НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

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

**Лабораторна робота №2**

з дисципліни **Бази даних і засоби управління**

*на тему: “Створення додатку бази даних, орієнтованого на взаємодію з СУБД PostgreSQL”*

Виконав:

студент ІII курсу

групи КВ-01

Щикалюк Г. І.

Перевірив:

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.

|  |  |  |
| --- | --- | --- |
| *№ варіанта* | *Види індексів* | *Умови для тригера* |
| *25* | *BTree, GIN* | *after delete, insert* |

**Логічна модель бази даних**

Нижче (Рисунок 1) наведено логічну модель бази даних:

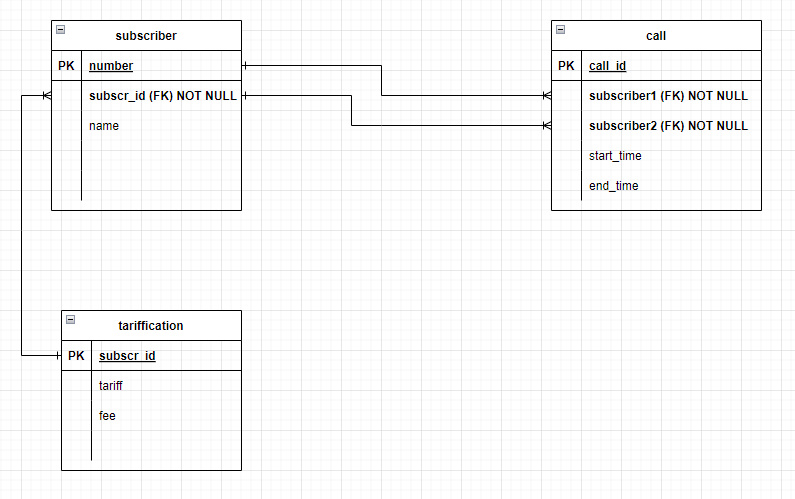


Рисунок 1 – Логічна модель бази даних

**Середовише розробки та налаштування підключення до бази даних**

Застосунок був розроблений мовою програмування Python, середовище розробки – Microsoft Visual Studio 2022. Для розробки програми також була використана стороння бібліотека – psycopg2, яка надає API для доступу до баз даних на основі PostgreSQL

**Посилання на гіт із текстом програми:**

<https://github.com/TwilightstoneZeCryptix/KPI_DBMS.git>

**Структура програми та її опис**

Структура програми зображена на рисунку 2:

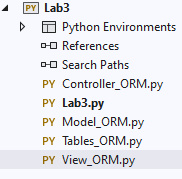


Рисунок 2 – Структура програми

Програма складається з п’яти файлів: Lab3.py, Model\_ORM.py, View\_ORM.py, Controller\_ORM.py, Tables\_ORM.py.

У файлі Model\_ORM.py описаний клас, який реалізує методи управління базою данних.

У файлі View\_ORM.py описаний клас, що надає методи для виведення меню а також подання результатів запитів у консолі.

У файлі Controller\_ORM.py описаний клас, що забезпечує взаємодію з користувачем. У своїй роботі він використовує об’єкти класів що описані в файлах Model\_ORM.py та View\_ORM.py

У файлі Tables\_ORM.py описані класи усіх таблиць із відповідними зв’язками

**Лістинги програм перетворених для ORM**

Лістинг фрагменту програми для внесення даних

**def** **insert\_data**(self, table\_name, values):

s = self.\_\_session()

**if** table\_name == 'tarification':

data = Tarification(fee = values[**0**], tariff = values[**1**])

**elif** table\_name == 'subscriber':

data = Subscriber(number = values[**0**], subscr\_id = [**1**], name = values[**2**])

**elif** table\_name == 'call':

data = Call(subscriber1 = values[**0**], subscriber2 = values[**1**], start\_time = values[**2**], end\_time = values[**3**])

s.add(data)

s.commit()

s.close()

Лістинг фрагменту програми для видалення даних

**def** **delete\_data**(self, table\_name, cond):

s = self.\_\_session()

**if** table\_name == 'tarification':

data = s.query(Tarification).get(cond)

**elif** table\_name == 'subscriber':

data = s.query(Subscriber).get(cond)

**elif** table\_name == 'call':

data = s.query(Call).get(cond)

s.delete(data)

s.commit()

s.close()

Лістинг фрагменту програми для зміни даних

**def** **change\_data**(self, table\_name, new\_values,cond):

s = self.\_\_session()

**if** table\_name == 'tarification':

data = s.query(Tarification).get(cond)

data.fee = new\_values[**0**]

data.tariff = new\_values[**1**]

**elif** table\_name == 'subscriber':

data = s.query(Subscriber).get(cond)

data.number = new\_values[**0**]

data.subscr\_id = new\_values[**1**]

data.name = new\_values[**2**]

**elif** table\_name == 'call':

data = s.query(Call).get(cond)

data.subscriber1 = new\_values[**0**]

data.subscriber2 = new\_values[**1**]

data.start\_time = new\_values[**2**]

data.end\_time = new\_values[**3**]

s.commit()

s.close()

**Текст програми**

**Lab3.py**

**from** **Model\_ORM** **import** db\_model

**from** **View\_ORM** **import** db\_view

**from** **Controller\_ORM** **import** db\_controller

m = db\_model("postgres", "postgres", "anonimus", "127.0.0.1")

v = db\_view()

c = db\_controller(m, v)

**pass**

**Model\_ORM.py**

**import** **sqlalchemy**

**from** **sqlalchemy** **import** create\_engine

**from** **sqlalchemy.orm** **import** sessionmaker

**from** **Tables\_ORM** **import** Tarification, Subscriber, Call

**import** **time**

**import** **re**

**class** **db\_model**():

**def** **\_\_init\_\_**(self, dbname, user\_name, password, host):

bd\_acces\_dir = 'postgresql' + '://' + user\_name + ':' + password + '@' + host + ':5432/' + dbname

**try**:

self.\_\_engine = create\_engine(bd\_acces\_dir)

self.\_\_session = sessionmaker(bind = self.\_\_engine)

**except** **Exception** **as** init\_ex:

print("Initialisation error: ", init\_ex)

**return** **None**

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

**pass**

**def** **get\_table\_data**(self, table\_name):

s = self.\_\_session()

**if** table\_name == 'tarification':

data = s.query(Tarification).all()

**elif** table\_name == 'subscriber':

data = s.query(Subscriber).all()

**elif** table\_name == 'call':

data = s.query(Call).all()

s.close()

**return** data

**def** **get\_table\_list**(self):

**return** ['tarification', 'subscriber', 'call']

**def** **get\_column\_names**(self, table\_name):

**if** table\_name == 'tarification':

**return** ['fee', 'tariff']

**elif** table\_name == 'subscriber':

**return** ['number', 'subscr\_id', 'name']

**elif** table\_name == 'call':

**return** ['subscriber1', 'subscriber2', 'start\_time', 'end\_time']

**def** **insert\_data**(self, table\_name, values):

s = self.\_\_session()

**if** table\_name == 'tarification':

data = Tarification(fee = values[**0**], tariff = values[**1**])

**elif** table\_name == 'subscriber':

data = Subscriber(number = values[**0**], subscr\_id = [**1**], name = values[**2**])

**elif** table\_name == 'call':

data = Call(subscriber1 = values[**0**], subscriber2 = values[**1**], start\_time = values[**2**], end\_time = values[**3**])

s.add(data)

s.commit()

s.close()

**def** **change\_data**(self, table\_name, new\_values, cond):

s = self.\_\_session()

**if** table\_name == 'tarification':

data = s.query(Tarification).get(cond)

data.fee = new\_values[**0**]

data.tariff = new\_values[**1**]

**elif** table\_name == 'subscriber':

data = s.query(Subscriber).get(cond)

data.number = new\_values[**0**]

data.subscr\_id = new\_values[**1**]

data.name = new\_values[**2**]

**elif** table\_name == 'call':

data = s.query(Call).get(cond)

data.subscriber1 = new\_values[**0**]

data.subscriber2 = new\_values[**1**]

data.start\_time = new\_values[**2**]

data.end\_time = new\_values[**3**]

s.commit()

s.close()

**def** **delete\_data**(self, table\_name, cond):

s = self.\_\_session()

**if** table\_name == 'tarification':

data = s.query(Tarification).get(cond)

**elif** table\_name == 'subscriber':

data = s.query(Subscriber).get(cond)

**elif** table\_name == 'call':

data = s.query(Call).get(cond)

s.delete(data)

s.commit()

s.close()

**View\_ORM.py**

**class** **db\_view**():

**def** **starting\_dialog**(self):

print("""Welcome to phone node database application.

To choose options from program menus, enter it's number which is written in front of it""")

**def** **main\_menu**(self):

print("""MAIN MENU

1. Data operations

2. Quit""")

**def** **operations\_menu**(self):

print("""OPERATIONS

1. Insert data

2. Delete data

3. Modify data""")

**def** **table\_menu**(self, tables):

print("""Choose table:""")

**for** i **in** range(len(tables)):

print(f"{i+1}.", tables[i])

**def** **show\_rows**(self, table\_name, rows):

**if** table\_name == 'tarification':

i = **1**

**for** r **in** rows:

print(str(i) + '. ' + str(r.subscr\_id) + ' ' + str(r.fee) + ' ' + r.tariff)

i = i + **1**

**elif** table\_name == 'subscriber':

i = **1**

**for** r **in** rows:

print(str(i) + '. ' + r.number + ' ' + str(r.subscr\_id) + ' ' + r.name)

i = i + **1**

**elif** table\_name == 'call':

i = **1**

**for** r **in** rows:

print(str(i) + '. ' + str(r.call\_id) + ' ' + r.subscriber1 + ' ' + str(r.subscriber2) + ' ' + str(r.start\_time) + ' ' + str(r.end\_time))

i = i + **1**

**Controller\_ORM.py**

**import** **re**

**class** **db\_controller**():

**def** **\_\_init\_\_**(self, model, view):

self.\_\_model = model

self.\_\_view = view

self.main\_process()

**def** **insert**(self):

table\_names = self.\_\_model.get\_table\_list()

self.\_\_view.table\_menu(table\_names)

table = int(input())

table = table\_names[table - **1**]

columns = self.\_\_model.get\_column\_names(table)

values = []

**for** i **in** range(len(columns)):

print("Enter " + columns[i] + " value:")

values.append(input())

self.\_\_model.insert\_data(table, tuple(values))

**def** **delete**(self):

table\_names = self.\_\_model.get\_table\_list()

self.\_\_view.table\_menu(table\_names)

table = int(input())

table = table\_names[table - **1**]

table\_data = self.\_\_model.get\_table\_data(table)

self.\_\_view.show\_rows(table, table\_data)

prime\_key = input('Enter primary key value (first value that occurs after row number): ')

self.\_\_model.delete\_data(table, prime\_key)

**def** **modify**(self):

table\_names = self.\_\_model.get\_table\_list()

self.\_\_view.table\_menu(table\_names)

table = int(input())

table = table\_names[table - **1**]

table\_data = self.\_\_model.get\_table\_data(table)

self.\_\_view.show\_rows(table, table\_data)

prime\_key = input('Enter primary key value (first value that occurs after row number): ')

columns = self.\_\_model.get\_column\_names(table)

values = []

**for** i **in** range(len(columns)):

print("Enter new " + columns[i] + " value:")

values.append(input())

self.\_\_model.change\_data(table, tuple(values), prime\_key)

**def** **main\_process**(self):

self.\_\_view.starting\_dialog()

**while** **1**:

self.\_\_view.main\_menu()

action = int(input())

**if** action == **2**:

**break**

**elif** action == **1**:

self.\_\_view.operations\_menu()

operation = int(input())

**if** operation == **1**:

self.insert()

**elif** operation == **2**:

self.delete()

**elif** operation == **3**:

self.modify()

**Tables\_ORM.py**

**from** **sqlalchemy** **import** Column, Integer, Text, Time, ForeignKey

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** relationship, backref

Base = declarative\_base()

**class** **Tarification**(Base):

\_\_tablename\_\_ = 'tarification'

subscr\_id = Column(Integer, primary\_key = **True**)

fee = Column(Integer)

tariff = Column(Text)

**class** **Subscriber**(Base):

\_\_tablename\_\_ = 'subscriber'

number = Column(Text, primary\_key = **True**)

subscr\_id = Column(Integer, ForeignKey('tarification.subscr\_id'))

name = Column(Text)

tarification = relationship("Tarification", backref = backref('tarification'))

**class** **Call**(Base):

\_\_tablename\_\_ = 'call'

call\_id = Column(Integer, primary\_key = **True**)

subscriber1 = Column(Text, ForeignKey('subscriber.number'))

subscriber2 = Column(Integer, ForeignKey('subscriber.subscr\_id'))

start\_time = Column(Time)

end\_time = Column(Time)

subscr\_id = relationship("Subscriber", foreign\_keys = [subscriber1])

number = relationship("Subscriber", foreign\_keys = [subscriber2])

**Приклад запитів ORM:**

s = Session()

#Вставка елементу

tarification = Tarification(fee = '75', tariff = 'Standart')

s.add(tarification)

#Видалення знайденого елементу

tarification = s.query(Tarification).get('15')#пошук по id

s.delete(tarification)

#Зміна елементу

tarification = s.query(Tarification).get('13')

tarification.tariff = 'Standart+'

s.commit()

s.close()

**Аналіз індексів**

Для аналізу заданих за варіантом індексів було взято два пошукових запити. Обидва запити використовують таблиці tarification та subscriber, в яких заздалегідь було створено близько 100000 випадкових записів. Перший запит та результат його виконання зображено на рисунку 3, а другий запит, разом з його результатом зображено на рисунку 4:

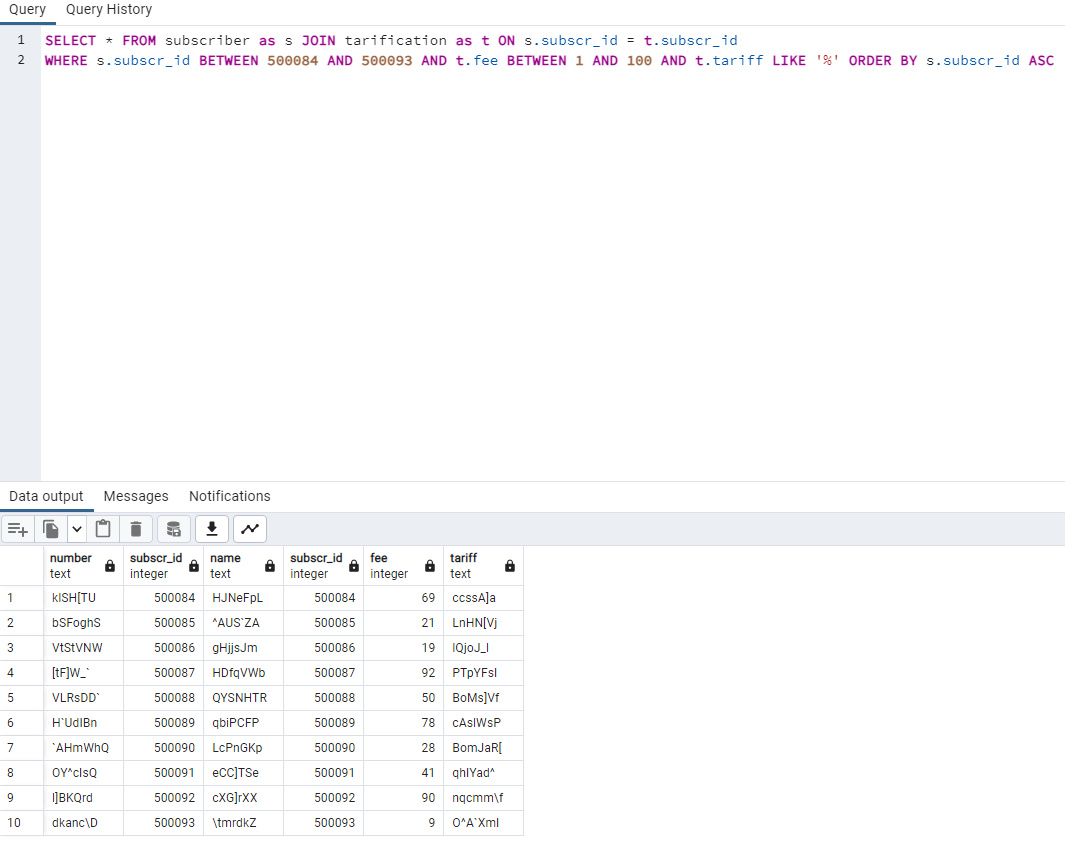


Рисунок 3 – Запит №1



Рисунок 4 – Запит №2

Як видно зі структури полів WHERE та результатів виконання, перший запит, окрім меншого діапазону значень для поля subscr\_id таблиці subscriber, в операторі LIKE містить шаблон що відповідає будь-якому набору символів котрий може з’являтися в полі tariff таблиці tariffication, в той час як шаблон LIKE в другому запиті відповідає рядкам, що починаються з ‘i’ та закінчуються ‘f’, тобто присутній пошук тексту що відповідає конкретному шаблону. В зв’язку з цим, перший запит було вибрано для дослідження індексу B-tree, а другий – індексу GIN. Нижче (рисунки 5 та 6) наведено запити на створення відповідних індексів:



Рисунок 5 – Створення індексу B-tree



Рисунок 6 – Створення індексу GIN

Для дослідження індексу B-tree, його було застосовано до атрибуту subscr\_id в таблиці subscriber. Індекс GIN з триграмним пошуком було застосовано до атрибуту tariff в таблиці tariffication.

Далі наведено результати виконання директиви EXPLAIN ANALYZE для кожного з запитів з використання обраних індексів та без (рисунки 7-10):

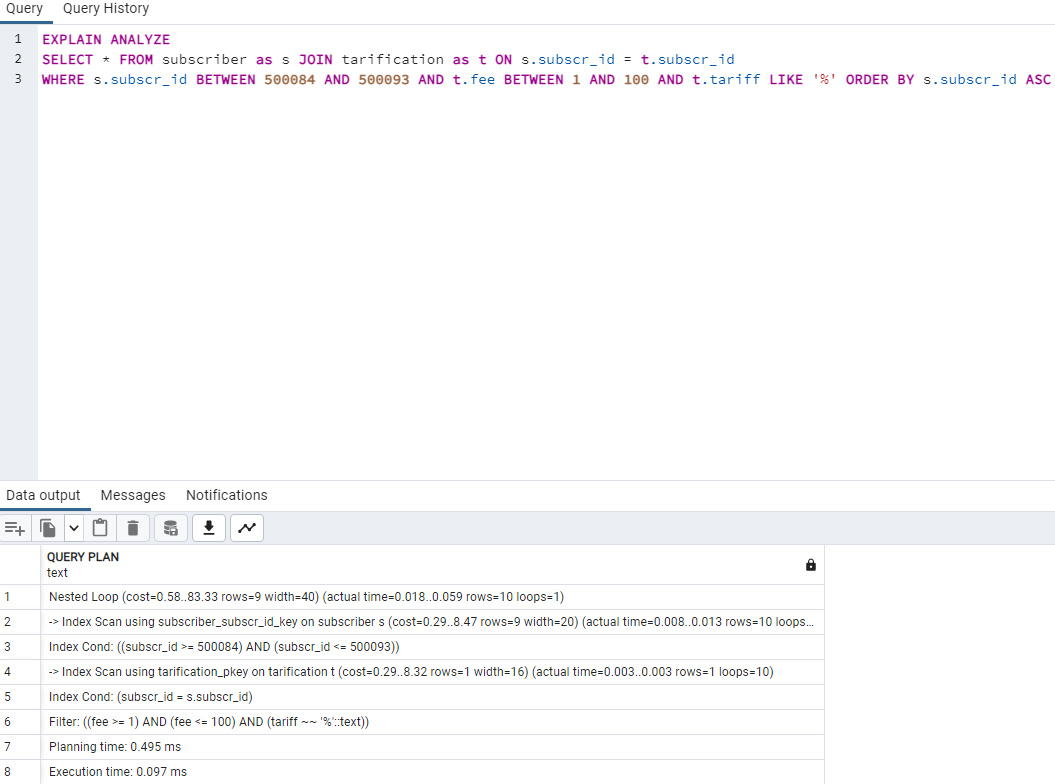


Рисунок 7 – План виконання запиту №1

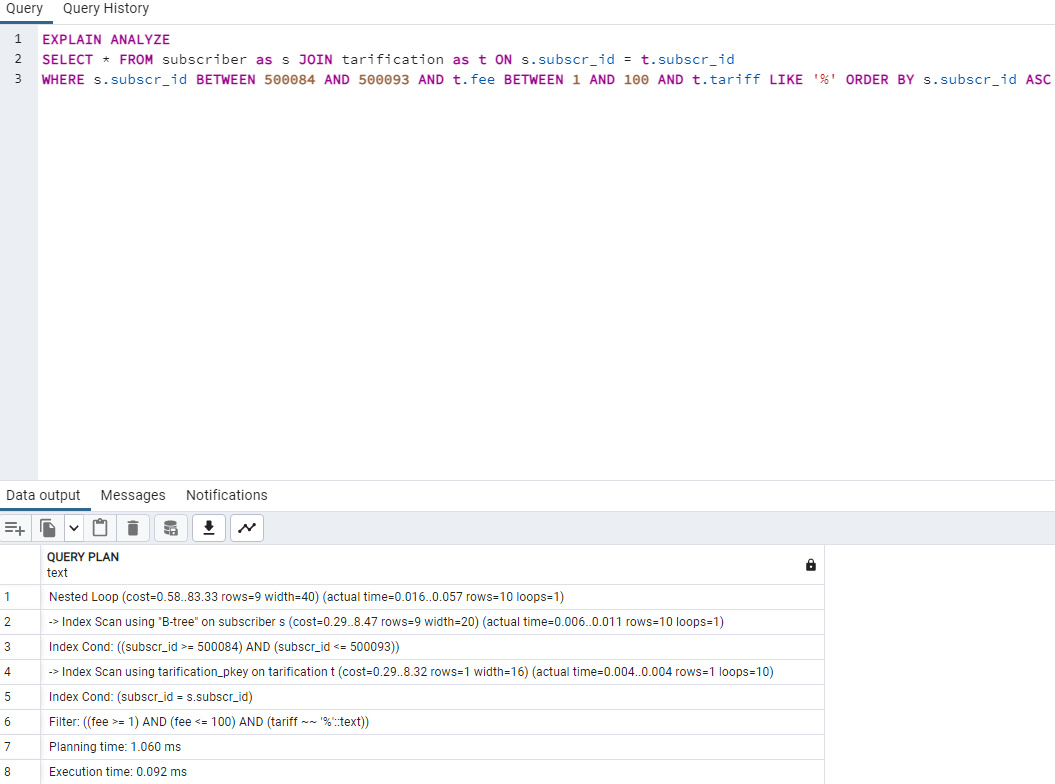


Рисунок 8 – План виконання запиту №1 з використанням індексу B-tree

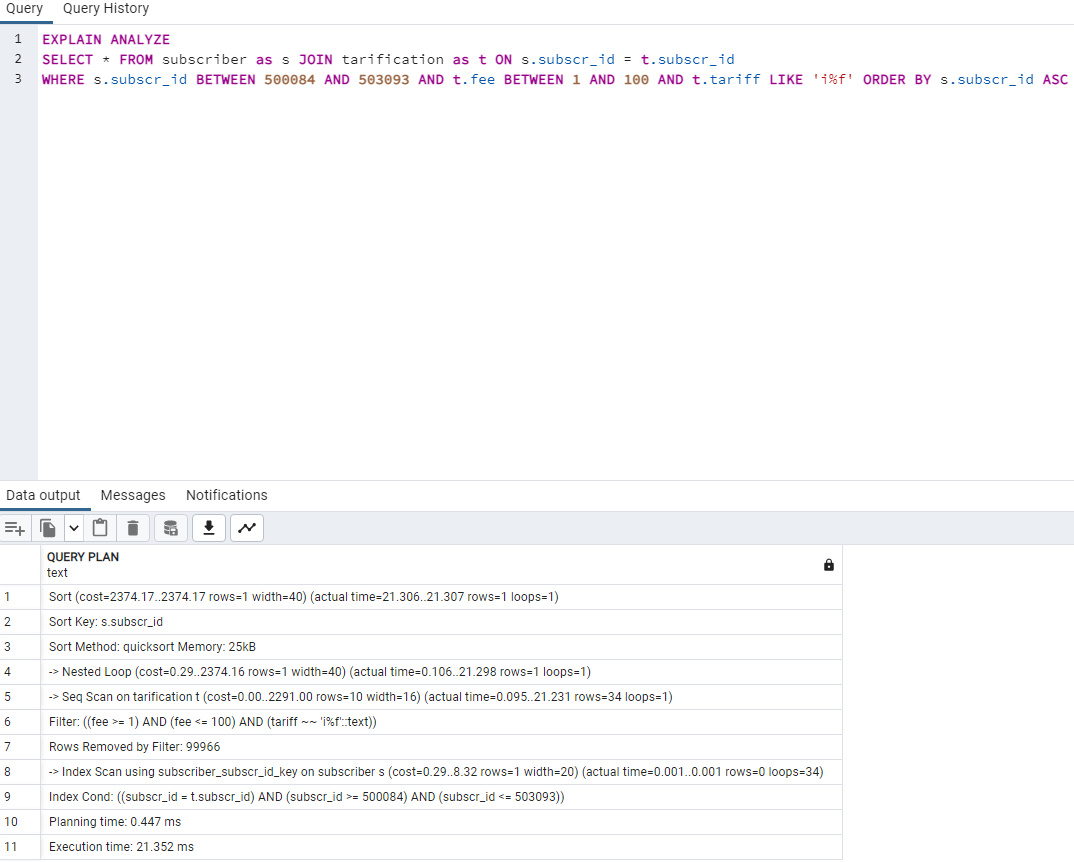


Рисунок 9 – План виконання запиту №2

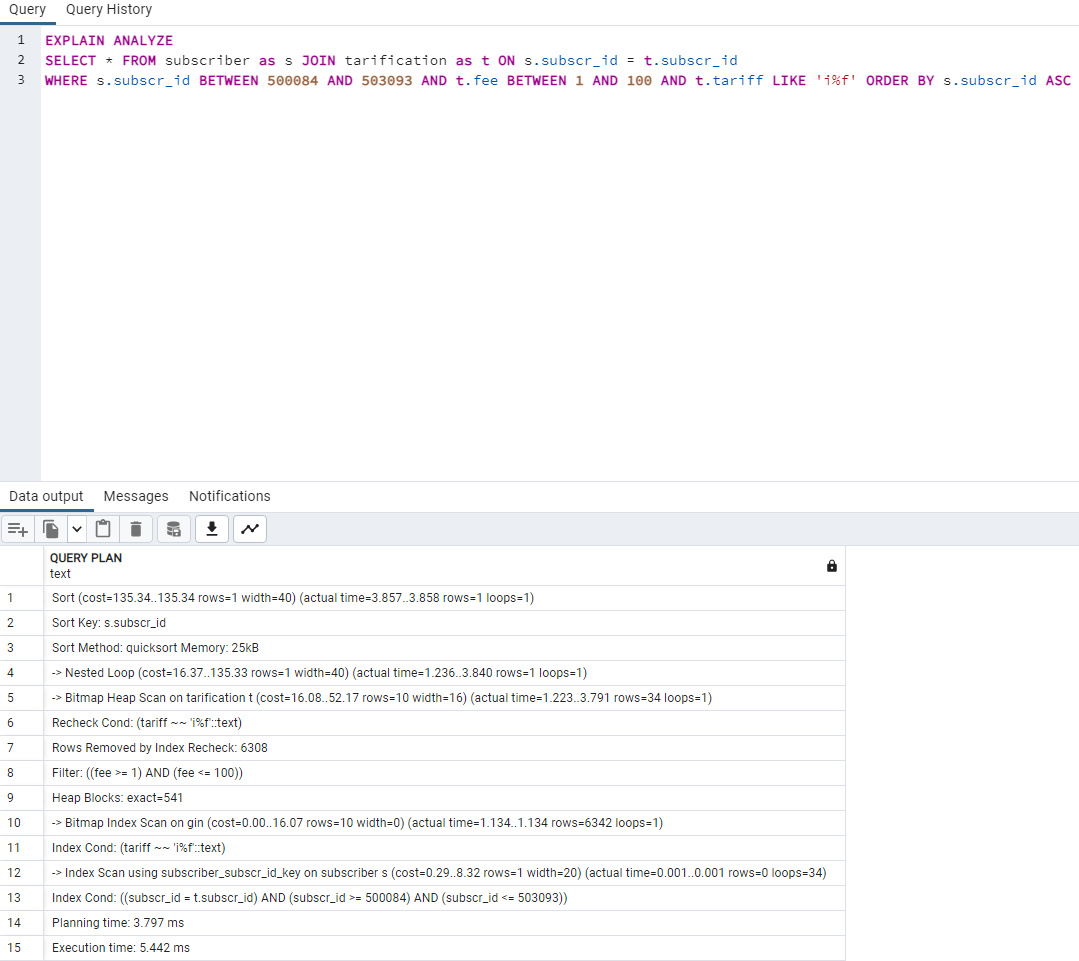


Рисунок 10 – План виконання запиту №2 з використанням індексу GIN з триграмним пошуком

Як видно з результатів, перший запит був не значно прискорений індексом B-tree, проте варто звернути увагу на перші два рядки в плані виконання запиту як з індексом так без індексу. При виконанні запиту з індексом, маємо хоч і незначне але зменшення показника actual time. Особливістю індексу є використання деревовидної структури данних, кожна гілка якої має однакову довжину, а самі розгалужені знаходяться на певних рівнях. На останньому рівні знаходяться власне записи в таблиці. Тому, в даному випадку, це може впливати на незначне, але пришвидшення швидкості пошуку.

Другий запит був значно прискорений індексом GIN, що виражено у разючій відмінності показника execution time (час виконання) до та після використання індексу. Це можна пояснити тим, що цей індекс ідеально пристосований для роботи зі складеними типами данних, на відміну від B-tree. У нашому випадку, було застосовано його різновидність котра гарно підходить для роботи з текстовими рядками – GIN з триграмним пошуком. Враховуючи тип атрибуту, до якого його було застосовано (text), значне покращення ефективності пошуку є очікуваним результатом.

Загалом, індекси, при правильному використанні, дозволяють покращити швидкість вибірки данних оператором SELECT, проте якщо починають часто використовуватись оператори INSERT, UPDATE та DELETE, структура таблиці може зазнати змін, зокрема може статися перестановка або розділення сторінки. В такому випадку, індексу може знадобитися реорганізація, що стосується всіх пов’язаних операцій та індексів, що не найкращим чином вплине на продуктивність