МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра №806 «Вычислительная математика и программирование»

**Лабораторная работа 1**

**по курсу «Базы Данных»**

Выполнил: Корнев М. С.

Группа: М8О-312Б-22

Преподаватель: А. В. Малахов

Москва, 2025

Оглавление

[1.0 Данные 2](#_Toc196860266)

[1.1 Индексы 7](#_Toc196860267)

[1.1.1 BTREE 7](#_Toc196860268)

[1.1.2 GIN 8](#_Toc196860269)

[1.1.3 BRIN 10](#_Toc196860270)

[1.2 Транзакции 11](#_Toc196860271)

[1.3 Использование расширений PostgreSQL для полнотекстового поиска и криптографических операций 15](#_Toc196860272)

[Вывод 18](#_Toc196860273)

# **1.0 Данные**

Датасет найти не получилось, поэтому сгенерировал сам при помощи следующего скрипта:

# Генерация данных

import psycopg2

import uuid

from faker import Faker

import random

from tqdm import tqdm

from dotenv import load\_dotenv

import os

# Загружаем переменные из .env

load\_dotenv()

# Читаем из .env

DB\_HOST = os.getenv('DB\_HOST')

DB\_PORT = os.getenv('DB\_PORT')

DB\_NAME = os.getenv('DB\_NAME')

DB\_USER = os.getenv('DB\_USER')

DB\_PASSWORD = os.getenv('DB\_PASSWORD')

SECRET\_KEY = os.getenv('SECRET\_KEY')

fake = Faker()

# Настройки подключения к базе

conn = psycopg2.connect(

dbname=DB\_NAME,

user=DB\_USER,

password=DB\_PASSWORD,

host=DB\_HOST,

port=DB\_PORT

)

conn.autocommit = True

cur = conn.cursor()

# Число записей

NUM\_USERS = 500\_000

NUM\_CATEGORIES = 50

NUM\_PRODUCTS = 500\_000

NUM\_ORDERS = 1\_000\_000

NUM\_REVIEWS = 500\_000

NUM\_SUPPORT\_SESSIONS = 100\_000

NUM\_MESSAGES\_PER\_SESSION = 5

def save\_secret\_key():

# Для примера сохраним SECRET\_KEY в таблицу настроек приложения

cur.execute("""

CREATE TABLE IF NOT EXISTS app\_settings (

key VARCHAR PRIMARY KEY,

value VARCHAR NOT NULL

)

""")

cur.execute("""

INSERT INTO app\_settings (key, value) VALUES ('SECRET\_KEY', %s)

ON CONFLICT (key) DO UPDATE SET value = EXCLUDED.value

""", (SECRET\_KEY,))

def create\_roles():

# Сначала создаем уникальное ограничение, если его нет

cur.execute("""

ALTER TABLE roles ADD CONSTRAINT roles\_name\_unique UNIQUE (name)

""")

# Затем выполняем вставку с ON CONFLICT

cur.execute("""

INSERT INTO roles (name) VALUES ('User'), ('Admin')

ON CONFLICT (name) DO NOTHING

""")

def create\_categories():

categories = [(str(uuid.uuid4()), fake.word()) for \_ in range(NUM\_CATEGORIES)]

cur.executemany("""

INSERT INTO categories (category\_id, name)

VALUES (%s, %s)

ON CONFLICT (category\_id) DO NOTHING

""", categories)

return [c[0] for c in categories]

def create\_users():

users = [(str(uuid.uuid4()), fake.user\_name(), fake.password(), fake.unique.email()) for \_ in range(NUM\_USERS)]

cur.executemany("""

INSERT INTO users (user\_id, username, hashed\_password, email)

VALUES (%s, %s, %s, %s)

ON CONFLICT (email) DO NOTHING

""", users)

return [u[0] for u in users]

def assign\_user\_roles(user\_ids):

roles = [1, 2]

user\_roles = [(uid, random.choice(roles)) for uid in user\_ids]

cur.executemany("""

INSERT INTO user\_roles (user\_id, role\_id)

VALUES (%s, %s)

ON CONFLICT (user\_id, role\_id) DO NOTHING

""", user\_roles)

def create\_products(category\_ids):

products = []

for \_ in range(NUM\_PRODUCTS):

products.append((

str(uuid.uuid4()),

fake.catch\_phrase(),

fake.text(max\_nb\_chars=200),

random.choice(category\_ids),

round(random.uniform(5.0, 500.0), 2),

random.randint(10, 1000),

fake.company()

))

cur.executemany("""

INSERT INTO products (product\_id, name, description, category\_id, price, stock, manufacturer)

VALUES (%s, %s, %s, %s, %s, %s, %s)

ON CONFLICT (product\_id) DO NOTHING

""", products)

return [p[0] for p in products]

def create\_orders(user\_ids):

orders = []

for \_ in range(NUM\_ORDERS):

orders.append((

str(uuid.uuid4()),

random.choice(user\_ids),

fake.date\_time\_this\_decade(),

random.choice(['Pending', 'Completed', 'Cancelled']),

round(random.uniform(20.0, 2000.0), 2)

))

cur.executemany("""

INSERT INTO orders (order\_id, user\_id, order\_date, status, total\_cost)

VALUES (%s, %s, %s, %s, %s)

ON CONFLICT (order\_id) DO NOTHING

""", orders)

return [o[0] for o in orders]

def create\_order\_items(order\_ids, product\_ids):

order\_items = []

for order\_id in tqdm(order\_ids, desc="Inserting order\_items"):

products\_in\_order = set() # Множество для хранения уникальных product\_id

for \_ in range(random.randint(1, 5)): # Случайное количество товаров в заказе

product\_id = random.choice(product\_ids)

while product\_id in products\_in\_order: # Проверяем, чтобы товар не повторялся

product\_id = random.choice(product\_ids)

products\_in\_order.add(product\_id) # Добавляем товар в заказ

quantity = random.randint(1, 10)

price = round(random.uniform(5.0, 500.0), 2)

order\_items.append((order\_id, product\_id, quantity, price))

cur.executemany("""

INSERT INTO order\_items (order\_id, product\_id, quantity, price)

VALUES (%s, %s, %s, %s)

ON CONFLICT (order\_id, product\_id) DO NOTHING

""", order\_items)

def create\_reviews(user\_ids, product\_ids):

reviews = []

for \_ in tqdm(range(NUM\_REVIEWS), desc="Inserting reviews"):

reviews.append((

str(uuid.uuid4()),

random.choice(product\_ids),

random.choice(user\_ids),

random.randint(1, 5),

fake.sentence(),

fake.date\_time\_this\_year()

))

cur.executemany("""

INSERT INTO reviews (review\_id, product\_id, user\_id, rating, comment, review\_date)

VALUES (%s, %s, %s, %s, %s, %s)

ON CONFLICT (review\_id) DO NOTHING

""", reviews)

def create\_support\_sessions(user\_ids):

sessions = []

for \_ in range(NUM\_SUPPORT\_SESSIONS):

start\_time = fake.date\_time\_this\_year()

end\_time = None if random.random() < 0.3 else fake.date\_time\_between(start\_date=start\_time)

sessions.append((

str(uuid.uuid4()),

random.choice(user\_ids),

start\_time,

end\_time,

random.choice(['open', 'closed'])

))

cur.executemany("""

INSERT INTO support\_sessions (session\_id, user\_id, start\_time, end\_time, status)

VALUES (%s, %s, %s, %s, %s)

ON CONFLICT (session\_id) DO NOTHING

""", sessions)

return [s[0] for s in sessions]

def create\_support\_messages(session\_ids):

messages = []

for session\_id in tqdm(session\_ids, desc="Inserting support\_messages"):

for \_ in range(NUM\_MESSAGES\_PER\_SESSION):

messages.append((

str(uuid.uuid4()),

session\_id,

random.choice(['user', 'support']),

fake.sentence(),

fake.date\_time\_this\_year()

))

cur.executemany("""

INSERT INTO support\_messages (message\_id, session\_id, sender, message, sent\_at)

VALUES (%s, %s, %s, %s, %s)

ON CONFLICT (message\_id) DO NOTHING

""", messages)

def main():

print("Starting data generation...")

save\_secret\_key()

create\_roles()

category\_ids = create\_categories()

user\_ids = create\_users()

assign\_user\_roles(user\_ids)

product\_ids = create\_products(category\_ids)

order\_ids = create\_orders(user\_ids)

create\_order\_items(order\_ids, product\_ids)

create\_reviews(user\_ids, product\_ids)

session\_ids = create\_support\_sessions(user\_ids)

create\_support\_messages(session\_ids)

print("Data generation completed!")

if \_\_name\_\_ == "\_\_main\_\_":

main()

По итогу вышла база данных около 7 млн строк

# **1.1 Индексы**

## 1.1.1 BTREE

Индекс BTREE эффективен для широкого набора задач и может быть использован практически для любых столбцов где реализовано сравнение.

Запрос без индекса и с индексом:

CREATE INDEX idx\_users\_email ON users USING btree(email);

CREATE INDEX idx\_products\_category\_id ON products USING btree(category\_id);

EXPLAIN ANALYZE

SELECT \*

FROM users

WHERE email = 'test@example.com';

EXPLAIN ANALYZE

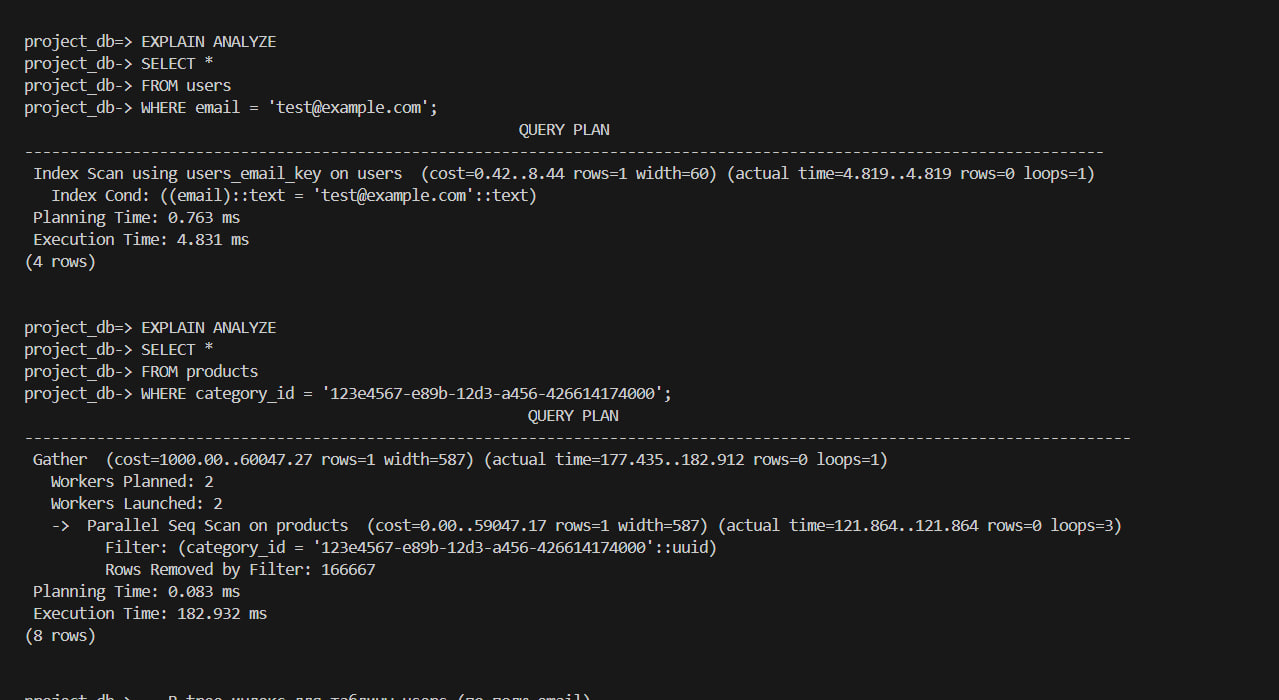
SELECT \*

FROM products

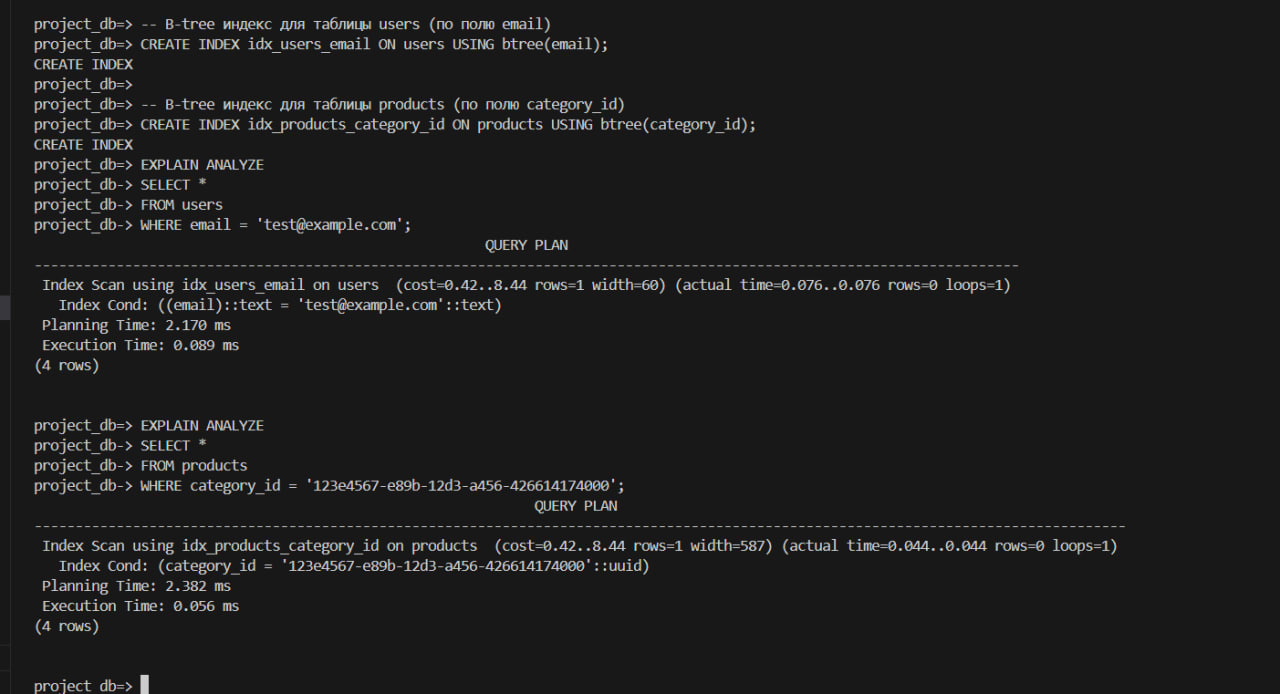
WHERE category\_id = '123e4567-e89b-12d3-a456-426614174000';

Сравним время выполнения:

До индексов:



После индексации:



Запросы стали намного быстрее.

## 1.1.2 GIN

Данный индекс как правило используется для поиска в массивах, больших текстах, а также полнотекстового поиска.

Запрос:

CREATE INDEX idx\_products\_description\_gin ON products USING gin(to\_tsvector('english', description));

CREATE INDEX idx\_reviews\_comment\_gin ON reviews USING gin(to\_tsvector('english', comment));

-- Запрос

EXPLAIN ANALYZE

SELECT \* FROM products

WHERE to\_tsvector('english', description) @@ to\_tsquery('laptop');

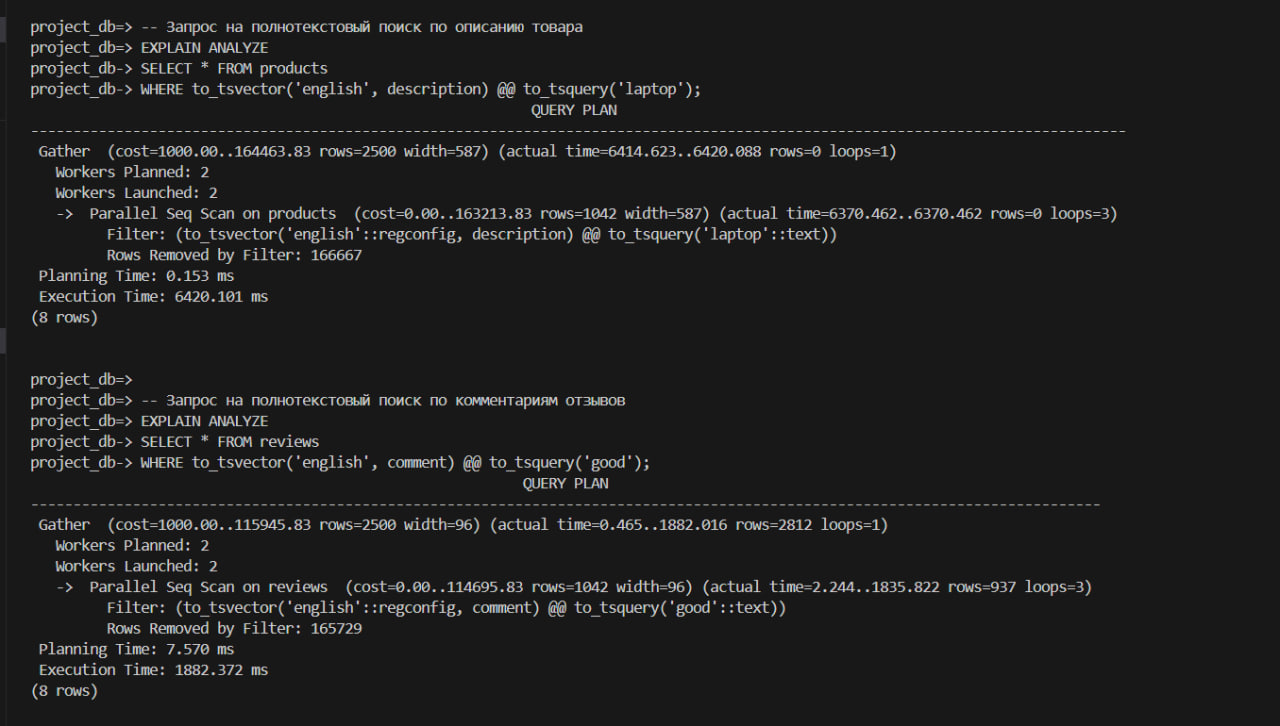
-- Запрос

EXPLAIN ANALYZE

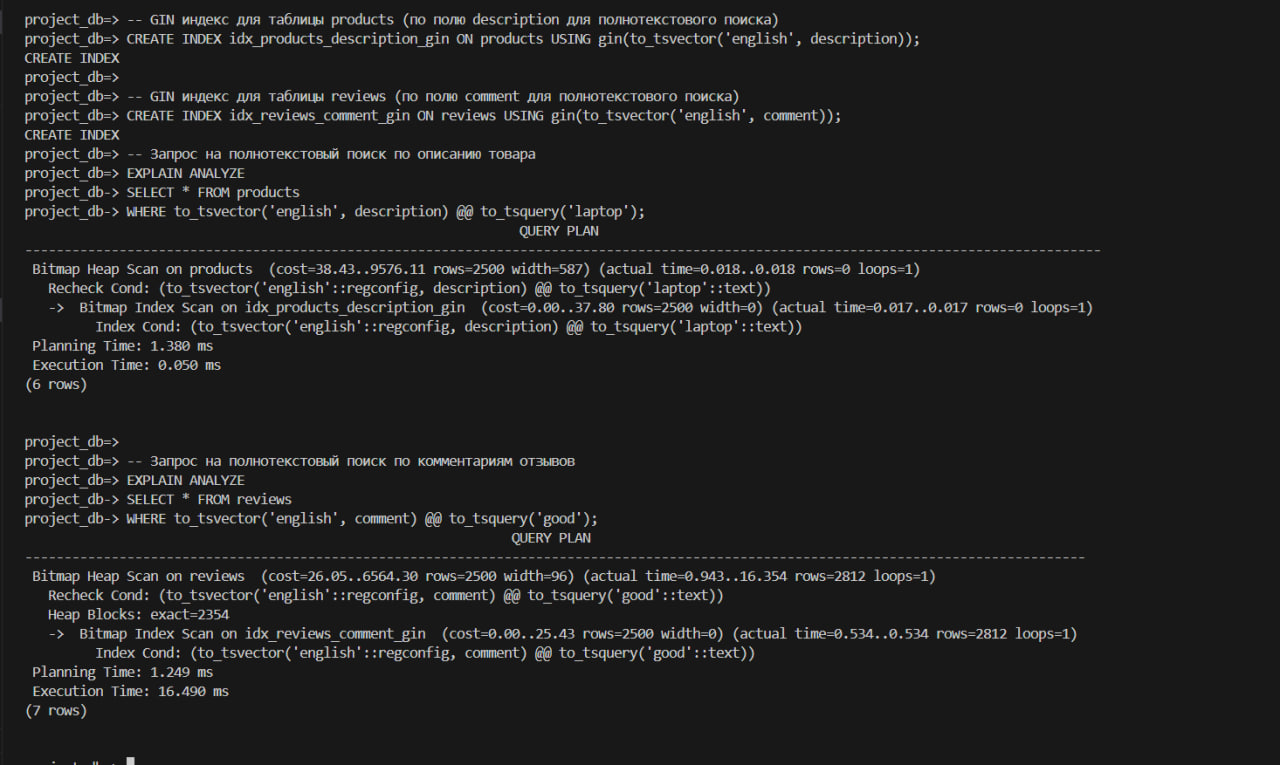
SELECT \* FROM reviews

WHERE to\_tsvector('english', comment) @@ to\_tsquery('good');

До индекса:



После индекса:



## 1.1.3 BRIN

Индекс эффективен для отсортированных величин, например в случае когда данные заносятся в БД с указыванием их времени добавления.

CREATE INDEX idx\_orders\_order\_date\_brin ON orders USING brin(order\_date) WITH (pages\_per\_range = 64);

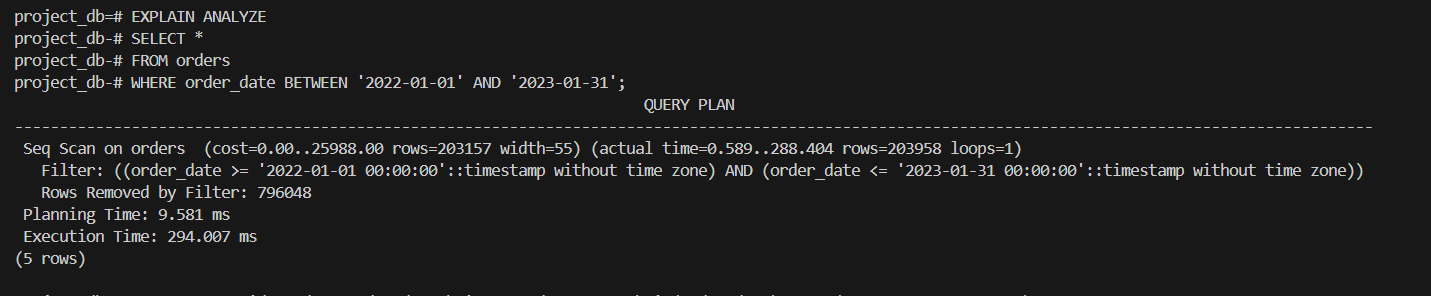
-- Запрос

EXPLAIN ANALYZE

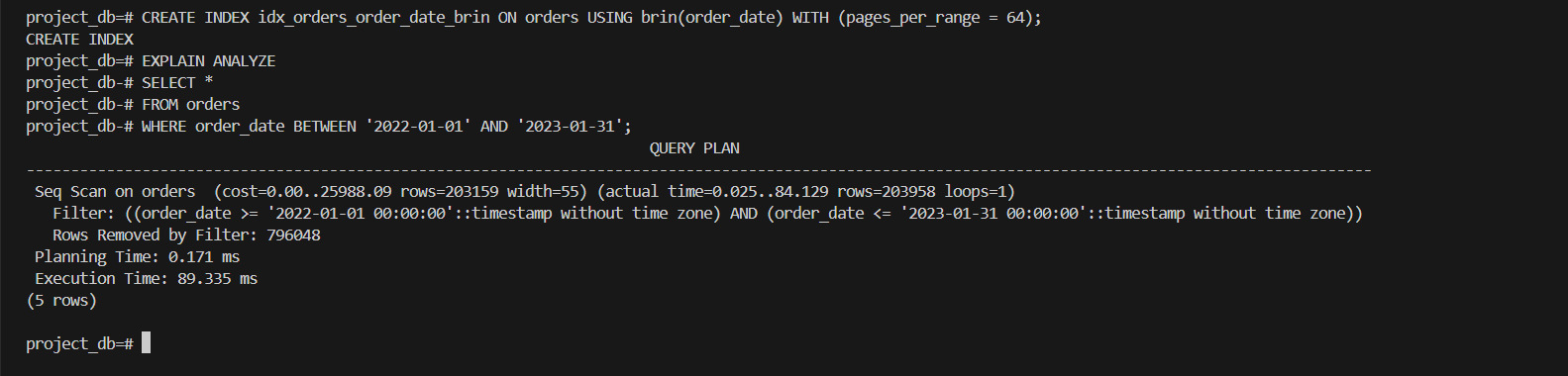
SELECT \*

FROM orders

WHERE order\_date BETWEEN '2022-01-01' AND '2023-01-31';

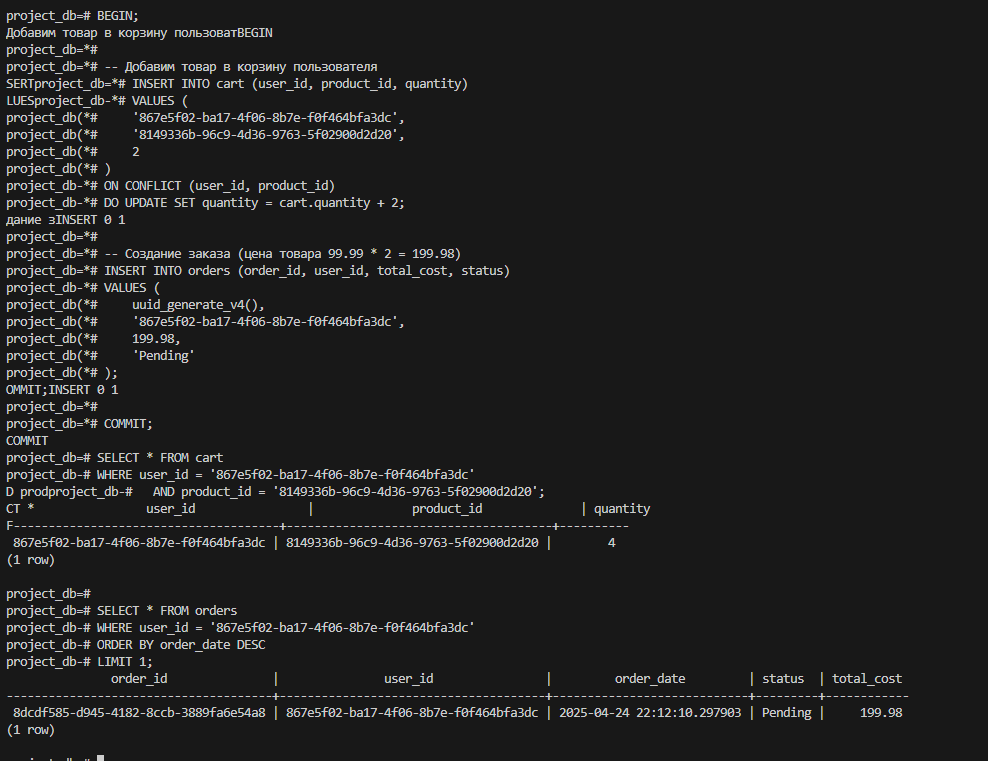
До индекса:  


После

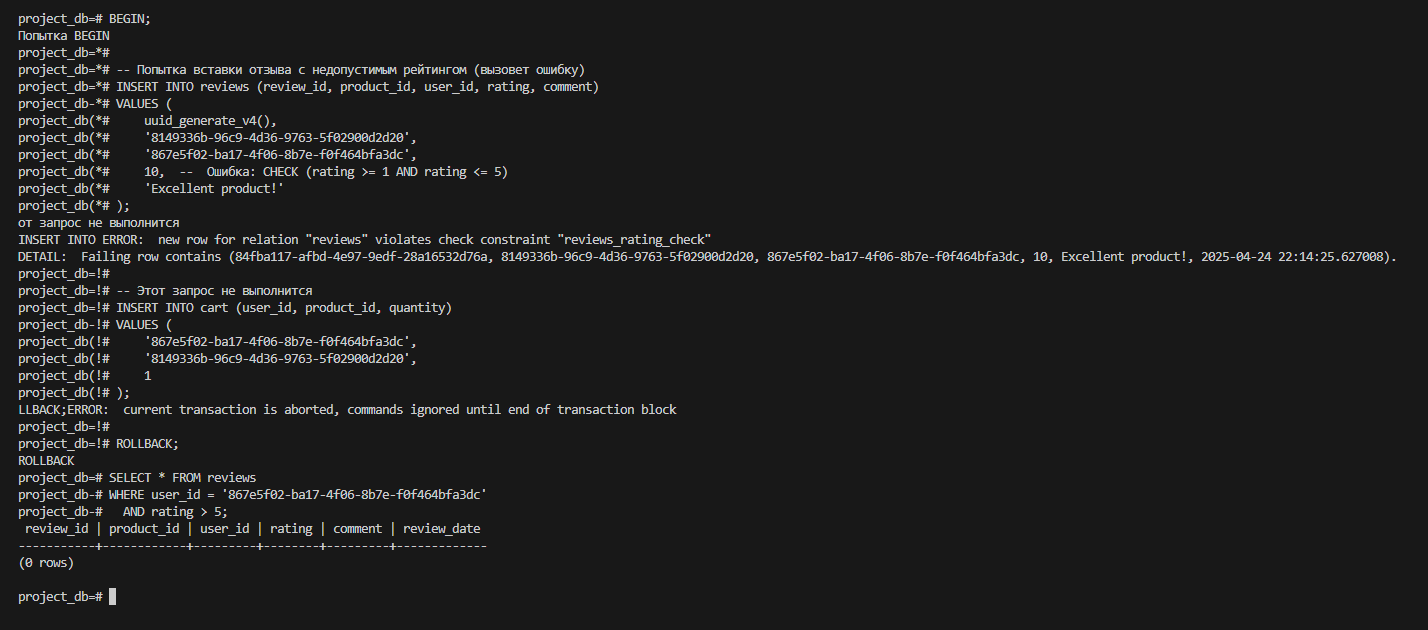


# **1.2 Транзакции**

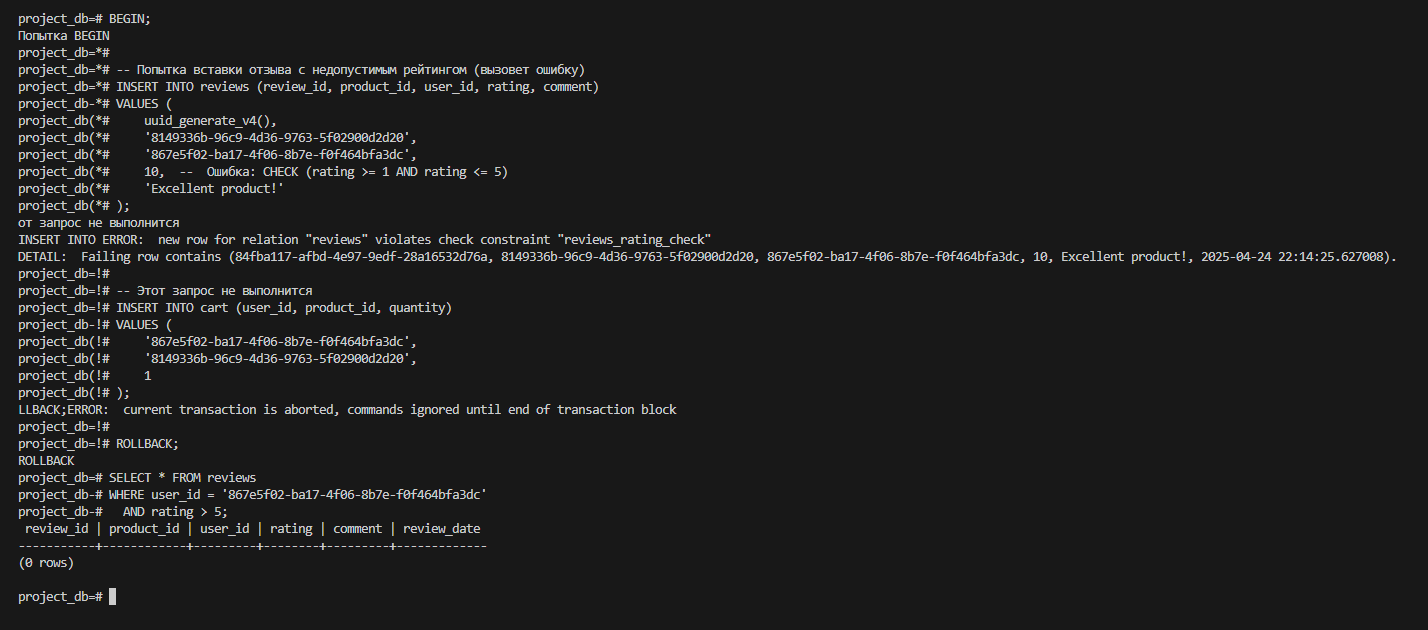
Первая транзакция:



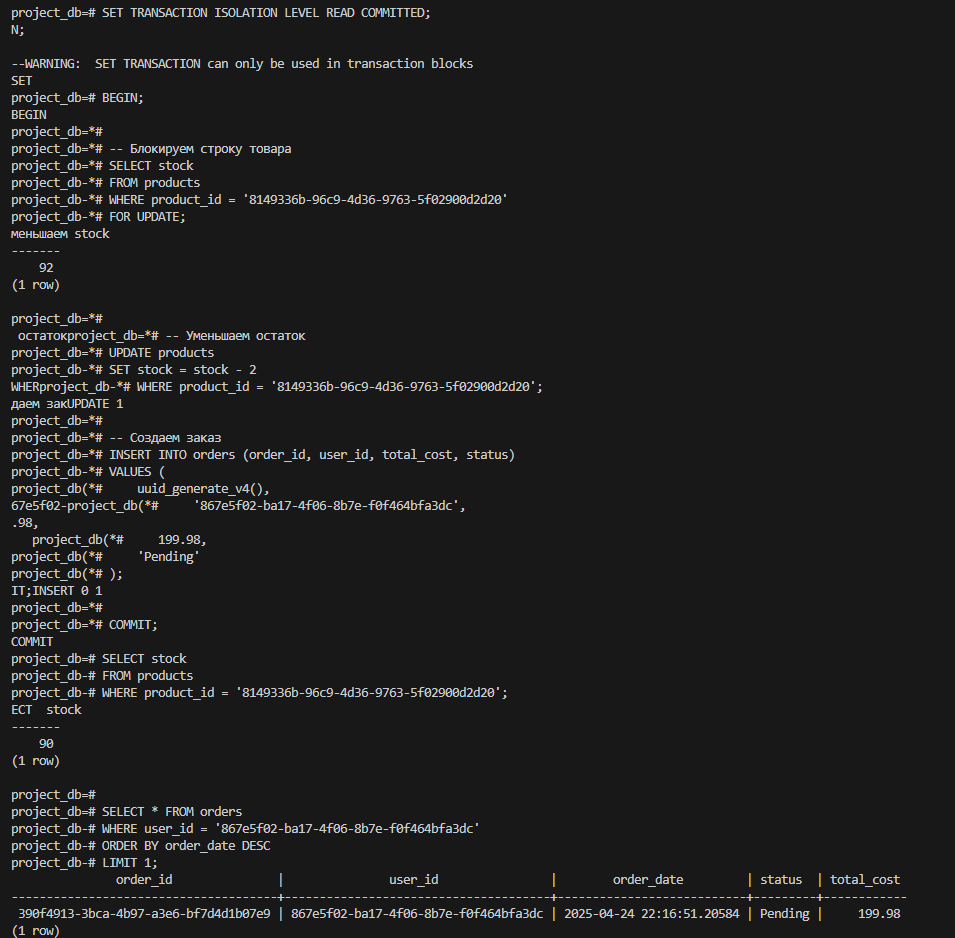
**Вторая транзакция:**



**Третья транзакция:**

****

**Четвертая транзакция:**

****

**Аномалии**

Грязное чтение (Dirty Read)

Грязное чтение возникает, когда одна транзакция читает данные, измененные другой незавершенной транзакцией. Эти данные могут быть позже откатаны, что приведет к несогласованности.

-- Сессия 1

BEGIN;

UPDATE products

SET price = 3000

WHERE product\_id = '8149336b-96c9-4d36-9763-5f02900d2d20';

-- Сессия 2

BEGIN ISOLATION LEVEL READ COMMITTED;

SELECT price

FROM products

WHERE product\_id = '8149336b-96c9-4d36-9763-5f02900d2d20';

-- Здесь может быть видно старое или измененное значение

COMMIT;

-- Возврат к сессии 1

ROLLBACK;

Вторая сессия может считать промежуточное (грязное) значение, которое позже будет отменено в первой сессии.

Неповторяемое чтение (Non-repeatable Read)

Неповторяемое чтение возникает, когда при повторном считывании тех же данных в рамках одной транзакции они изменяются другой транзакцией.

-- Сессия 1

BEGIN ISOLATION LEVEL READ COMMITTED;

SELECT price

FROM products

WHERE product\_id = '8149336b-96c9-4d36-9763-5f02900d2d20';

-- Цена: 3000.00

-- Сессия 2

BEGIN;

UPDATE products

SET price = 30000

WHERE product\_id = '8149336b-96c9-4d36-9763-5f02900d2d20';

COMMIT;

-- Сессия 1

SELECT price

FROM products

WHERE product\_id = '8149336b-96c9-4d36-9763-5f02900d2d20';

-- Цена: 30000.00

COMMIT;

Данные, считанные первой транзакцией, изменились между двумя запросами в рамках одной транзакции.

Фантомное чтение (Phantom Read)

Фантомное чтение возникает, когда после выполнения запроса выборки данных в той же транзакции другая транзакция добавляет новые строки, удовлетворяющие условиям первоначального запроса.

-- Сессия 1

INSERT INTO categories (category\_id, name)

VALUES ('00000000-0000-0000-0000-000000000099', 'Electronics')

ON CONFLICT DO NOTHING;

BEGIN ISOLATION LEVEL READ COMMITTED;

SELECT COUNT(\*)

FROM products p

JOIN categories c ON p.category\_id = c.category\_id

WHERE c.name = 'Electronics';

-- Количество товаров: 1

-- Сессия 2

BEGIN;

INSERT INTO products (product\_id, name, category\_id, price, stock, manufacturer)

VALUES (

uuid\_generate\_v4(),

'Headphones',

'00000000-0000-0000-0000-000000000099',

149.99,

20,

'Sony'

);

COMMIT;

-- Сессия 1

SELECT COUNT(\*)

FROM products p

JOIN categories c ON p.category\_id = c.category\_id

WHERE c.name = 'Electronics';

-- Количество товаров: 2

COMMIT;

Во время выполнения первой транзакции количество строк, удовлетворяющих запросу, изменилось за счёт действий другой транзакции.

# **1.3 Использование расширений PostgreSQL для полнотекстового поиска и криптографических операций**

Использование расширения pgcrypto в PostgreSQL для шифрования данных значительно повышает уровень безопасности системы. В рамках лабораторной работы было продемонстрировано шифрование пароля пользователя, что обеспечивает несколько важных уровней защиты.

1. Шифрование паролей

В данном примере применено симметричное шифрование с использованием функции pgp\_sym\_encrypt, позволяющее безопасно хранить пароли в базе данных.

UPDATE users

SET hashed\_password = pgp\_sym\_encrypt(hashed\_password, 'encryption\_key')

WHERE hashed\_password NOT LIKE '-----BEGIN PGP MESSAGE-----%';

username | hashed\_password

-----------+--------------------------------------------------------------------------------------------------------------------------------------------------------------------

test\_user | \xc30d040703023999a080048be39e7fd23f01539ac0422ad829ec01b54a5f1b250f44de05be453ebb8480e520d41216da3dd7d023cdb245c46b7a731e6a4fb7efc6db2cbd6b4352d1bd5832d756c4cbc6

(1 row)

username | pgp\_sym\_decrypt

-----------+-----------------

test\_user | securepassword

(1 row)

**Что получили:**

* **Защита данных**: пароли не хранятся в открытом виде, что существенно снижает риск утечек.
* **Конфиденциальность**: даже в случае компрометации базы данных злоумышленники не смогут получить доступ к паролям без ключа шифрования.

2. Симметричное шифрование

Примененное симметричное шифрование предполагает использование одного и того же ключа как для шифрования, так и для расшифровки данных (например, 'encryption\_key').

Преимущества:

* **Простота использования**: механизм симметричного шифрования легко внедряется и обслуживается.

**Недостатки:**

* **Уязвимость при компрометации ключа**: раскрытие ключа приводит к риску полной утечки зашифрованных данных.
* **Необходимость защиты ключей**: ключи шифрования должны храниться в безопасных хранилищах и подлежать регулярной смене (ротации).

Сравнение pg\_bigm и pg\_trgm

В рамках эксперимента были выполнены запросы с использованием индексов, созданных на базе расширений pg\_bigm и pg\_trgm.

Пример планов выполнения запросов:

CREATE INDEX IF NOT EXISTS trgm\_idx\_products\_name

ON products USING gin (name gin\_trgm\_ops);

CREATE INDEX IF NOT EXISTS bigm\_idx\_products\_name

ON products USING gin (name gin\_bigm\_ops);

EXPLAIN ANALYZE

SELECT \*

FROM products

WHERE name LIKE '%Adaptive zero administration conglomeration%';

**Запрос с использованием pg\_trgm**:

Bitmap Heap Scan on products

Recheck Cond: ((name)::text ~~ '%Adaptive zero administration conglomeration%'::text)

-> Bitmap Index Scan on trgm\_idx\_products\_name

Planning Time: 1.582 ms

Execution Time: 82.190 ms

**Запрос с использованием pg\_bigm**:

Bitmap Heap Scan on products

Recheck Cond: ((name)::text ~~ '%Adaptive zero administration conglomeration%'::text)

-> Bitmap Index Scan on bigm\_idx\_products\_name

Planning Time: 0.170 ms

Execution Time: 28.292 ms

Таблица сравнения:

| **Параметр** | **pg\_trgm** | **pg\_bigm** |
| --- | --- | --- |
| **Тип индекса** | GIN (gin\_trgm\_ops) | GIN (gin\_bigm\_ops) |
| **План запроса** | Bitmap Heap Scan + Bitmap Index Scan | Bitmap Heap Scan + Bitmap Index Scan |
| **Стоимость запроса** | 832.39..1024.94 | 1168.39..1360.94 |
| **Фактическое время** | 82.169 мс | 28.262 мс |
| **Количество строк** | 0 | 0 |
| **Время планирования** | 1.582 мс | 0.170 мс |

1. **Типы индексов**: Оба индекса используют структуру GIN, но оптимизированы под разные виды поиска:
   * pg\_trgm (триграммы) эффективно работает с произвольными подстроками.
   * pg\_bigm ориентирован на поиск по биграммам (парам символов).
2. **Производительность**:
   * Запрос через pg\_bigm отработал быстрее (28.262 мс против 82.169 мс для pg\_trgm).
   * Время планирования запроса с pg\_bigm оказалось значительно меньше (0.170 мс против 1.582 мс).
3. **Стоимость запроса**:
   * У pg\_trgm стоимость запроса ниже, что указывает на его большую общую "эффективность" с точки зрения планировщика, однако фактическое время выполнения оказалось хуже.
4. **Итог**:
   * В данном тесте использование pg\_bigm обеспечило лучшее время выполнения, что делает его предпочтительным для поиска длинных подстрок.

# **Вывод**

В ходе выполнения данной лабораторной работы вспомнил информацию об индексах и транзакциях. Закрепил знания об индексах на практике, использовав индексы btree, gin, brin; замерил, какой прирост по скорости запросов дают данные индексы.

А также поработал с дополнениями pg\_bigm, pg\_trigm, pgcrypto и рассмотрел в каких сценариях они полезны, применил их в работе: pg\_trgm лучше подходит для поиска коротких подстрок или для полнотекстового поиска с приблизительным совпадением, а pg\_bigm будет предпочтительнее для длинных текстов и поиска длинных последовательностей символов.