

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 та засоби управління»**

ТЕМА: «Засоби оптимізації роботи СУБД PostgreSQL»

Виконала:

студентка ІII курсу

групи КВ-02

Дахал К.

Перевірив:

доц. Павловський В.І.

Київ 2022

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.

Номер залікової книжки – **0207**.

|  |  |  |
| --- | --- | --- |
| № варіанта | Види індексів | Умови для тригера |
| 7 | GIN, BRIN | before insert, delete |

**Логічна модель (схема) БД «Музичний гурт»**

Логічну модель (схему бази даних) наведено на рисунку 1.

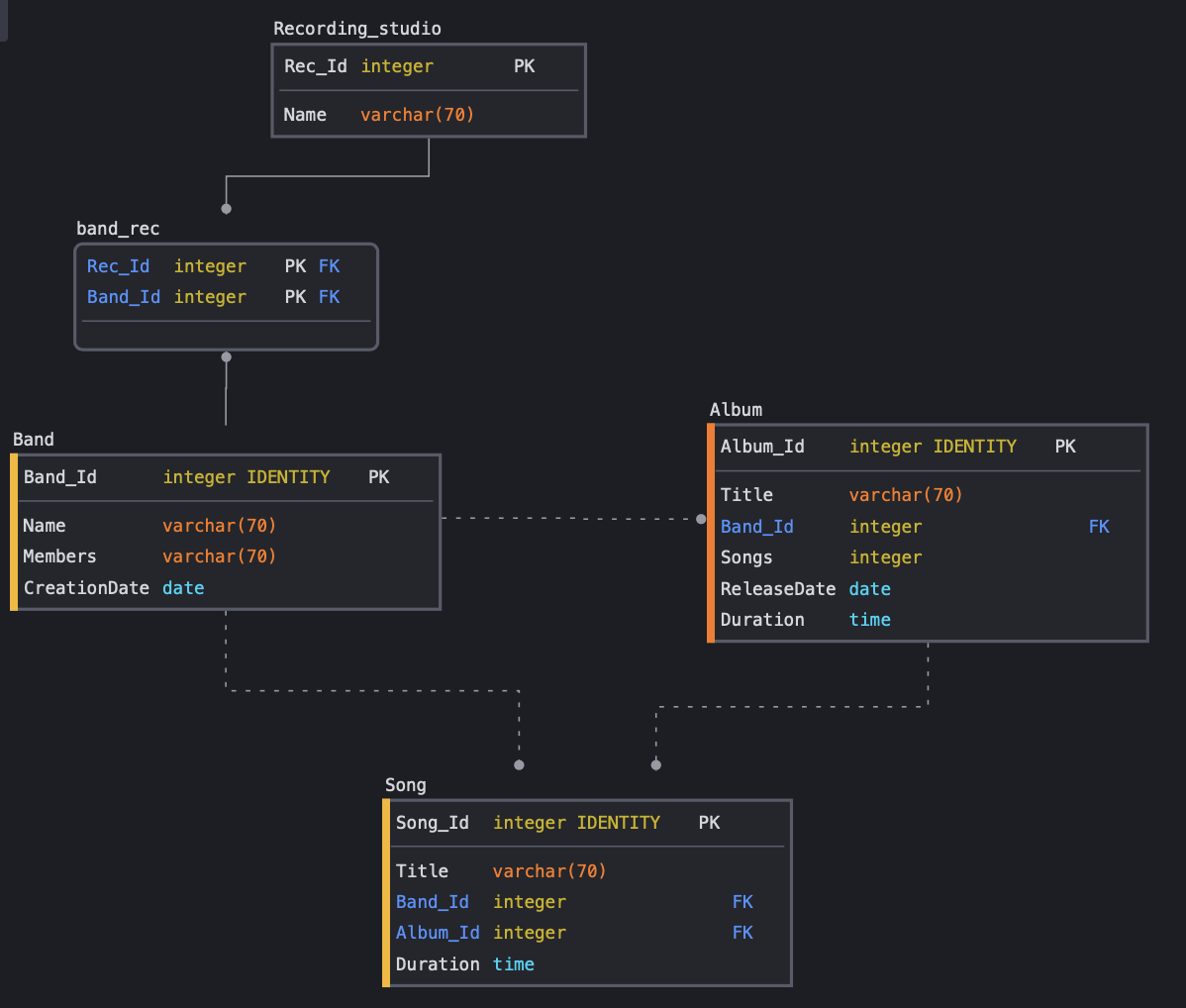


Рисунок 1 – Схема бази даних (інструмент: sqldbm.com)

**Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM)**

using Microsoft.EntityFrameworkCore;

using System;

using System.Collections.Generic;

public class Band{

public int Band\_Id;

public string Name;

public string Members;

public Date CreationDate;

public int getBand\_Id(){

return Band\_Id;

}

public void setBand\_Id(int Band\_Id){

this.Band\_Id = Band\_Id;

}

public string getName(){

return Name;

}

public void setName(string Name){

this.Name = Name;

}

public string getMembers(){

return Members;

}

public void setMembers(string Members){

this.Members = Members;

}

public Date getCreationDate(){

return CreationDate;

}

public void setCreationDate(Date CreationDate){

this.CreationDate = CreationDate;

}

}

public class Album{

public int Album\_Id;

public string Title;

public virtual Band Band\_Id;

public int Song;

public Date ReleaseDate;

public Time Duration;

public int getAlbum\_Id(){

return Album\_Id;

}

public void setAlbum\_Id(int Album\_Id){

this.Album\_Id = Album\_Id;

}

public string getTitle(){

return Title;

}

public void setTitle(string Title){

this.Title = Title;

}

public int getBand\_Id(){

return Band\_Id;

}

public void setBand\_Id(int Band\_Id){

this.Band\_Id = Band\_Id;

}

public int getSong(){

return Song;

}

public void setSong(int Song){

this.Song = Song;

}

public Date getReleaseDate(){

return ReleaseDate;

}

public void setReleaseDate(Date ReleaseDate){

this.ReleaseDate = ReleaseDate;

}

public Time getDuration(){

return Duration;

}

public void setDuration(Time Duration){

this.Duration = Duration;

}

}

public class Song{

public int Song\_Id;

public string Title;

public virtual Band Band\_Id;

public virtual Album Album\_Id;

public Time Duration;

public int getSong\_Id(){

return Song\_Id;

}

public void setSong\_Id(int Song\_Id){

this.Song\_Id = Song\_Id;

}

public string getTitle(){

return Title;

}

public void setTitle(string Title){

this.Title = Title;

}

public int getBand\_Id(){

return Band\_Id;

}

public void setBand\_Id(int Band\_Id){

this.Band\_Id = Band\_Id;

}

public int getAlbum\_Id(){

return Album\_Id;

}

public void setAlbum\_Id(int Album\_Id){

this.Album\_Id = Album\_Id;

}

public Time getDuration(){

return Duration;

}

public void setDuration(Time Duration){

this.Duration = Duration;

}

}

public class Recording\_studio{

public int Rec\_Id;

public string Name;

public int getRec\_Id(){

return Rec\_Id;

}

public void setRec\_Id(int Rec\_Id){

this.Rec\_Id = Rec\_Id;

}

public string getTitle(){

return Title;

}

public void setTitle(string Title){

this.Title = Title;

}

}

public class band\_rec{

public virtual Recording\_studio Rec\_Id;

public virtual Band Band\_Id;

public int getRec\_Id(){

return Rec\_Id;

}

public void setRec\_Id(int Rec\_Id){

this.Rec\_Id = Rec\_Id;

}

public int getBand\_Id(){

return Band\_Id;

}

public void setBand\_Id(int Band\_Id){

this.Band\_Id = Band\_Id;

}

}

Реалізація вставки:

var album = db.Song.SingleOrDefault();

var song = new Song() { Created = DateTime.Now };

db.Songs.Add(song);

db.SaveChanges();

Реалізація редагування:

var song = db.Song.SingleOrDefault();

song.Quantity++;

db.SaveChanges();

Реалізація вилучення:

var song = db.Song.SingleOrDefault();

db.Song.Remove(product);

db.SaveChanges();

Реалізація пошуку:

// find all songs with names starting with B

var song1 = from b in context.Song

where b.Name.StartsWith("B")

select b;

// find the Band named Arctic Monkeys

var band1 = context.Band

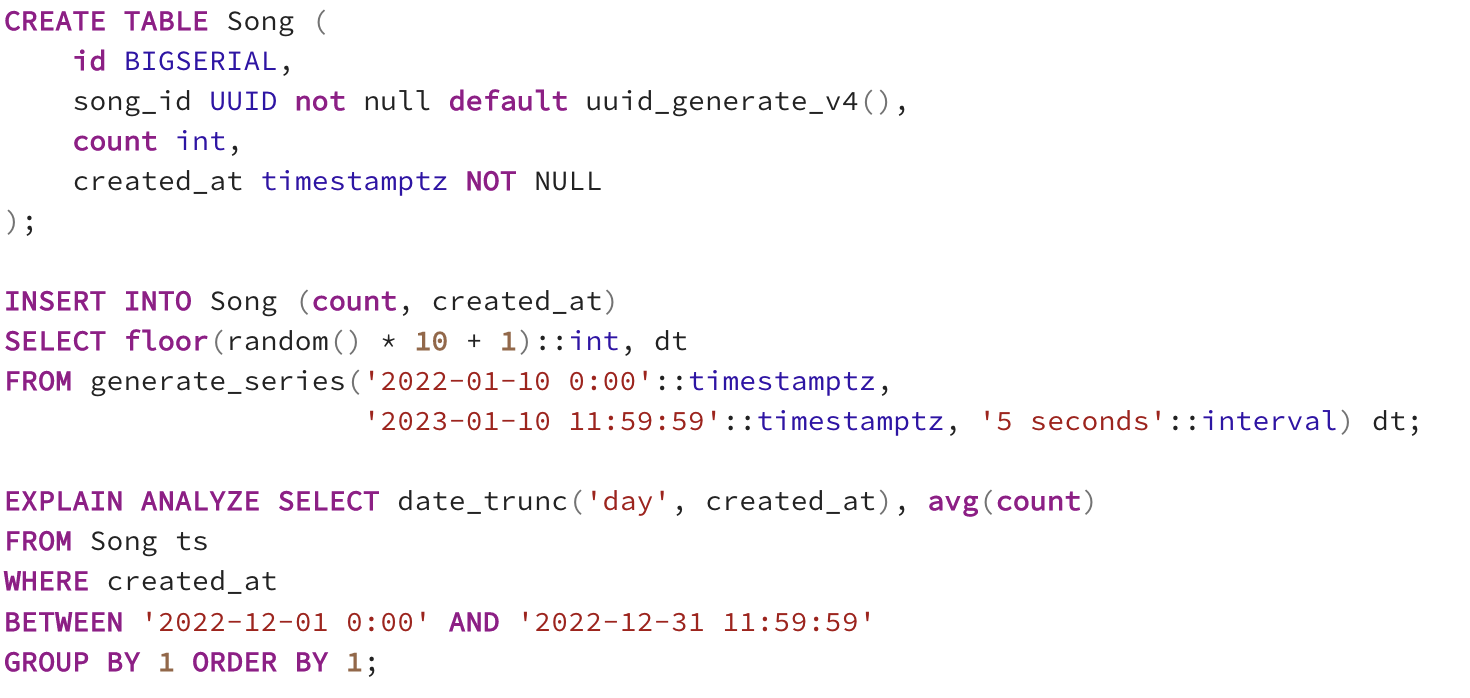
.Where(b => b.Name == "Arctic Monkeys")

.FirstOrDefault();

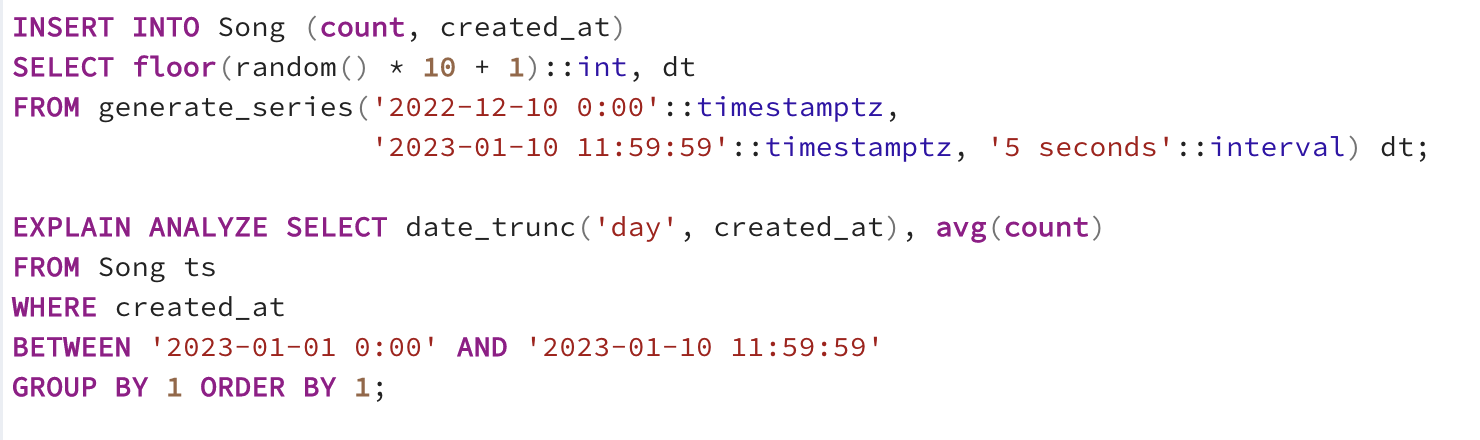
**Створити та проаналізувати різні типи індексів у PostgreSQL**

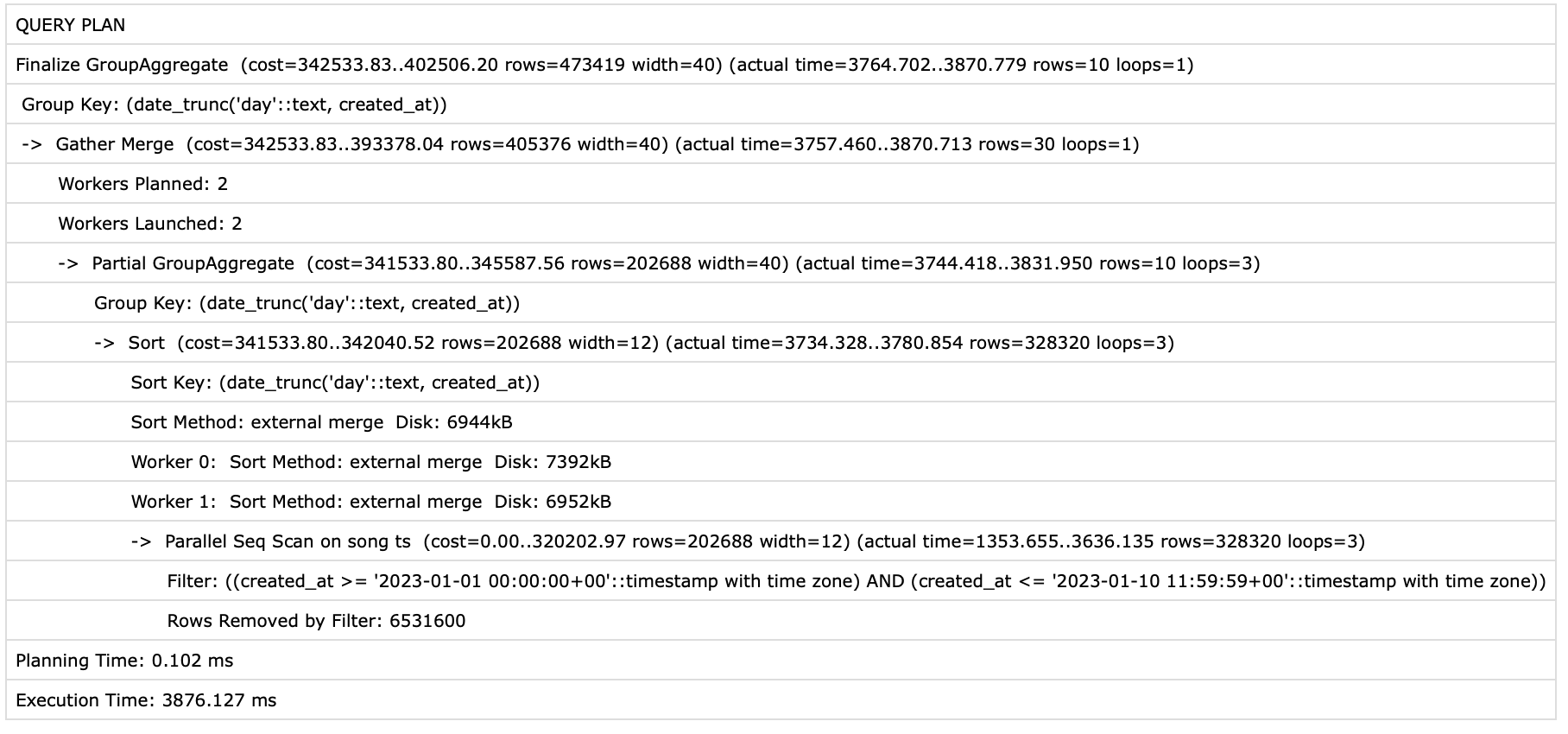
**BRIN**

Створимо таблицю:



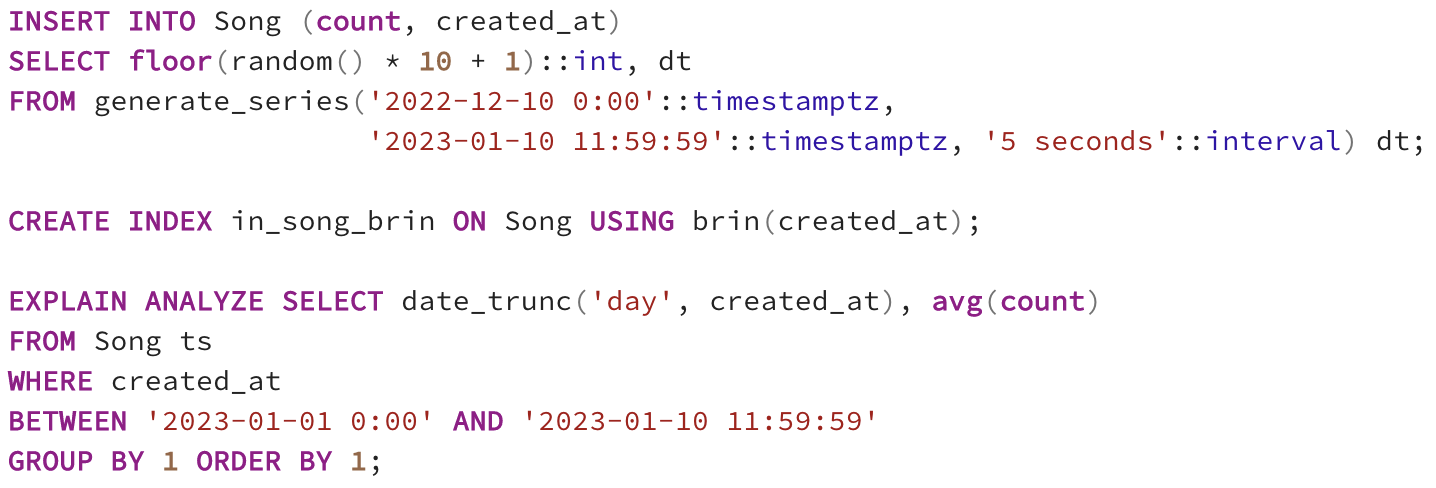
Перевіримо час виконання без індексу:

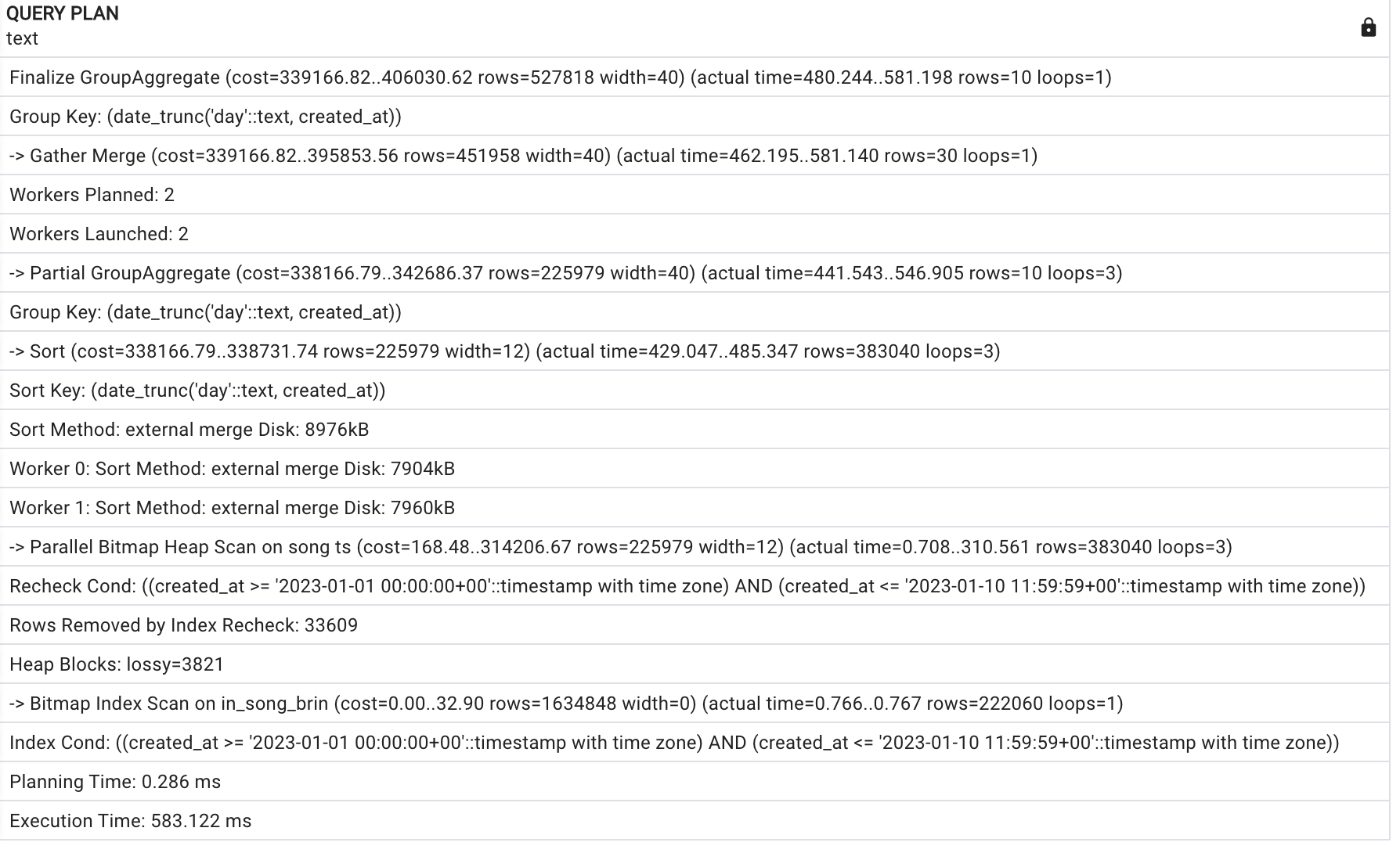


****

Час виконання: 3876 мс.

Перевіримо час роботи з індексом:





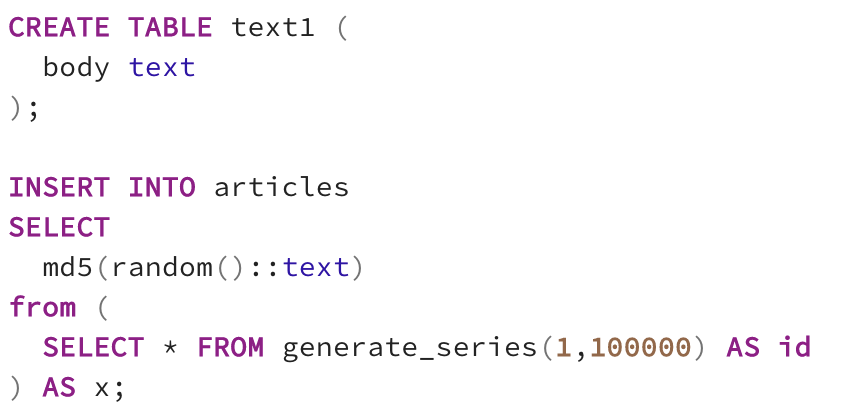
Час виконання: 583 мс.

Проаналізувавши дані можна зазначити, що час виконання з використанням індексу BRIN зменшився в 6,6 разів. Це свідчить про ефективну роботу індексу з великими об’ємами даних. Індекс BRIN корисно використовувати для впорядкованих наборів даних, адже він значно заощаджує місце для аналогічної, а іноді й кращої продуктивності.

Окрім цього, сьогодні багато програм записують дані, які мають спільний атрибут: мітку часу, яка постійно збільшується. Ця позначка часу є дуже цінною, оскільки вона служить основою для аналітичних запитів.

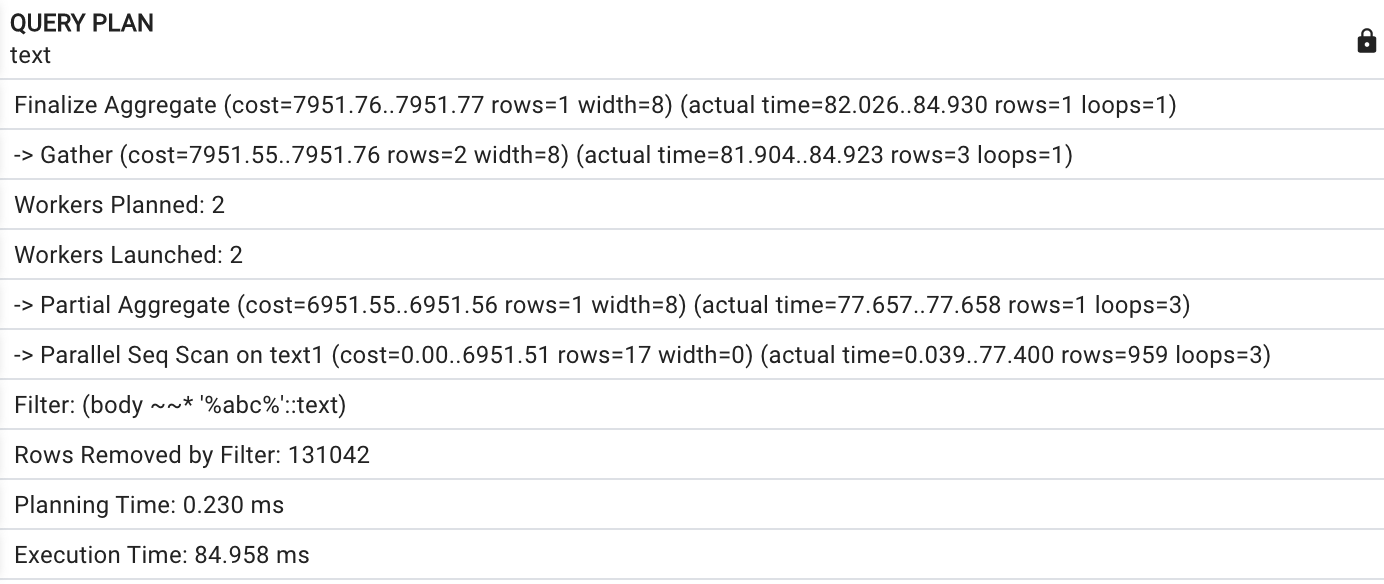
**GIN**

Створимо нову таблицю та заповнимо її текстовими даними:



Проаналізуємо роботу без використання індексу:

****

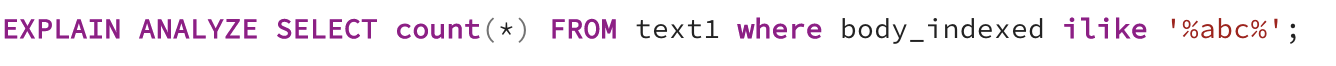
****

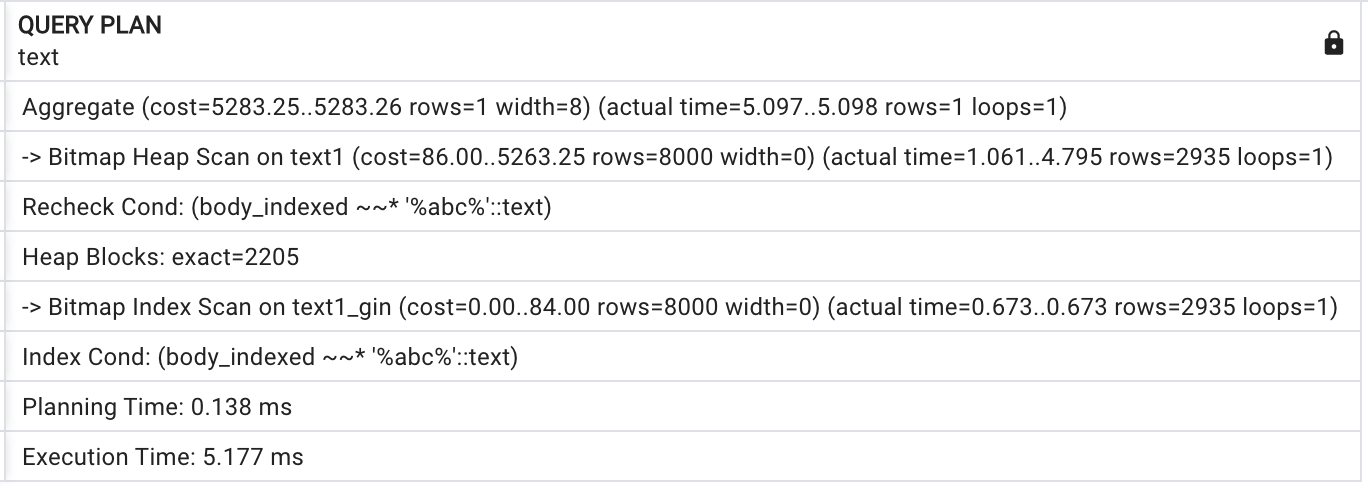
Час виконання: 85 мс.

Використаємо індекс:

*CREATE EXTENSION pg\_trgm;*







Час виконання: 5 мс.

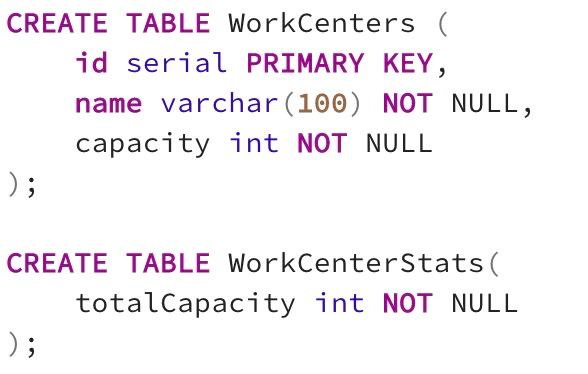
Отже, час виконання зменшився в 17 разів. Індекс GIN створено для роботи з типами даних, які можна розділити, і шукати значення окремих компонентів (елементи масиву, лексеми в текстовому документі). Тобто індекс GIN є ефективним в разі застосування текстового пошуку.

**Розробка тригерів бази даних PostgreSQL**

**Before insert**

Створимо тригер Before insert для отримання сумарної таблиці з іншої таблиці. Даний тригер запускається автоматично до того, як у таблиці відбудеться подія вставки.

Створюємо дві таблиці:



Створюємо тригер:

DELIMITER $$

CREATE TRIGGER before\_workcenters\_insert

BEFORE INSERT

ON WorkCenters FOR EACH ROW

BEGIN

DECLARE rowcount INT;

SELECT COUNT(\*)

INTO rowcount

FROM WorkCenterStats;

IF rowcount > 0 THEN

UPDATE WorkCenterStats

SET totalCapacity = totalCapacity + new.capacity;

ELSE

INSERT INTO WorkCenterStats(totalCapacity)

VALUES(new.capacity);

END IF;

END $$

DELIMITER ;

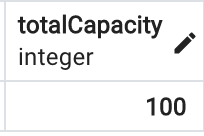
Протестуємо результат:

Спочатку додаємо новий ряд в таблицю WorkCenter:

*INSERT INTO WorkCenters(name, capacity)*

*VALUES('Mold Machine',100);*

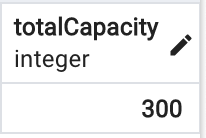
*SELECT \* FROM WorkCenterStats;*



Повторюємо дії:

*INSERT INTO WorkCenters(name, capacity)*

*VALUES('Packing',200);*

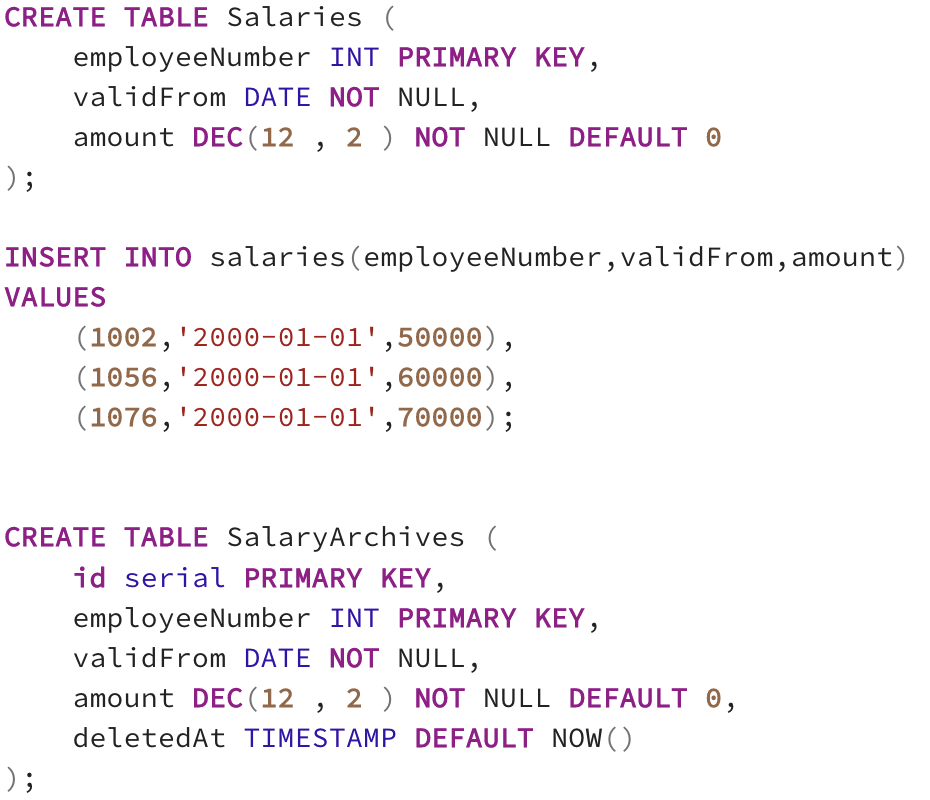


Тригер оновив загальну ємність зі 100 до 300, як і очікувалося.

**Before delete**

Створимо тригер Before delete для додавання видалених строчок в архівовану таблицю. Даний тригер запускається автоматично до того, як у таблиці відбудеться подія видалення.

Створимо дві таблиці:



Створимо тригер:

DELIMITER $$

CREATE TRIGGER before\_salaries\_delete

BEFORE DELETE

ON salaries FOR EACH ROW

BEGIN

INSERT INTO SalaryArchives(employeeNumber,validFrom,amount)

VALUES(OLD.employeeNumber,OLD.validFrom,OLD.amount);

END $$

DELIMITER ;

Протестуємо:

*DELETE FROM salaries*

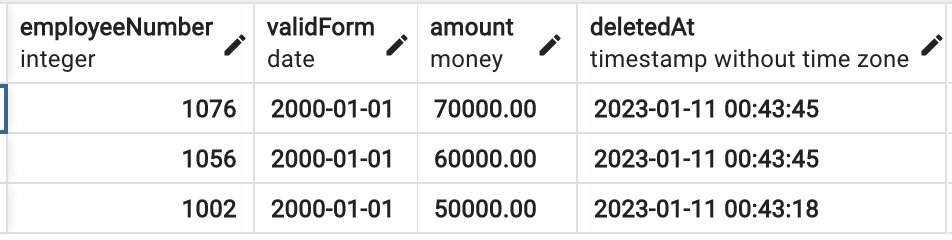
*WHERE employeeNumber = 1002;*

*SELECT \* FROM SalaryArchives;*



*DELETE FROM salaries;*

*SELECT \* FROM SalaryArchives;*



Як бачимо, з таблиці «Зарплатня» було видалено 2 рядки.

**Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL**

Рівні ізоляції транзакції:

* Read uncommitted.

У стандарті SQL дозволяється читання даних, які були внесені ще не завершеними транзакціями, однак у PostgreSQL ця можливість відсутня, тому read uncommitted не представлений.

* Read committed.

Виконується безпечне читання даних, тобто результати транзакцій, які ще не були успішно завершені не будуть видимі на цьому рівні, однак, якщо в процесі роботи однієї транзакції інша була завершена, то вона буде зафіксована. У PostgreSQL встановлений за замовчуванням.

* Repeatable read.

Бачить лише ті дані, які були у базі даних на момент початку транзакції, однак дані, що були попередньо внесені у цій ж транзакції будуть доступні.

* Serializable.

Повна ізоляція транзакцій одна від одної.

Типи феноменів, які можуть виникати на різних рівнях ізоляції транзакцій:

* Dirty read.
* Дозволяється читання даних незавершених транзакцій. Не можливий у PostgreSQL.
* Non-Repeatable read.