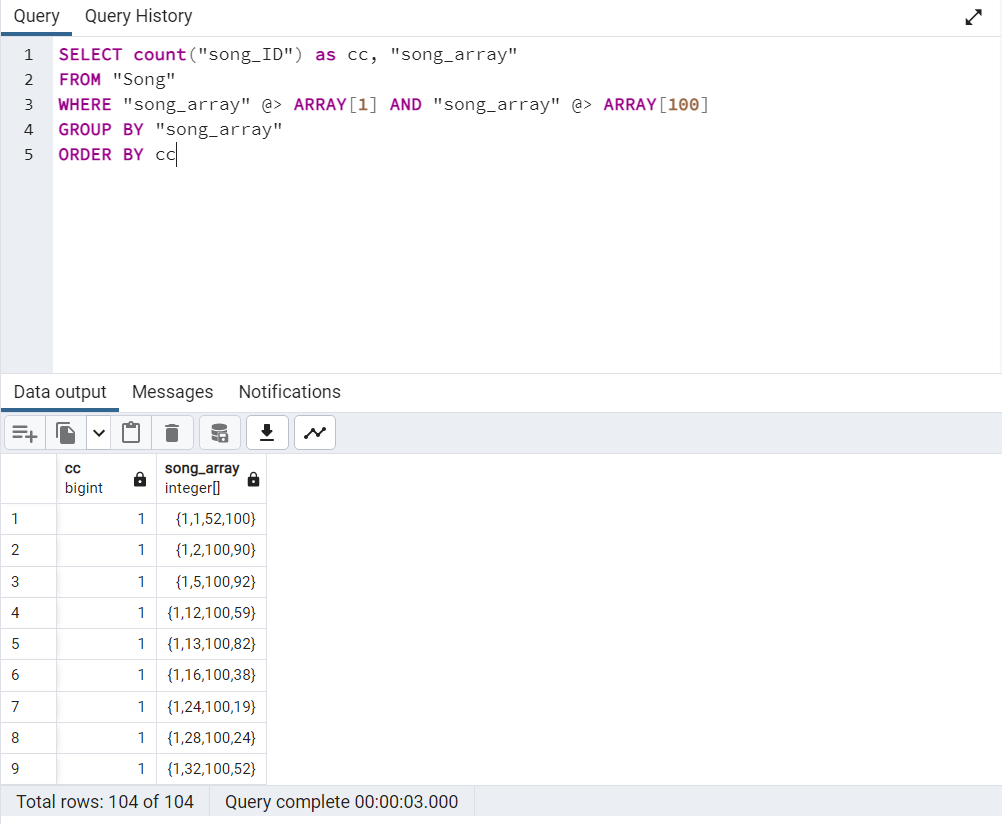
***Третій запит***

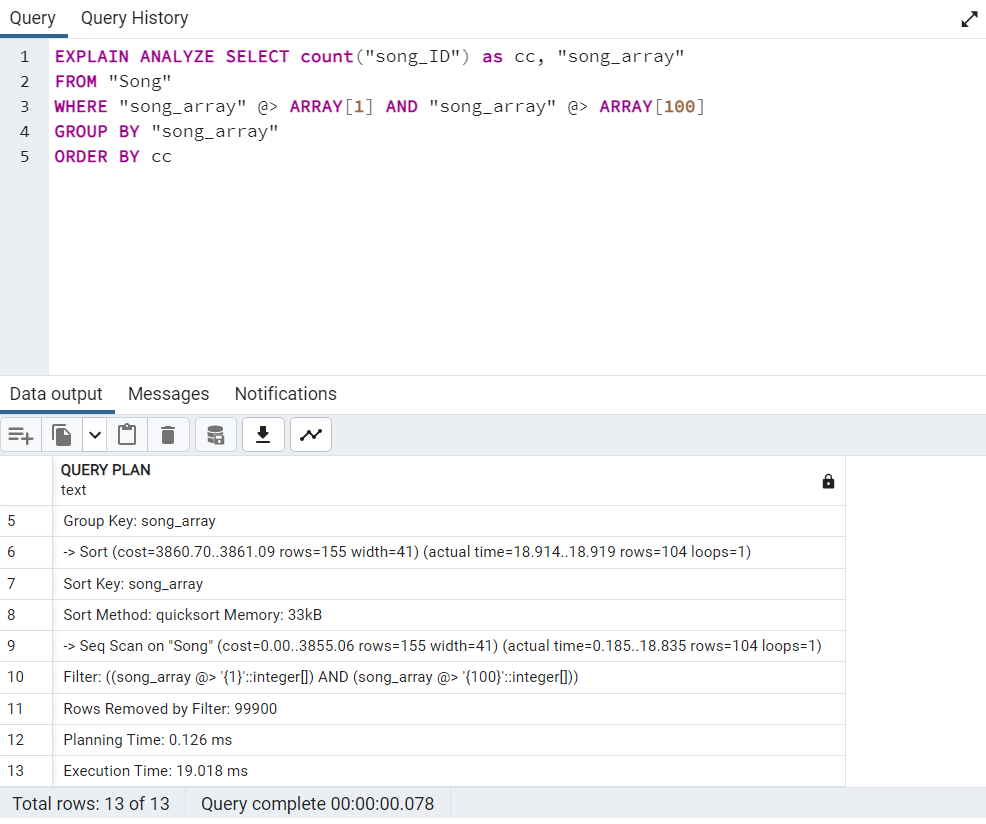




Результат – Execution Time: **20.351 ms**; seq scan.

***Четвертий запит***





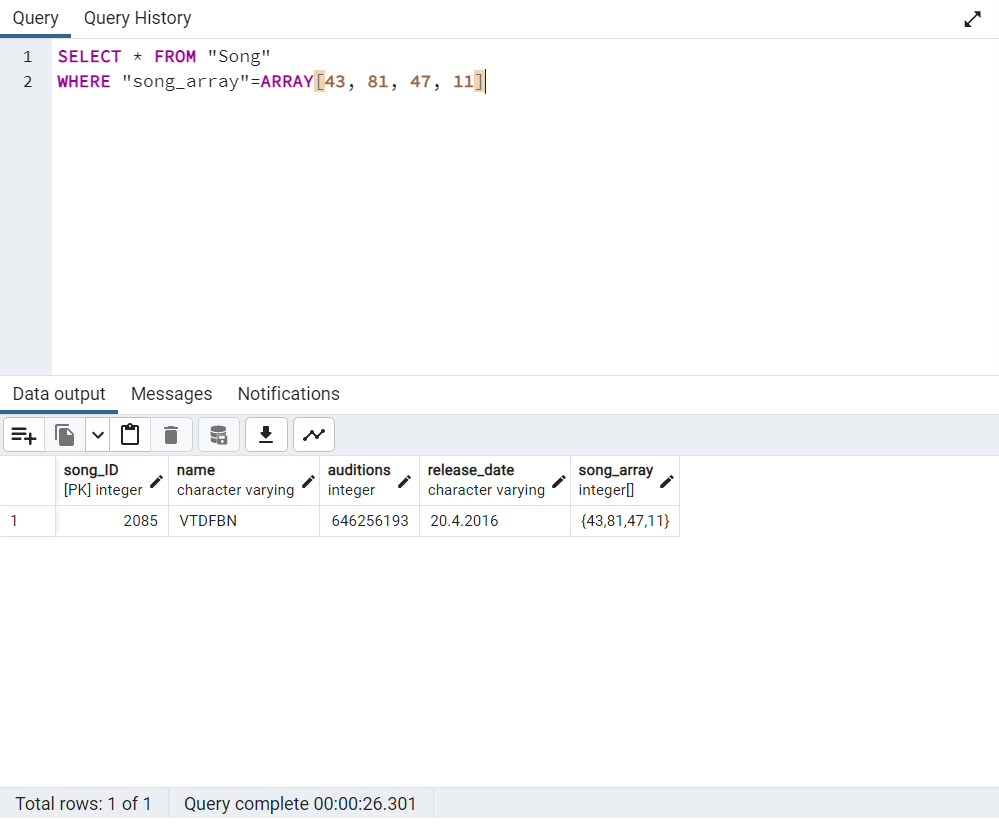
Результат – Execution Time: **19.018 ms**; seq scan.

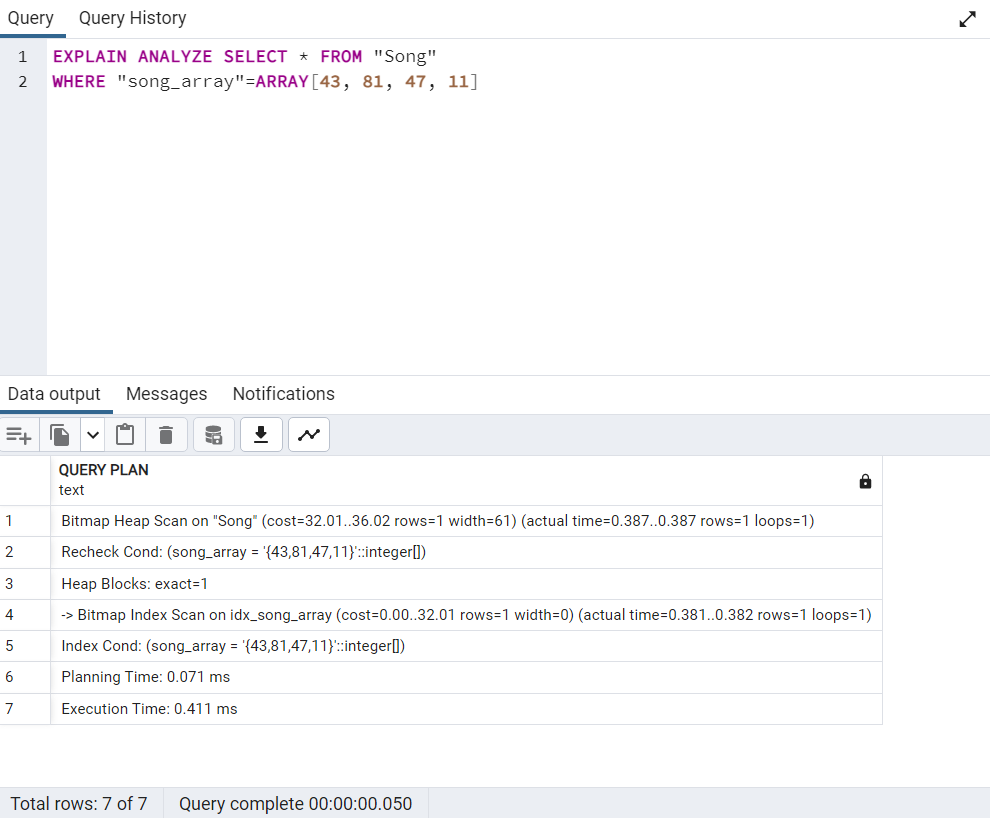
Тепер створимо gin-індекс для стовпця song\_array:



Після цього повторимо відповідні запити, подивимось результати та порівняємо з результатами без індексу.

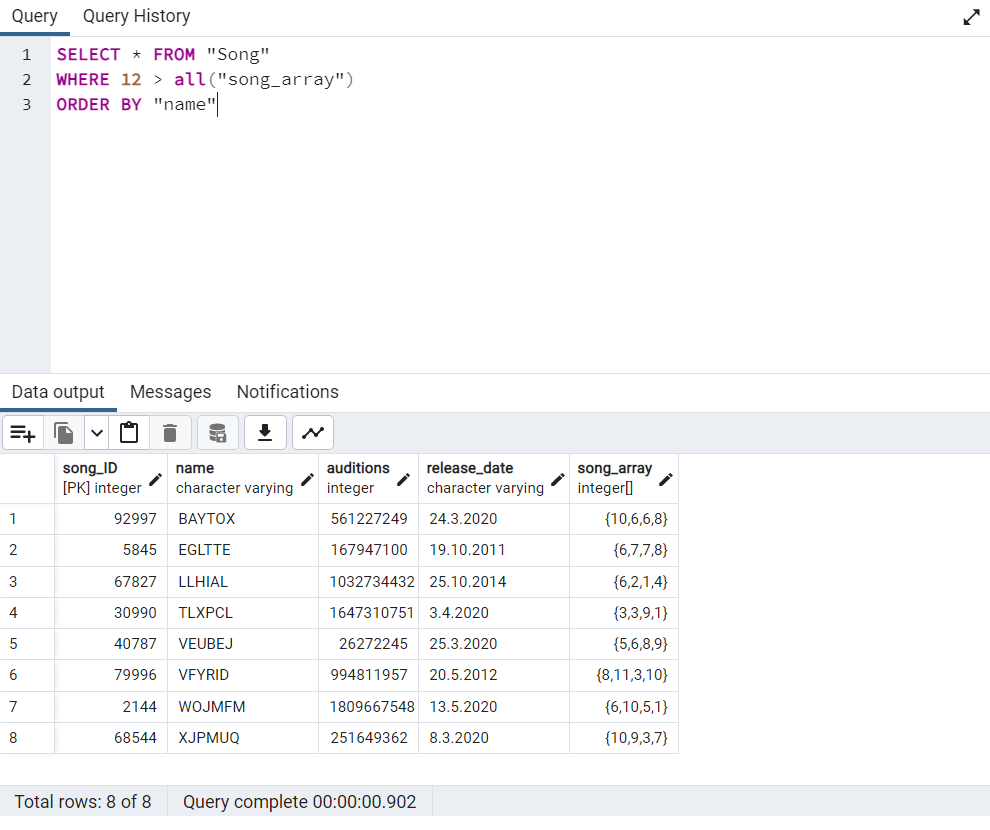
***Перший запит***

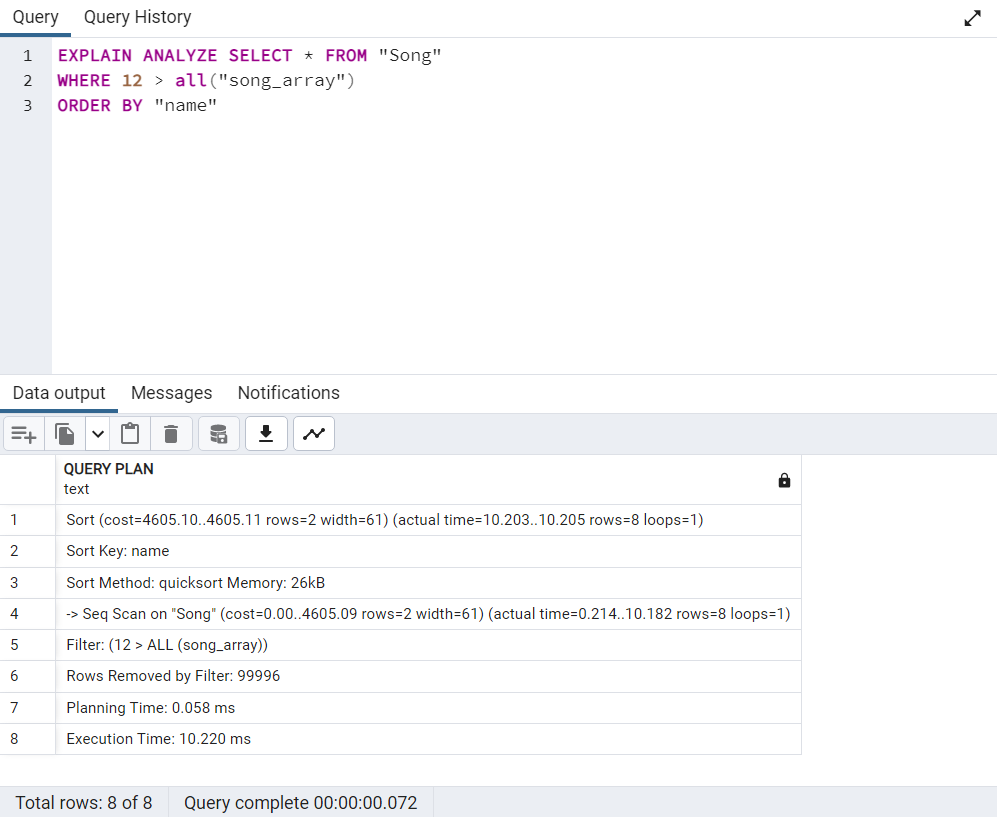




Результат – Execution Time: **0.411 ms**; index scan.

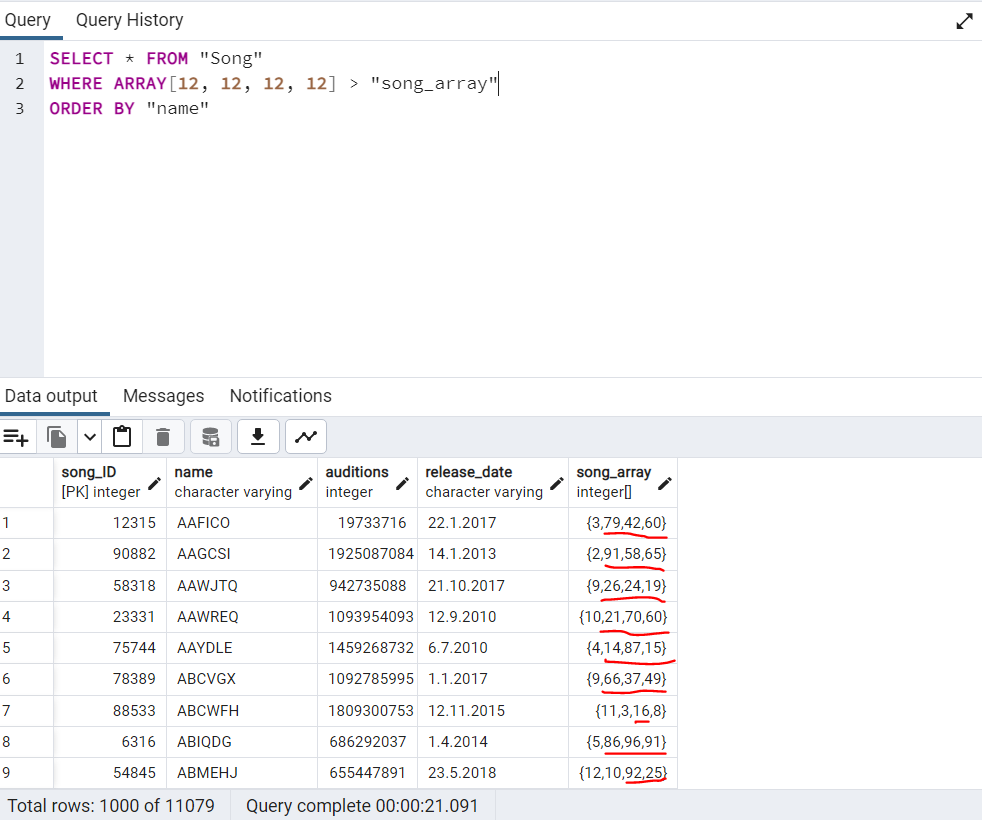
***Другий запит***



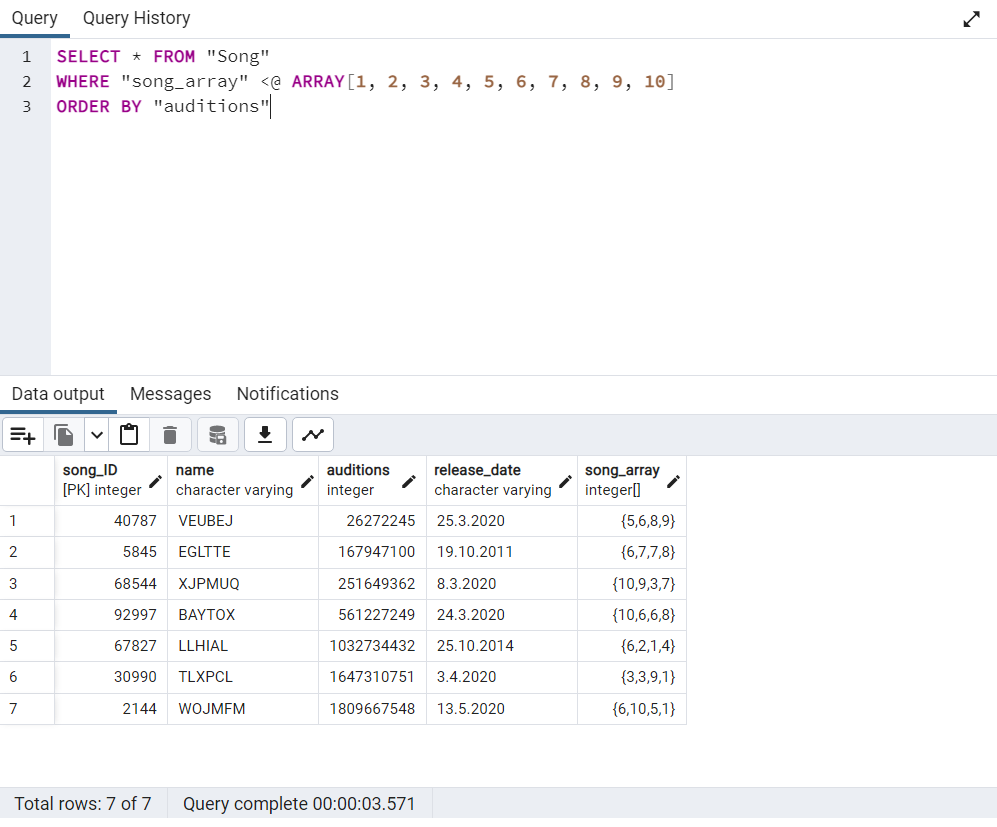


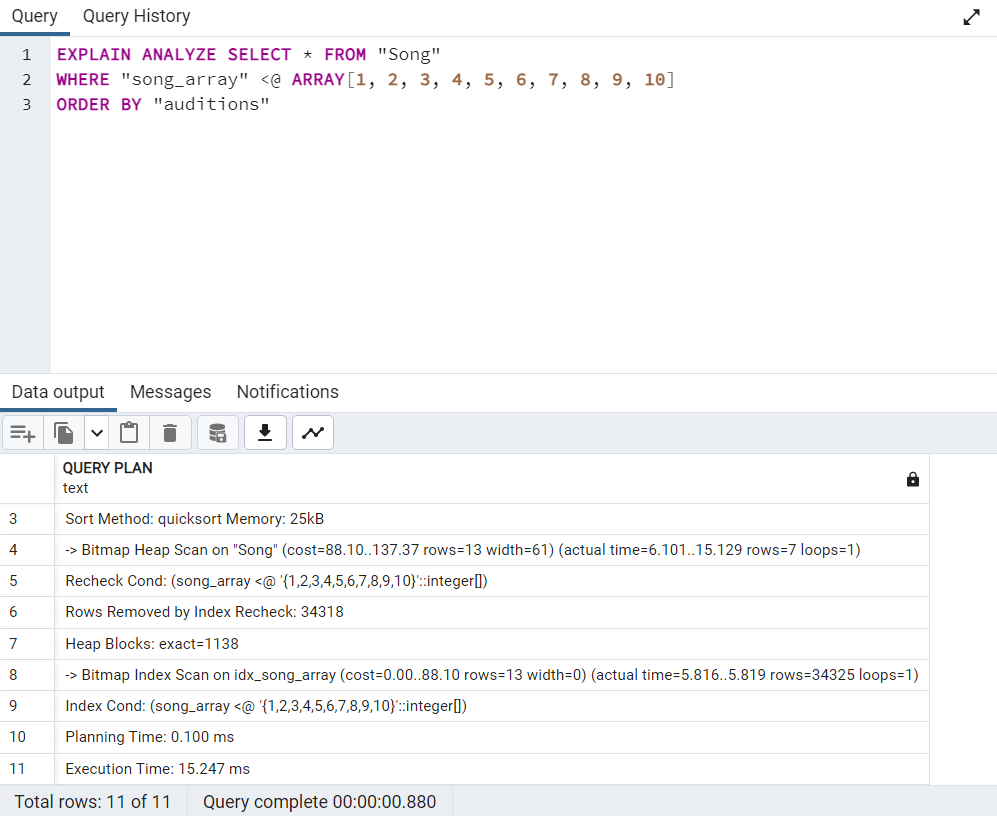
Результат – Execution Time: **10.220 ms**; seq scan.

У цьому випадку знову ж таки працює seq scan, а не індекс. Скоріше за все це пояснюється тим, що ми порівнюємо числа, а не самі масиви. Я хотів порівняти адекватно масиви, але sql чомусь порівнював лише перше число в масиві:



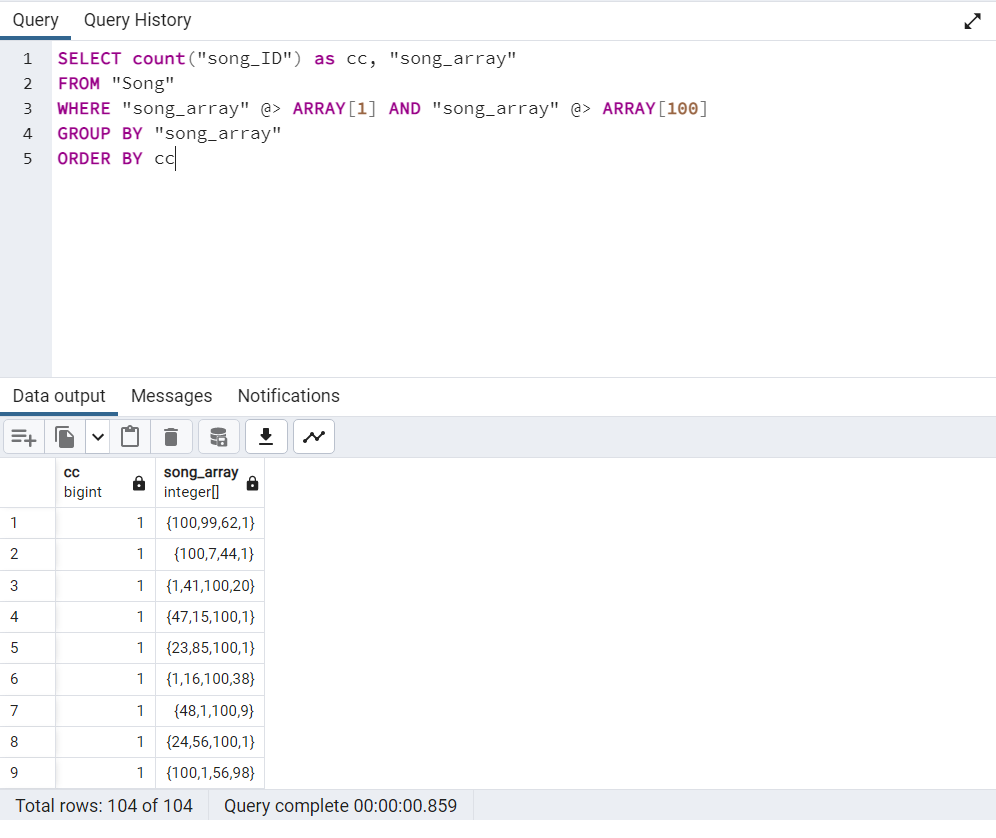
***Третій запит***





Результат – Execution Time: **15.247 ms**; index scan.

***Четвертий запит***





Результат – Execution Time: **0.398 ms**; index scan.

**Підсумок по gin-індексу**

Розглянувши теорію та виконавши дані тестові запити, можна зробити певні висновки по індексу gin:

1. індекс gin, як і будь-який інший індекс, доцільно використовувати, коли нам потрібна вибірка з невеликої кількості даних, адже якщо ми завжди при запиті будемо повертати майже всю таблицю, то втрачається сенс індексів. При цьому сама база даних має містити великий обсяг даних, недоцільно використовувати індекси при малій кількості записів;
2. індекс gin доцільно використовувати при роботі з такими форматами даних: ключ-значення, масиви, JSON, діапазони та повнотекстовий пошук;
3. дуже важливо правильно формулювати запит, щоб система використала саме потрібний індекс;
4. як бачимо зі скрінів роботи запитів, після створення індексу gin, 3 з 4 запитів використали саме його, і одразу можна побачити, що в усіх набагато зменшились cost та execution time, окрім хіба що 3 запиту, де економія часу не така значна. Отже даний індекс набагато пришвидшує роботу запиту, але лише при роботі з потрібними типами даних.

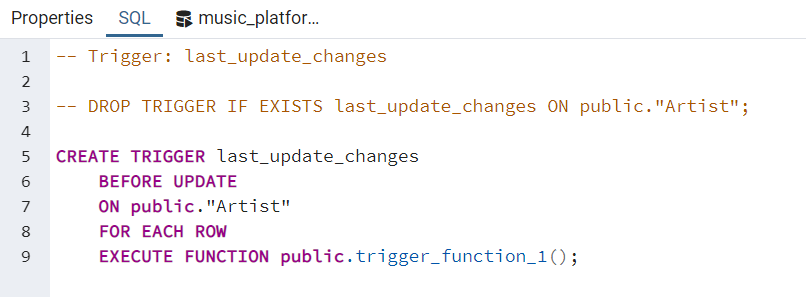
**Завдання 3**

***Код тригерної функції***

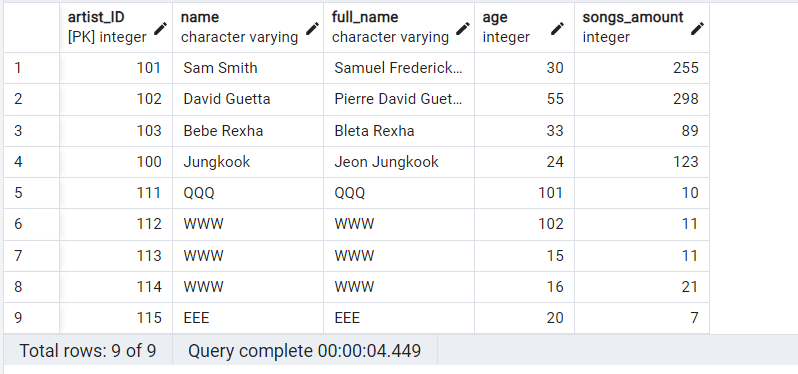


Тобто, якщо нове значення age менше за старе, то видаляємо всі рядки, де age більше 100, інакше – всі рядки, де age менше 18.

***Інформація про тригер для таблиці “Artist”***



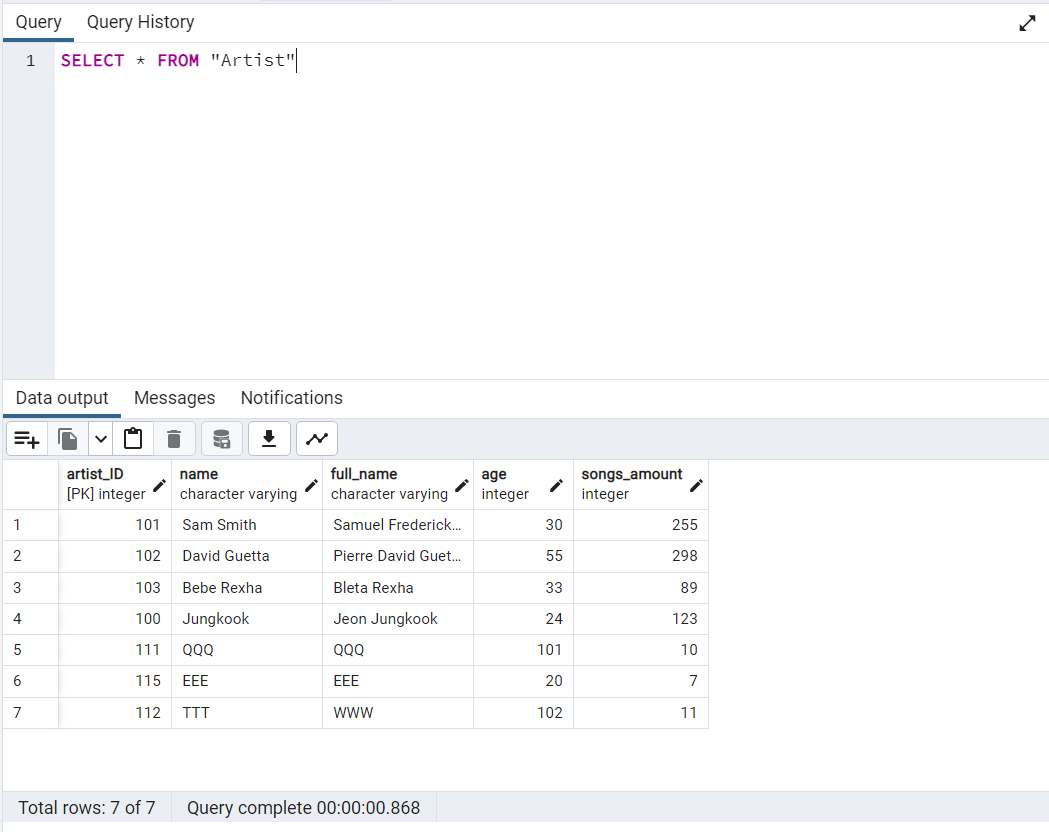
Для перевірки роботи тригерної функції заповнимо таблицю певними даними:



Тепер виконаємо певний запит UPDATE, який не змінює age. У цьому випадку тригерна функція має видалити рядки, де age менше 18:



Перевіряємо:



Тригер спрацював правильно, видаливши 2 записи, де age менше 18.

Тепер зробимо запит, де змінимо вік на менший:



Перевіримо результат:



Отже, потрібні записи були видалено, а нове значення age записано. Тригерна функція працює правильно.

**Завдання 4**

Всього існує 4 аномалії:

1. Dirty reads

Транзакція A вставляє рядок в таблицю; транзакція B читає новий рядок, а транзакція A робить відкат.

Тобто, транзакції бачать зміни в інших транзакціях, які ще не завершилися. Даний вид аномалії не відбувається в мові PostgreSQL, але може бути в інших мовах SQL.

1. Non repeatable reads

Транзакція A читає рядок; транзакція B змінює рядок та робить коміт; транзакція A читає той же рядок вдруге та отримує новий результат.

Тобто, транзакції можуть бачити зафіксовані зміни інших транзакцій.

1. Phantom reads

Транзакція A читає всі рядки що задовольняють умову WHERE; транзакція B вставляє новий рядок, що задовольняє умову WHERE, а потім транзакція A отримує це нове значення.

Тобто, транзакції не бачать змін, що роблять інші транзакції, але бачать результат операції INSERT.

1. Serialization

Декілька паралельних транзакцій записують та читають інформацію; результат залежить від послідовності запуску та комітів.

Щоб вирішити всі ці аномалії, у мовах SQL існують 4 рівні ізоляції, які вирішують певний вид (або види) цих феноменів. У даній роботі ми розглянемо 3 з 4 видів: read committed, repeatable read та serializable.

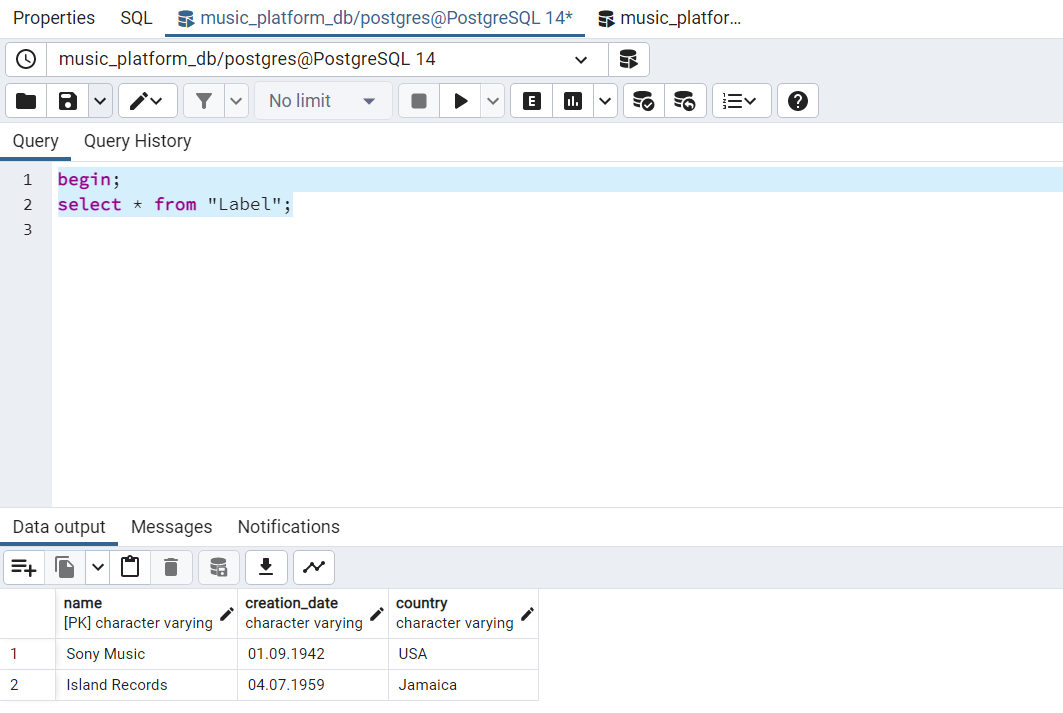
***READ COMMITED***

Почнемо з цього рівня ізоляції, адже він є рівнем ізоляції за замовчанням у PG. Він гарантує, що кожна дата, на момент зчитування, є закоміченою.

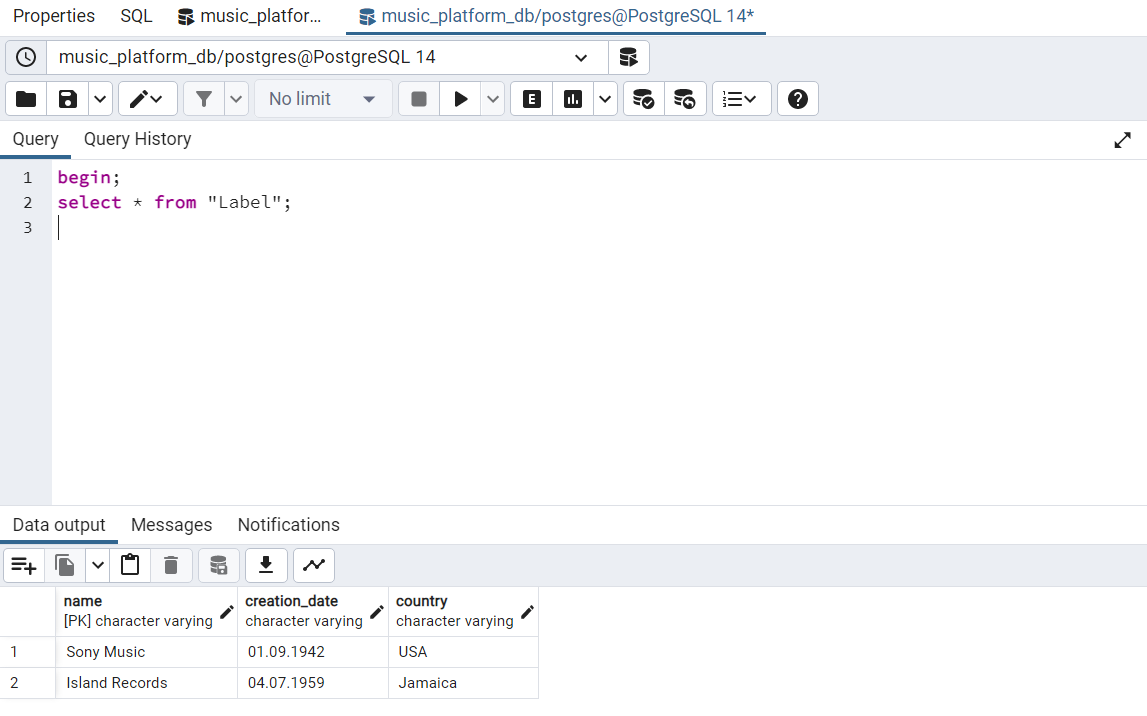
Даний вид ізоляції є найслабший і вирішує лише проблему dirty read, хоч дана аномалія взагалі не виникає в PG, тому перевіряти на прикладі dirty read не має сенсу для PostgreSQL. Тому краще подивитись приклади аномалій, які існують у PG та які дана ізоляція не в змозі вирішити.

Для початку переглянемо приклад для *nonrepeatable read*.

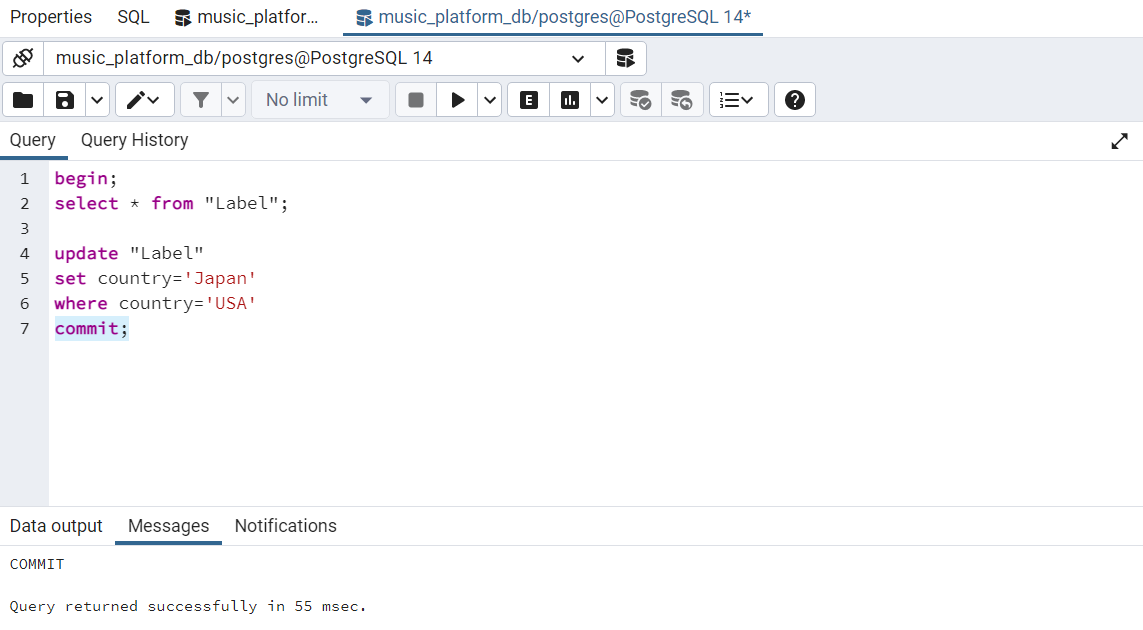
У першому екрані зчитуємо дані з таблиці Label:



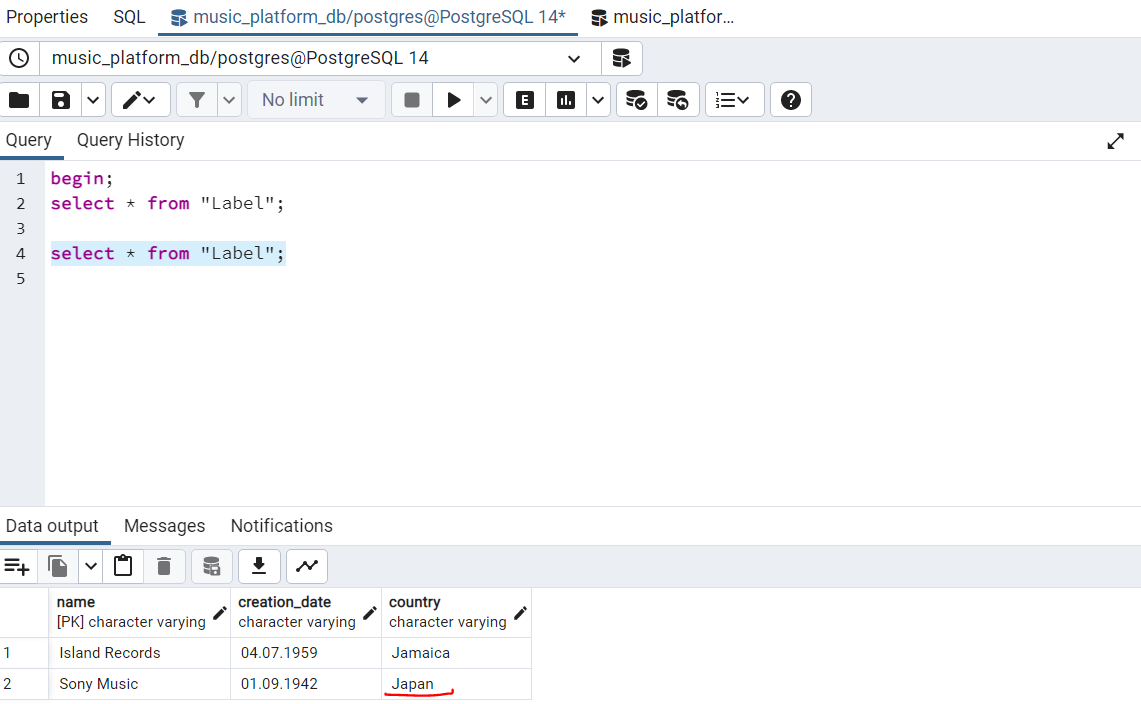
Далі робимо аналогічну дію в 2 екрані:



Потім в 2 екрані виконуємо певний запит, що змінить один рядок таблиці та закомітимо це:



Перевіряємо значення таблиці через перше вікно:



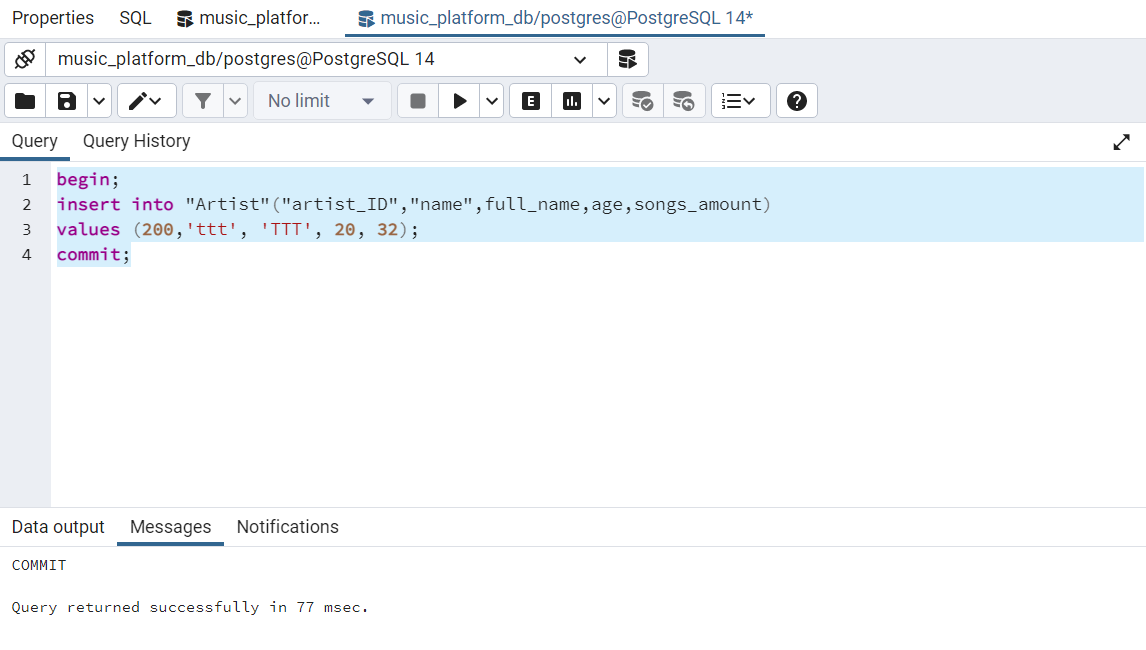
Як бачимо, закомічені дії в другому вікні змінили значення таблиці для першого вікна. Тобто, дана ізоляція не вирішує феномен *nonrepeatable reads*. І ми переконались в цьому на прикладі.

Йдемо далі, переглянемо приклад для феномену *phantom read*.

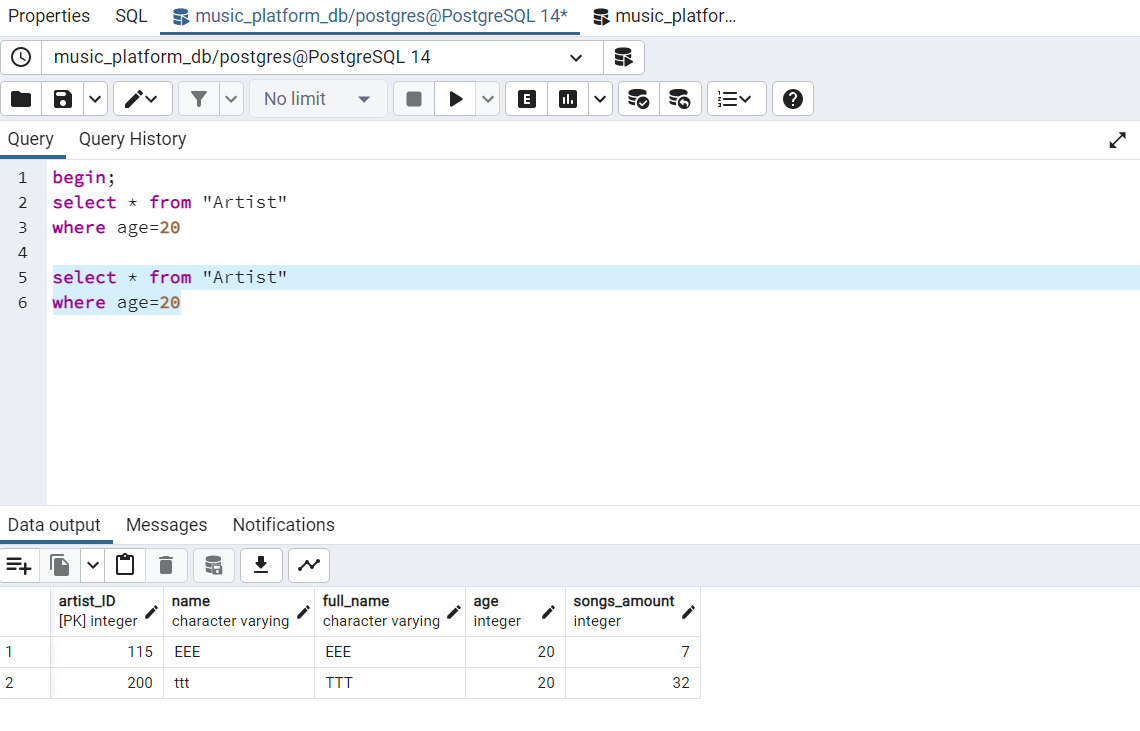
Зробимо такий запит для першого вікна:



Тепер зробимо такий запит для другого вікна та закомітимо:



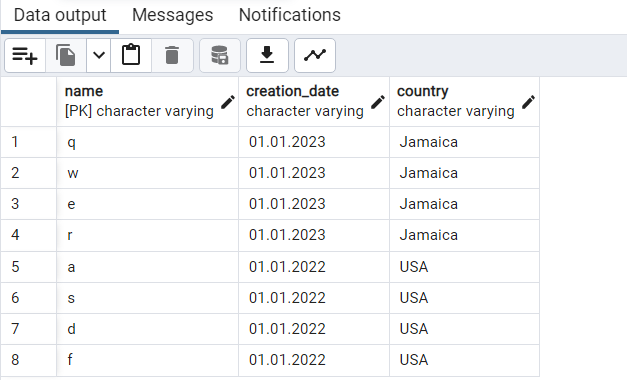
Тепер поглянемо, як це змінило дані для першого вікна:



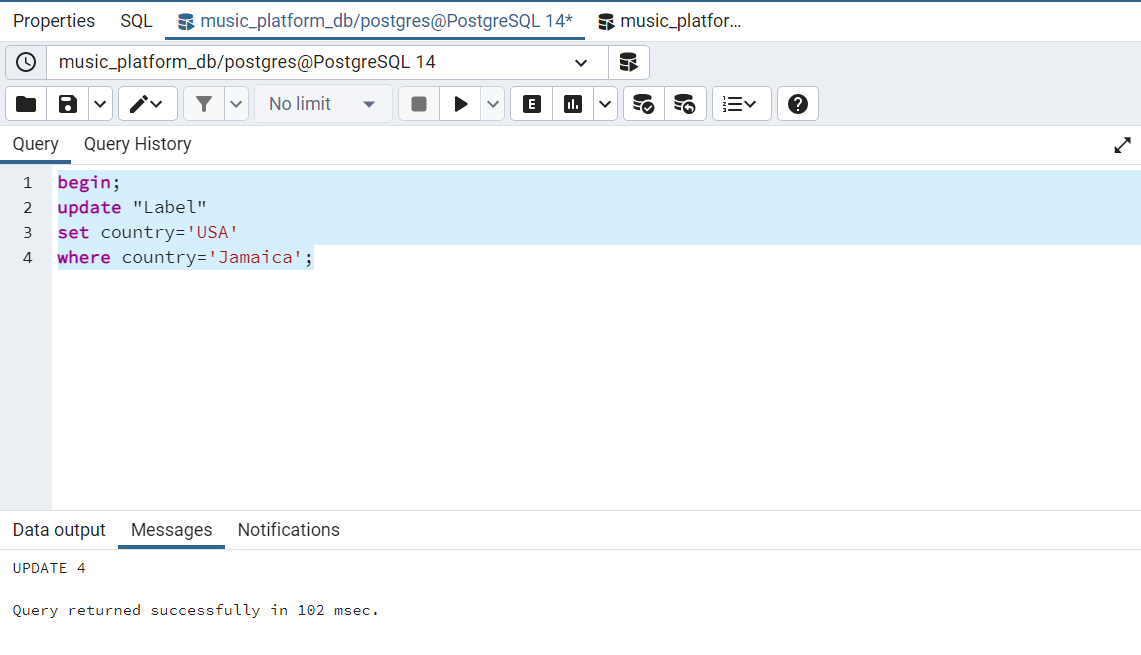
Отже, як бачимо, феномен phantom reads відбувається при ізоляції read committed.

Йдемо далі, переглянемо приклад для феномену *serialization*.

Для прикладу даної аномалії візьмемо таку таблицю:

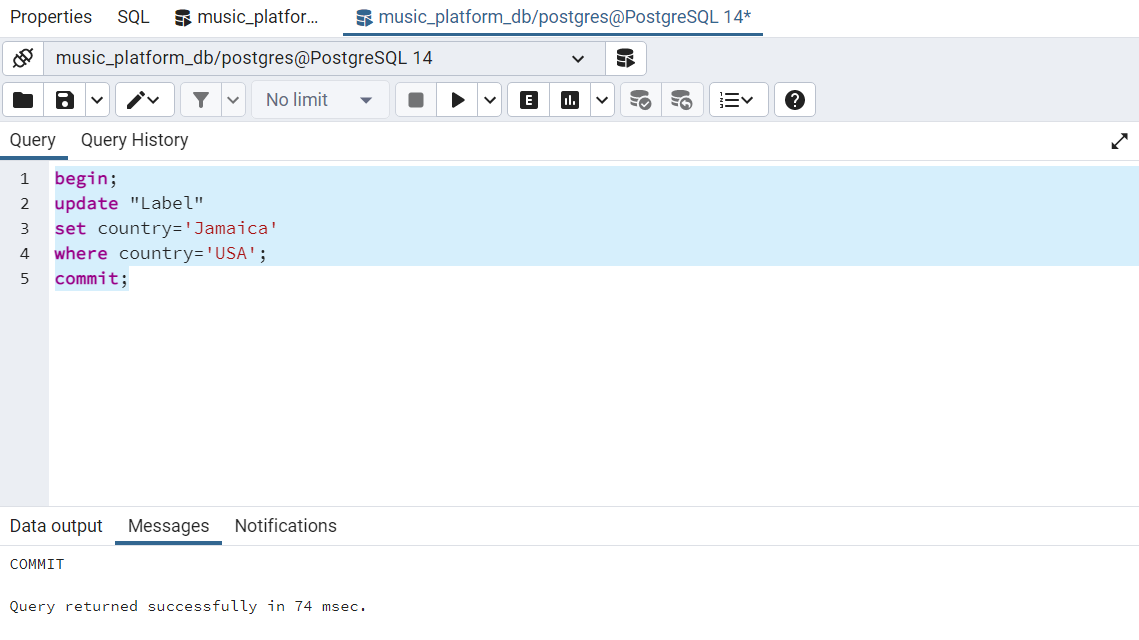


Тоді в першому вікні зробимо такий запит:

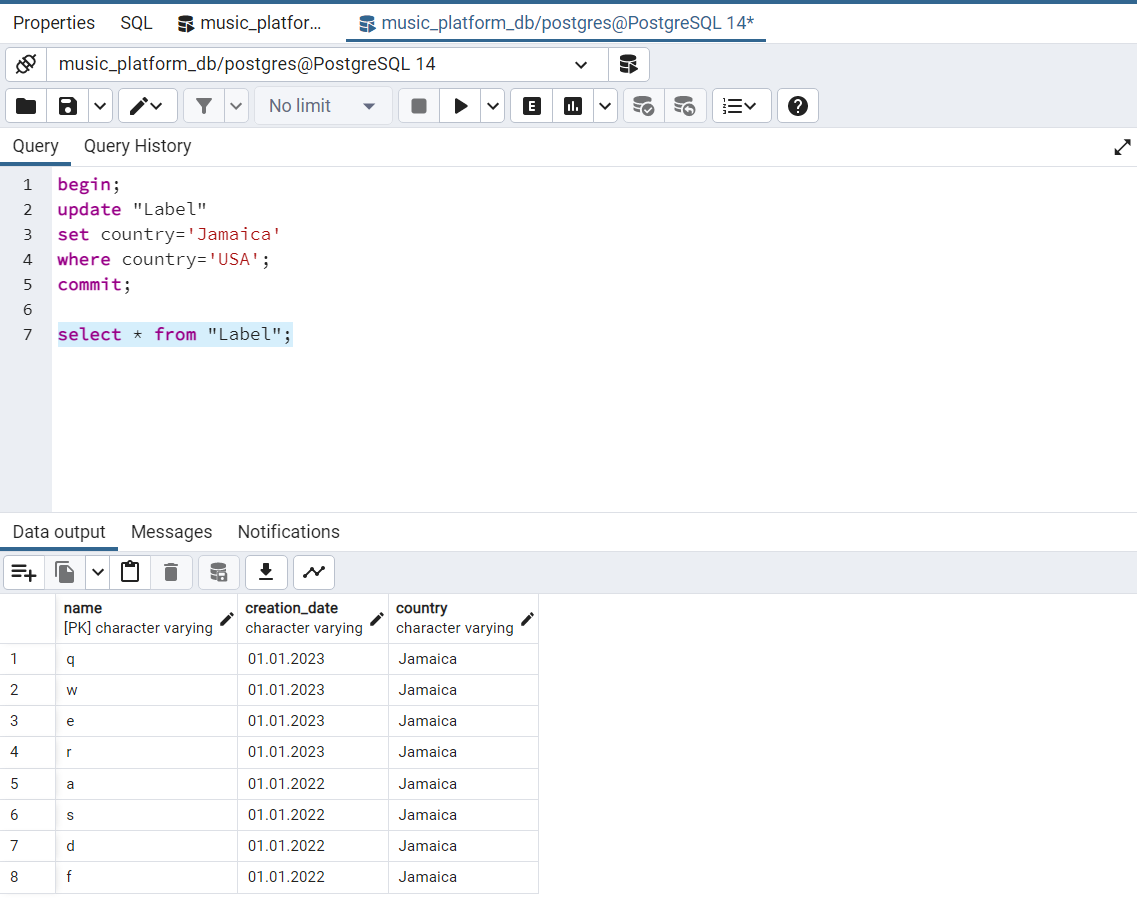


Тобто, перетворили все в значення ‘USA’.

А в другому вікні зробимо такий запит та закомітимо:

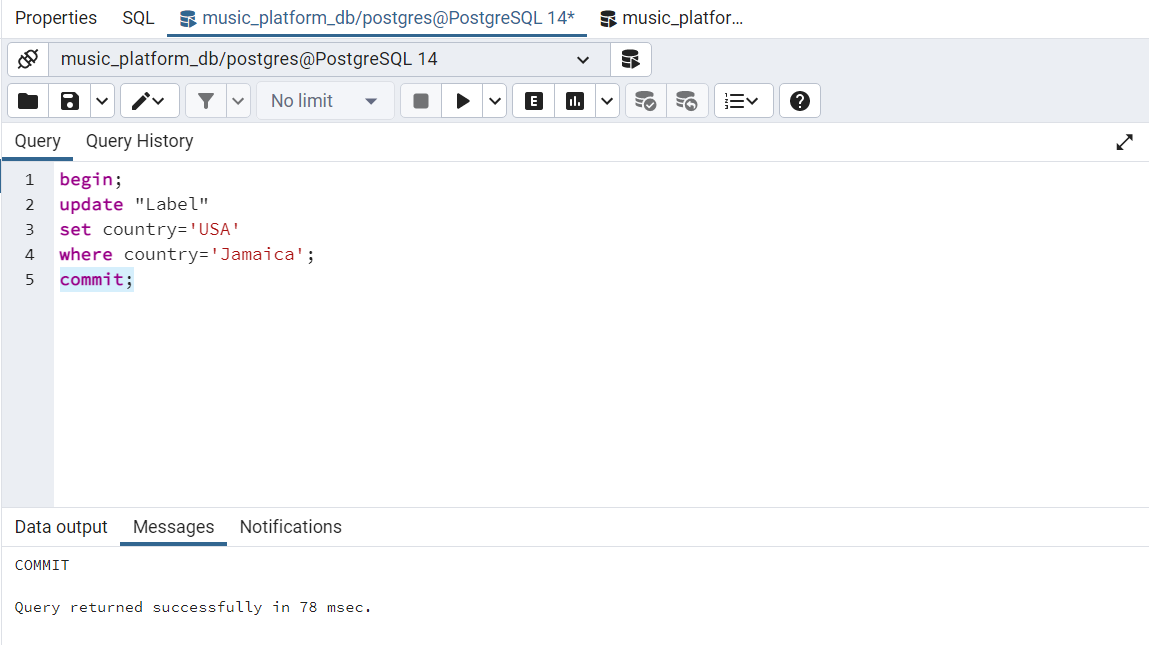


Тепер перевіримо стан таблиці у другому вікні:

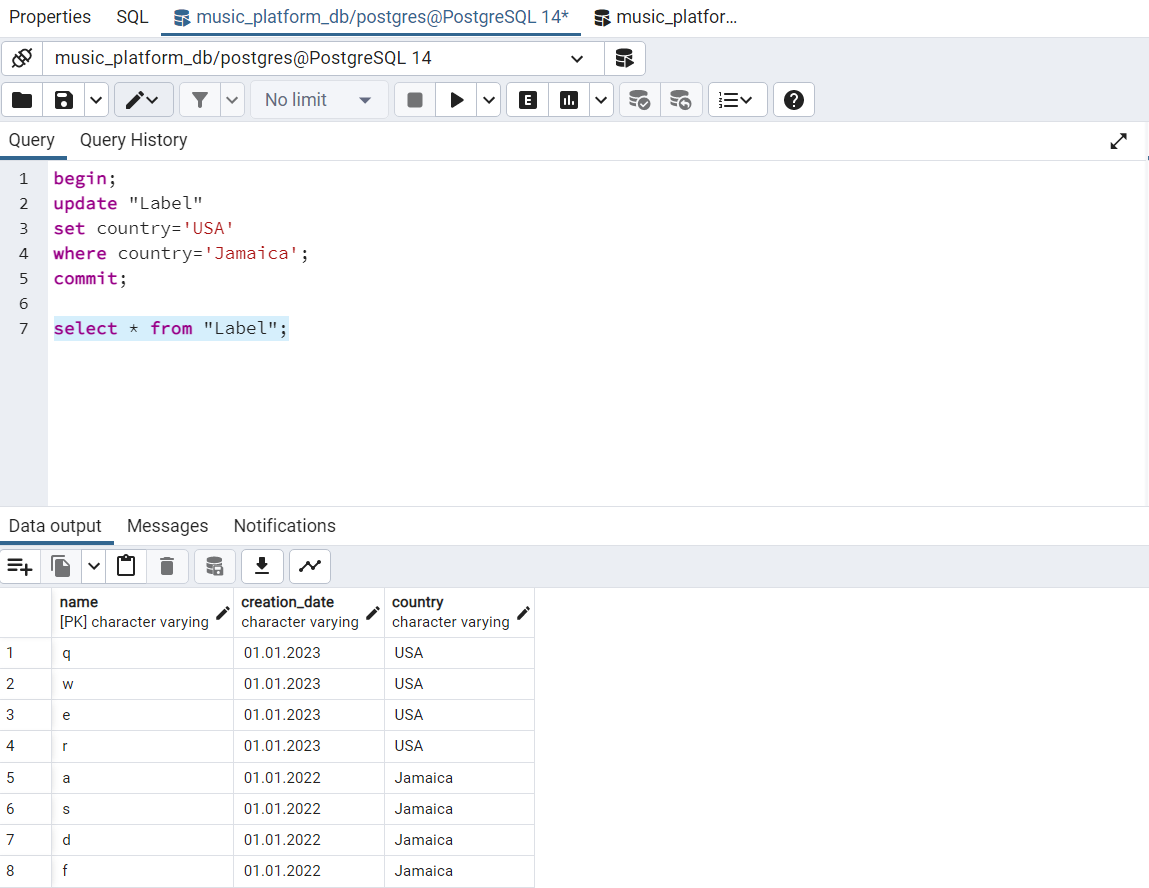


Тобто, перетворили все в значення ‘Jamaica’.

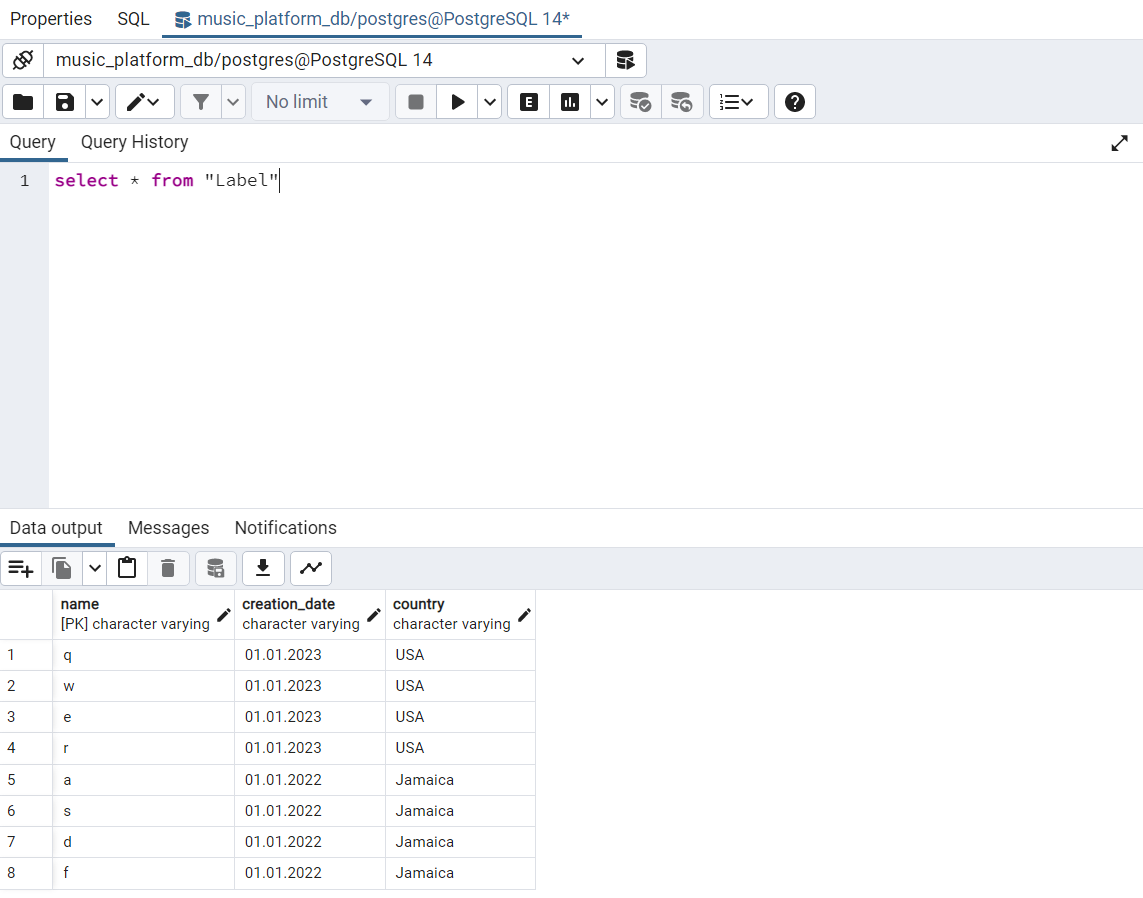
Тепер спробуємо закомітити зміни в першому вікні:



Переглянемо стан таблиці:



Тепер закриємо вікна та переглянемо остаточний результат:



Маємо взагалі щось дивне в результаті. Адже спочатку ми зберегли зміни, які перетворили всю таблицю в ‘Jamaica’, а потім зберегли зміни, де все перетворили в ‘USA’. А в результаті отримали таблицю, обернену до таблиці, яка була на початку. Я думаю, що це через те, що коли ми змінили та зберегли таблицю у другому вікні, то змінилось чотири записи. І ці чотири змінені записи були зчитані у першому вікні, як нові значення, і були записані в таблицю в першому вікні. І при коміті ми маємо таблицю з чотирьох нових значень першого вікна + чотири нові значення другого вікна. У результаті отримали обернену таблицю. Тобто як бачимо, втрачається послідовність запитів, що не є добре. Отже на прикладі переконались, що read committed не рятує від феномена serialization.

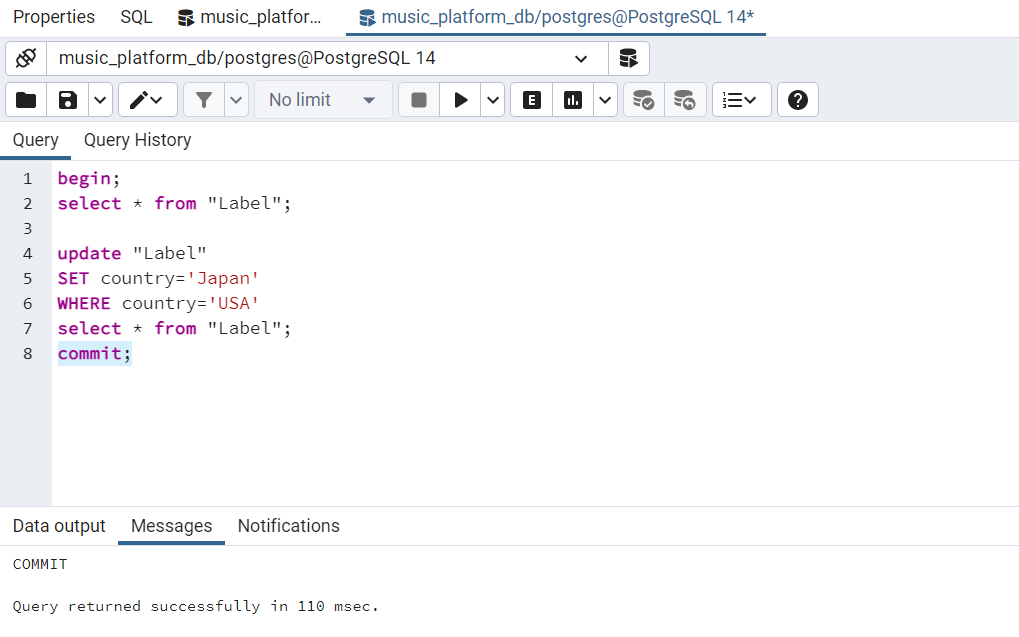
**REPEATABLE READ**

Найбільш обмежуючий рівень ізоляції. Транзакція зберігає блокування читання для всіх рядків, на які вона посилається, і записує блокування для всіх рядків, які вона вставляє/оновлює/видаляє. Даний рівень ізоляції захищає від феноменів dirty read (як було зазначено раніше, у PG даний феномен взагалі не виникає, тому опустимо приклад для нього) та nonrepeatable read. Але в PG він захищає ще й від phantom read. Тому переконаємось на прикладах.

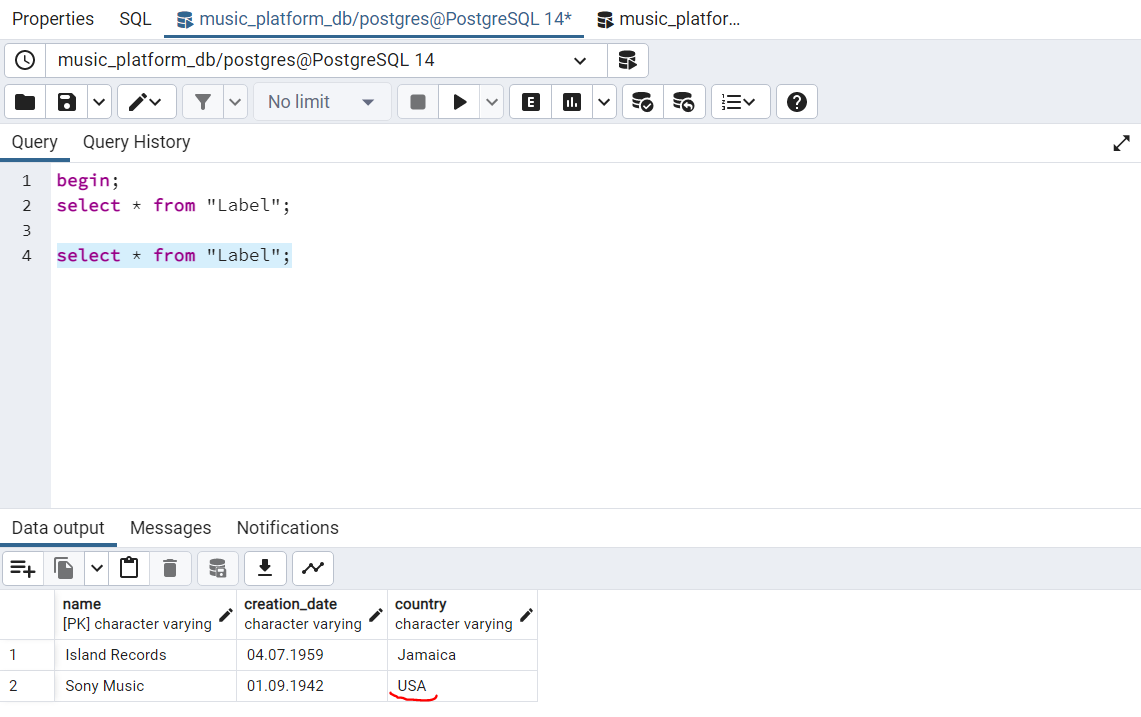
Спочатку встановимо даний рівень ізоляції для нашої бази даних.



Розглянемо феномен *nonrepeatable read*.

У другому вікні зробимо такий запит та закомітимо: 

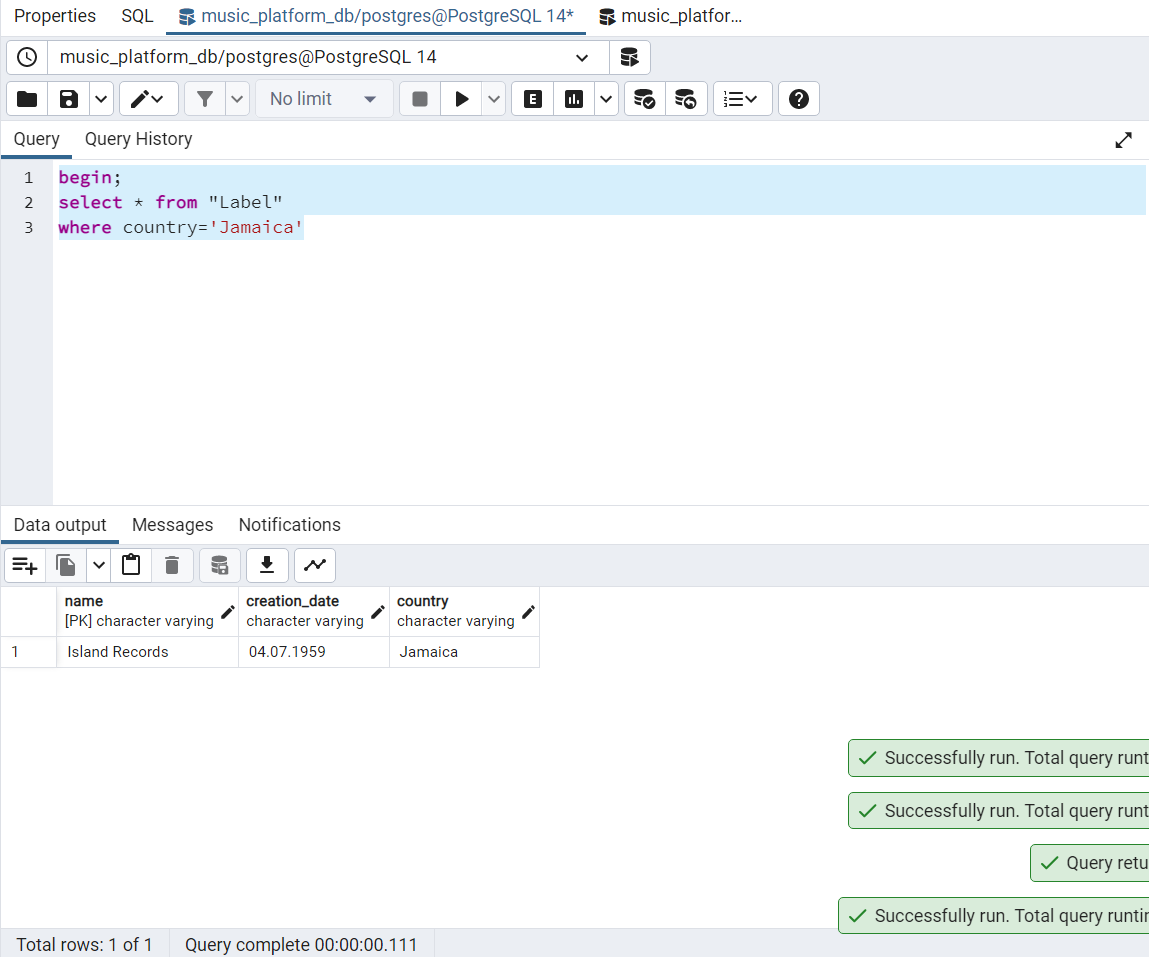
Тоді в першому вікні перевіримо результат:



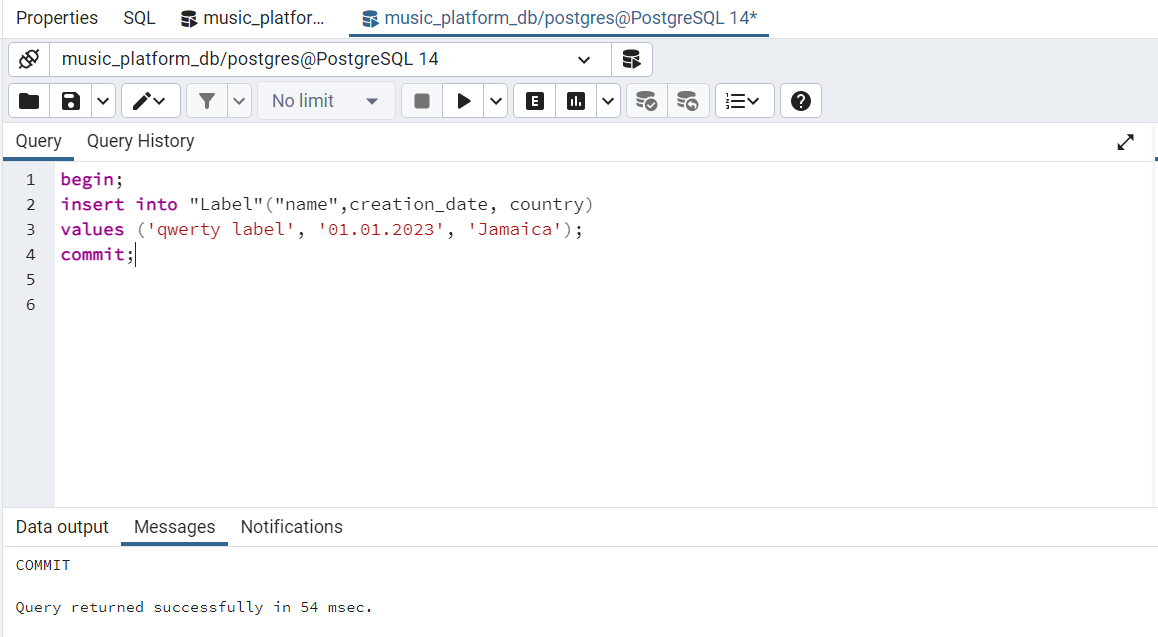
Запит у другому вікні мав змінити ‘USA’ на ‘Japan’, але в першому вікні маємо все ще ‘USA’. Отже, на прикладі переконались, що даний рівень ізоляції вирішує проблему феномена repeatable read.

Далі переглянемо на прикладі проблему феномена *phantom read.*

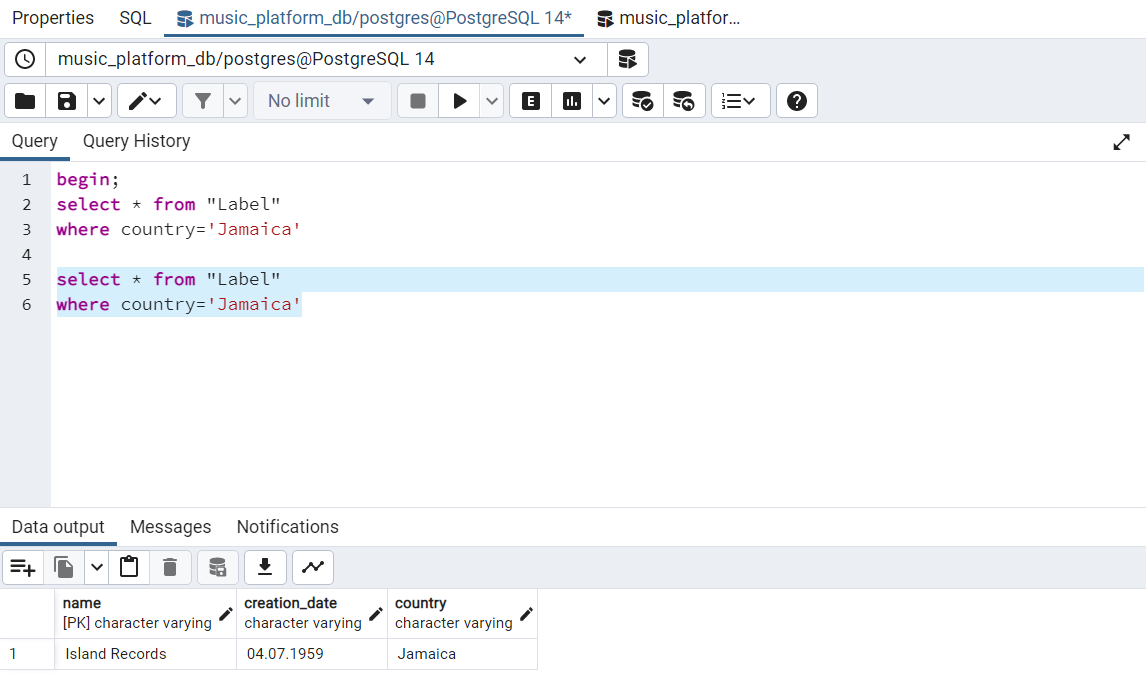
У першому вікні робимо такий запит:



Тоді у другому вікні робимо такий запит та комітимо:

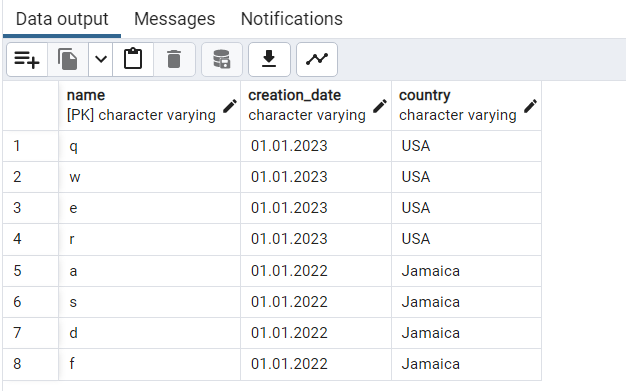


Далі в першому вікні повторюємо запит:

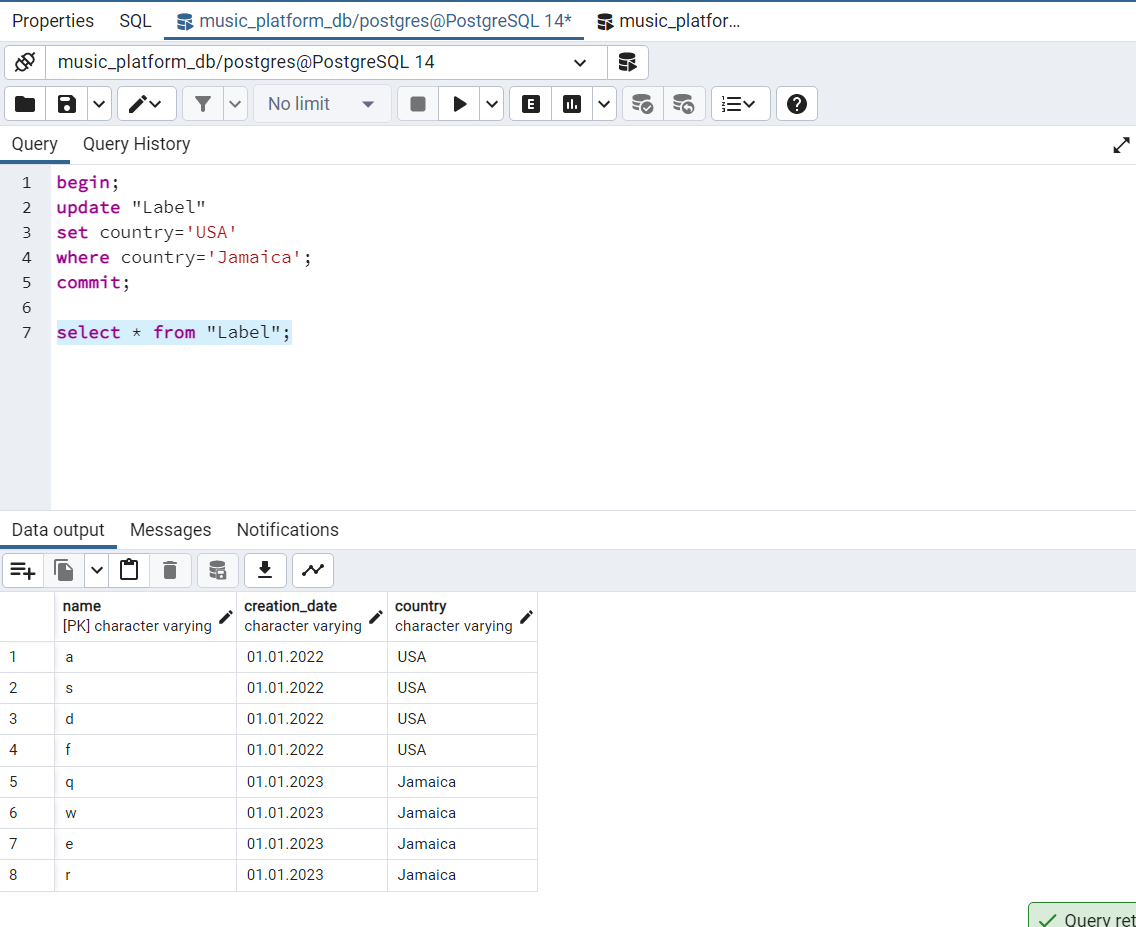


Як бачимо, нового запису при читанні у першому вікні не з’явилось. Отже, на прикладі переконались, що даний рівень ізоляції вирішує проблему феномена phantom read.

Тепер розглянемо проблему феномена *serialization.* Візьмемо таку таблицю:



Та повторимо ті ж дії, що робили для попереднього рівня ізоляції. Швидше за все, результат буде той же, тобто отримаємо обернену таблицю. Тож перевіримо. Отримуємо:



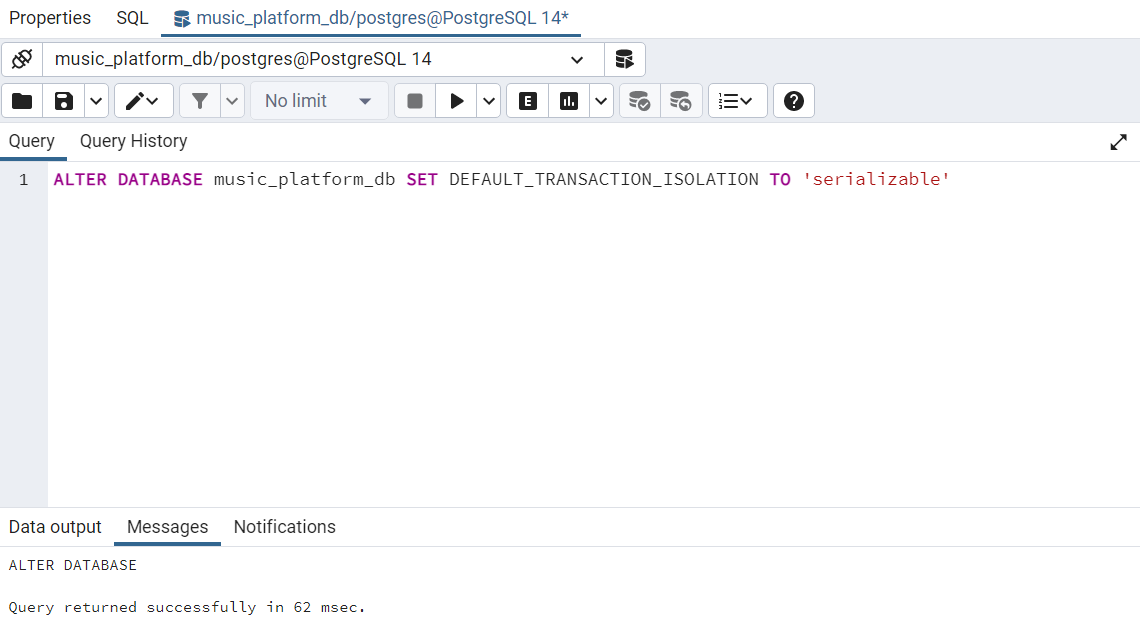
Так, результат очікуваний. Значення country просто змінились на протилежні. Тобто як бачимо, втрачається послідовність запитів, що не є добре. Отже на прикладі переконались, що repeatable read не рятує від феномена serialization.

**SERIALIZABLE**

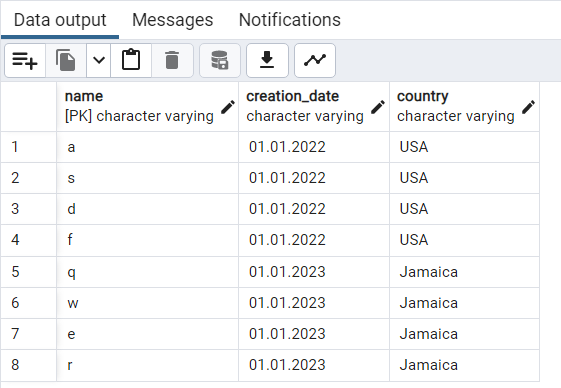
Даний рівень ізоляції є найвищим рівнем ізоляції та найскладнішим. Він гарантує серіалізацію та вирішує проблему всіх феноменів (у т.ч. serialization).

Ми вже побачили всі проблеми та їх вирішення на прикладі минулих рівнів ізоляції, окрім вирішення проблеми феномена serialization, тому опустимо приклади з вирішенням проблем інших феноменів даного рівня ізоляції та перейдемо безпосередньо до прикладу феномена *serialization*.

Але спочатку встановимо даний рівень ізоляції для нашої бази даних:

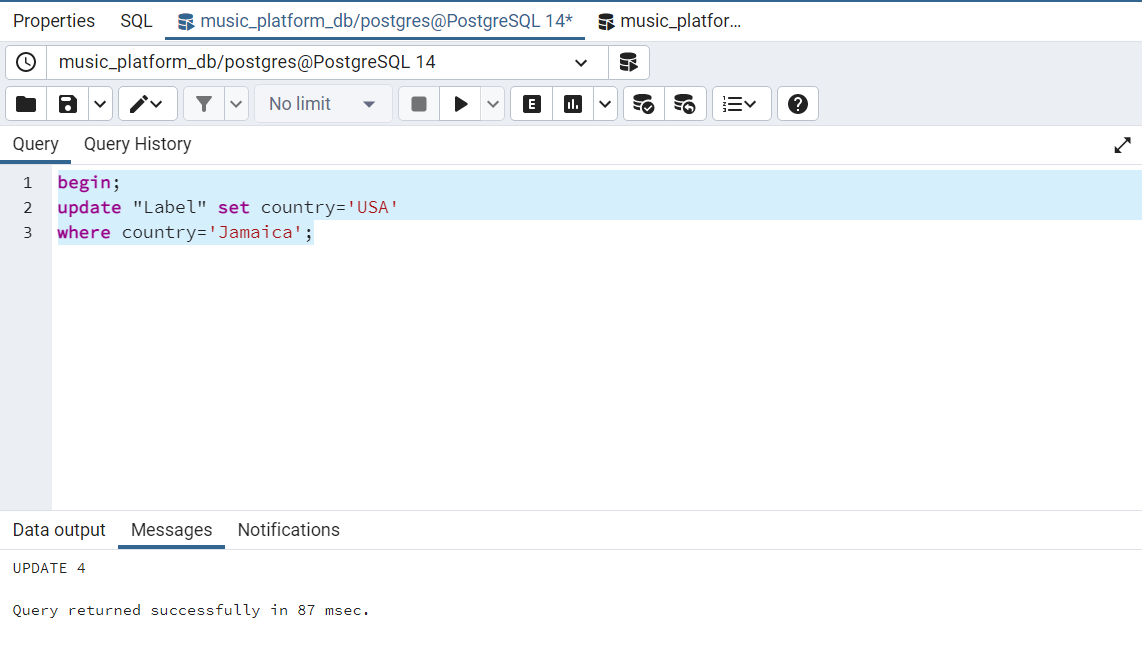


Знову ж таки, працюватимемо з тією ж таблицею:

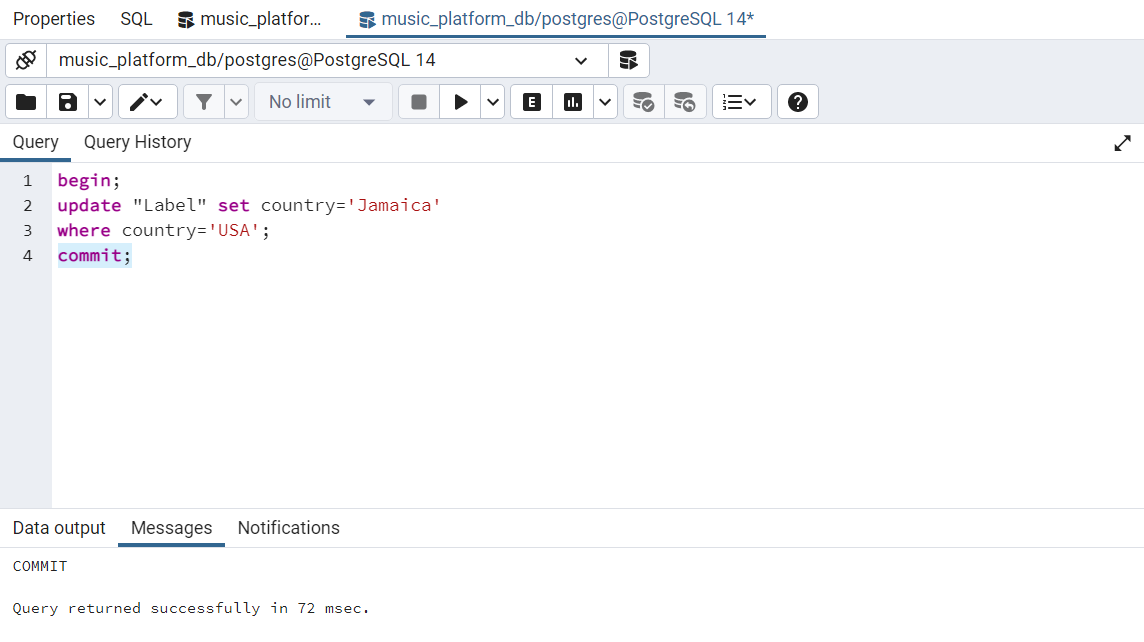


І використаємо ті ж самі запити.

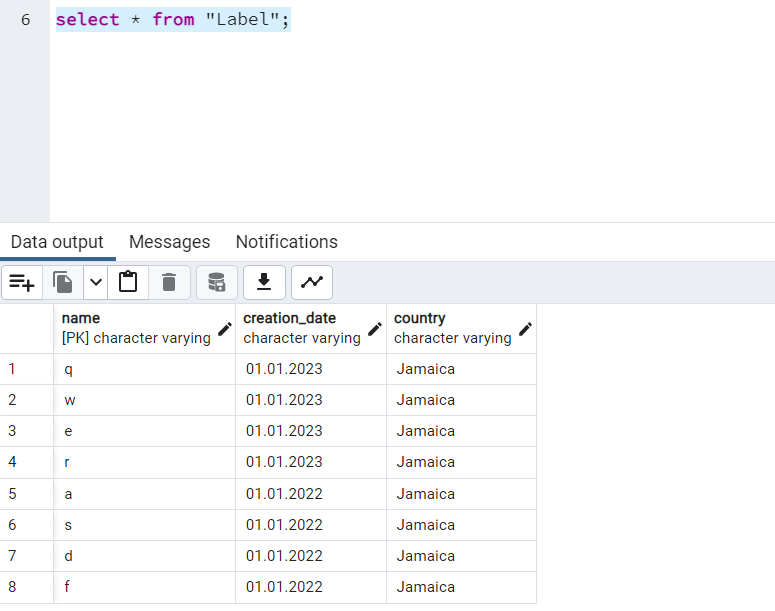
У першому вікні:



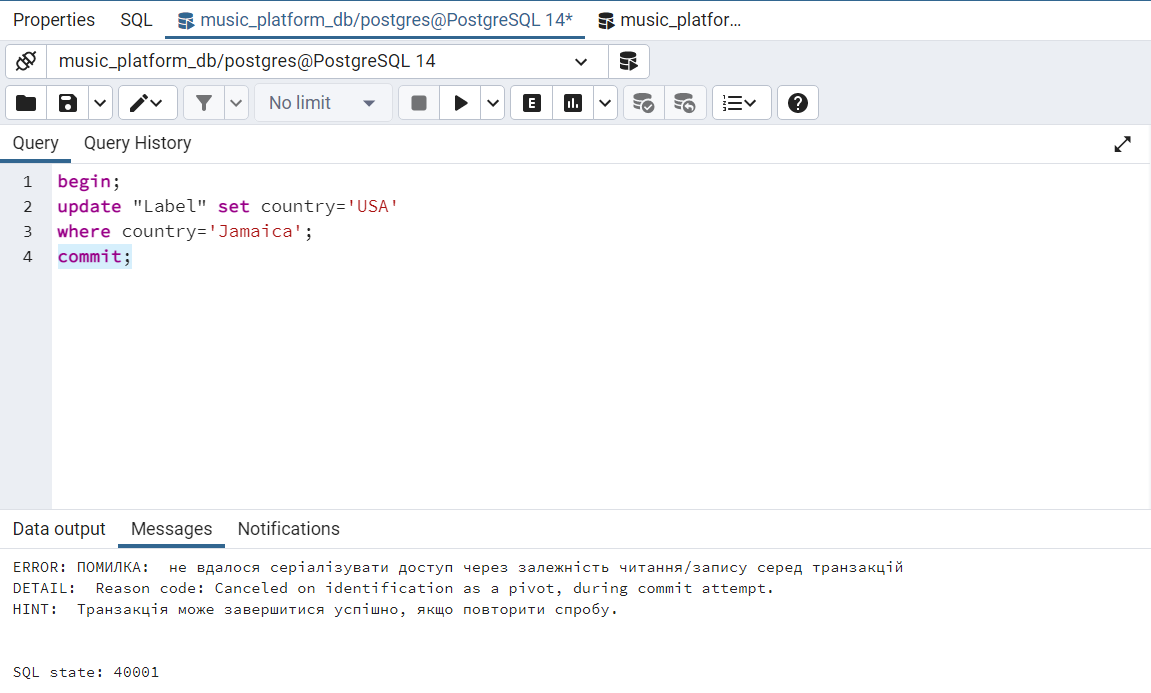
У другому вікні:



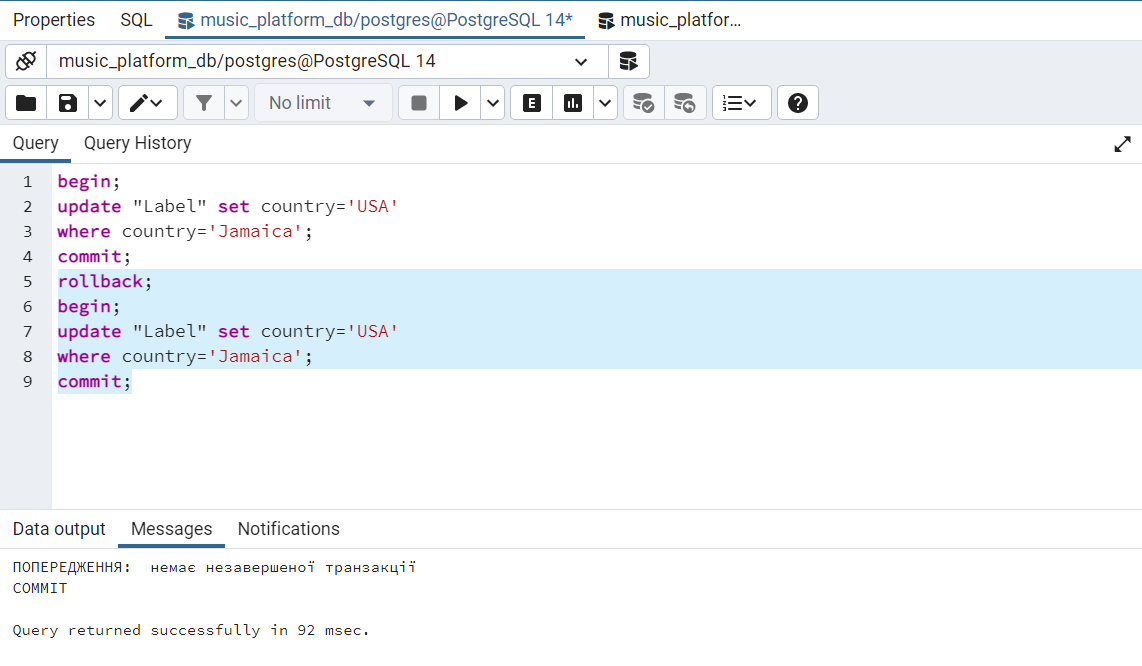
Перевіримо таблицю у другому вікні:



Тепер зробимо коміт у першому вікні:



Отримали повідомлення про помилку. Отже даний рівень ізоляції працює, і бере до уваги послідовність запитів. Тож, зробимо відкат у першому вікні та повторимо запит:



Тож, спочатку ми змінили всі значення ‘USA’ на ‘Jamaica’ у другому вікні, і вся таблиця складається з ‘Jamaica’. Потім у першому вікні ми зробили відкат першого запиту та повторили його, і за логікою, обернули всі значення ‘Jamaica’ на ‘USA’. Тобто після всіх комітів таблиця має повністю складатись із записів з ‘USA’. Перевіримо це:



Так, отримали те, що й мали отримати. Отже, на прикладі переконались, що даний рівень ізоляції вирішує проблему феномена serialization, яку не можуть вирішити інші рівні ізоляції.

**Підсумкова таблиця для PG**

|  |  |  |  |
| --- | --- | --- | --- |
| **Isolation Level** | **Nonrepeatable Read** | **Phantom Read** | **Serialization Anomaly** |
| Read committed | Не захищає | Не захищає | Не захищає |
| Repeatable read | Захищає | Захищає | Не захищає |
| Serializable | Захищає | Захищає | Захищає |