Министерство науки и высшего образования РФ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт
Программная инженерия
кафедра

ОТЧЕТ О ПРАКТИЧЕСКОЙ РАБОТЕ

Повышение производительности
тема

Преподаватель _____ А. Д. Вожжов
подпись, дата — инициалы, фамилия

Студент КИ23-17/1б, 032320521 _____ А. С. Лысаковский
номер группы, зачётной книжки — подпись, дата — инициалы, фамилия

Красноярск 2025

1 ВВЕДЕНИЕ

1.1 Цель работы

Изучить теоретический материал по теме «Повышение производительности». Выполнить задания.

1.2 Задачи

В рамках данной практической работы необходимо выполнить следующие задачи:

1 изучить теоретический материал по предложенной теме;

2 выполнить задание;

3 предоставить отчёт преподавателю.

1.3 Задание

Задание данной практической работы состоит из следующих частей:

1 Выполнить задания из главы 10 из книги на е-курсах.

2 ХОД РАБОТЫ

2.1 Задание 1

Задание. Как вы думаете, почему при сканировании по индексу оценка стоимости ресурсов, требующихся для выдачи первых результатов, не равна нулю, хотя используется индекс, совпадающий с порядком сортировки?

На рисунке 1 показан результат выполнения задания.

```
demo=# EXPLAIN
demo-# SELECT *
demo-# FROM bookings
demo-# ORDER BY book_ref;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Index Scan using bookings_pkey on bookings  (cost=0.42..8549.24 rows=262788 width=21)
(1 строка)
```

Рисунок 1 – Запрос

Для выдачи первых результатов необходимо сначала найти индекс в специальной таблице. Поэтому начальные затраты и не равны нулю.

2.2 Задание 2

Задание. Как вы думаете, если в запросе присутствует предложение ORDER BY, и создан индекс по тем столбцам, которые фигурируют в предложении ORDER BY, то всегда ли будет использоваться этот индекс или нет? Почему? Проверьте ваши предположения с помощью команды EXPLAIN.

Ответ. Не всегда.

На рисунке 2 показан результат выполнения задания.

```
demo=# EXPLAIN SELECT * FROM bookings ORDER BY book_ref;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using bookings_pkey on bookings  (cost=0.42..8549.24 rows=262788 width=21)
(1 строка)


demo=# EXPLAIN SELECT * FROM bookings;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on bookings  (cost=0.00..4339.88 rows=262788 width=21)
(1 строка)


demo=# EXPLAIN SELECT * FROM bookings ORDER BY book_ref;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using bookings_pkey on bookings  (cost=0.42..8549.24 rows=262788 width=21)
(1 строка)


demo=# EXPLAIN SELECT * FROM bookings ORDER BY book_ref LIMIT 10;
                                      QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=0.42..0.75 rows=10 width=21)
   ->  Index Scan using bookings_pkey on bookings  (cost=0.42..8549.24 rows=262788 width=21)
(2 строки)
```

Рисунок 2 – Эксперименты

Как видно из примеров, не всегда используется поиск по индексу. Потому-что планировщик подготавливает несколько планов выполнения запросов и выбирает лучший. Не всегда выгодно обращаться к индексной таблице.

2.3 Задание 3

Задание. Самостоятельно выполните команду EXPLAIN для запроса, содержащего общее табличное выражение (CTE). Посмотрите, на каком уровне находится узел плана, отвечающий за это выражение, как он оформляется. Учтите, что общие табличные выражения всегда материализуются, т.е. вычисляются однократно и результат их вычисления сохраняется в памяти, а затем все последующие обращения в рамках запроса направляются уже к этому материализованному результату.

На рисунке 3 показан прогресс работы.

```
demo=# EXPLAIN
demo-# WITH cte AS MATERIALIZED
demo-# (
demo(# SELECT passenger_id, passenger_name, contact_data
demo(# FROM tickets
demo(# )
demo-# SELECT * FROM cte
demo-# WHERE passenger_name ~ '^IVAN';
                                QUERY PLAN
-------------------------------------------------------------------------
 CTE Scan on cte  (cost=9843.35..18094.89 rows=6771 width=122)
   Filter: (passenger_name ~ '^IVAN'::text)
   CTE cte
     ->  Seq Scan on tickets  (cost=0.00..9843.35 rows=366735 width=83)
(4 строки)
```

Рисунок 3 – CTE запрос

CTE находится в самом низу.

2.4 Задание 4

Прокомментируйте следующий план, попробуйте объяснить значения всех его узлов и параметров

На рисунке 4 показан результат выполнения задания.

```
demo=# EXPLAIN
demo-# SELECT total_amount
demo-# FROM bookings
demo-# ORDER BY total_amount DESC
demo-# LIMIT 5;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=6825.36..6825.93 rows=5 width=6)
   ->  Gather Merge  (cost=6825.36..24602.17 rows=154581 width=6)
         Workers Planned: 1
         ->  Sort  (cost=5825.35..6211.80 rows=154581 width=6)
               Sort Key: total_amount DESC
               ->  Parallel Seq Scan on bookings  (cost=0.00..3257.81 rows=154581 width=6)
(6 строк)
```

Рисунок 4 – Анализ

Производится запрос на возврат из таблицы bookings столбца total_amount. Затем данные сортируются по убыванию столбца total_amount. Стоит отметить, что данные задачи выполняются параллельно. Следующим шагом происходит объединение параллельной работы запросов выше. Наконец, ограничение вывода первыми пятью результатами с учётом сортировки.

2.5 Задание 5

Задание. В подавляющем большинстве городов только один аэропорт, но есть и такие города, в которых более одного аэропорта. Давайте их выявим.

Как вы думаете, чем можно объяснить, что вторая оценка стоимости в параметре cost для узла Seq Scan, равная 3,04, не совпадает с первой оценкой стоимости в параметре cost для узла HashAggregate?

На рисунке 5 показан прогресс работы.

```
demo=# EXPLAIN
demo-# SELECT city, count( * )
demo-# FROM airports
demo-# GROUP BY city
demo-# HAVING count( * ) > 1;
                          QUERY PLAN
---------------------------------------------------------------
 HashAggregate  (cost=3.56..4.82 rows=34 width=25)
   Group Key: city
   Filter: (count(*) > 1)
   ->  Seq Scan on airports  (cost=0.00..3.04 rows=104 width=17)
(4 строки)
```

Рисунок 5 – Анализ запроса

Расходы не сходятся, потому что создаются дополнительные структуры данных, например хэш-таблица, и фильтрация. Эти действия создают дополнительные расходы.

2.6 Задание 6

Выполните команду EXPLAIN для запроса, в котором использована какая-нибудь из оконных функций. Найдите в плане выполнения запроса узел с именем WindowAgg. Попробуйте объяснить, почему он занимает именно этот уровень в плане.

На рисунках 6 показан результат выполнения задания.

```
demo=# CREATE TABLE sales (
demo(#     id SERIAL PRIMARY KEY,
demo(#     product TEXT,
demo(#     sale_date DATE,
demo(#     amount NUMERIC
demo(# );
CREATE TABLE
demo=#
demo=# INSERT INTO sales (product, sale_date, amount) VALUES
demo-# ('Product A', '2023-01-01', 100),
demo-# ('Product A', '2023-01-02', 150),
demo-# ('Product B', '2023-01-01', 200),
demo-# ('Product B', '2023-01-02', 250),
demo-# ('Product A', '2023-01-03', 300);
INSERT 0 5
demo=#
demo=# EXPLAIN
demo-# SELECT
demo-#     product,
demo-#     sale_date,
demo-#     amount,
demo-#     SUM(amount) OVER (PARTITION BY product ORDER BY sale_date) AS cumulative_sum
demo-# FROM sales;
                            QUERY PLAN
-------------------------------------------------------------------
 WindowAgg  (cost=57.25..73.43 rows=810 width=100)
   ->  Sort  (cost=57.23..59.26 rows=810 width=68)
         Sort Key: product, sale_date
         ->  Seq Scan on sales  (cost=0.00..18.10 rows=810 width=68)
(4 строки)
```

Рисунок 6 – Анализ

Такая позиция WindowAgg обусловлена тем, что оконные функции выполняются после сортировок и фильтров, но до ограничений выводов (LIMIT).

2.7 Задание 7

Проанализируйте план выполнения операций вставки и удаления строк. Причем сделайте это таким образом, чтобы данные в таблицах фактически изменены не были.

На рисунке 7 показан прогресс работы.

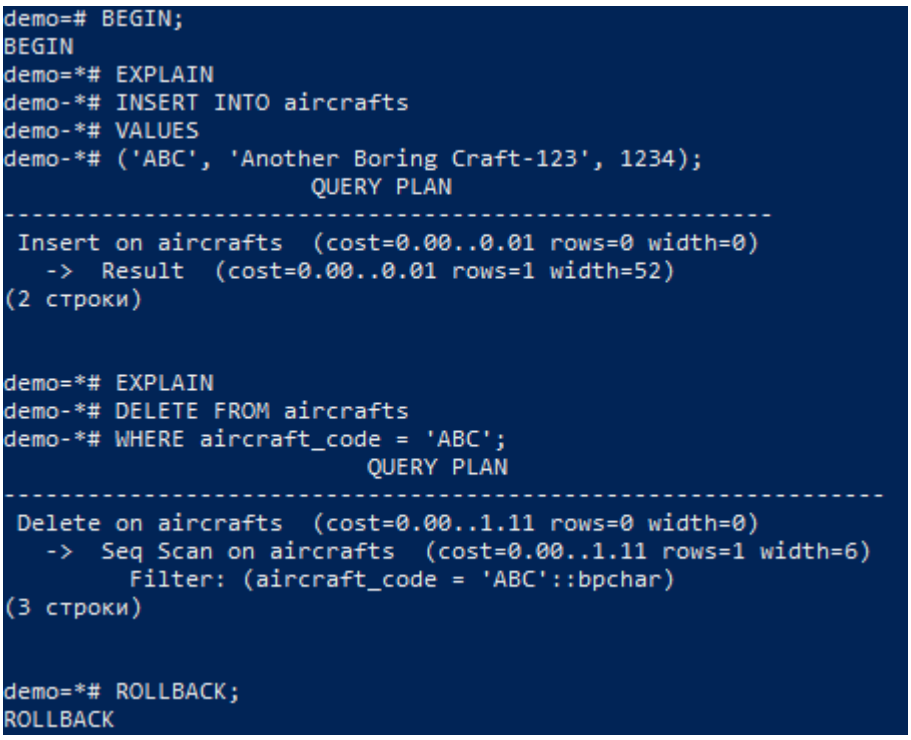

Рисунок 7 – Анализ операций

2.8 Задание 8

Задание. Замена коррелированного подзапроса соединением таблиц является одним из способов повышения производительности. Предположим, что мы задались вопросом: сколько маршрутов обслуживают самолеты каждого типа? При этом нужно учитывать, что может иметь место такая ситуация, когда самолеты какого-либо типа не обслуживают ни одного маршрута. Поэтому необходимо использовать не только представление «Маршруты» (routes), но и таблицу «Самолеты» (aircrafts).

Это первый вариант запроса, в нем используется коррелированный подзапрос.

```
EXPLAIN ANALYZE
SELECT a.aircraft_code AS a_code,
a.model,
( SELECT count( r.aircraft_code )
FROM routes r
WHERE r.aircraft_code = a.aircraft_code
) AS num_routes
FROM aircrafts a
GROUP BY 1, 2
ORDER BY 3 DESC;
```

А в этом варианте коррелированный подзапрос раскрыт и заменен внешним соединением:

```
EXPLAIN ANALYZE
SELECT a.aircraft_code AS a_code,
a.model,
count( r.aircraft_code ) AS num_routes
FROM aircrafts a
LEFT OUTER JOIN routes r
ON r.aircraft_code = a.aircraft_code
GROUP BY 1, 2
ORDER BY 3 DESC;
```

Причина использования внешнего соединения в том, что может найтись модель самолета, не обслуживающая ни одного маршрута, и если не использовать внешнее соединение, она вообще не попадет в результирующую выборку.

Исследуйте планы выполнения обоих запросов. Попытайтесь найти объяснение различиям в эффективности их выполнения. Чтобы получить усредненную картину, выполните каждый запрос несколько раз. Поскольку таблицы, участвующие в запросах, небольшие, то различие по абсолютным затратам времени выполнения будет незначительным. Но если бы число строк в таблицах было большим, то экономия ресурсов сервера могла оказаться заметной.

Предложите аналогичную пару запросов к базе данных «Авиаперевозки». Проведите необходимые эксперименты с вашими запросами.

На рисунках с 8 по 15 показан прогресс работы.

```
demo=# EXPLAIN ANALYZE
demo-# SELECT a.aircraft_code AS a_code,
demo-# a.model,
demo-# ( SELECT count( r.aircraft_code )
demo(# FROM routes r
demo(# WHERE r.aircraft_code = a.aircraft_code
demo(# ) AS num_routes
demo-# FROM aircrafts a
demo-# GROUP BY 1, 2
demo-# ORDER BY 3 DESC;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=371.31..371.34 rows=9 width=56) (actual time=0.804..0.805 rows=9 loops=1)
   Sort Key: ((SubPlan 1)) DESC
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (cost=1.11..371.17 rows=9 width=56) (actual time=0.146..0.799 rows=9 loops=1)
         Group Key: a.aircraft_code
         Batches: 1  Memory Usage: 24kB
         ->  Seq Scan on aircrafts a  (cost=0.00..1.09 rows=9 width=48) (actual time=0.009..0.010 rows=9 loops=1)
         SubPlan 1
           ->  Aggregate  (cost=41.10..41.11 rows=1 width=8) (actual time=0.085..0.085 rows=1 loops=9)
                 ->  Seq Scan on routes r  (cost=0.00..40.88 rows=89 width=4) (actual time=0.016..0.078 rows=79 loops=9)
                       Filter: (aircraft_code = a.aircraft_code)
                       Rows Removed by Filter: 631
 Planning Time: 0.133 ms
 Execution Time: 0.832 ms
(14 строк)
```

Рисунок 8 – Коррелированный подзапрос 1

```
demo=# EXPLAIN ANALYZE
demo-# SELECT a.aircraft_code AS a_code,
demo-# a.model,
demo-# ( SELECT count( r.aircraft_code )
demo(# FROM routes r
demo(# WHERE r.aircraft_code = a.aircraft_code
demo(# ) AS num_routes
demo-# FROM aircrafts a
demo-# GROUP BY 1, 2
demo-# ORDER BY 3 DESC;
                                                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=371.31..371.34 rows=9 width=56) (actual time=0.804..0.805 rows=9 loops=1)
   Sort Key: ((SubPlan 1)) DESC
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (cost=1.11..371.17 rows=9 width=56) (actual time=0.146..0.799 rows=9 loops=1)
         Group Key: a.aircraft_code
         Batches: 1  Memory Usage: 24kB
         ->  Seq Scan on aircrafts a  (cost=0.00..1.09 rows=9 width=48) (actual time=0.009..0.010 rows=9 loops=1)
         SubPlan 1
           ->  Aggregate  (cost=41.10..41.11 rows=1 width=8) (actual time=0.085..0.085 rows=1 loops=9)
                 ->  Seq Scan on routes r  (cost=0.00..40.88 rows=89 width=4) (actual time=0.016..0.078 rows=79 loops=9)
                       Filter: (aircraft_code = a.aircraft_code)
                       Rows Removed by Filter: 631
 Planning Time: 0.133 ms
 Execution Time: 0.832 ms
(14 строк)
```

Рисунок 9 – Коррелированный подзапрос 2

```
demo=# EXPLAIN ANALYZE
demo-# SELECT a.aircraft_code AS a_code,
demo-# a.model,
demo-# count( r.aircraft_code ) AS num_routes
demo-# FROM aircrafts a
demo-# LEFT OUTER JOIN routes r
demo-# ON r.aircraft_code = a.aircraft_code
demo-# GROUP BY 1, 2
demo-# ORDER BY 3 DESC;
                                                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=46.83..46.85 rows=9 width=56) (actual time=0.430..0.432 rows=9 loops=1)
   Sort Key: (count(r.aircraft_code)) DESC
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (cost=46.60..46.69 rows=9 width=56) (actual time=0.424..0.426 rows=9 loops=1)
         Group Key: a.aircraft_code
         Batches: 1  Memory Usage: 24kB
         ->  Hash Right Join  (cost=1.20..43.05 rows=710 width=52) (actual time=0.026..0.313 rows=711 loops=1)
               Hash Cond: (r.aircraft_code = a.aircraft_code)
               ->  Seq Scan on routes r  (cost=0.00..39.10 rows=710 width=4) (actual time=0.007..0.101 rows=710 loops=1)
               ->  Hash  (cost=1.09..1.09 rows=9 width=48) (actual time=0.015..0.015 rows=9 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 9kB
                     ->  Seq Scan on aircrafts a  (cost=0.00..1.09 rows=9 width=48) (actual time=0.010..0.011 rows=9 loops=1)
 Planning Time: 0.177 ms
 Execution Time: 0.456 ms
(14 строк)
```

Рисунок 10 – Внешнее соединение 1

```
demo=# EXPLAIN ANALYZE
demo-# SELECT a.aircraft_code AS a_code,
demo-# a.model,
demo-# count( r.aircraft_code ) AS num_routes
demo-# FROM aircrafts a
demo-# LEFT OUTER JOIN routes r
demo-# ON r.aircraft_code = a.aircraft_code
demo-# GROUP BY 1, 2
demo-# ORDER BY 3 DESC;
                                                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=46.83..46.85 rows=9 width=56) (actual time=0.428..0.430 rows=9 loops=1)
   Sort Key: (count(r.aircraft_code)) DESC
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (cost=46.60..46.69 rows=9 width=56) (actual time=0.402..0.403 rows=9 loops=1)
         Group Key: a.aircraft_code
         Batches: 1  Memory Usage: 24kB
         ->  Hash Right Join  (cost=1.20..43.05 rows=710 width=52) (actual time=0.038..0.286 rows=711 loops=1)
               Hash Cond: (r.aircraft_code = a.aircraft_code)
               ->  Seq Scan on routes r  (cost=0.00..39.10 rows=710 width=4) (actual time=0.019..0.084 rows=710 loops=1)
               ->  Hash  (cost=1.09..1.09 rows=9 width=48) (actual time=0.014..0.015 rows=9 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 9kB
                     ->  Seq Scan on aircrafts a  (cost=0.00..1.09 rows=9 width=48) (actual time=0.010..0.011 rows=9 loops=1)
 Planning Time: 0.125 ms
 Execution Time: 0.452 ms
(14 строк)
```

Рисунок 11 – Внешнее соединение 2

```
demo=# EXPLAIN ANALYZE
demo-# SELECT ap.airport_code AS ap_code,
demo-#        ap.airport_name,
demo-#        (SELECT count(f.flight_id)
demo(#         FROM flights f
demo(#         WHERE f.departure_airport = ap.airport_code) AS num_flights
demo-# FROM airports ap
demo-# GROUP BY 1, 2
demo-# ORDER BY 3 DESC;
                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=83930.47..83930.73 rows=104 width=29) (actual time=313.030..313.035 rows=104 loops=1)
   Sort Key: ((SubPlan 1)) DESC
   Sort Method: quicksort  Memory: 30kB
   ->  Group  (cost=0.14..83926.98 rows=104 width=29) (actual time=4.179..312.969 rows=104 loops=1)
         Group Key: ap.airport_code
         ->  Index Scan using airports_pkey on airports ap  (cost=0.14..17.70 rows=104 width=21) (actual time=0.011..0.104 rows=104 loops=1)
         SubPlan 1
           ->  Aggregate  (cost=806.81..806.82 rows=1 width=8) (actual time=3.007..3.007 rows=1 loops=104)
                 ->  Seq Scan on flights f  (cost=0.00..806.01 rows=318 width=4) (actual time=1.951..2.988 rows=318 loops=104)
                       Filter: (departure_airport = ap.airport_code)
                       Rows Removed by Filter: 32803
 Planning Time: 15.244 ms
 Execution Time: 313.072 ms
(13 строк)
```

Рисунок 12 – Свой коррелированный подзапрос 1

```
demo=# EXPLAIN ANALYZE
demo-# SELECT ap.airport_code AS ap_code,
demo-#        ap.airport_name,
demo-#        (SELECT count(f.flight_id)
demo(#         FROM flights f
demo(#         WHERE f.departure_airport = ap.airport_code) AS num_flights
demo-# FROM airports ap
demo-# GROUP BY 1, 2
demo-# ORDER BY 3 DESC;
                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=83930.47..83930.73 rows=104 width=29) (actual time=323.835..323.840 rows=104 loops=1)
   Sort Key: ((SubPlan 1)) DESC
   Sort Method: quicksort  Memory: 30kB
   ->  Group  (cost=0.14..83926.98 rows=104 width=29) (actual time=4.348..323.777 rows=104 loops=1)
         Group Key: ap.airport_code
         ->  Index Scan using airports_pkey on airports ap  (cost=0.14..17.70 rows=104 width=21) (actual time=0.007..0.098 rows=104 loops=1)
         SubPlan 1
           ->  Aggregate  (cost=806.81..806.82 rows=1 width=8) (actual time=3.111..3.111 rows=1 loops=104)
                 ->  Seq Scan on flights f  (cost=0.00..806.01 rows=318 width=4) (actual time=2.026..3.092 rows=318 loops=104)
                       Filter: (departure_airport = ap.airport_code)
                       Rows Removed by Filter: 32803
 Planning Time: 0.168 ms
 Execution Time: 323.872 ms
(13 строк)
```

Рисунок 13 – Свой коррелированный подзапрос 2

```
demo=# EXPLAIN ANALYZE
demo-# SELECT ap.airport_code AS ap_code,
demo-#        ap.airport_name,
demo-#        count(f.flight_id) AS num_flights
demo-# FROM airports ap
demo-# LEFT OUTER JOIN flights f
demo-# ON f.departure_airport = ap.airport_code
demo-# GROUP BY 1, 2
demo-# ORDER BY 3 DESC;
                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=988.16..988.42 rows=104 width=29) (actual time=12.768..12.772 rows=104 loops=1)
   Sort Key: (count(f.flight_id)) DESC
   Sort Method: quicksort  Memory: 30kB
   ->  HashAggregate  (cost=983.64..984.68 rows=104 width=29) (actual time=12.739..12.751 rows=104 loops=1)
         Group Key: ap.airport_code
         Batches: 1  Memory Usage: 32kB
         ->  Hash Right Join  (cost=4.34..818.03 rows=33121 width=25) (actual time=0.058..8.248 rows=33121 loops=1)
               Hash Cond: (f.departure_airport = ap.airport_code)
               ->  Seq Scan on flights f  (cost=0.00..723.21 rows=33121 width=8) (actual time=0.006..1.373 rows=33121 loops=1)
               ->  Hash  (cost=3.04..3.04 rows=104 width=21) (actual time=0.047..0.047 rows=104 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 14kB
                     ->  Seq Scan on airports ap  (cost=0.00..3.04 rows=104 width=21) (actual time=0.011..0.034 rows=104 loops=1)
 Planning Time: 0.355 ms
 Execution Time: 12.799 ms
(14 строк)
```

Рисунок 14 – Свой запрос с внешним соединением 1

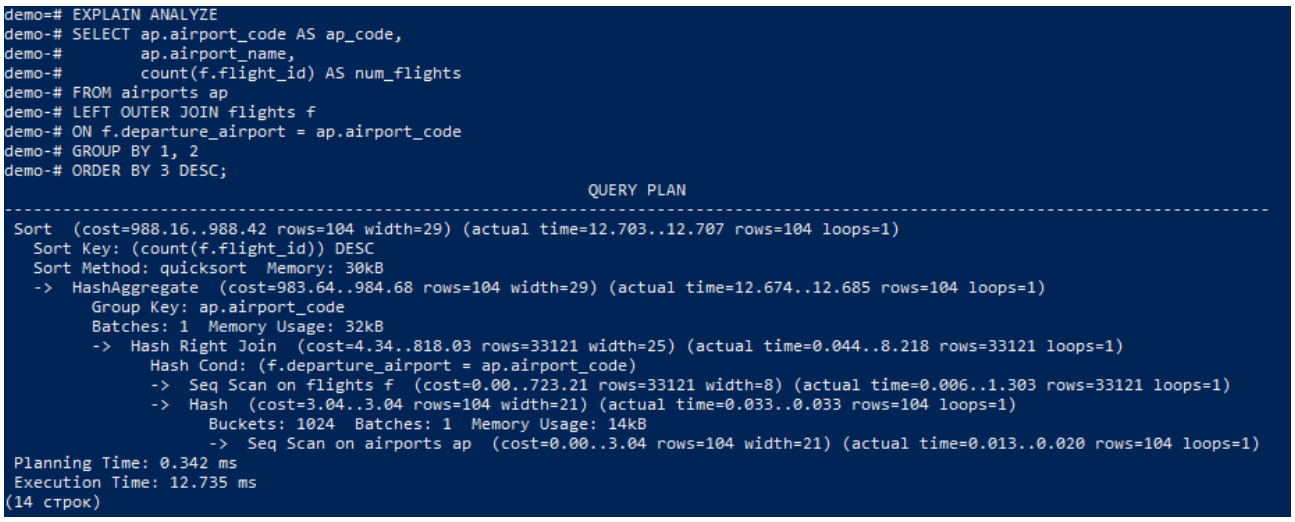

Рисунок 15 – Свой запрос с внешним соединением 2

Низкая скорость коррелированных подзапросов обусловлена в необходимости анализировать таблицу много раз. Внешние соединения позволяют выполнить анализ таблиц условно 1 раз и далее уже выполнять группировку. По итогу внешние соединения использовать выгоднее на больших выборках, нежели коррелированные подзапросы.

2.9 Задание 9

Одним из способов повышения производительности является изменение схемы данных, связанное с денормализацией, а именно: создание материализованных представлений. В главе 5 было описано такое материализованное представление - "Маршруты" (routes). Команда для его создания была приведена в главе 6.

Проведите эксперимент: сначала выполните выборку из готового представления, а затем ту выборку, которая это представление формирует.

```
EXPLAIN ANALYZE
SELECT * FROM routes;

EXPLAIN ANALYZE
WITH f3 AS (SELECT f2.flight_no, ...
```

Поскольку второй запрос очень громоздкий, то можно поступить таким образом: сначала сохраните его в текстовом файле, а затем выполните с помощью команды \i утилиты psql.

Вы увидите, что затраты времени отличаются практически на два порядка. Конечно, нужно помнить, что материализованные представления необходимо периодически обновлять, чтобы их содержимое было актуальным.

На рисунках 16-17 показан прогресс работы.

```
demo=# EXPLAIN ANALYZE
demo-# SELECT * FROM routes;
                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Seq Scan on routes  (cost=0.00..39.10 rows=710 width=147) (actual time=0.027..0.137 rows=710 loops=1)
 Planning Time: 0.137 ms
 Execution Time: 0.162 ms
(3 строки)
```

Рисунок 16 – Запрос к представлению

```
                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=2442.86..2638.23 rows=276 width=135)
   Hash Cond: (flights.arrival_airport = arr.airport_code)
   ->  Hash Join  (cost=2438.52..2632.47 rows=530 width=101)
         Hash Cond: (flights.departure_airport = dep.airport_code)
         ->  GroupAggregate  (cost=2434.18..2625.39 rows=1020 width=67)
               Group Key: flights.flight_no, flights.departure_airport, flights.arrival_airport, flights.aircraft_code, ((flights.scheduled_arrival - flights.scheduled_departure))
               ->  Sort  (cost=2434.18..2459.67 rows=10198 width=39)
                     Sort Key: flights.flight_no, flights.departure_airport, flights.arrival_airport, flights.aircraft_code, ((flights.scheduled_arrival - flights.scheduled_departure)), ((to_char(flights.scheduled_departure, 'ID'::text))::integer)
                     ->  HashAggregate  (cost=1551.24..1755.20 rows=10198 width=39)
                           Group Key: flights.flight_no, flights.departure_airport, flights.arrival_airport, flights.aircraft_code, (flights.scheduled_arrival - flights.scheduled_departure), (to_char(flights.scheduled_departure, 'ID'::text))::integer
                           ->  Seq Scan on flights  (cost=0.00..1054.42 rows=33121 width=39)
         ->  Hash  (cost=3.04..3.04 rows=104 width=38)
               ->  Seq Scan on airports dep  (cost=0.00..3.04 rows=104 width=38)
   ->  Hash  (cost=3.04..3.04 rows=104 width=38)
         ->  Seq Scan on airports arr  (cost=0.00..3.04 rows=104 width=38)
(15 строк)
```

Рисунок 17 – Запрос к исходному запросу представления

2.10 Задание 10

Одним из способов повышения производительности является изменение схемы данных, связанное с денормализацией, а именно: использование вычисляемых столбцов. Для примера рассмотрим таблицу "Бронирования" (bookings). В ней столбец "Полная сумма бронирования" (total_amount) является вычисляемым. Мы не будем сейчас говорить о том, каким образом его значения синхронизируются с данными в таблице "Перелеты" (ticket_flights), а лишь рассмотрим два запроса, возвращающие полные суммы бронирований.

Предположим, что указанного столбца в таблице bookings не было бы. Тогда запрос, возвращающий полные суммы бронирований, выглядел бы так:

```
EXPLAIN ANALYZE
SELECT b.book_ref, sum(tf.amount)
FROM bookings b, tickets t, ticket_flights tf
WHERE b.book_ref = t.book_ref
AND t.ticket_no = tf.ticket_no
GROUP BY 1
ORDER BY 1;
```

Но благодаря наличию вычисляемого столбца total_amount те же сведения можно получить с гораздо меньшими затратами ресурсов:

```
EXPLAIN ANALYZE
SELECT book_ref, total_amount
FROM bookings
ORDER BY 1;
```

В качестве другого примера можно предложить добавить вычисляемый столбец "Количество билетов" (tickets_count) в таблицу bookings, который будет

хранить количество билетов в каждом бронировании. Это позволит избежать сложных соединений таблиц при подсчете количества билетов:

```
-- Без вычисляемого столбца:
EXPLAIN ANALYZE
SELECT b.book_ref, count(t.ticket_no)
FROM bookings b JOIN tickets t ON b.book_ref = t.book_ref
GROUP BY 1;

-- С вычисляемым столбцом:
EXPLAIN ANALYZE
SELECT book_ref, tickets_count
FROM bookings;
```

Эксперименты покажут значительное сокращение времени выполнения запроса и уменьшение нагрузки на сервер, особенно при частых обращениях к этой информации. Однако следует помнить о необходимости поддержания актуальности данных в вычисляемом столбце при изменении связанных таблиц.

На рисунках 18-19 показан результат работы.

```
demo=# EXPLAIN ANALYZE
demo-# SELECT b.book_ref, COUNT(tf.flight_id) AS flight_count
demo-# FROM bookings b
demo-# JOIN tickets t ON b.book_ref = t.book_ref
demo-# JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
demo-# GROUP BY b.book_ref
demo-# ORDER BY b.book_ref;
                                                                             QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize GroupAggregate  (cost=86384.69..158200.73 rows=262788 width=15) (actual time=2011.446..2770.487 rows=262790 loops=1)
   Group Key: b.book_ref
   ->  Gather Merge  (cost=86384.69..152944.97 rows=525576 width=15) (actual time=2011.441..2685.224 rows=393425 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial GroupAggregate  (cost=85384.67..91280.44 rows=262788 width=15) (actual time=1965.734..2277.786 rows=131142 loops=3)
               Group Key: b.book_ref
               ->  Sort  (cost=85384.67..86473.96 rows=435719 width=11) (actual time=1965.726..2213.581 rows=348576 loops=3)
                     Sort Key: b.book_ref
                     Sort Method: external merge  Disk: 7568kB
                     Worker 0:  Sort Method: external merge  Disk: 7576kB
                     Worker 1:  Sort Method: external merge  Disk: 7448kB
                     ->  Parallel Hash Join  (cost=15700.21..37124.96 rows=435719 width=11) (actual time=275.428..507.252 rows=348576 loops=3)
                           Hash Cond: (t.book_ref = b.book_ref)
                           ->  Parallel Hash Join  (cost=10510.14..30791.10 rows=435719 width=11) (actual time=252.688..372.675 rows=348576 loops=3)
                                 Hash Cond: (tf.ticket_no = t.ticket_no)
                                 ->  Parallel Seq Scan on ticket_flights tf  (cost=0.00..13133.19 rows=435719 width=18) (actual time=0.463..137.981 rows=348576 loops=3)
                                 ->  Parallel Hash  (cost=7704.06..7704.06 rows=152806 width=21) (actual time=41.084..41.084 rows=122245 loops=3)
                                       Buckets: 131072  Batches: 4  Memory Usage: 6144kB
                                       ->  Parallel Seq Scan on tickets t  (cost=0.00..7704.06 rows=152806 width=21) (actual time=0.013..16.391 rows=122245 loops=3)
                           ->  Parallel Hash  (cost=3257.81..3257.81 rows=154581 width=7) (actual time=21.946..21.947 rows=87597 loops=3)
                                 Buckets: 524288  Batches: 1  Memory Usage: 14432kB
                                 ->  Parallel Seq Scan on bookings b  (cost=0.00..3257.81 rows=154581 width=7) (actual time=0.013..5.348 rows=87597 loops=3)
 Planning Time: 0.517 ms
 Execution Time: 2781.530 ms
(25 строк)
```

Рисунок 18 – Запрос без вычисляемого столбца

```
demo=# EXPLAIN ANALYZE
demo-# SELECT book_ref, flight_count
demo-# FROM bookings
demo-# ORDER BY book_ref;
                                                    QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------
 Index Scan using bookings_pkey on bookings  (cost=0.42..8549.24 rows=262788 width=11) (actual time=0.019..48.756 rows=262790 loops=1)
 Planning Time: 0.092 ms
 Execution Time: 53.744 ms
(3 строки)
```

Рисунок 19 – Запрос с вычисляемым столбцом

2.11 Задание 11

Одним из способов повышения производительности является использование временных таблиц. Если нужно сделать много выборок из представления "Рейсы" (flights_v), стоит создать временную таблицу:

CREATE TEMP TABLE flights_tt AS SELECT * FROM flights_v;

Сравним планы выполнения запросов:
EXPLAIN ANALYZE SELECT * FROM flights_v;
EXPLAIN ANALYZE SELECT * FROM flights_tt;

План для представления многоуровневый, потому что представление каждый раз пересчитывает данные из исходных таблиц. Для временной таблицы план проще, так как данные уже сохранены.

Для более сложных запросов:
EXPLAIN ANALYZE SELECT departure_city, count(*) FROM flights_v WHERE status = 'Scheduled' GROUP BY 1;
EXPLAIN ANALYZE SELECT departure_city, count(*) FROM flights_tt WHERE status = 'Scheduled' GROUP BY 1;
На рисунках показан прогресс работы.

```
demo=# CREATE TEMP TABLE flights_tt AS
demo-# SELECT * FROM flights_v;
SELECT 33121
demo=#
demo=# EXPLAIN ANALYZE
demo-# SELECT * FROM flights_v;
                                                                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=8.68..1409.67 rows=33121 width=195) (actual time=0.089..46.087 rows=33121 loops=1)
   Hash Cond: (f.arrival_airport = arr.airport_code)
   ->  Hash Join  (cost=4.34..818.03 rows=33121 width=112) (actual time=0.038..11.465 rows=33121 loops=1)
         Hash Cond: (f.departure_airport = dep.airport_code)
         ->  Seq Scan on flights f  (cost=0.00..723.21 rows=33121 width=63) (actual time=0.007..1.565 rows=33121 loops=1)
         ->  Hash  (cost=3.04..3.04 rows=104 width=53) (actual time=0.027..0.028 rows=104 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 17kB
               ->  Seq Scan on airports dep  (cost=0.00..3.04 rows=104 width=53) (actual time=0.004..0.013 rows=104 loops=1)
   ->  Hash  (cost=3.04..3.04 rows=104 width=53) (actual time=0.036..0.037 rows=104 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 17kB
         ->  Seq Scan on airports arr  (cost=0.00..3.04 rows=104 width=53) (actual time=0.010..0.020 rows=104 loops=1)
 Planning Time: 0.385 ms
 Execution Time: 46.748 ms
(13 строк)


demo=# EXPLAIN ANALYZE
demo-# SELECT * FROM flights_tt;
                                                                                                                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on flights_tt  (cost=0.00..1075.20 rows=17920 width=362) (actual time=0.025..2.791 rows=33121 loops=1)
 Planning Time: 0.320 ms
 Execution Time: 3.465 ms
(3 строки)
```

Рисунок 20 – Работа

Большой план к представлению обусловлен тем, что представление – это просто ссылка на сохранённый запрос. Потому, когда мы инспектируем представление, мы инспектируем запрос этого представления.

Сравнение сложных планов представлено на рисунке 21.

```
demo=# EXPLAIN ANALYZE
demo-# SELECT departure_airport, departure_airport_name, COUNT(*) AS flight_count
demo-# FROM flights_v
demo-# WHERE scheduled_departure BETWEEN '2023-01-01' AND '2023-12-31'
demo-# GROUP BY departure_airport, departure_airport_name
demo-# ORDER BY flight_count DESC;
                                                     QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=860.23..860.24 rows=1 width=29) (actual time=4.841..4.842 rows=0 loops=1)
   Sort Key: (count(*)) DESC
   Sort Method: quicksort  Memory: 25kB
   ->  GroupAggregate  (cost=860.20..860.22 rows=1 width=29) (actual time=4.836..4.837 rows=0 loops=1)
         Group Key: f.departure_airport, dep.airport_name
         ->  Sort  (cost=860.20..860.20 rows=1 width=21) (actual time=4.835..4.836 rows=0 loops=1)
               Sort Key: f.departure_airport, dep.airport_name
               Sort Method: quicksort  Memory: 25kB
               ->  Nested Loop  (cost=0.29..860.19 rows=1 width=21) (actual time=4.829..4.830 rows=0 loops=1)
                     Join Filter: (arr.airport_code = f.arrival_airport)
                     ->  Nested Loop  (cost=0.29..855.85 rows=1 width=25) (actual time=4.829..4.829 rows=0 loops=1)
                           Join Filter: (dep.airport_code = f.departure_airport)
                           ->  Index Scan using flights_flight_no_scheduled_departure_key on flights f  (cost=0.29..851.51 rows=1 width=8) (actual time=4.828..4.829 rows=0 loops=1)
                                 Index Cond: ((scheduled_departure >= '2023-01-01 00:00:00+07'::timestamp with time zone) AND (scheduled_departure <= '2023-12-31 00:00:00+07'::timestamp with time zone))
                           ->  Seq Scan on airports dep  (cost=0.00..3.04 rows=104 width=21) (never executed)
                     ->  Seq Scan on airports arr  (cost=0.00..3.04 rows=104 width=4) (never executed)
 Planning Time: 0.471 ms
 Execution Time: 4.877 ms
(18 строк)


demo=# EXPLAIN ANALYZE
demo-# SELECT departure_airport, departure_airport_name, COUNT(*) AS flight_count
demo-# FROM flights_tt
demo-# WHERE scheduled_departure BETWEEN '2023-01-01' AND '2023-12-31'
demo-# GROUP BY departure_airport, departure_airport_name
demo-# ORDER BY flight_count DESC;
                                                     QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=1172.34..1172.56 rows=88 width=56) (actual time=4.153..4.154 rows=0 loops=1)
   Sort Key: (count(*)) DESC
   Sort Method: quicksort  Memory: 25kB
   ->  GroupAggregate  (cost=1167.72..1169.50 rows=88 width=56) (actual time=4.150..4.150 rows=0 loops=1)
         Group Key: departure_airport, departure_airport_name
         ->  Sort  (cost=1167.72..1167.95 rows=90 width=48) (actual time=4.149..4.149 rows=0 loops=1)
               Sort Key: departure_airport, departure_airport_name
               Sort Method: quicksort  Memory: 25kB
               ->  Seq Scan on flights_tt  (cost=0.00..1164.80 rows=90 width=48) (actual time=4.143..4.144 rows=0 loops=1)
                     Filter: ((scheduled_departure >= '2023-01-01 00:00:00+07'::timestamp with time zone) AND (scheduled_departure <= '2023-12-31 00:00:00+07'::timestamp with time zone))
                     Rows Removed by Filter: 33121
 Planning Time: 0.105 ms
 Execution Time: 4.179 ms
(13 строк)
```

Рисунок 21 – Сравнение запросов

Временная таблица выигрывает по времени.

Решение для базы данных «Авиаперевозки» представлено на рисунках 22-24.

```
demo=# CREATE TEMP TABLE bookings_analysis_tt AS
demo-# SELECT b.book_ref, b.total_amount, COUNT(tf.flight_id) AS flight_count
demo-# FROM bookings b
demo-# JOIN tickets t ON b.book_ref = t.book_ref
demo-# JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
demo-# JOIN flights f ON tf.flight_id = f.flight_id
demo-# WHERE b.book_date BETWEEN '2016-08-19 17:05:00+07' AND '2016-08-28 07:15:00+07'
demo-# GROUP BY b.book_ref, b.total_amount;
SELECT 5033
```

Рисунок 22 – Создание временной таблицы

```
demo=# EXPLAIN ANALYZE
demo-# SELECT b.book_ref, SUM(tf.amount) AS total_spent, COUNT(tf.flight_id) AS flight_count
demo-# FROM bookings b
demo-# JOIN tickets t ON b.book_ref = t.book_ref
demo-# JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
demo-# JOIN flights f ON tf.flight_id = f.flight_id
demo-# WHERE b.book_date BETWEEN '2016-08-19 17:05:00+07' AND '2016-08-28 07:15:00+07'
demo-# GROUP BY b.book_ref
demo-# ORDER BY flight_count DESC;
                                                     QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=17601.98..17614.18 rows=4880 width=47) (actual time=130.534..132.020 rows=5033 loops=1)
   Sort Key: (count(tf.flight_id)) DESC
   Sort Method: quicksort  Memory: 389kB
   ->  Finalize HashAggregate  (cost=17242.01..17303.01 rows=4880 width=47) (actual time=128.613..131.179 rows=5033 loops=1)
         Group Key: b.book_ref
         Batches: 1  Memory Usage: 2257kB
         ->  Gather  (cost=16107.41..17144.41 rows=9760 width=47) (actual time=124.563..127.255 rows=5043 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Partial HashAggregate  (cost=15107.41..15168.41 rows=4880 width=47) (actual time=88.759..89.321 rows=1681 loops=3)
                     Group Key: b.book_ref
                     Batches: 1  Memory Usage: 1233kB
                     Worker 0:  Batches: 1  Memory Usage: 977kB
                     Worker 1:  Batches: 1  Memory Usage: 977kB
                     ->  Parallel Hash Join  (cost=5048.29..15046.73 rows=8091 width=17) (actual time=8.410..85.257 rows=6623 loops=3)
                           Hash Cond: (tf.flight_id = f.flight_id)
                           ->  Nested Loop  (cost=4067.03..14044.23 rows=8091 width=17) (actual time=6.052..80.684 rows=6623 loops=3)
                                 ->  Parallel Hash Join  (cost=4066.61..12171.79 rows=2838 width=21) (actual time=5.950..37.100 rows=2336 loops=3)
                                       Hash Cond: (t.book_ref = b.book_ref)
                                       ->  Parallel Seq Scan on tickets t  (cost=0.00..7704.06 rows=152806 width=21) (actual time=0.042..15.479 rows=122245 loops=3)
                                       ->  Parallel Hash  (cost=4030.72..4030.72 rows=2871 width=7) (actual time=5.657..5.657 rows=1678 loops=3)
                                             Buckets: 8192  Batches: 1  Memory Usage: 288kB
                                             ->  Parallel Seq Scan on bookings b  (cost=0.00..4030.72 rows=2871 width=7) (actual time=0.015..16.117 rows=5033 loops=1)
                                                   Filter: ((book_date >= '2016-08-19 17:05:00+07'::timestamp with time zone) AND (book_date <= '2016-08-28 07:15:00+07'::timestamp with time zone))
                                                   Rows Removed by Filter: 257757
                                 ->  Index Scan using ticket_flights_pkey on ticket_flights tf  (cost=0.42..0.63 rows=3 width=24) (actual time=0.012..0.018 rows=3 loops=7009)
                                       Index Cond: (ticket_no = t.ticket_no)
                           ->  Parallel Hash  (cost=737.72..737.72 rows=19483 width=4) (actual time=2.229..2.229 rows=11040 loops=3)
                                 Buckets: 65536  Batches: 1  Memory Usage: 1824kB
                                 ->  Parallel Index Only Scan using flights_pkey on flights f  (cost=0.29..737.72 rows=19483 width=4) (actual time=0.015..2.408 rows=33121 loops=1)
                                       Heap Fetches: 126
 Planning Time: 0.617 ms
 Execution Time: 132.743 ms
(33 строки)
```

Рисунок 23 – Запрос без временной таблицы

```
demo=# EXPLAIN ANALYZE
demo-# SELECT book_ref, total_amount AS total_spent, flight_count
demo-# FROM bookings_analysis_tt
demo-# ORDER BY flight_count DESC;
                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=542.91..559.23 rows=6528 width=52) (actual time=0.776..0.976 rows=5033 loops=1)
   Sort Key: flight_count DESC
   Sort Method: quicksort  Memory: 389kB
   ->  Seq Scan on bookings_analysis_tt  (cost=0.00..129.28 rows=6528 width=52) (actual time=0.013..0.305 rows=5033 loops=1)
 Planning Time: 0.277 ms
 Execution Time: 1.082 ms
(6 строк)
```

Рисунок 24 – Запрос с временной таблицей

2.12 Задание 12

Одним из способов повышения производительности является изменение схемы данных, связанное с денормализацией, а именно: создание индексов. Выполните следующий простой запрос к таблице «Билеты»:

EXPLAIN ANALYZE SELECT count(*) FROM tickets WHERE passenger_name = 'IVAN IVANOV';

Создайте индекс по столбцу passenger_name:

CREATE INDEX passenger_name_key ON tickets (passenger_name);

Теперь повторите запрос и сравните полученные планы и фактические результаты. Предложите какой-нибудь запрос к базе данных «Авиаперевозки», для выполнения которого было бы целесообразно создать индекс. Создайте индекс и повторите запрос. Изучите полученный план, посмотрите, используется ли индекс планировщиком.

На рисунках 25-26 представлен прогресс работы.

```
demo=#
demo=# EXPLAIN ANALYZE
demo-# SELECT count( * )
demo-# FROM tickets
demo-# WHERE passenger_name = 'IVAN IVANOV';
                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=9.08..9.09 rows=1 width=8) (actual time=0.069..0.069 rows=1 loops=1)
   ->  Index Only Scan using tickets_passenger_name_pattern_idx on tickets  (cost=0.42..9.00 rows=33 width=0) (actual time=0.039..0.054 rows=200 loops=1)
         Index Cond: (passenger_name = 'IVAN IVANOV'::text)
         Heap Fetches: 0
 Planning Time: 0.200 ms
 Execution Time: 0.106 ms
(6 строк)


demo=# CREATE INDEX passenger_name_key
demo-# ON tickets ( passenger_name );
CREATE INDEX
demo=# EXPLAIN ANALYZE
demo-# SELECT count( * )
demo-# FROM tickets
demo-# WHERE passenger_name = 'IVAN IVANOV';
                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=9.08..9.09 rows=1 width=8) (actual time=0.136..0.136 rows=1 loops=1)
   ->  Index Only Scan using passenger_name_key on tickets  (cost=0.42..9.00 rows=33 width=0) (actual time=0.115..0.126 rows=200 loops=1)
         Index Cond: (passenger_name = 'IVAN IVANOV'::text)
         Heap Fetches: 0
 Planning Time: 0.326 ms
 Execution Time: 0.151 ms
(6 строк)
```

Рисунок 25 – Прогресс работы

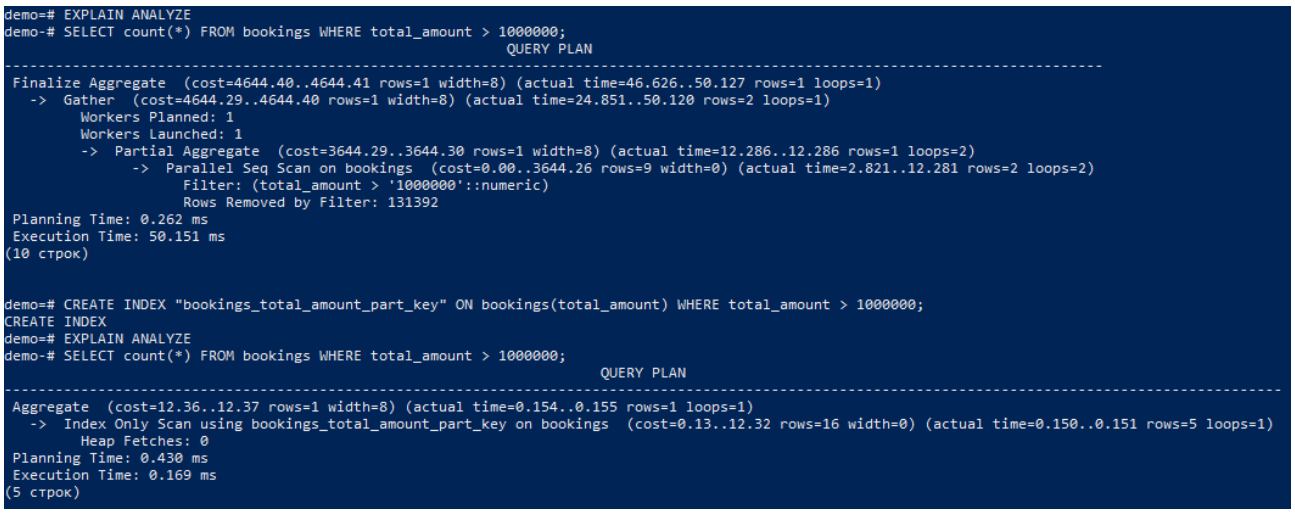

Рисунок 26 – Своё решение для индексов

2.13 Задание 13

В самом конце главы мы выполняли оптимизацию запроса путем создания индекса и модификации текста запроса. Был сформирован такой запрос:

```
EXPLAIN ANALYZE
SELECT num_tickets, count() AS num_bookings
FROM
(SELECT b.book_ref, count()
FROM bookings b, tickets t
WHERE date_trunc('mon', b.book_date) = '2016-09-01'
AND t.book_ref = b.book_ref
GROUP BY b.book_ref
) AS count_tickets(book_ref, num_tickets)
GROUP by num_tickets
ORDER BY num_tickets DESC;
```

Мы экспериментировали с параметрами планировщика enable_hashjoin и enable_nestloop при наличии индекса по таблице tickets:

```
SET enable_hashjoin = off;
SET enable_nestloop = off;
```

Однако полученные планы детально рассмотрены не были.

Задание. Проанализируйте эти планы. Посмотрите, в каких случаях используются и в каких не используются индексы по таблицам bookings и tickets. Вспомните о таком понятии, как селективность, т.е. доля строк, выбираемых из таблицы.

На рисунках 27-29 показан прогресс работы.

```
demo=# EXPLAIN ANALYZE
demo-# SELECT num_tickets, count( * ) AS num_bookings
demo-# FROM
demo-# ( SELECT b.book_ref, count( * )
demo(# FROM bookings b, tickets t
demo(# WHERE date_trunc( 'mon', b.book_date ) = '2016-09-01'
demo(# AND t.book_ref = b.book_ref
demo(# GROUP BY b.book_ref
demo(# ) AS count_tickets( book_ref, num_tickets )
demo-# GROUP by num_tickets
demo-# ORDER BY num_tickets DESC;
                                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=7543.77..7555.62 rows=200 width=16) (actual time=846.293..858.898 rows=5 loops=1)
   Group Key: count_tickets.num_tickets
   ->  Sort  (cost=7543.77..7547.05 rows=1314 width=8) (actual time=846.285..850.716 rows=165543 loops=1)
         Sort Key: count_tickets.num_tickets DESC
         Sort Method: quicksort  Memory: 4096kB
         ->  Subquery Scan on count_tickets  (cost=7303.76..7475.71 rows=1314 width=8) (actual time=700.657..834.839 rows=165543 loops=1)
               ->  Finalize GroupAggregate  (cost=7303.76..7462.57 rows=1314 width=15) (actual time=700.657..827.216 rows=165543 loops=1)
                     Group Key: b.book_ref
                     ->  Gather Merge  (cost=7303.76..7444.03 rows=1079 width=15) (actual time=700.652..788.122 rows=165543 loops=1)
                           Workers Planned: 1
                           Workers Launched: 1
                           ->  Partial GroupAggregate  (cost=6303.75..6322.63 rows=1079 width=15) (actual time=668.722..696.398 rows=82772 loops=2)
                                 Group Key: b.book_ref
                                 ->  Sort  (cost=6303.75..6306.45 rows=1079 width=7) (actual time=668.713..672.416 rows=115345 loops=2)
                                       Sort Key: b.book_ref
                                       Sort Method: quicksort  Memory: 3073kB
                                       Worker 0:  Sort Method: quicksort  Memory: 3073kB
                                       ->  Nested Loop  (cost=0.42..6249.39 rows=1079 width=7) (actual time=0.088..636.107 rows=115345 loops=2)
                                             ->  Parallel Seq Scan on bookings b  (cost=0.00..4030.74 rows=773 width=7) (actual time=0.012..40.935 rows=82772 loops=2)
                                                   Filter: (date_trunc('mon'::text, book_date) = '2016-09-01 00:00:00+07'::timestamp with time zone)
                                                   Rows Removed by Filter: 48624
                                             ->  Index Only Scan using tickets_book_ref_key on tickets t  (cost=0.42..2.85 rows=2 width=7) (actual time=0.007..0.007 rows=1 loops=165543)
                                                   Index Cond: (book_ref = b.book_ref)
                                                   Heap Fetches: 206
 Planning Time: 0.361 ms
 Execution Time: 859.659 ms
(26 строк)
```

Рисунок 27 – Запрос 1

```
demo=# SET enable_hashjoin = off;
SET
demo=# EXPLAIN ANALYZE
demo-# SELECT num_tickets, count( * ) AS num_bookings
demo-# FROM
demo-# ( SELECT b.book_ref, count( * )
demo(# FROM bookings b, tickets t
demo(# WHERE date_trunc( 'mon', b.book_date ) = '2016-09-01'
demo(# AND t.book_ref = b.book_ref
demo(# GROUP BY b.book_ref
demo(# ) AS count_tickets( book_ref, num_tickets )
demo-# GROUP by num_tickets
demo-# ORDER BY num_tickets DESC;
                                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=7543.77..7555.62 rows=200 width=16) (actual time=805.915..820.406 rows=5 loops=1)
   Group Key: count_tickets.num_tickets
   ->  Sort  (cost=7543.77..7547.05 rows=1314 width=8) (actual time=805.907..810.779 rows=165543 loops=1)
         Sort Key: count_tickets.num_tickets DESC
         Sort Method: quicksort  Memory: 4096kB
         ->  Subquery Scan on count_tickets  (cost=7303.76..7475.71 rows=1314 width=8) (actual time=660.902..796.429 rows=165543 loops=1)
               ->  Finalize GroupAggregate  (cost=7303.76..7462.57 rows=1314 width=15) (actual time=660.902..788.810 rows=165543 loops=1)
                     Group Key: b.book_ref
                     ->  Gather Merge  (cost=7303.76..7444.03 rows=1079 width=15) (actual time=660.898..748.960 rows=165543 loops=1)
                           Workers Planned: 1
                           Workers Launched: 1
                           ->  Partial GroupAggregate  (cost=6303.75..6322.63 rows=1079 width=15) (actual time=582.076..609.605 rows=82772 loops=2)
                                 Group Key: b.book_ref
                                 ->  Sort  (cost=6303.75..6306.45 rows=1079 width=7) (actual time=582.070..585.764 rows=115345 loops=2)
                                       Sort Key: b.book_ref
                                       Sort Method: quicksort  Memory: 4096kB
                                       Worker 0:  Sort Method: quicksort  Memory: 3073kB
                                       ->  Nested Loop  (cost=0.42..6249.39 rows=1079 width=7) (actual time=0.071..550.289 rows=115345 loops=2)
                                             ->  Parallel Seq Scan on bookings b  (cost=0.00..4030.74 rows=773 width=7) (actual time=0.011..35.921 rows=82772 loops=2)
                                                   Filter: (date_trunc('mon'::text, book_date) = '2016-09-01 00:00:00+07'::timestamp with time zone)
                                                   Rows Removed by Filter: 48624
                                             ->  Index Only Scan using tickets_book_ref_key on tickets t  (cost=0.42..2.85 rows=2 width=7) (actual time=0.006..0.006 rows=1 loops=165543)
                                                   Index Cond: (book_ref = b.book_ref)
                                                   Heap Fetches: 206
 Planning Time: 0.333 ms
 Execution Time: 821.110 ms
(26 строк)
```

Рисунок 28 – Запрос 2

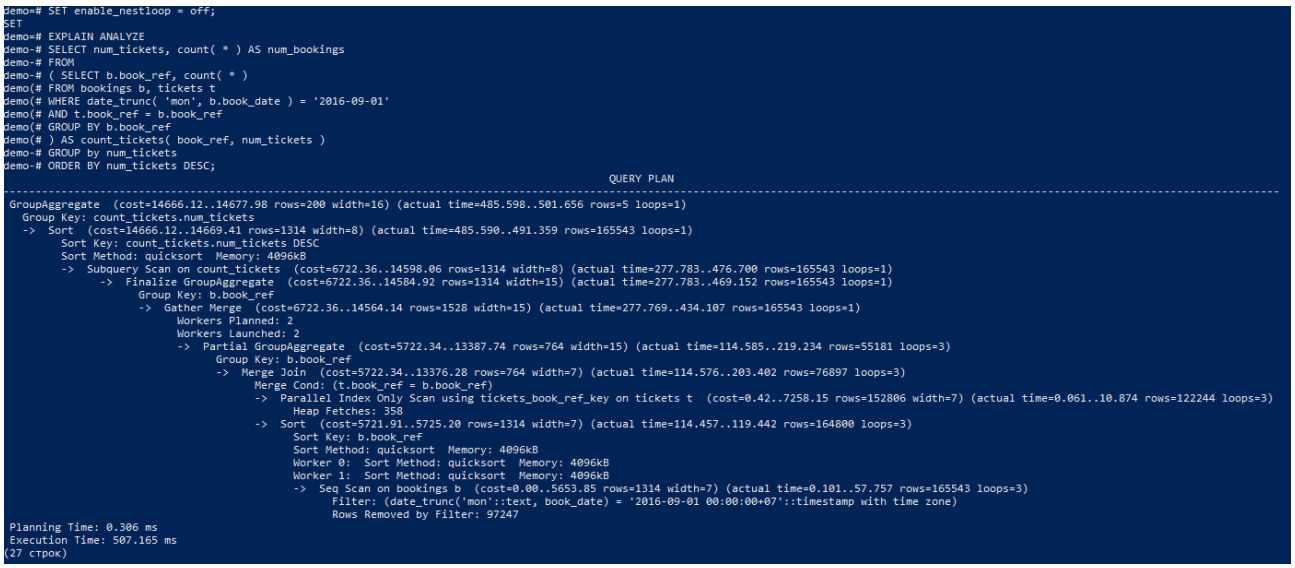

Рисунок 29 – Запрос 3

Индексы в таблицах используются, когда селективность высокая. То есть объём выборки по отношению к таблице мал. Индексы не используются, когда селективность низкая. То есть объём выборки по отношению к таблице велик.

Проанализировав данные запросы, можно сказать, что планировщик может допускать ошибки и выбирать не эффективные планы.

2.14 Задание 14

В столбцах таблиц могут содержаться значения NULL. При сортировке строк по значениям таких столбцов СУБД по умолчанию ведет себя так, как будто значение NULL превосходит по величине любые другие значения. В результате получается, что если задан возрастающий порядок сортировки, то значения NULL будут идти последними, если же порядок сортировки убывающий, тогда они будут первыми. Принимая решение о создании индексов, нужно учитывать требуемый порядок сортировки и желаемое расположение строк со значениями NULL в выборке.

Давайте создадим таблицу, содержащую такое число строк, что использование индекса планировщиком становится очень вероятным:

```
CREATE TABLE nulls AS
SELECT num::integer, 'TEXT' || num::text AS txt
FROM generate_series(1, 200000) AS gen_ser(num);
```

Проиндексируем таблицу по числовому столбцу:

```
CREATE INDEX nulls_ind
ON nulls(num);
```

Добавим в таблицу одну строку, содержащую значение NULL в индексируемом столбце:

```
INSERT INTO nulls
VALUES (NULL, 'TEXT');
```

Проверим использование индекса:

```
EXPLAIN
SELECT *
FROM nulls
ORDER BY num;
```

Убедимся, что строка со значением NULL окажется в выводе самой последней:

```
SELECT *
FROM nulls
ORDER BY num
OFFSET 199995;
```

Модифицируем запрос, указав, что значения NULL должны располагаться в начале выборки:

```
EXPLAIN
SELECT *
FROM nulls
ORDER BY num NULLS FIRST;
```

Задание 1. Проверьте, будет ли использоваться индекс:

```
EXPLAIN
SELECT *
FROM nulls
ORDER BY num DESC NULLS FIRST;
```

Задание 2. Модифицируйте команду создания индекса так, чтобы он использовался при выполнении выборки:

```
SELECT *
FROM nulls
ORDER BY num NULLS FIRST;
```

Задание 3. Выполните аналогичные эксперименты, задавая убывающий порядок сортировки с помощью DESC и изменяя расположение значений NULL с помощью NULLS FIRST и NULLS LAST. Проверьте фактическое время выполнения команд с помощью EXPLAIN ANALYZE.

На рисунках 30-32 показан прогресс работы.

```
demo=# CREATE TABLE nulls AS
demo-# SELECT num::integer, 'TEXT' || num::text AS txt
demo-# FROM generate_series( 1, 200000 ) AS gen_ser( num );
SELECT 200000
demo=#
demo=# CREATE INDEX nulls_ind
demo-# ON nulls ( num );
CREATE INDEX
demo=#
demo=# INSERT INTO nulls
demo-# VALUES ( NULL, 'TEXT' );
INSERT 0 1
demo=#
demo=# EXPLAIN
demo-# SELECT *
demo-# FROM nulls
demo-# ORDER BY num;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using nulls_ind on nulls  (cost=0.42..9556.42 rows=200000 width=36)
(1 строка)


demo=#
demo=# SELECT *
demo-# FROM nulls
demo-# ORDER BY num
demo-# OFFSET 199995;
  num   |    txt
--------+------------
 199996 | TEXT199996
 199997 | TEXT199997
 199998 | TEXT199998
 199999 | TEXT199999
 200000 | TEXT200000
        | TEXT
(6 строк)


demo=# EXPLAIN
demo-# SELECT *
demo-# FROM nulls
demo-# ORDER BY num NULLS FIRST;
                             QUERY PLAN
---------------------------------------------------------------------
 Sort  (cost=26168.14..26668.14 rows=200000 width=36)
   Sort Key: num NULLS FIRST
   ->  Seq Scan on nulls  (cost=0.00..3088.00 rows=200000 width=36)
(3 строки)
```

Рисунок 30 – Прогресс работы

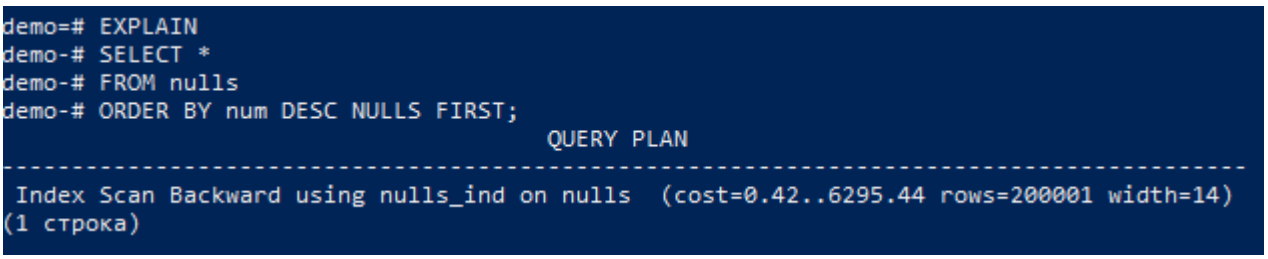

Рисунок 31 – Проверка гипотезы

Слово Backward в плане означает, что поиск по таблице индексов будет производиться снизу вверх. По умолчанию «PostgreSQL» создаёт таблицы индексов в возрастающем порядке.

На рисунке 32 показано под задание 2.

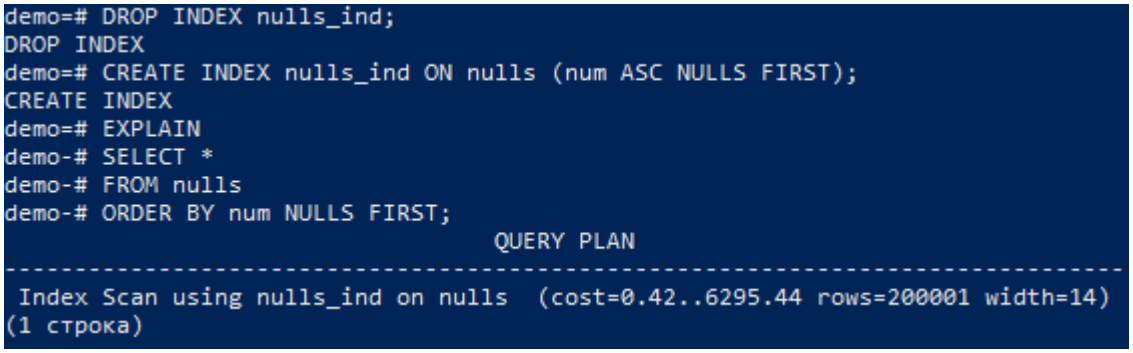

Рисунок 32 – Решение для использования индекса

```
demo=# EXPLAIN ANALYZE
demo-# SELECT *
demo-# FROM nulls
demo-# ORDER BY num ASC NULLS FIRST;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Index Scan using nulls_ind on nulls  (cost=0.42..6295.44 rows=200001 width=14) (actual time=0.023..19.463 rows=200001 loops=1)
 Planning Time: 0.067 ms
 Execution Time: 23.236 ms
(3 строки)


demo=# EXPLAIN ANALYZE
demo-# SELECT *
demo-# FROM nulls
demo-# ORDER BY num ASC NULLS LAST;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Sort  (cost=24117.25..24617.25 rows=200001 width=14) (actual time=31.591..43.638 rows=200001 loops=1)
   Sort Key: num
   Sort Method: external merge  Disk: 4800kB
   ->  Seq Scan on nulls  (cost=0.00..3088.01 rows=200001 width=14) (actual time=0.011..7.622 rows=200001 loops=1)
 Planning Time: 0.060 ms
 Execution Time: 48.184 ms
(6 строк)


demo=# EXPLAIN ANALYZE
demo-# SELECT *
demo-# FROM nulls
demo-# ORDER BY num DESC NULLS FIRST;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Sort  (cost=24117.25..24617.25 rows=200001 width=14) (actual time=45.103..57.040 rows=200001 loops=1)
   Sort Key: num DESC
   Sort Method: external merge  Disk: 4800kB
   ->  Seq Scan on nulls  (cost=0.00..3088.01 rows=200001 width=14) (actual time=0.012..7.553 rows=200001 loops=1)
 Planning Time: 0.060 ms
 Execution Time: 61.516 ms
(6 строк)


demo=# EXPLAIN ANALYZE
demo-# SELECT *
demo-# FROM nulls
demo-# ORDER BY num DESC NULLS LAST;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Index Scan Backward using nulls_ind on nulls  (cost=0.42..6295.44 rows=200001 width=14) (actual time=0.056..19.761 rows=200001 loops=1)
 Planning Time: 0.086 ms
 Execution Time: 23.554 ms
(3 строки)
```

Рисунок 33 – Эксперименты с запросами

2.15 Задание 15

Обратитесь к запросам в главе 6. Выполните команду EXPLAIN для всех этих запросов и ознакомьтесь с планами, которые создаст планировщик. В планах могут встречаться наименования методов, которые не были рассмотрены в тексте главы, однако они должны быть вам интуитивно понятны.

На рисунках 34-42 показан прогресс работы.

```
demo=# EXPLAIN
demo-# SELECT * FROM aircrafts
demo-# WHERE model NOT LIKE 'Airbus%'
demo-# AND model NOT LIKE 'Boeing%';
                                QUERY PLAN
---------------------------------------------------------------------------
 Seq Scan on aircrafts  (cost=0.00..1.14 rows=9 width=52)
   Filter: ((model !~~ 'Airbus%'::text) AND (model !~~ 'Boeing%'::text))
(2 строки)
```

Рисунок 34 – EXPLAIN Запросов, часть 1

```
demo=# EXPLAIN
demo-# SELECT DISTINCT timezone FROM airports ORDER BY 1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=3.82..3.86 rows=17 width=15)
   Sort Key: timezone
   ->  HashAggregate  (cost=3.30..3.47 rows=17 width=15)
         Group Key: timezone
         ->  Seq Scan on airports  (cost=0.00..3.04 rows=104 width=15)
(5 строк)
```

Рисунок 35 – EXPLAIN Запросов, часть 2

```
demo=# EXPLAIN
demo-# SELECT model, range,
demo-# CASE WHEN range < 2000 THEN 'Ближнемагистральный'
demo-# WHEN range < 5000 THEN 'Среднемагистральный'
demo-# ELSE 'Дальнемагистральный'
demo-# END AS type
demo-# FROM aircrafts
demo-# ORDER BY model;
                              QUERY PLAN
---------------------------------------------------------------------
 Sort  (cost=1.28..1.30 rows=9 width=68)
   Sort Key: model
   ->  Seq Scan on aircrafts  (cost=0.00..1.14 rows=9 width=68)
(3 строки)
```

Рисунок 36 – EXPLAIN Запросов, часть 3

```
demo=# EXPLAIN
demo-# SELECT a.aircraft_code, a.model, s.seat_no, s.fare_conditions
demo-# FROM seats AS s
demo-# JOIN aircrafts AS a
demo-# ON s.aircraft_code = a.aircraft_code
demo-# WHERE a.model ~ '^Cessna'
demo-# ORDER BY s.seat_no;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Sort  (cost=23.28..23.65 rows=149 width=59)
   Sort Key: s.seat_no
   ->  Nested Loop  (cost=5.43..17.90 rows=149 width=59)
         ->  Seq Scan on aircrafts a  (cost=0.00..1.11 rows=1 width=48)
               Filter: (model ~ '^Cessna'::text)
         ->  Bitmap Heap Scan on seats s  (cost=5.43..15.29 rows=149 width=15)
               Recheck Cond: (aircraft_code = a.aircraft_code)
               ->  Bitmap Index Scan on seats_pkey  (cost=0.00..5.39 rows=149 width=0)
                     Index Cond: (aircraft_code = a.aircraft_code)
(9 строк)
```

Рисунок 37 – EXPLAIN Запросов, часть 4

```
demo=# EXPLAIN
demo-# SELECT count( * )
demo-# FROM airports a1 CROSS JOIN airports a2
demo-# WHERE a1.city <> a2.city;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=195.34..195.35 rows=1 width=8)
   ->  Nested Loop  (cost=0.00..168.58 rows=10704 width=0)
         Join Filter: (a1.city <> a2.city)
         ->  Seq Scan on airports a1  (cost=0.00..3.04 rows=104 width=17)
         ->  Materialize  (cost=0.00..3.56 rows=104 width=17)
               ->  Seq Scan on airports a2  (cost=0.00..3.04 rows=104 width=17)
(6 строк)
```

Рисунок 38 – EXPLAIN Запросов, часть 5

```
demo=# EXPLAIN
demo-# SELECT a.aircraft_code AS a_code,
demo-# a.model,
demo-# r.aircraft_code AS r_code,
demo-# count( r.aircraft_code ) AS num_routes
demo-# FROM aircrafts a
demo-# LEFT OUTER JOIN routes r ON r.aircraft_code = a.aircraft_code
demo-# GROUP BY 1, 2, 3
demo-# ORDER BY 4 DESC;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Sort  (cost=51.31..51.49 rows=72 width=60)
   Sort Key: (count(r.aircraft_code)) DESC
   ->  HashAggregate  (cost=48.37..49.09 rows=72 width=60)
         Group Key: a.aircraft_code, r.aircraft_code
         ->  Hash Right Join  (cost=1.20..43.05 rows=710 width=52)
               Hash Cond: (r.aircraft_code = a.aircraft_code)
               ->  Seq Scan on routes r  (cost=0.00..39.10 rows=710 width=4)
               ->  Hash  (cost=1.09..1.09 rows=9 width=48)
                     ->  Seq Scan on aircrafts a  (cost=0.00..1.09 rows=9 width=48)
(9 строк)
```

Рисунок 39 – EXPLAIN Запросов, часть 6

```
demo=# EXPLAIN
demo-# SELECT r.min_sum, r.max_sum, count( b.* )
demo-# FROM bookings b
demo-# RIGHT OUTER JOIN
demo-# ( VALUES ( 0, 100000 ), ( 100000, 200000 ),
demo(# ( 200000, 300000 ), ( 300000, 400000 ),
demo(# ( 400000, 500000 ), ( 500000, 600000 ),
demo(# ( 600000, 700000 ), ( 700000, 800000 ),
demo(# ( 800000, 900000 ), ( 900000, 1000000 ),
demo(# ( 1000000, 1100000 ), ( 1100000, 1200000 ),
demo(# ( 1200000, 1300000 )
demo(# ) AS r ( min_sum, max_sum )
demo-# ON b.total_amount >= r.min_sum AND b.total_amount < r.max_sum
demo-# GROUP BY r.min_sum, r.max_sum
demo-# ORDER BY r.min_sum;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=118081.38..118081.41 rows=13 width=16)
   Sort Key: "*VALUES*".column1
   ->  HashAggregate  (cost=118081.01..118081.14 rows=13 width=16)
         Group Key: "*VALUES*".column1, "*VALUES*".column2
         ->  Nested Loop Left Join  (cost=0.00..115234.11 rows=379586 width=57)
               Join Filter: ((b.total_amount >= ("*VALUES*".column1)::numeric) AND (b.total_amount < ("*VALUES*".column2)::numeric))
               ->  Values Scan on "*VALUES*"  (cost=0.00..0.16 rows=13 width=8)
               ->  Materialize  (cost=0.00..8220.85 rows=262790 width=55)
                     ->  Seq Scan on bookings b  (cost=0.00..4339.90 rows=262790 width=55)
(9 строк)
```

Рисунок 40 – EXPLAIN Запросов, часть 7

```
demo=# EXPLAIN
demo-# SELECT arrival_city FROM routes
demo-# WHERE departure_city = 'Москва'
demo-# INTERSECT
demo-# SELECT arrival_city FROM routes
demo-# WHERE departure_city = 'Санкт-Петербург'
demo-# ORDER BY arrival_city;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Sort  (cost=85.79..85.87 rows=30 width=36)
   Sort Key: "*SELECT* 2".arrival_city
   ->  HashSetOp Intersect  (cost=0.00..85.06 rows=30 width=36)
         ->  Append  (cost=0.00..84.58 rows=189 width=36)
               ->  Subquery Scan on "*SELECT* 2"  (cost=0.00..41.23 rows=35 width=21)
                     ->  Seq Scan on routes  (cost=0.00..40.88 rows=35 width=17)
                           Filter: (departure_city = 'Санкт-Петербург'::text)
               ->  Subquery Scan on "*SELECT* 1"  (cost=0.00..42.41 rows=154 width=21)
                     ->  Seq Scan on routes routes_1  (cost=0.00..40.88 rows=154 width=17)
                           Filter: (departure_city = 'Москва'::text)
(10 строк)


demo=# EXPLAIN
demo-# SELECT arrival_city FROM routes
demo-# WHERE departure_city = 'Санкт-Петербург'
demo-# EXCEPT
demo-# SELECT arrival_city FROM routes
demo-# WHERE departure_city = 'Москва'
demo-# ORDER BY arrival_city;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Sort  (cost=85.79..85.87 rows=30 width=36)
   Sort Key: "*SELECT* 1".arrival_city
   ->  HashSetOp Except  (cost=0.00..85.06 rows=30 width=36)
         ->  Append  (cost=0.00..84.58 rows=189 width=36)
               ->  Subquery Scan on "*SELECT* 1"  (cost=0.00..41.23 rows=35 width=21)
                     ->  Seq Scan on routes  (cost=0.00..40.88 rows=35 width=17)
                           Filter: (departure_city = 'Санкт-Петербург'::text)
               ->  Subquery Scan on "*SELECT* 2"  (cost=0.00..42.41 rows=154 width=21)
                     ->  Seq Scan on routes routes_1  (cost=0.00..40.88 rows=154 width=17)
                           Filter: (departure_city = 'Москва'::text)
(10 строк)
```

Рисунок 41 – EXPLAIN Запросов, часть 8

```
demo=# EXPLAIN
demo-# SELECT avg( total_amount ) FROM bookings;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=4644.40..4644.41 rows=1 width=32)
   ->  Gather  (cost=4644.28..4644.39 rows=1 width=32)
         Workers Planned: 1
         ->  Partial Aggregate  (cost=3644.28..3644.29 rows=1 width=32)
               ->  Parallel Seq Scan on bookings  (cost=0.00..3257.82 rows=154582 width=6)
(5 строк)


demo=# EXPLAIN
demo-# SELECT max( total_amount ) FROM bookings;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=4644.39..4644.40 rows=1 width=32)
   ->  Gather  (cost=4644.28..4644.39 rows=1 width=32)
         Workers Planned: 1
         ->  Partial Aggregate  (cost=3644.28..3644.29 rows=1 width=32)
               ->  Parallel Seq Scan on bookings  (cost=0.00..3257.82 rows=154582 width=6)
(5 строк)


demo=# EXPLAIN
demo-# SELECT min( total_amount ) FROM bookings;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=4644.39..4644.40 rows=1 width=32)
   ->  Gather  (cost=4644.28..4644.39 rows=1 width=32)
         Workers Planned: 1
         ->  Partial Aggregate  (cost=3644.28..3644.29 rows=1 width=32)
               ->  Parallel Seq Scan on bookings  (cost=0.00..3257.82 rows=154582 width=6)
(5 строк)
```

Рисунок 42 – EXPLAIN Запросов, часть 9

2.16 Задание 16

В разделе документации 19.7 «Планирование запросов» приведены параметры, с помощью которых можно влиять на решения, принимаемые планировщиком. В тексте главы мы уже говорили о параметрах, управляющих выбором способа соединения наборов строк, и показали простой пример. Также было сказано и о том, что при установке значений параметров enable_hashjoin, enable_mergejoin и enable_nestloop в off не накладывается полного запрета на использование соответствующих методов. Вместо этого конкретному методу назначается очень высокая стоимость.

Давайте проведем следующий эксперимент: запретим использование всех методов соединения наборов строк и выполним запрос, в котором соединяются две таблицы:

```
SET enable_hashjoin = off;
SET enable_mergejoin = off;
SET enable_nestloop = off;
```

Запрос выводит информацию о числе мест в самолетах всех моделей:

```
EXPLAIN
SELECT a.model, count(*)
FROM aircrafts a, seats s
WHERE a.aircraft_code = s.aircraft_code
GROUP BY a.aircraft_code;
```

Обратите внимание на оценки стоимости выполнения запроса. Резкое повышение оценок происходит именно в узле, отвечающем за соединение наборов строк. Эти оценки не означают, что время выполнения запроса будет стремиться к бесконечности. С помощью команды EXPLAIN ANALYZE выполните запрос и убедитесь в этом сами.

Задание. Самостоятельно ознакомьтесь с содержанием раздела документации 19.7 «Планирование запросов», а также раздела 14.3 «Управление планировщиком с помощью явных предложений JOIN» и проведите эксперименты с запросами, приведенными в главе 6 пособия, получая различные варианты планов и сравнивая их.

Ваша задача — понять, как изменения значений этих параметров влияют на план выполнения запроса. Однако для того чтобы понимать, когда и почему нужно изменять значения конкретных параметров, правильно оценивать степень и направленность их влияния, понимать взаимосвязь параметров, требуется опыт и изучение документации.

На рисунке 43 представлен прогресс работы.

```
demo=# SET enable_hashjoin = off;
SET
demo=# SET enable_mergejoin = off;
SET
demo=# SET enable_nestloop = off;
SET
demo=# EXPLAIN
demo-# SELECT a.model, count( * )
demo-# FROM aircrafts a, seats s
demo-# WHERE a.aircraft_code = s.aircraft_code
demo-# GROUP BY a.aircraft_code;
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=10000000000.41..10000000082.43 rows=9 width=56)
   Group Key: a.aircraft_code
   ->  Nested Loop  (cost=10000000000.41..10000000075.65 rows=1339 width=48)
         ->  Index Scan using aircrafts_pkey on aircrafts a  (cost=0.14..12.27 rows=9 width=48)
         ->  Index Only Scan using seats_pkey on seats s  (cost=0.28..5.55 rows=149 width=4)
               Index Cond: (aircraft_code = a.aircraft_code)
(6 строк)
```

Рисунок 43 – Анализ запроса

```
demo=# SET enable_nestloop = on;
SET
demo=#
demo=# SET work_mem = '64MB';
SET
demo=# EXPLAIN ANALYZE
demo-# SELECT *
demo-# FROM flights
demo-# ORDER BY departure_airport;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Sort  (cost=3209.85..3292.65 rows=33121 width=63) (actual time=21.839..22.850 rows=33121 loops=1)
   Sort Key: departure_airport
   Sort Method: quicksort  Memory: 4255kB
   ->  Seq Scan on flights  (cost=0.00..723.21 rows=33121 width=63) (actual time=0.011..1.491 rows=33121 loops=1)
 Planning Time: 0.064 ms
 Execution Time: 23.806 ms
(6 строк)


demo=#
demo=# SET work_mem = '256MB';
SET
demo=# EXPLAIN ANALYZE
demo-# SELECT *
demo-# FROM flights
demo-# ORDER BY departure_airport;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Sort  (cost=3209.85..3292.65 rows=33121 width=63) (actual time=19.585..20.572 rows=33121 loops=1)
   Sort Key: departure_airport
   Sort Method: quicksort  Memory: 4255kB
   ->  Seq Scan on flights  (cost=0.00..723.21 rows=33121 width=63) (actual time=0.014..1.362 rows=33121 loops=1)
 Planning Time: 0.072 ms
 Execution Time: 21.535 ms
(6 строк)
```

Рисунок 44 – Анализ «nestloop»


```
demo=# SET enable_nestloop = on;
SET
demo=#
demo=# SET work_mem = '64MB';
SET
demo=# EXPLAIN ANALYZE
demo-# SELECT *
demo-# FROM flights
demo-# ORDER BY departure_airport;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Sort  (cost=3209.85..3292.65 rows=33121 width=63) (actual time=21.839..22.850 rows=33121 loops=1)
   Sort Key: departure_airport
   Sort Method: quicksort  Memory: 4255kB
   ->  Seq Scan on flights  (cost=0.00..723.21 rows=33121 width=63) (actual time=0.011..1.491 rows=33121 loops=1)
 Planning Time: 0.064 ms
 Execution Time: 23.806 ms
(6 строк)


demo=#
demo=# SET work_mem = '256MB';
SET
demo=# EXPLAIN ANALYZE
demo-# SELECT *
demo-# FROM flights
demo-# ORDER BY departure_airport;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Sort  (cost=3209.85..3292.65 rows=33121 width=63) (actual time=19.585..20.572 rows=33121 loops=1)
   Sort Key: departure_airport
   Sort Method: quicksort  Memory: 4255kB
   ->  Seq Scan on flights  (cost=0.00..723.21 rows=33121 width=63) (actual time=0.014..1.362 rows=33121 loops=1)
 Planning Time: 0.072 ms
 Execution Time: 21.535 ms
(6 строк)
```


Рисунок 45 – Анализ «work_mem»

```
demo=# SET random_page_cost = 4.0;
SET
demo=# EXPLAIN ANALYZE
demo-# SELECT *
demo-# FROM flights
demo-# WHERE departure_airport = 'SVO';
                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..806.01 rows=2979 width=63) (actual time=0.478..3.060 rows=2981 loops=1)
   Filter: (departure_airport = 'SVO'::bpchar)
   Rows Removed by Filter: 30140
 Planning Time: 0.064 ms
 Execution Time: 3.127 ms
(5 строк)


demo=#
demo=# SET random_page_cost = 1.0;
SET
demo=# EXPLAIN ANALYZE
demo-# SELECT *
demo-# FROM flights
demo-# WHERE departure_airport = 'SVO';
                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..806.01 rows=2979 width=63) (actual time=0.643..3.618 rows=2981 loops=1)
   Filter: (departure_airport = 'SVO'::bpchar)
   Rows Removed by Filter: 30140
 Planning Time: 0.085 ms
 Execution Time: 3.712 ms
(5 строк)
```

Рисунок 46 – Анализ «random_page_count»

2.17 Задание 17

Самостоятельно ознакомьтесь с разделом документации 14.2 «Статистика, используемая планировщиком».

Раздел «Статистика, используемая планировщиком» был изучен.

2.18 Задание 18

Команда EXPLAIN имеет опцию BUFFERS. Ознакомьтесь с ней самостоятельнопо разделу документации 14.1 «Использование EXPLAIN».

На рисунках 47-48 показан прогресс работы.

```
demo=# EXPLAIN (ANALYZE, BUFFERS)
demo-# SELECT * FROM flights WHERE departure_airport = 'SVO';
                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..806.01 rows=2979 width=63) (actual time=0.704..4.540 rows=2981 loops=1)
   Filter: (departure_airport = 'SVO'::bpchar)
   Rows Removed by Filter: 30140
   Buffers: shared hit=392
 Planning Time: 0.139 ms
 Execution Time: 4.665 ms
(6 строк)


demo=# EXPLAIN (ANALYZE, BUFFERS)
demo-# SELECT * FROM flights WHERE flight_id = 12345;
                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Index Scan using flights_pkey on flights  (cost=0.29..8.31 rows=1 width=63) (actual time=0.025..0.026 rows=1 loops=1)
   Index Cond: (flight_id = 12345)
   Buffers: shared hit=3
 Planning Time: 0.088 ms
 Execution Time: 0.042 ms
(5 строк)
```

Рисунок 47 – Анализ BUFFERS, часть 1

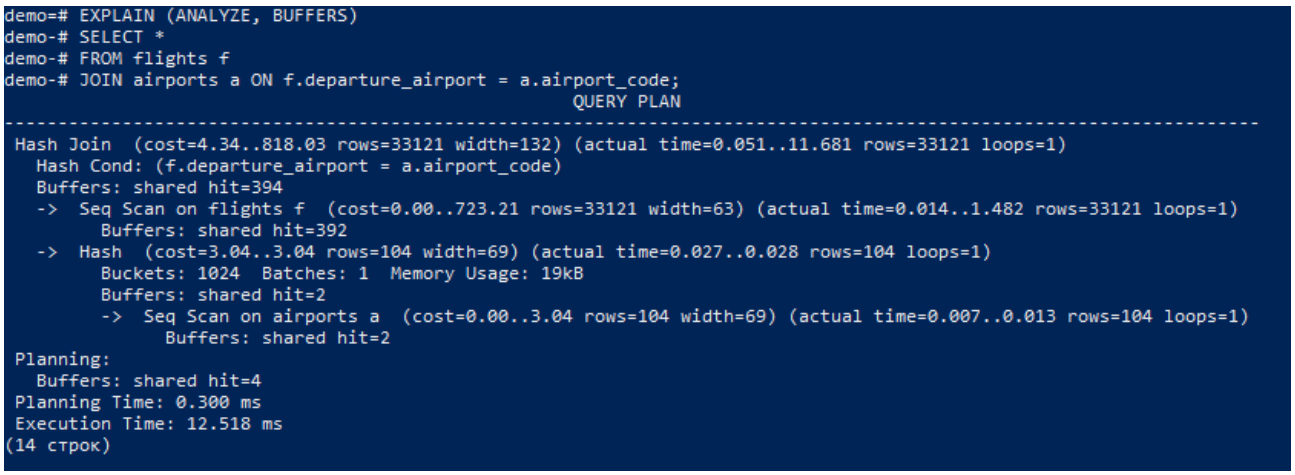

Рисунок 48 – Анализ BUFFERS, часть 2

2.19 Задание 19

При массовом вводе данных в базу данных производительность СУБД может снижаться по ряду причин. Например, при наличии индексов они обновляются при вводе каждой новой строки в таблицу, что требует дополнительных затрат ресурсов.

Для повышения производительности СУБД в подобных ситуациях в документации предлагается ряд мер. В частности, рекомендуется удаление индексов перед началом массового ввода данных и их пересоздание после завершения такого ввода.

Ознакомьтесь с этими мерами самостоятельно по разделу документации 14.4 «Наполнение базы данных».

На рисунках 49-52 показан прогресс работы.

```
demo=# CREATE TABLE test_table (
demo(#     id SERIAL PRIMARY KEY,
demo(#     value INT
demo(# );
CREATE TABLE
Время: 3,398 мс
demo=#
demo=# CREATE INDEX idx_test_table_value ON test_table(value);
CREATE INDEX
Время: 1,806 мс
demo=#
demo=# EXPLAIN ANALYZE
demo-# INSERT INTO test_table (value)
demo-# SELECT floor(random() * 100) FROM generate_series(1, 1000000);
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Insert on test_table  (cost=0.00..25000.00 rows=0 width=0) (actual time=5210.390..5210.391 rows=0 loops=1)
   ->  Subquery Scan on "*SELECT*"  (cost=0.00..25000.00 rows=1000000 width=8) (actual time=74.402..780.713 rows=1000000 loops=1)
         ->  Function Scan on generate_series  (cost=0.00..17500.00 rows=1000000 width=8) (actual time=74.344..298.688 rows=1000000 loops=1)
 Planning Time: 0.059 ms
 Execution Time: 5212.271 ms
(5 строк)


Время: 5215,899 мс (00:05,216)
demo=#
demo=# DROP INDEX idx_test_table_value;
DROP INDEX
Время: 2,257 мс
demo=#
demo=# EXPLAIN ANALYZE
demo-# INSERT INTO test_table (value)
demo-# SELECT floor(random() * 100) FROM generate_series(1, 1000000);
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Insert on test_table  (cost=0.00..25000.00 rows=0 width=0) (actual time=3314.203..3314.203 rows=0 loops=1)
   ->  Subquery Scan on "*SELECT*"  (cost=0.00..25000.00 rows=1000000 width=8) (actual time=75.665..767.396 rows=1000000 loops=1)
         ->  Function Scan on generate_series  (cost=0.00..17500.00 rows=1000000 width=8) (actual time=75.653..276.644 rows=1000000 loops=1)
 Planning Time: 0.034 ms
 Execution Time: 3315.999 ms
(5 строк)


Время: 3319,613 мс (00:03,320)
demo=#
demo=# CREATE INDEX idx_test_table_value ON test_table(value);
CREATE INDEX
Время: 990,236 мс
```

Рисунок 49 – Эксперименты, часть 1

```
demo=# CREATE TABLE test_table2 (
demo(#     id SERIAL PRIMARY KEY,
demo(#     value INT UNIQUE
demo(# );
CREATE TABLE
Время: 4,110 мс
demo=#
demo=# EXPLAIN ANALYZE
demo-# INSERT INTO test_table2 (value)
demo-# SELECT * FROM generate_series(1, 1000000);
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Insert on test_table2  (cost=0.00..15000.00 rows=0 width=0) (actual time=4444.090..4444.090 rows=0 loops=1)
   ->  Function Scan on generate_series  (cost=0.00..15000.00 rows=1000000 width=8) (actual time=73.891..606.117 rows=1000000 loops=1)
 Planning Time: 0.032 ms
 Execution Time: 4445.874 ms
(4 строки)


Время: 4449,480 мс (00:04,449)
demo=# ALTER TABLE test_table2 DROP CONSTRAINT test_table2_value_key;
ALTER TABLE
Время: 3,818 мс
demo=#
demo=# EXPLAIN ANALYZE
demo-# INSERT INTO test_table2 (value)
demo-# SELECT * FROM generate_series(1000001, 2000001);
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Insert on test_table2  (cost=0.00..15000.02 rows=0 width=0) (actual time=3338.697..3338.698 rows=0 loops=1)
   ->  Function Scan on generate_series  (cost=0.00..15000.02 rows=1000001 width=8) (actual time=73.660..649.699 rows=1000001 loops=1)
 Planning Time: 0.045 ms
 Execution Time: 3340.461 ms
(4 строки)


Время: 3342,949 мс (00:03,343)
```

Рисунок 50 – Эксперименты, часть 2

```
demo=# SET maintenance_work_mem = '64MB';
SET
Время: 0,318 мс
demo=# CREATE INDEX idx_test_table_value ON test_table(value);
CREATE INDEX
Время: 957,547 мс
demo=# DROP INDEX idx_test_table_value;
DROP INDEX
Время: 2,389 мс
demo=# SET maintenance_work_mem = '256MB';
SET
Время: 0,300 мс
demo=# CREATE INDEX idx_test_table_value ON test_table(value);
CREATE INDEX
Время: 1026,022 мс (00:01,026)
```

Рисунок 51 – Эксперименты, часть 3

```
demo=# ALTER TABLE test_table SET (autovacuum_enabled = off);
ALTER TABLE
Время: 0,932 мс
demo=# EXPLAIN ANALYZE
demo-# INSERT INTO test_table (value)
demo-# SELECT * FROM generate_series(5000001, 6000001);
                                                              QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Insert on test_table  (cost=0.00..15000.02 rows=0 width=0) (actual time=5262.794..5262.794 rows=0 loops=1)
   ->  Function Scan on generate_series  (cost=0.00..15000.02 rows=1000001 width=8) (actual time=75.169..608.669 rows=1000001 loops=1)
 Planning Time: 0.029 ms
 Execution Time: 5264.557 ms
(4 строки)


Время: 5268,500 мс (00:05,268)
demo=# ALTER TABLE test_table SET (autovacuum_enabled = on);
ALTER TABLE
Время: 0,606 мс
demo=# EXPLAIN ANALYZE
demo-# INSERT INTO test_table (value)
demo-# SELECT * FROM generate_series(6000001, 7000001);
                                                              QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Insert on test_table  (cost=0.00..15000.02 rows=0 width=0) (actual time=4382.882..4382.883 rows=0 loops=1)
   ->  Function Scan on generate_series  (cost=0.00..15000.02 rows=1000001 width=8) (actual time=74.880..602.782 rows=1000001 loops=1)
 Planning Time: 0.033 ms
 Execution Time: 4384.774 ms
(4 строки)


Время: 4388,306 мс (00:04,388)
```

Рисунок 52 – Эксперименты, часть 4

3 ЗАКЛЮЧЕНИЕ

По результатам работы был изучен теоретический материал по теме «Повышение производительности». Все поставленные цели и задачи были выполнены.