ДЕМОНСТРАЦИОННЫЕ ЗАДАНИЯ

Для получения плана конкретного запроса используется команда EXPLAIN. Без дополнительных параметров она предоставляет ожидаемый план выполнения запроса. Чтобы запрос выполнился и был получен реальный план, используется команда EXPLAIN (ANALYZE).

При анализе планов важно помнить о необходимости обновления статистики командой ANALYZE, так как именно на основе собираемых ею данных планировщик решает, какой план является наилучшим. Она выбирает некоторое количество строк случайным образом и собирает статистику по каждой колонке таблицы. Также статистика собирается на уровне отношений для каждого столбца. Если статистику не обновлять, то получаемые данные могут быть не актуальны.

После каждого задания необходимо удалять созданные индексы.

Пример №1. Найти информацию об оборудовании, у которого истек срок эксплуатации, но его взяли на объект в количестве равном 3.

В листинге 1 представлено самое простое решение данной задачи.

Листинг 1. Решение задачи (№1)

```
--EXPLAIN ANALYZE
SELECT *
FROM equipment e
JOIN equipment_on_departure eod ON e.EQ_ID = eod.EQ_ID
WHERE eod.EQ_COUNT = 3 AND
e.EQ_LIFE < 2024 - e.DO_DATE::INT;
```

Выполним запрос с помощью команды EXPLAIN ANALYZE, убрав символы комментария в первой строке листинга 1 (рис. 2).

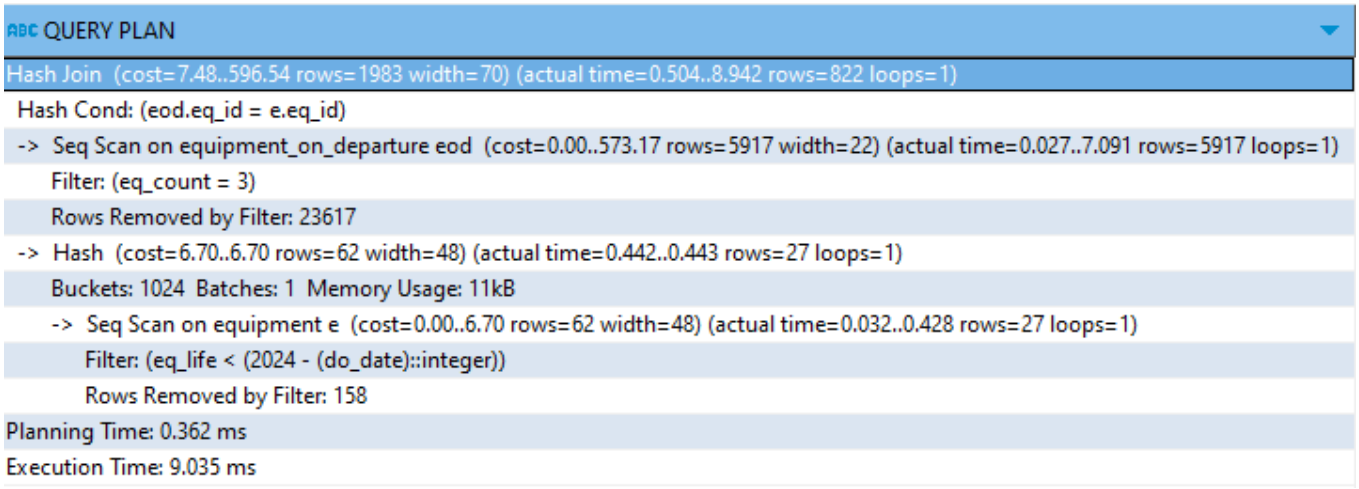


Рис. 2. Начальный план выполнения запроса (№1)

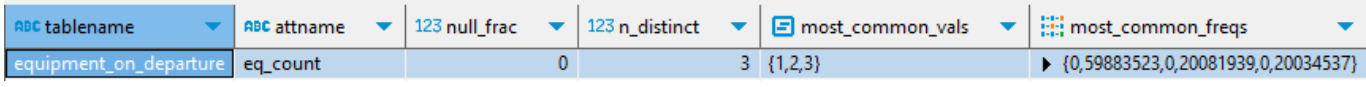
План содержит узлы Seq Scan — последовательное сканирование таблицы equipment_on_departure и equipment. В каждом из этих узлов указаны filter — условие выборки строк, использованное в предложении WHERE, и Rows Removed by Filter — количество строк, которые не попали в итоговую выборку, так как не соответствовали условию в WHERE. Таблицы соединяются с помощью соединения хешированием Hash Join: Hash Cond — условие соединения; узел Hash — построение хэш-таблицы для строк таблицы equipment, Buckets — количество корзин для построения хеш-таблицы, Batches — число пакетов (1, следовательно, данные полностью поместились в памяти), Memory Usage — использованная память. После названия каждого узла в скобках указаны стоимость (cost), кардинальность (rows), время выполнения (actual time) и количество циклов по таблице (loops). В cost и actual time первые значения — стоимость и время соответственно, потраченные на получение первой строки результата, вторые значения — стоимость и время, потраченные на получение всех строк результата. В конце плана — планируемое (Planning Time) и действительное (Execution Time) время выполнения запросов.

Для данного запроса планировщик решил, что лучше всего будет использовать последовательное сканирование таблиц. Метод Seq Scan

применяется при низкой селективности (или ее отсутствии), когда необходимо либо полностью прочитать таблицу, либо прочитать большую ее часть. Для таблицы equipment_on_departure запрос имеет среднюю селективность (из плана видно, что число выбираемых строк — 5917, а число строк, не прошедших проверку — 23617). Следовательно, выборка данных с помощью индексного дерева в данном случае будет в разы быстрее последовательного сканирования, поэтому лучшим вариантом было бы использование Bitmap Scan или Index Only Scan. Исходя из этого, можно сделать вывод о необходимости создания индекса по полю eq_count, так как оно участвует в предложении WHERE. Но посмотрев статистику по этому полю с помощью запроса из листинга 2, увидим (рисунок 3), что столбец содержит всего три уникальных значения (n_distinct): 1, 2 и 3 (most_common_vals — часто встречаемые значения, в данном случае совпадают со всеми уникальными значениями). Причем чаще всего встречается значение 1 — с вероятностью почти 60% (most_common_freqs). Тогда чтобы индекс не занимал слишком много памяти, можно создать частичный индекс, то есть индекс, в который не входит значение 1 (при выборке по условию eq_count = 1, будет использован Seq Scan вместо индексов).

Листинг 2. Статистика по столбцу eq_count

```
select tablename, attname, null_frac, n_distinct,
       most_common_vals, most_common_freqs, histogram_bounds
from pg_stats where tablename = 'equipment_on_departure'
                    and attname = 'eq_count';
```



tablename	attname	null_frac	n_distinct	most_common_vals	most_common_freqs
equipment_on_departure	eq_count	0	3	{1,2,3}	{0,59883523,0,20081939,0,20034537}

Рис. 3. Статистика по столбцу eq_count (№1)

Также необходимо создать индекс по столбцу eq_id, так как по нему происходит соединение таблиц. Вообще говоря, eq_id является внешним ключом и ссылается на первичный ключ таблицы equipment, поэтому считается рекомендованным создание индекса по такому полю.

Далее можно создать индекс для таблицы equipment. Однако индексный доступ работает, когда индексируемый столбец сравнивается с каким-то уже вычисленным значением. Но в данном же случае для выборки по условию необходимо посчитать выражение. Решением этой проблемы будет создание индекса по выражению. Перенесем поля таблицы в левую сторону неравенства, оставив справа скаляр, и создадим индекс по выражению в левой части.

В листинге 3 приведены запросы для создания индексов.

Листинг 3. Создание индексов

```
create index on equipment_on_departure(EQ_COUNT) WHERE
EQ_COUNT != 1;
create index on equipment_on_departure(EQ_ID);
create index on equipment((do_date::int + eq_life));
```

После создания индексов снова выполним запрос и посмотрим на план его выполнения (рис. 4).

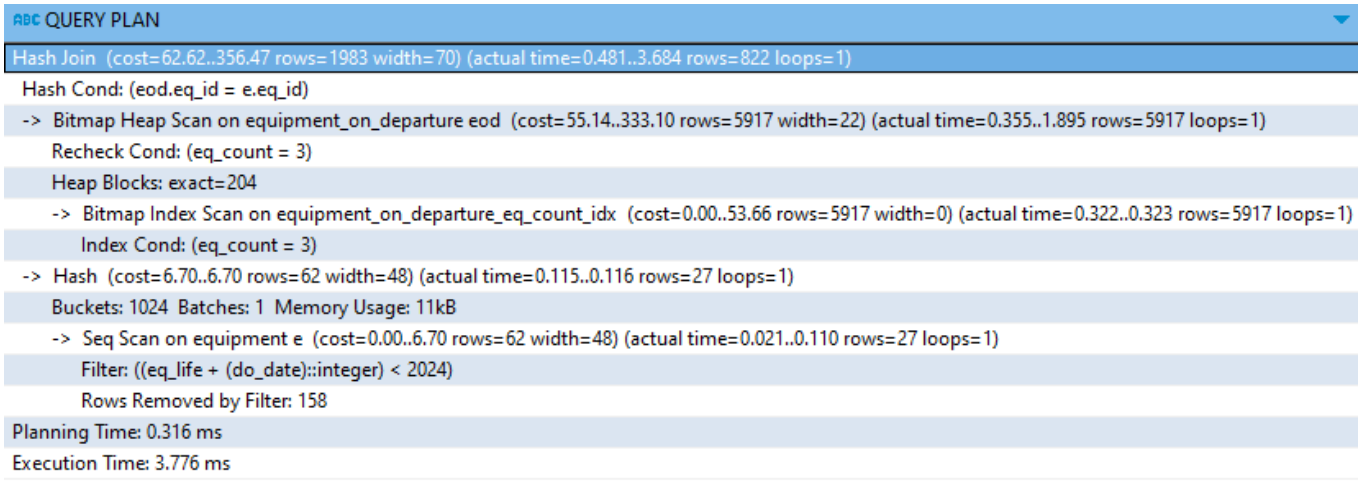


Рис. 4. План запроса после создания индексов (№1)

Планировщик решил использовать Bitmap Scan. При построении битовой карты (Bitmap Index Scan) обращение к индексу осуществляется по условию Index Cond. При сканировании битовой карты (Bitmap Heap Scan) строка Heap Blocks: exact говорит о том, что все фрагменты битовой карты построены с точностью до строк, то есть перепроверка (Recheck Cond) не выполняется (lossy в Heap Blocks отвечает за неточные фрагменты).

Итак, в связи с оптимальным доступом к данным стоимость и время выполнения запроса улучшили свои показатели: cost уменьшилась с 596.54 до 356.47, а Execution Time — с 9.035 ms до 3.776 ms. Однако так как алгоритм доступа к данным Index Only Scan является более эффективным, попытаемся добиться того, чтобы планировщик использовал именно его.

Метод Index Only Scan применяется, когда данные, хранящиеся в индексе, покрывают все потребности запроса для работы с определенной таблицей, то есть нет необходимости чтении табличных страниц. Поэтому следует создать составной индекс по полям eq_count и eq_id, так как если для этих полей используются отдельные индексы, то чтобы их значения соотнести, придется все же посмотреть таблицу. Обратим также внимание на то, что в предложении SELECT указывается «*», то есть в результате будут возвращены все столбцы двух таблиц: equipment и equipment_on_departure. Это тоже мешает планировщику использовать Index Only Scan, ведь нужно будет читать таблицу equipment_on_departure для возвращения значений всех полей (не проиндексированных в том числе). Поэтому, так как в задании сказано найти информацию о самом оборудовании, данных из таблицы equipment достаточно для получения результата. Учитывая все вышесказанное, создадим индекс и скорректируем запрос (листинг 4).

Листинг 4. Корректировка запроса

```
create index on EQUIPMENT_ON_DEPARTURE(EQ_ID,EQ_COUNT);

--EXPLAIN ANALYZE
SELECT e.eq_id, e.eq_name, e.do_date, e.eq_life, e.eq_cost,
e.eq_count
FROM equipment e
JOIN equipment_on_departure eod ON e.EQ_ID = eod.EQ_ID
WHERE eod.EQ_COUNT = 3 and
e.do_date::int + e.eq_life < 2024;
```

Получим план выполнения запроса из листинга 4 (рисунок 5).

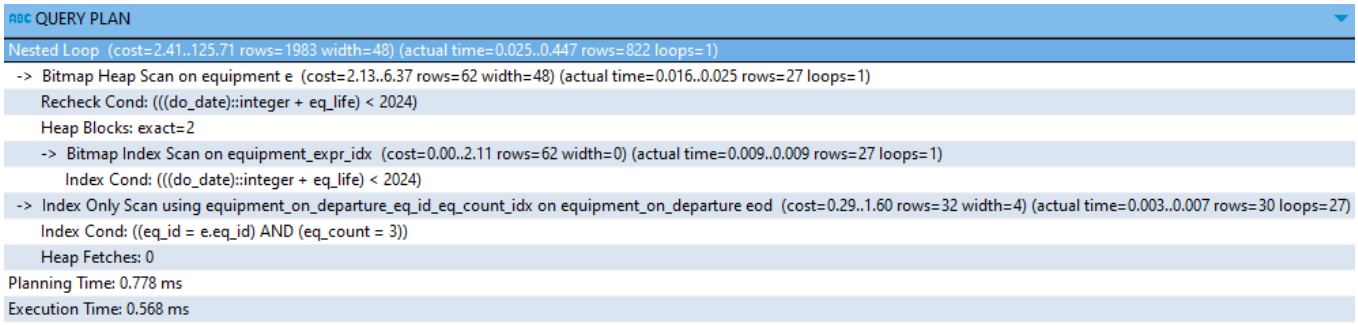


Рис. 5. План скорректированного запроса (№1)

Мы добились того, что планировщик построил план с применением сканирования только по индексу к таблице equipment_on_departure, объединив два условия благодаря составному индексу по столбцам в них (Heap Fetches показывает, сколько версий строк было проверено с помощью таблицы). Благодаря эффективному доступу к строкам, а, следовательно, оптимальной выборке данных (количество выбранных данных из таблицы equipment_on_departure сильно уменьшилось), планировщик использовал уже другой алгоритм соединения таблиц — метод вложенных циклов (Neasted Loop), который является менее затратным (для небольших наборов данных), чем Hash Join.

Время выполнения сократилось до 0.568 ms, а стоимость — до 125.71. Мы убедились в том, что индекс ускоряет выборку данных и является хорошим способом увеличить производительность запроса.

Но выполнение запроса можно сделать еще более эффективным. Заметим, что проверка условия по соединенной таблице equipment_on_departure выполняется всегда, независимо от истинности других условий. Таблица оборудования на выезде содержит около 27 тыс. строк, на соединении с ней будет потрачено много ресурсов. Сделаем так, чтобы условие по этой таблице проверялось только в случае необходимости, и «спрячем» «тяжелый» запрос в CASE (листинг 5).

Листинг 5. Запрос с использованием CASE (№1)

```
--EXPLAIN ANALYZE
SELECT *
FROM equipment e
WHERE
     case
          when (e.EQ_LIFE < 2024::INT - e.DO_DATE::INT) THEN
          EXISTS(select NULL
          from equipment_on_departure eod
          WHERE e.EQ_ID = eod.EQ_ID and
          eod.EQ_COUNT = 3
          limit 1
          )
     end;
```

В данном случае следует писать «*», так как таблица equipment не соединяется с таблицей equipment_on_departure в основном запросе, поэтому и в результате будут только столбцы из первой таблицы. В подзапросе используются только индексируемые столбцы, необходимость чтения таблиц отсутствует, и, следовательно, сканирование только по индексу становится возможным.

Также стоит сказать, что индекс по выражению лучше удалить, так как он не будет задействован. В целом, применение такого индекса должно быть обусловлено частым использованием конкретного выражения в запросах или постоянным выполнением определенного запроса, использующего данный индекс, в функционирующей базе.

На рисунке 6 представлен план измененного запроса.

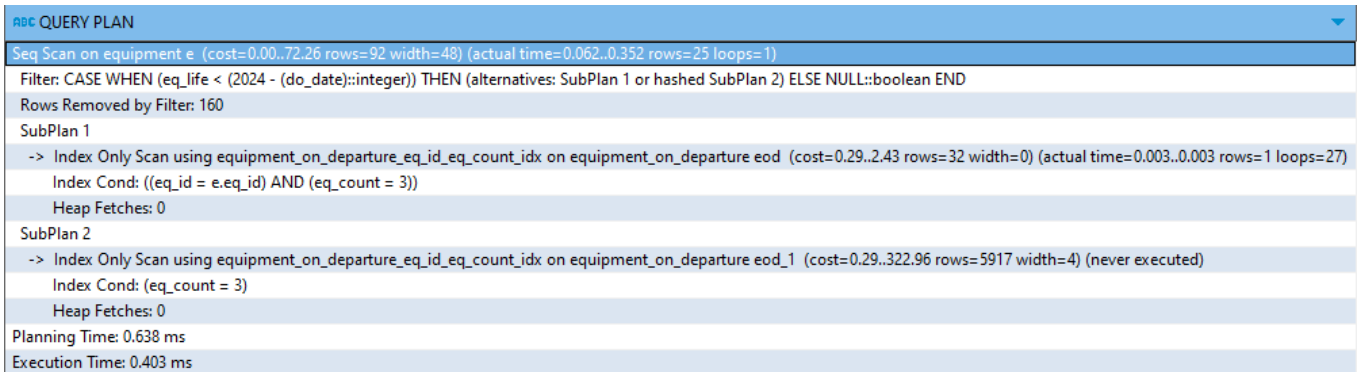


Рис. 6. План измененного запроса (№1)

Узел SubPlan — план, вызванный для выполнения подзапроса с данными, зависящими от текущей строки. Часть плана alternatives: SubPlan 1

or hashed SubPlan 2 говорит о том, что оптимизатор планирует две альтернативы: либо выполнить SubPlan 1 для каждой найденной строки, либо выполнить SubPlan 2 один раз, построить хэш-таблицу на основе результата и исследовать этот хэш для каждой найденной строки. Оптимизатор выбирает наилучший вариант и в данном случае он решил, что это SubPlan 1, использующий, как и ожидалось, Index Only Scan. Причем не примененный план помечен как never executed. Можно заметить, что loops=27 — это количество раз, которое условие в CASE проходило (27 строки из equipment было отобрано) и выполнялся подзапрос. Индекс по выражению не использовался, потому что в данном случае выражение, по которому происходит выборка из таблицы equipment — это вся конструкция CASE (невозможно создать такой индекс).

По сравнению с планом скорректированного запроса, после добавления CASE время выполнения уменьшилось с 0.568 ms до 0.403 ms, а стоимость — с 125.71 до 72.26. Сравнивая с начальным запросом, Execution Time сократилось в 22 раза, а cost почти в 8 раз.

Пример №2. Найти список идентификаторов объектов с суммой выполненных услуг больше 50000 рублей.

Решение пункта задания в листинге 6.

Листинг 6. Решение задачи (№2)

```
--explain analyze
select fac_id
from departure
where dep_sum > 50000;
```

Выполним запрос с помощью команды EXPLAIN ANALYZE, убрав символы комментария в первой строке листинга 6 (рис. 7).

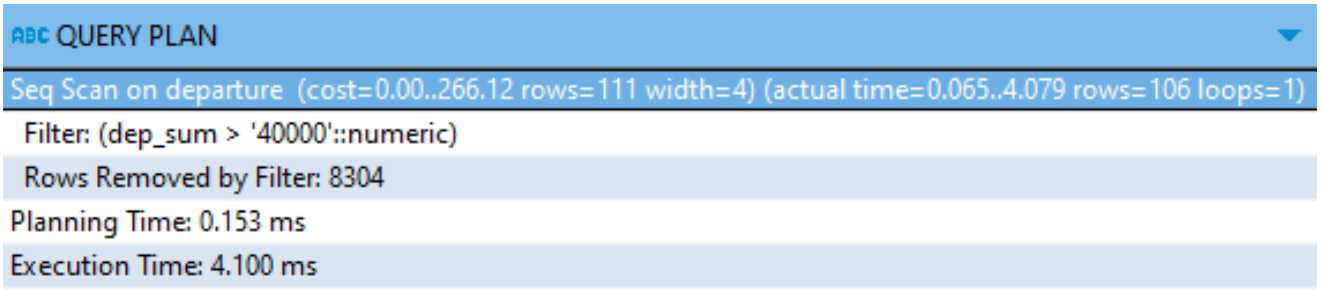


Рис. 7. Начальный план выполнения запроса (№2)

Запрос имеет среднюю селективность, следовательно, лучшим вариантом было бы использование Bitmap Scan, а еще лучше — Index Only Scan. Создав индекс по полю dep_sum, мы добьемся использования сканирования по битовой карте. Но как уже было рассмотрено раннее, для применения сканирования только по индексу необходимо, чтобы индекс был покрывающим. Эту проблему решают include-индексы: при создании индекса добавляется предложение INCLUDE, в котором указывается имя неключевого столбца. Его значения хранятся вместе с индексом, но сам он таковым не является. Создание include-индекса приведено в листинге 7.

Листинг 7. Создание include-индекса (№2)

```
create index on departure(dep_sum) include (fac_id);
```

План решения задачи теперь будет иметь вид как на рисунке 8.

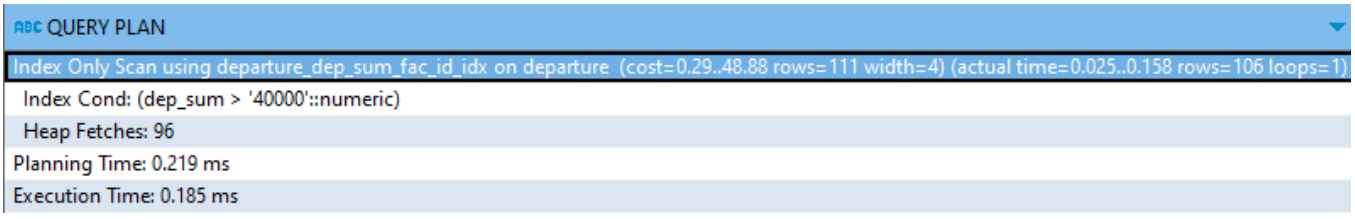


Рис. 8. План после создания include-индекса (№2)

Оправдывая ожидания, значения стоимости и времени выполнения уменьшились: Execution Time в 11 раз, cost в 12 раз.

Пример №3. Сравнить планы запросов, использующих и не использующих вычисляемые столбцы на примере столбца dep_sum.

Поле dep_sum таблицы departure вычисляется на основе данных, хранящихся в ней, а также в таблицах facility и service. Из таблицы departure берется информация о количестве сотрудников (emp_cnt) и времени работы на объекте (dep_time), из таблицы facility — расстояние до населенного пункта, в котором находится объект, из service — цена для всех перечисленных параметров.

Напишем запрос, который вычисляет сумму работ по каждому выезду (листинг 8).

Листинг 8. Вычисление суммы работ (№3)

```
     --explain analyze
     select d.dep_id, f.place,
     case
               when d.emp_cnt = 1 and d.dep_time > '01:00:00'
then round((
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 1) +
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 4) *
          ((extract(hour from d.dep_time)::numeric - 1) +
extract(minutes from d.dep_time)::numeric/60) +
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 5) *
f.distance), 2)
               when d.emp_cnt = 1 and d.dep_time <= '01:00:00'
then round((
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 1) +
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 5) *
f.distance), 2)
               when d.emp_cnt = 2 and d.dep_time > '01:00:00'
then round((
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 2) +
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 4) *
          ((extract(hour from d.dep_time)::numeric - 1) +
extract(minutes from d.dep_time)::numeric/60) *
         d.emp_cnt +
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 5) *
f.distance), 2)
               when d.emp_cnt = 2 and d.dep_time <= '01:00:00'
then round((
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 2) +
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 5) *
f.distance), 2)
               when d.emp_cnt > 2 and d.dep_time > '01:00:00'
then round((
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 2) +
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 3) *
(d.emp_cnt - 2) +
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 4) *
          ((extract(hour from d.dep_time)::numeric - 1) +
extract(minutes from d.dep_time)::numeric/60) *
         d.emp_cnt +
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 5) *
f.distance), 2)
               when d.emp_cnt > 2 and d.dep_time <= '01:00:00'
then round((
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 2) +
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 3) *
(d.emp_cnt - 2) +
          (SELECT SERV_COST FROM SERVICE WHERE SERV_ID = 5) *
f.distance), 2)
          end as sum_dep
          from departure d
          join facility f using(fac_id);
```

План выполнения этого запроса слишком длинный, поэтому на рисунке 9 показаны его начало и конец.

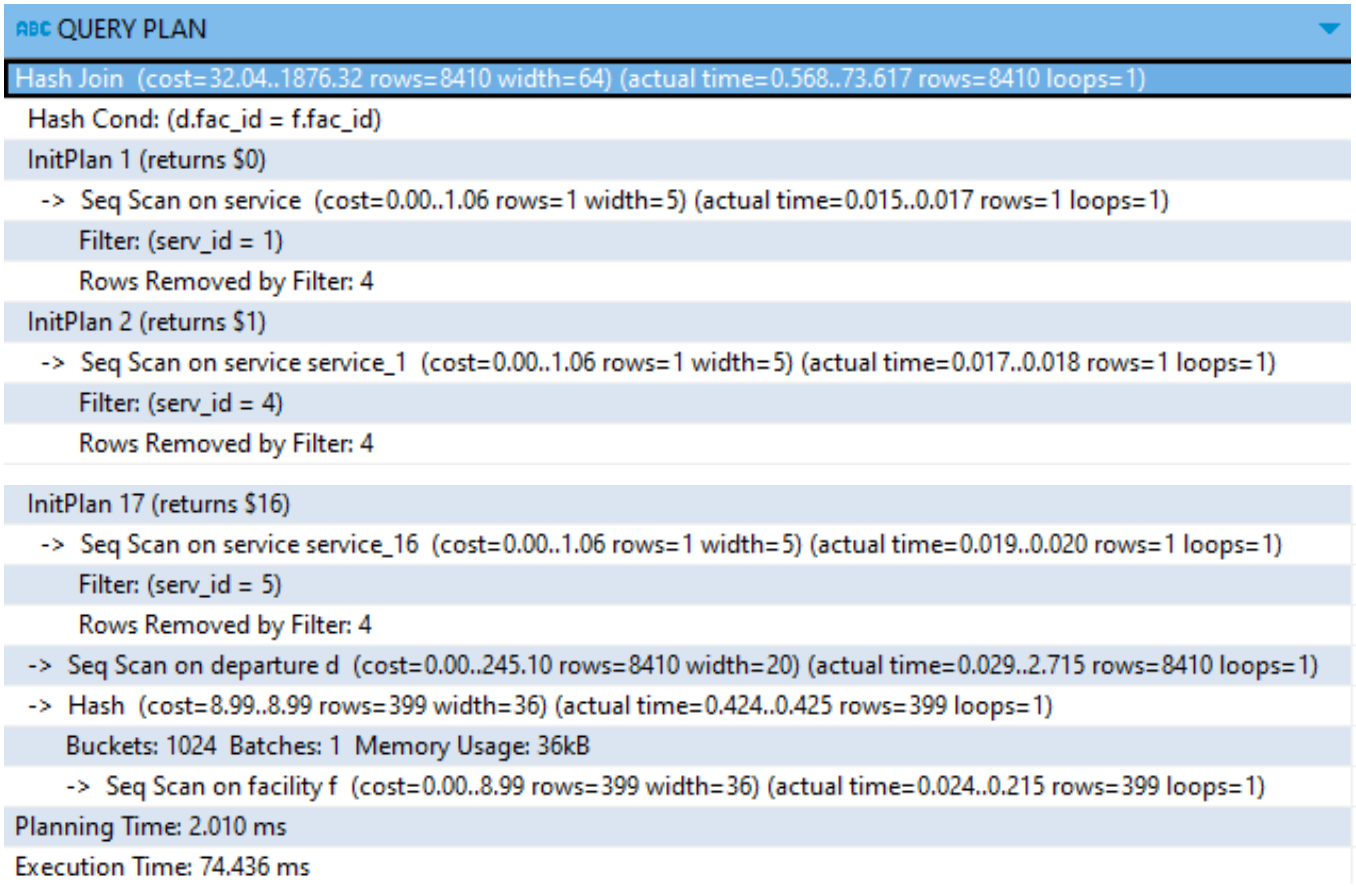


Рис. 9. План запроса вычисления суммы (№3)

В запросе присутствуют InitPlan — планы, которые применяется, когда часть запроса должна выполниться перед остальными и не зависит от них. Планировщик не знает, что будет возвращено из подзапроса — рядом с InitPlan (или, например, SubPlan) в скобках указано return $1, return $2 и т.д.

Для таблиц facility и departure применялось последовательное сканирование, так как должна быть просмотрена вся таблица. Поэтому создание и использование индексов здесь будет излишним. Учитывая также оценки кардинальности узлов, можно сказать, что запрос написан оптимально.

Теперь найдем сумму работ по каждому выезду, но используя уже вычисленный столбец dep_sum таблицы departure (листинг 9).

Листинг 9. Запрос с использованием вычисляемого столбца (№3)

```
explain analyze
select d.dep_id,f.place, d.dep_sum
from departure d
join facility f using(fac_id);
```

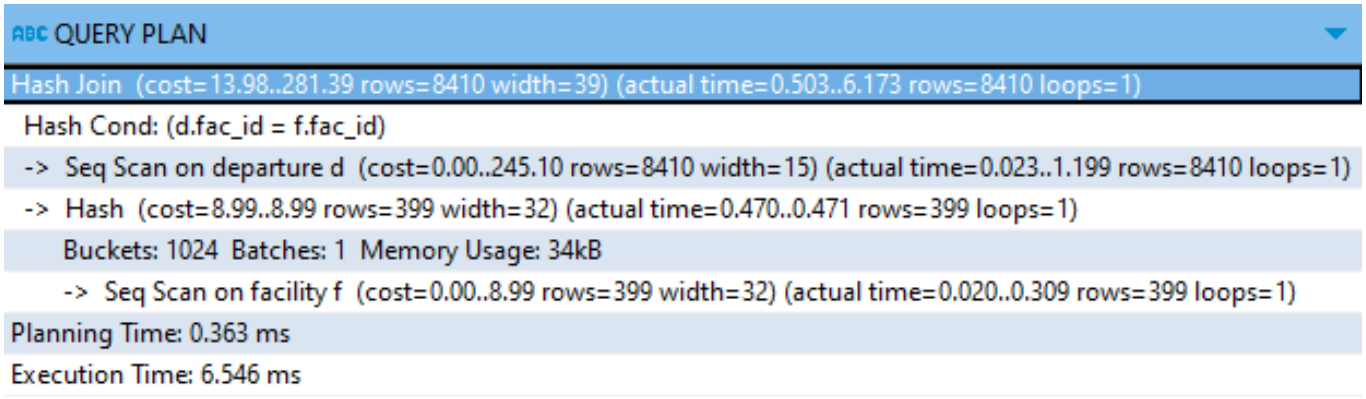


Рис. 10. План запроса с использованием вычисляемого столбца (№3)

На рисунке 10 представлен план запроса с использованием столбца dep_sum.

Сравнивая планы можно сделать вывод, что наличие вычисляемого столбца помогло ускорить выполнения запроса. При подсчете суммы запрос выполнялся за 74.436 ms и его стоимость была 1876.32, при использовании же столбца dep_sum время выполнения сократилось до 6.546 ms, а стоимость — до 281.39.

Конечно, создание вычисляемого столбца приобретает смысл, когда на получение результата его вычисления необходимо много ресурсов. В ином случае на значения производительности это не повлияет, а только создаст ненужную избыточность данных. Тем не менее, при сложных вычислениях этот прием позволяет экономить время, тем самым оптимизируя работу системы.

Еще большего ускорения работы с такими столбцами можно добиться созданием индекса по ним. В данной задаче при вычислении суммы индекс по выражению будет не очень хорошей идеей хотя бы потому, что придется создавать два почти одинаковых сложных индекса. А использование индекса на столбце dep_ sum будет оптимальным вариантом, так как значения уже вычислены и поле dep_sum часто используется в предложениях WHERE.

Пример №4. Найти идентификаторы выездов в 2020 году, которые длились 6 и более часов. Дан запрос (листинг 10), который нужно оптимизировать.

Листинг 10. Запрос из условия задания (№4)

```
--explain analyze
with time_year as (
select d.dep_id,
case
 when extract(year from d.dep_date) = 2020 and
     extract(hour from d.dep_time) = 6 then 1
 when extract(year from d.dep_date) = 2020 and
   extract(hour from d.dep_time) > 6 then 10
else 0
end as report
from departure d )
select *
from time_year
where report != 0;
```

На рисунке 11 представлен план выполнения запроса из листинга 10 (так как выражение CASE длинное, оно не поместилось на рисунок).

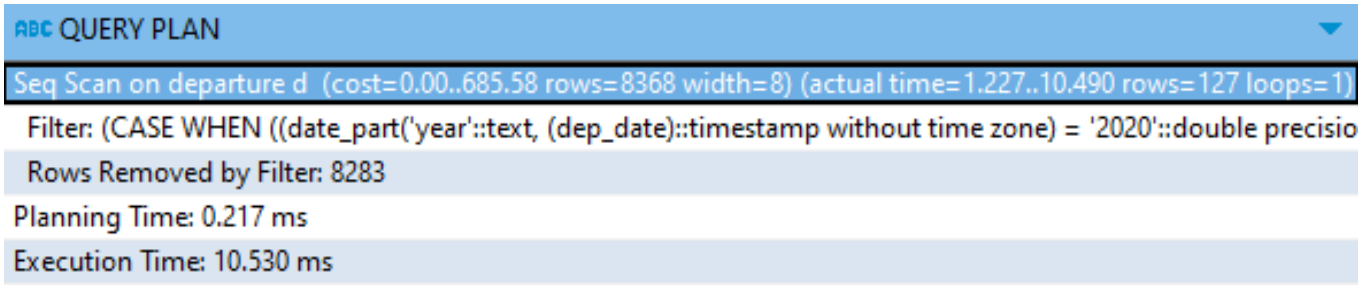


Рис. 11. План запроса из условия задания (№4)

Важно заметить сильную разницу между расчетным и действительным значениями кардинальности. При написании и оптимизации запроса необходимо сделать так, чтобы определенный узел плана не просто работал быстро, а чтобы количества строк в нем были сопоставимы.

Анализируя запрос, можно сказать, что, так как таблица departure содержит большой объем данных, фильтрация и вычисления в CTE могут оказаться ресурсоемкими, особенно если они выполняются часто. Одним из принципов оптимизации является упрощение запроса. Следуя ему, запрос можно переписать, не используя CTE и CASE, так как в данном случае их применение является избыточным (листинг 11).

Листинг 11. Упрощение запроса (№4)

```
--explain analyze
select d.dep_id
from departure d
where extract(year from d.dep_date) = 2020
and extract(hour from d.dep_time) >= 6;
```

На рисунке 12 показан план выполнения запроса из листинга 11. Видно, что стоимость уменьшилась примерно в 2 раза, а время выполнения в 2.5 раза. Но все еще возможно улучшить эти показатели.

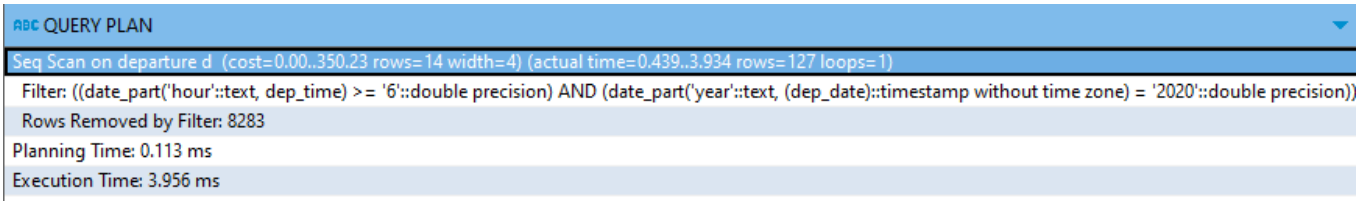


Рис. 12. План выполнения измененного запроса (№4)

Ситуация такая же как и в предыдущих задачах: последовательное сканирование при средней селективности данных. Соответственно может показаться, что опять нужно создать индекс по полям dep_date и dep_time. Но в данном задании это решение будет спорным. Следует отметить, что индексы используются в основном для запросов к отсортированным данным, а набор функций, которые могут быть применены на самих индексах, сильно ограничен (EXTRACT не поддерживается). Поэтому создание индекса на весь столбец dep_date и столбец dep_time становится бессмысленным. В этом случае эффективнее будет создать индекс по выражению (листинг 12).

Листинг 12. Индекс по выражению (№4)

```
create index on departure(extract(year from dep_date));
create index on departure(extract(hour from dep_time));
```

Снова получим план выполнения запроса (рис. 13).

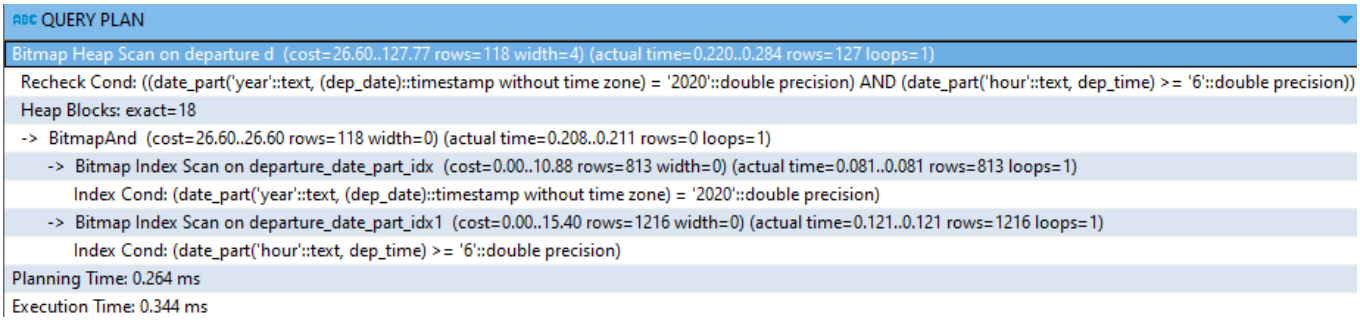


Рис. 13. План выполнения измененного запроса (№4)

После создания индекса метод доступа к данным изменился на Bitmap Scan, и ощутимо уменьшилась стоимость (была 350.23, стала 127.77), как и время выполнения запроса (было 3.956, стало 0.344). Поэтому можно сделать вывод, что создание индекса по выражению поспособствовало ускорению работы запроса.

Однако стоит сказать, что индексы не являются решением всех проблем, так как за ускорение доступа к данным приходится «платить» памятью. Поэтому приведем другой способ оптимизации запроса. Для этого рассмотрим такое понятие как категории изменчивости функций.

Выделяют 3 категории изменчивости функции:

- VOLATILE (изменчивая) функция может делать все, что угодно, поэтому планировщик не может сделать предположений о ее поведении. Вследствие этого функция выполняется каждый раз для каждой строки исходного множества даже с учетом того, что аргументы одинаковые;
- STABLE (стабильная) функция ищет, но не меняет данные в БД. Не выполняется на этапе планирования, но вычисляется один раз для одних и тех же аргументов в рамках одного запроса. Пользуется информацией о локалях и временной зоны сервера;
- IMMUTABLE (постоянная) функция не ищет и не меняет данные в БД. Возвращает один и тот же результат при тех же значениях аргумента, поэтому может быть выполнена на этапе планирования. Не использует данные временных зон и локалях сервера.

Информацию о спецификаторах доступных в PostgreSQL функций можно найти в файле pg_proc.

Итак, make_date и make_time — неизменяемые функции, поэтому могут быть выполнены во время планирования. Изменим запрос, заменив extract на make_date (листинг 13).

Листинг 13. Итоговый запрос (№4)

```
--explain analyze
select d.dep_id
from departure d
where d.dep_date between make_date(2020, 1, 1)
                         and make_date(2020, 12, 31)
and d.dep_time >= make_time(6, 0, 0);
```

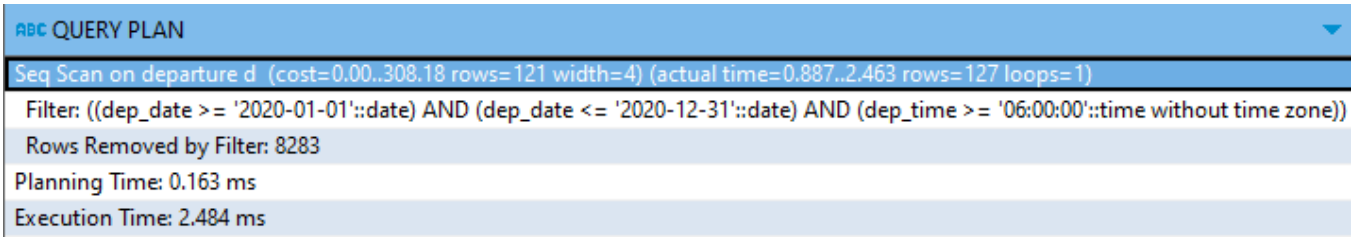


Рис. 14. План выполнения итогового запроса (№4)

На этапе планирования планировщик понимает, какой диапазон дат указан в запросе. Применив IMMUTABLE-функцию, мы добились ускорения в среднем в 1.5 раза (рис. 14). Также, так как мы убрали вычисляемые значения, планировщик может использовать статистику, которая есть по полям, следовательно, расчетное количество строк стало сопоставимо с действительным.

Стоит сказать, что использование индексов по выражению все же имело бы меньшую стоимость и время выполнения. Но создание такого индекса может стать полезным, только если в функционирующей базе запросы часто используют обращение к выделяемым из столбца данным, например, году из даты и часу из времени, как в данном случае.

Пример №5. Найти список выездов, сумма по которым превышает 80 тыс. руб. Вывести строки в порядке возрастания суммы.

Решение задачи представлено в листинге 14.

Листинг 14. Решение задачи (№5)

```
--explain analyze
SELECT *
FROM departure
where dep_sum > 80000
order by dep_sum;

create index on departure(dep_sum);
```

Выполним запрос из листинга 14 и получим план его выполнения (рис. 15).

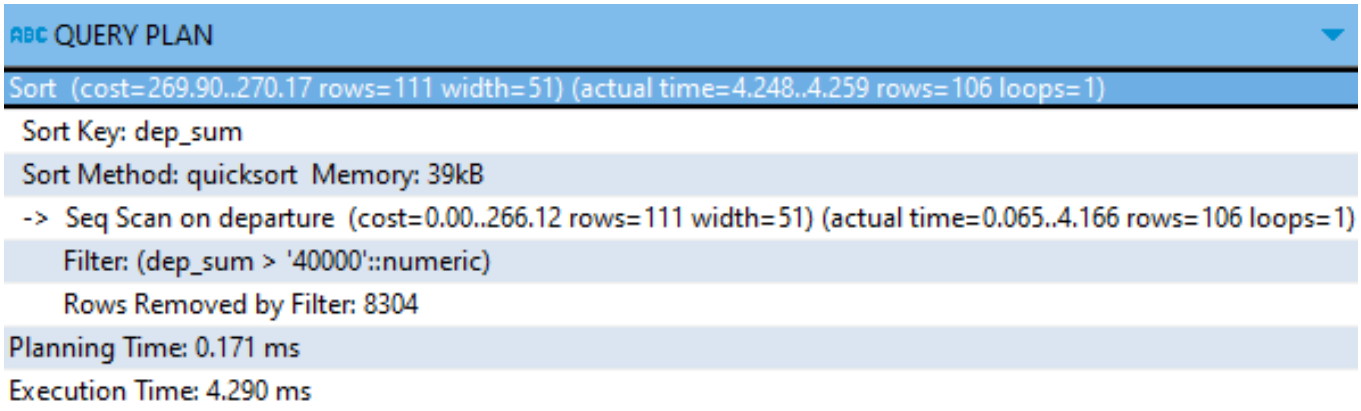


Рис. 15. План выполнения запроса-решения (№5)

Узел Sort — сортировка данных, Sort Key — поле, по которому производилась сортировка, Sort Method — алгоритм сортировки (в данном случае быстрая сортировка quicksort), Memory — память, использованная для сортировки.

Попробуем создать индекс по полю dep_sum (последняя строка из листинга 14), чтобы данные из индекса возвращались уже отсортированными и узел Sort не использовался. Снова посмотрим на план выполнения запроса из листинга 14 (рис. 16).

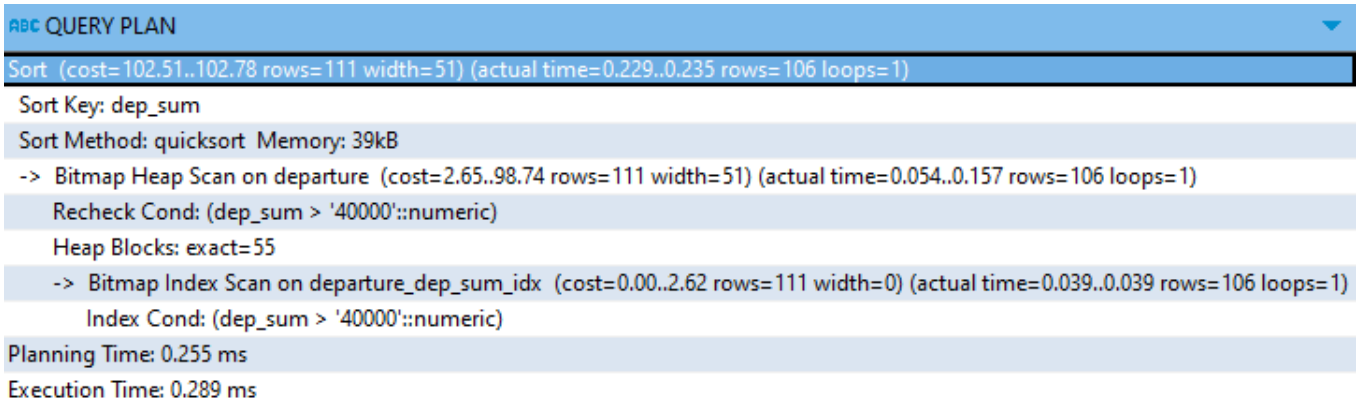


Рис. 16. План выполнения запроса после создания индекса (№5)

Несмотря на то, что упорядочивание производится по индексируемому столбцу, все равно используется явная сортировка. Индекс применяется для выборки данных по условию. Но производительней было бы применить его и для сортировки. Решить эту проблему может с помощью кластеризации по индексу (листинг 15). Она упорядочивает строки так же, как и индекс.

Листинг 15. Кластеризация (№5)

```
CLUSTER departure USING departure_dep_sum_idx;
ANALYZE departure;
```

Снова выполним запрос из листинга 14 и получим план его выполнения (рис. 17).

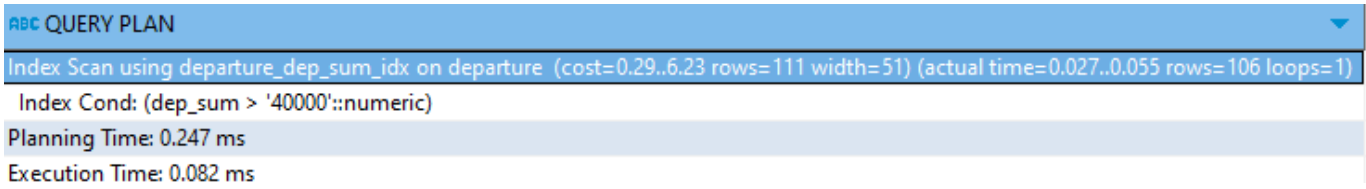


Рис. 17. План выполнения запроса после кластеризации (№5)

После кластеризации для упорядочивания данных использовалось только индексное сканирование, а сканирование по битовой карте и явная сортировка стали излишними для выполнения запроса. Время выполнения улучшилось почти в 3 раза, а стоимость — примерно в 15 раз.

Однако, заметим, что если данные упорядочены, то и при написании запроса можно опускать предложение ORDER BY, так как они и так будут возвращаться отсортированными. Удалим последнюю строку запроса и выполним его снова, получив план на рисунке 18.

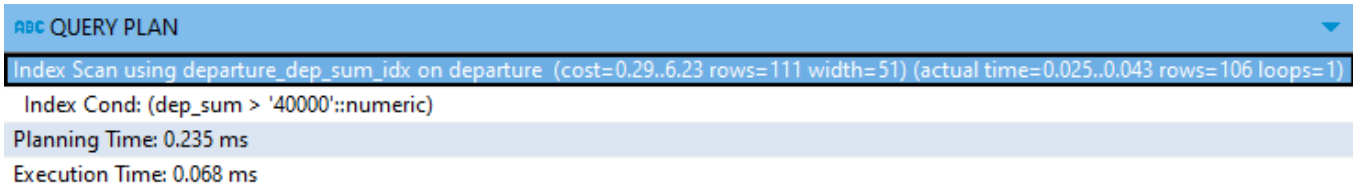


Рис. 18. Исключение ORDER BY из запроса (№5)

Время выполнения уменьшилось (с 0.082 до 0.068). Более явно польза этого замечания будет видна, если так же сравнить планы запроса, но без предложения WHERE, то есть случай, когда надо отсортировать всю таблицу. Тогда планы запросов будут выглядеть так, как показано на рисунке 19.

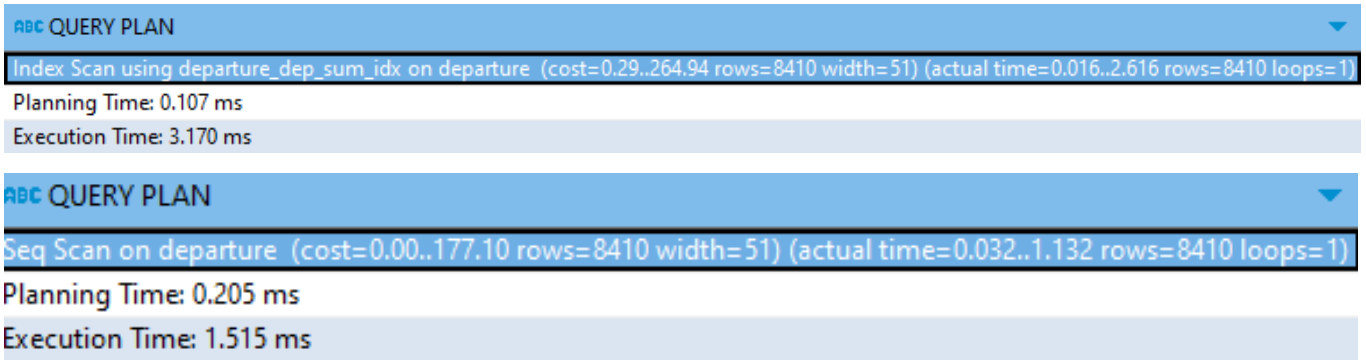


Рис. 19. Сортировка всей таблицы (№5)

Первый план выполнения запроса с предложением ORDER BY, а второй без. Время выполнения запроса сократилось почти вдвое. Так как таблица уже отсортирована, то достаточно просто выбрать все строки по порядку с помощью последовательного доступа. При указании же в запросе ORDER BY планировщик будет использовать индексный доступ к данным, чтобы получить отсортированный набор, не зная о том, что данные уже упорядочены. То есть сортировка будет производиться как бы с нуля, что повлечет за собой дополнительные затраты.

Интересно заметить, что если удалить индекс по отсортированному столбцу и снова проделать то же самое, то в случае отсутствия ORDER BY план выполнения будет таким же, как второй план на рисунке 19. А если оставить предложение ORDER BY, то план будет таким, как на рисунке 20.

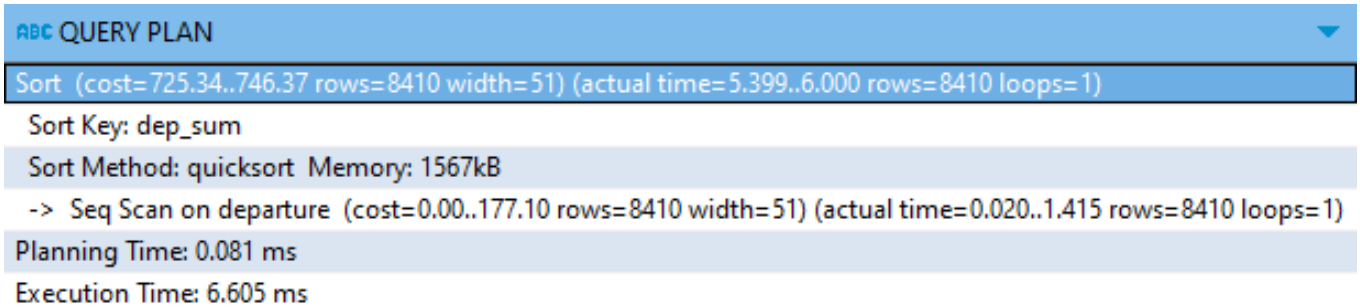


Рис. 20. Сортировка отсортированного (без индекса) (№5)

Сортировка будет производиться узлом Sort, но индексы при этом уже использоваться не будут, так как их нет. Сравнивая планы, упомянутые раннее, время выполнения запроса увеличилось в 6 раз, а стоимость — в 4

раза. То есть можно сделать вывод, что если в таблице данные отсортированы по определенному столбцу, то применение сортировки по этому же столбцу, особенно в случае, когда по нему нет индекса, приведет к заметному увеличению затрат на выполнение запроса.

Также из всего вышесказанного следует, что сортировка с помощью индексов эффективнее явной сортировки.

Пример №6. Вывести первые 10 идентификаторов выездов в порядке возрастания даты. Причем сначала должны быть выезды, в которых нет главного садовника, а затем те, в которых его идентификатор 23 (главным садовником является Ларионов Д.В.). Дан запрос (листинг 16), требующий оптимизации.

Листинг 16. Решение задачи (№6)

```
--explain analyze
select dep_id, dep_date
from departure d
where main_emp is null or main_emp = 23
order by main_emp nulls first, dep_date desc
limit 10;
```

План этого запроса представлен на рисунке 21.

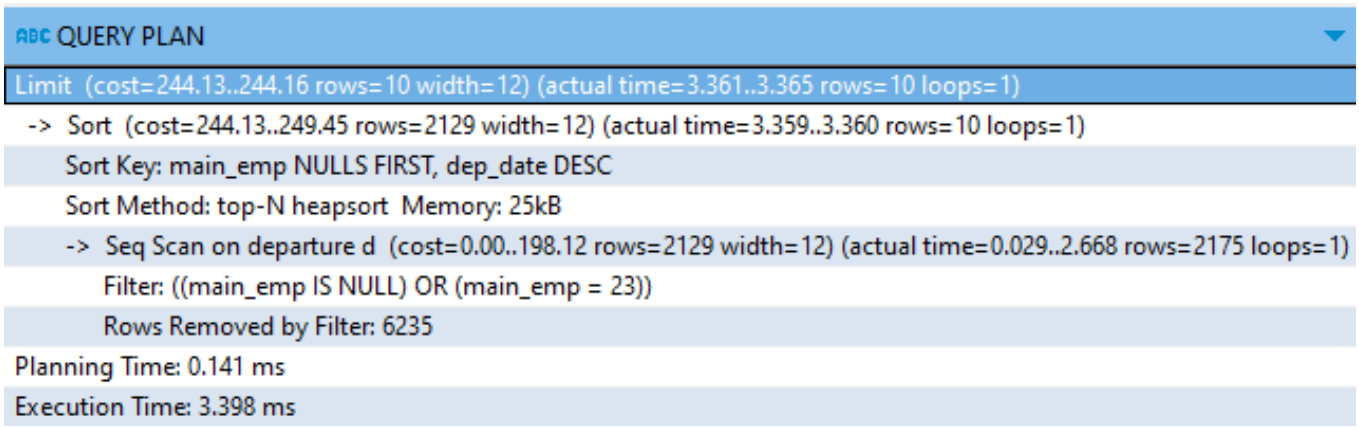


Рис. 21. Решение задачи (№6)

Limit в плане появился из-за предложения LIMIT в запросе.

Таблица departure сканируется полностью, хотя из нее выбираются не все строки, а около 2 тыс., поэтому создадим индекс по столбцу в WHERE. Но заметим так же, что в узле Sort оценка кардинальности несопоставима, и

не существует индекса для сортировки. Тогда расширим индекс полем сортировки (листинг 17).

Листинг 17. Создание индекса (№6)

```
Create index on departure(main_emp, dep_date DESC);
```

Снова выполним запрос из листинга 16 и получим результат (рис. 22).

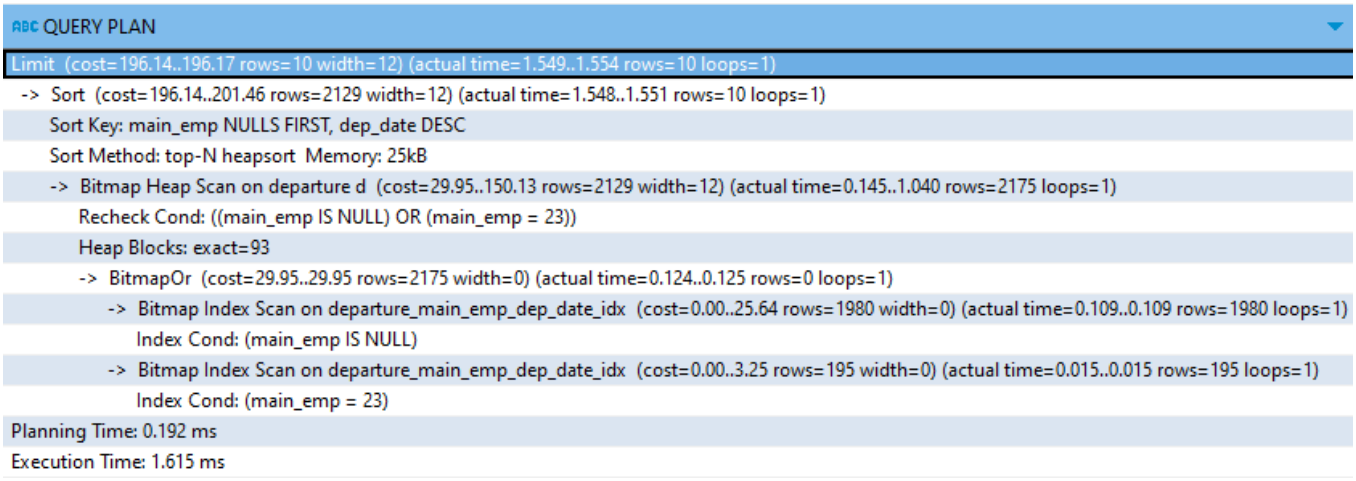


Рис. 22. План запроса после создания индекса (№6)

Стоимость и время выполнения улучшились (в 1.2 и в 2.1 раза соответственно), но не все так хорошо. На самом деле, созданный индекс не будет использоваться для сортировки, так как нет возможности создания индекса по заданным в предложении ORDER BY условиям. Еще одной проблемой является то, что в случае, когда достаточно много записей попадает под условие WHERE, сканирование по битовой карте может стоить очень дорого из-за присутствия в плане узла BitmapOr для условия OR. В таком случае одним из возможных путей оптимизации является разделение выборки на две непересекающиеся (листинг 18). Так как чтение из вложенного запроса сохраняет порядок, то данные вернутся в нужной нам последовательности. План выполнения измененного запроса представлен на рисунке 23.

Листинг 18. Запрос с использованием UNION ALL (№6)

```
--explain analyze
(select dep_id
from departure d
where main_emp is null
order by main_emp, dep_date desc
limit 10)
union all
(select dep_id
from departure d
where main_emp = 23
order by main_emp, dep_date desc
limit 10)
limit 10;
```

Причем, последний LIMIT гарантирует, что если записи найдутся в первом блоке, то выполнение второго блока производиться уже не будет. Это избавляет базу от выполнения лишней работы. В данном случае получается именно такая ситуация — из-за того, что нужно найти только первые 10 строк, все из которых будут получены с помощью первого запроса (never executed в узле Index Only Scan для второго запроса на рис. 23).

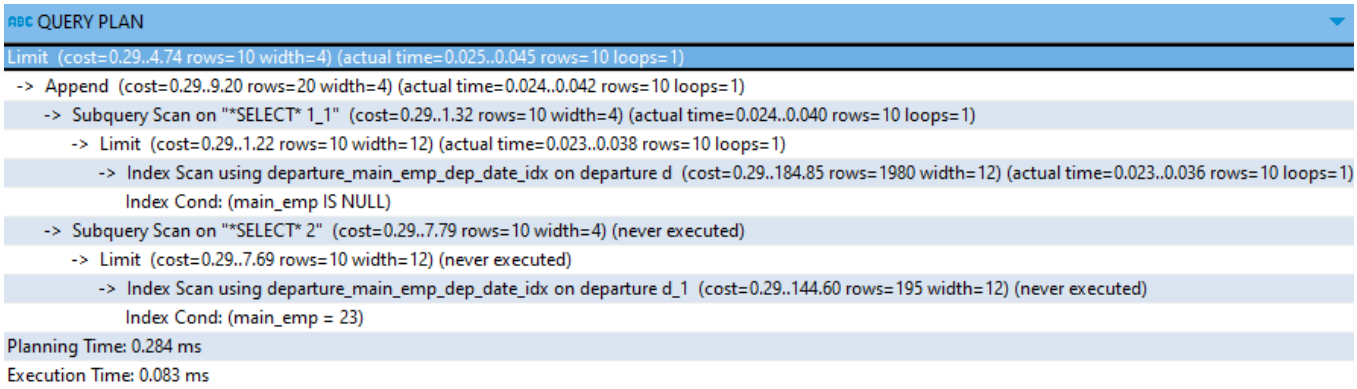


Рис. 23. План запроса с использованием UNION ALL (№6)

Вследствие применения UNION ALL появился узел Append, объединяющий выборки из двух запросов, соединение происходит по мере появления новых срок в результате второго запроса.

После разделения запроса по двум условиям время и стоимость его выполнения сильно улучшили свои показатели, а именно Execution Time уменьшилось в 19 раз, а стоимость в 41 раз. По сравнению с планом запроса до начала оптимизации время улучшилось в 41 раз, стоимость в 51 раз.

Стоит заметить, что если в плане использовался бы узел BitmapAnd (для условия AND), то оптимизацией в этом случае являлось бы создание составного индекса.

Пример №7. Вычислить среднюю стоимость выездов для каждого сотрудника, исключая при этом самые дорогие и самые дешёвые выезды. Дан запрос (листинг 19), который нужно оптимизировать.

Листинг 19. Решение задачи (№7)

```
--EXPLAIN ANALYZE
SELECT e.emp_id, (
SELECT round(avg(d.dep_sum))
FROM equipment_on_departure eod
JOIN departure d using(dep_id)
WHERE eod.emp_id = e.emp_id
AND d.dep_sum > (SELECT min(dep_sum) FROM departure)
AND d.dep_sum < (SELECT max(dep_sum) FROM departure)
)
FROM employee_on_departure e
GROUP BY e.emp_id;
```

Откомментируем первую строку (листинг 19) и получим план выполнения запроса (рис. 23).

```
QUERY PLAN
HashAggregate  (cost=606.75..43493.82 rows=36 width=36) (actual time=45.172..225.597 rows=36 loops=1)
  Group Key: e.emp_id
  Batches: 1  Memory Usage: 24kB
  ->  Seq Scan on employee_on_departure e  (cost=0.00..548.00 rows=23500 width=4) (actual time=0.199..4.474 rows=23500 loops=1)
  SubPlan 3
    ->  Aggregate  (cost=1191.28..1191.30 rows=1 width=32) (actual time=5.790..5.790 rows=1 loops=36)
          InitPlan 1 (returns $0)
            ->  Aggregate  (cost=198.13..198.14 rows=1 width=32) (actual time=6.601..6.602 rows=1 loops=1)
                  ->  Seq Scan on departure  (cost=0.00..177.10 rows=8410 width=7) (actual time=0.153..1.901 rows=8410 loops=1)
          InitPlan 2 (returns $1)
            ->  Aggregate  (cost=198.13..198.14 rows=1 width=32) (actual time=6.983..6.984 rows=1 loops=1)
                  ->  Seq Scan on departure departure_1  (cost=0.00..177.10 rows=8410 width=7) (actual time=0.039..1.774 rows=8410 loops=1)
          ->  Hash Join  (cost=219.67..795.00 rows=4 width=7) (actual time=0.656..5.594 rows=819 loops=36)
                Hash Cond: (eod.dep_id = d.dep_id)
                ->  Seq Scan on equipment_on_departure eod  (cost=0.00..573.17 rows=820 width=4) (actual time=0.017..4.638 rows=820 loops=36)
                      Filter: (emp_id = e.emp_id)
                      Rows Removed by Filter: 28714
                ->  Hash  (cost=219.15..219.15 rows=42 width=11) (actual time=21.983..21.983 rows=8381 loops=1)
                      Buckets: 16384 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 504kB
                      ->  Seq Scan on departure d  (cost=0.00..219.15 rows=42 width=11) (actual time=13.640..19.466 rows=8381 loops=1)
                            Filter: ((dep_sum > $0) AND (dep_sum < $1))
                            Rows Removed by Filter: 29
Planning Time: 1.225 ms
Execution Time: 225.822 ms
```

Рис. 23. План выполнения изначального запроса (№7)

Вместо обычной группировки в данном случае используется узел HashAggregate — группировки данных по полю, указанному в Group Key. Узел Aggregate — агрегирующая функция, которая в данном случае вычисляет среднее.

Основной запрос сканирует таблицу facility только по индексу и группирует данные по id объекта. Узел SubPlan — подзапрос, вычисляющий стоимость для каждого id объекта (выполняется много раз — в узле Aggregate loops=399). Узлы InitPlan — подзапросы, вычисляющие минимум и максимум (выполняются один раз, но для каждого SubPlan).

Заметим, что в узлах InitPlan выполняется полное сканирование таблицы departure. Чтобы этого избежать, создадим индекс по полю dep_sum, по которому выполняется вычисление минимума и максимума (листинг 20).

Листинг 20. Создание индекса по dep_sum (№7)

```
create index on departure(dep_sum);
```

Снова получим план выполнения запроса из листинга 14 (рис. 24).

```
QUERY PLAN
HashAggregate  (cost=606.75..21510.89 rows=36 width=36) (actual time=24.144..211.953 rows=36 loops=1)
  Group Key: e.emp_id
  Batches: 1  Memory Usage: 24kB
  ->  Seq Scan on employee_on_departure e  (cost=0.00..548.00 rows=23500 width=4) (actual time=0.187..3.097 rows=23500 loops=1)
  SubPlan 5
    ->  Aggregate  (cost=580.65..580.66 rows=1 width=32) (actual time=5.579..5.579 rows=1 loops=36)
      InitPlan 2 (returns $1)
        ->  Result  (cost=0.32..0.33 rows=1 width=32) (actual time=0.066..0.068 rows=1 loops=1)
          InitPlan 1 (returns $0)
            ->  Limit  (cost=0.29..0.32 rows=1 width=7) (actual time=0.064..0.065 rows=1 loops=1)
              ->  Index Only Scan using departure_dep_sum_idx on departure  (cost=0.29..285.96 rows=8410 width=7) (actual time=0.062..0.062 rows=1 loops=1)
                  Index Cond: (dep_sum IS NOT NULL)
                  Heap Fetches: 1
      InitPlan 4 (returns $3)
        ->  Result  (cost=0.32..0.33 rows=1 width=32) (actual time=0.016..0.017 rows=1 loops=1)
          InitPlan 3 (returns $2)
            ->  Limit  (cost=0.29..0.32 rows=1 width=7) (actual time=0.014..0.015 rows=1 loops=1)
              ->  Index Only Scan Backward using departure_dep_sum_idx on departure departure_1  (cost=0.29..285.96 rows=8410 width=7) (actual time=0.013..0.014 rows=1 loops=1)
                  Index Cond: (dep_sum IS NOT NULL)
                  Heap Fetches: 1
      ->  Hash Join  (cost=4.65..579.98 rows=4 width=7) (actual time=0.274..5.384 rows=819 loops=36)
          Hash Cond: (eod.dep_id = d.dep_id)
          ->  Seq Scan on equipment_on_departure eod  (cost=0.00..573.17 rows=820 width=4) (actual time=0.012..4.803 rows=820 loops=36)
            Filter: (emp_id = e.emp_id)
            Rows Removed by Filter: 28714
          ->  Hash  (cost=4.12..4.12 rows=42 width=11) (actual time=8.519..8.520 rows=8381 loops=1)
            Buckets: 16384 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 504kB
            ->  Index Scan using departure_dep_sum_idx on departure d  (cost=0.29..4.12 rows=42 width=11) (actual time=0.129..5.875 rows=8381 loops=1)
              Index Cond: ((dep_sum > $1) AND (dep_sum < $3))
Planning Time: 1.134 ms
Execution Time: 212.228 ms
```

Рис. 24. План запроса после создания индекса (№7)

Несмотря на то, что время и стоимость выполнения запроса уменьшились, все равно можно сказать, что ситуация ухудшилась. Действительно, подзапросы используют индексное сканирование (Index Only Scan), за счет чего ускоряются. Но по полю dep_sum происходят вычисления не только в узле InitPlan, но и в SubPlan, когда происходит сравнение значений поля с получаемыми значениями подзапросов. Из-за того, что планировщик не знает, что он получит из подзапросов (на этапе планирования это неизвестно, как и при вызове функции вместо подзапроса), непонятно какая оценка кардинальности применится к таблице departure. Но так как таких условий два, то кардинальность будет сильно занижена — при индексном сканировании (Index Scan) расчетное количество строк 42, а реальное 8381. Учитывая, что таблица departure содержит 8410 строк, приходим к выводу, что в данном случае эффективнее было бы последовательное сканирование всей таблицы (Seq Scan), а индекс только мешает и замедляет выполнение запроса.

Одним из способов запрета индексного доступа является замена индексируемого столбца выражением. Например, написать dep_sum + 0. Но лучшей практикой будет подправить оценку планировщика.

Проблема заключается в том, что планировщик не знает, какой результат будет в подзапросах. Поэтому посчитаем минимальную и максимальную суммы отдельно, до выполнения основного запроса, и подставим в него уже вычисленные значения, хранящиеся в переменных. Для этого создадим процедуру (листинг 21), так как в PostgreSQL нет возможности записывать значения в переменные в транзакциях (только в psql клиенте командой \gset).

Листинг 21. Создание и выполнение процедуры (№7)

```
CREATE OR REPLACE PROCEDURE number_7()
  LANGUAGE plpgsql AS
$proc$
DECLARE
   _line text;
   s_min numeric;
```

```
    s_max numeric;
BEGIN
    s_max := (SELECT max(dep_sum) FROM departure);
    s_min := (SELECT min(dep_sum) FROM departure);
    FOR _line IN
        EXPLAIN ANALYZE
        SELECT e.emp_id, (
         SELECT round(avg(d.dep_sum))
         FROM equipment_on_departure eod
           JOIN departure d using(dep_id)
         WHERE eod.emp_id = e.emp_id -- and eod.dep_id = e.dep_id
           AND d.dep_sum > s_min
            AND d.dep_sum < s_max
      )
      FROM employee_on_departure e
      GROUP BY e.emp_id
    LOOP
       RAISE NOTICE '%', _line;
    END LOOP;
END
$proc$;

CALL number_7();
```

Результатом выполнения процедуры будет план, представленный на рисунке 25. Следует отметить, что процедура была создана для отслеживания планов выполнения запроса, поэтому для дальнейшего использования она требует корректировки.

```
HashAggregate  (cost=606.75..33053.37 rows=36 width=36) (actual time=23.729..185.042 rows=36 loops=1)
  Group Key: e.emp_id
  Batches: 1  Memory Usage: 24kB
  ->  Seq Scan on employee_on_departure e  (cost=0.00..548.00 rows=23500 width=4) (actual time=0.145..2.862 rows=23500 loops=1)
  SubPlan 1
    ->  Aggregate  (cost=901.27..901.28 rows=1 width=32) (actual time=4.858..4.858 rows=1 loops=36)
          ->  Hash Join  (cost=323.90..899.23 rows=817 width=7) (actual time=0.280..4.696 rows=819 loops=36)
                Hash Cond: (eod.dep_id = d.dep_id)
                ->  Seq Scan on equipment_on_departure eod  (cost=0.00..573.17 rows=820 width=4) (actual time=0.012..4.146 rows=820 loops=36)
                      Filter: (emp_id = e.emp_id)
                      Rows Removed by Filter: 28714
                ->  Hash  (cost=219.15..219.15 rows=8380 width=11) (actual time=8.784..8.784 rows=8381 loops=1)
                      Buckets: 16384  Batches: 1  Memory Usage: 504kB
                      ->  Seq Scan on departure d  (cost=0.00..219.15 rows=8380 width=11) (actual time=0.053..6.040 rows=8381 loops=1)
                            Filter: ((dep_sum > 2900.00) AND (dep_sum < 58826.68))
                            Rows Removed by Filter: 29
Planning Time: 0.797 ms
Execution Time: 185.259 ms
```

Рис. 25. План выполнения процедуры (№7)

Заметим, что одни и те же таблицы читаются несколько раз. Это происходит из-за того, что значения вычисляются в коррелированном подзапросе. Поэтому избавимся от него, переписав запрос в процедуре так, как показано в листинге 22.

Листинг 22. Переписывание запроса в процедуре (№7)

```
SELECT e.emp_id, round(avg(d.dep_sum))
FROM employee_on_departure e
LEFT JOIN equipment_on_departure eod on e.dep_id=eod.dep_id
LEFT JOIN departure d on eod.dep_id=d.dep_id
WHERE eod.emp_id = e.emp_id
AND d.dep_sum > s_min
AND d.dep_sum < s_max
GROUP BY e.emp_id;
```

Изменив процедуру и выполнив ее вызов, получим план (рис. 26).

```
HashAggregate  (cost=2179.37..2179.91 rows=36 width=36) (actual time=61.797..61.825 rows=36 loops=1)
  Group Key: e.emp_id
  Batches: 1  Memory Usage: 32kB
  ->  Hash Join  (cost=1894.49..2167.93 rows=2287 width=11) (actual time=42.762..53.540 rows=29487 loops=1)
        Hash Cond: (d.dep_id = e.dep_id)
        ->  Seq Scan on departure d  (cost=0.00..219.15 rows=8380 width=11) (actual time=0.801..4.088 rows=8381 loops=1)
              Filter: ((dep_sum > 2900.00) AND (dep_sum < 58826.68))
              Rows Removed by Filter: 29
        ->  Hash  (cost=1865.80..1865.80 rows=2295 width=12) (actual time=41.848..41.850 rows=29534 loops=1)
              Buckets: 32768 (originally 4096)  Batches: 1 (originally 1)  Memory Usage: 1526kB
              ->  Hash Join  (cost=900.50..1865.80 rows=2295 width=12) (actual time=14.832..34.577 rows=29534 loops=1)
                    Hash Cond: ((eod.dep_id = e.dep_id) AND (eod.emp_id = e.emp_id))
                    ->  Seq Scan on equipment_on_departure eod  (cost=0.00..499.34 rows=29534 width=8) (actual time=0.045..3.434 rows=29534 loops=1)
                    ->  Hash  (cost=548.00..548.00 rows=23500 width=8) (actual time=14.550..14.550 rows=23500 loops=1)
                          Buckets: 32768  Batches: 1  Memory Usage: 1174kB
                          ->  Seq Scan on employee_on_departure e  (cost=0.00..548.00 rows=23500 width=8) (actual time=0.243..6.869 rows=23500 loops=1)
Planning Time: 0.645 ms
Execution Time: 62.372 ms
```

Рис. 26. План выполнения измененной процедуры (№7)

Теперь запрос выполняется эффективно — время выполнения улучшилось с 225.822 ms до 62.372 ms, применяется корректная оценка кардинальности, нет лишних чтений таблиц.