Санкт-Петербургский Государственный Университет
Математико-механический факультет

Математическое обеспечение и администрирование информационных систем

Шкуратов Илья Андреевич

Оптимизация запросов с обобщёнными табличными выражениями

Магистерская диссертация

Научный руководитель:
д. ф.-м. н., профессор Новиков Б. А.

Рецензент:
Яцышин И. А.

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Ilia Shkuratov

Optimization of queries with common table expressions

Graduation Thesis

Scientific supervisor:
professor Boris Novikov

Reviewer:
Ilya Yatsishin

Saint-Petersburg
2017

Оглавление

Введение **4**

1. Постановка задачи **5**

2. Оптимизация обобщённых табличных выражений **6**

3. Анализ планов запросов с ОТВ в PostgreSQL **8**

3.1. Потеря порядка сортировки 8

3.2. «Проталкивание» предикатов 10

4. Оптимизатор/планировщик PostgreSQL **17**

5. Возможность реализации алгоритма встраивания ОТВ в PostgreSQL **19**

5.1. Сравнение планов со встроенными ОТВ 21

6. Мнение сообщества разработчиков PostgreSQL о встраивании ОТВ **26**

Заключение **27**

Список литературы **28**

Введение

Обобщённые табличные выражения (ОТВ) были введены в стандарте `SQL/Foundation ISO/IEC 9075-2:1999`. Они увеличивают выразительность языка SQL, позволяя писать рекурсивные запросы, а также представляют средства для лучшего структурирования запроса, путём разбиения его на небольшие именованные подзапросы.

На обобщённые табличные выражения можно ссылаться по имени как из других ОТВ, которые объявлены после, так и из тела запроса. В запросе может быть несколько ссылок на ОТВ, аналогично ссылкам на таблицу. При этом, семантика ОТВ накладывает ограничение: результат, получаемый по этой ссылке должен быть таким, как-будто ОТВ выполнилось один раз.

В данной работе рассматриваются нерекурсиные ОТВ и возможность их оптимизации в PostgreSQL. Оптимизатор PostgreSQL реализует семантику простым способом — он материализует результат выполнения ОТВ, а затем использует его при каждом обращении. При этом он оптимизирует подзапрос соответствующий ОТВ в изоляции от остальной части, что может приводить к неэффективным планам в некоторых сценариях.

1. Постановка задачи

В связи с особенностями реализации обобщённых табличных выражений (ОТВ) в PostreSQL, целью данной работы были поставлены следующие задачи:

- исследовать механизм работы оптимизатора с ОТВ в текущей версии PostgreSQL,
- изучить существующие методы оптимизации ОТВ,
- предложить способ их реализации в текущей инфраструктуре оптимизатора.

2. Оптимизация обобщённых табличных выражений

Популярные СУБД, такие как DB2, Oracle Database, SQL Server поддерживают для ОТВ тот же набор оптимизаций, что и для обычных подзапросов или представлений [1]. В случае если оптимизатор считает, что замена ссылки на ОТВ соответсвующим подзапросом (встраивание) не изменит семантику запроса, он рассматривает планы с данной альтернативой. Встраивание позволяет использовать все методы оптимизации, обычно применяемые к подзапросам [7, 2, 1, 4, 5]:

- проталкивание предикатов,
- поднятие предикатов,
- слияние с внешним запросом (unnesting),
- слияние с другим подзапросом (coalescing),
- замена на оконную функцию.

Однако встраивание не возможно, когда ОТВ содержит недетерминированные (volatile) функции или функции с доступом ко внешним ресурсам, так как это может нарушить семантику. Кроме того, не смотря на широкий спектр возможных оптимизаций, встраивание не всегда будет оказываться выгодным [8, 3].

Чтобы принять оптимальное решение, нам нужно рассмотреть все возможные планы получающиеся при встраивании ОТВ вместо ссылки [8]. Однако узнать стоимость таких планов мы сможем только на уровне наименьшего общего предка (НОК) ссылок на ОТВ [3]. Это так в силу того, что мы не можем сразу добавить стоимость выполнения и материализации исходного ОТВ (C^{CTE}) к планам, которые используют обращение по ссылке, так как эта стоимость будет «разделяться» между всеми такими обращениями. А в случае плана, которые использует только встроенные ОТВ, мы не можем знать нужно ли добавлять

[1]http://modern-sql.com/feature/with/performance

стоимость C^{CTE}, так как в другой части запроса может быть выгодно использовать обращение к материализованному ОТВ.

Дойдя до НОК, мы сможем учесть C^{CTE} и сравнить стоимости планов, содержащих встроенные ОТВ, с остальными планами и отбросить те, что имеют большую стоимость.

Понятно, что количество дополнительных планов, которые нам нужно будет рассмотреть, экспоненциально зависит от количества ссылок на ОТВ, и при их большом количестве, нужно использовать эвристические подходы.

Кроме описанной выше оптимизации, разработчики Orca [9] реализовали в своей системе проталкивание предикатов в ОТВ и учёт физических свойств данных в местах использования [8]. При проталкивании, предикаты из разных контекстов (относящиеся к разным ссылкам на ОТВ) объединяются дизъюнкцией и добавляется к подзапросу ОТВ. При дальнейшей оптимизации ОТВ они, по возможности, будут проталкиваться дальше. При этом оригинальные предикаты остаются на месте, чтобы сохранить семантику запроса. Учёт физических свойств позволяет протолкнуть общие требования встречающиеся в контексте ссылок на ОТВ, в само ОТВ. Например, если во всех контекстах требуется определённый порядок записей или партиционирование по определённому ключу, то соответствующие операции будут применены один раз, при выполнении ОТВ.

3. Анализ планов запросов с ОТВ в PostgreSQL

Для проведения анализа использовалась демонстрационная база данных Postgres Professional [12] маленького размера и последняя версия PostgreSQL [10]. Эксперименты проводились на машине со следующими характеристиками:

RAM	8 Gb
CPU	Intel® Core™ i5-4200U CPU @ 1.60GHz × 4
OS	Ubuntu 16.04 LTS, 64-bit
HDD	ST500LM021-1KJ152

Таблица 1: Конфигурация экспериментальной машины

Тематикой демонстрационной базы являются авиаперевозки. Схема базы данных показана на Рис. 1. Мы не будем описывать данные, так как схемы должно быть достаточно для понимания представленных запросов. При необходимости, можно получить подробное описание на сайте с базой [12].

Рассмотрим несколько примеров, которые иллюстрируют недостатки отказа оптимизации ОТВ совместно с остальной частью запроса.

3.1. Потеря порядка сортировки

Посмотрим на Запрос 1, который выводит количество мест в самолёте, сгруппированное по классу обслуживания. Сравним его план выполнения (План 1), с планом выполнения (План 2) семантически эквивалентного Запроса 2, использующего обобщённое табличное выражение.

Обратим внимание на повторную сортировку результата в Плане 2 (строка 10). Ввиду того, что оптимизатор не запоминает порядок записей в ОТВ, он вынужден добавлять в план сортировку, даже когда в этом нет необходимости. При работе с подзапросами, эта информация сохраняется, поэтому оптимизатор может сгенерировать план без лишней сортировки.

```
SELECT    s2.aircraft_code,
          string_agg (s2.fare_conditions || '(' || s2.num::text || ')',
                      ', ') as fare_conditions
FROM      (
            SELECT    s.aircraft_code, s.fare_conditions, count(*) as num
            FROM      seats s
            GROUP BY s.aircraft_code, s.fare_conditions
            ORDER BY s.aircraft_code, s.fare_conditions
) s2
GROUP BY s2.aircraft_code
ORDER BY s2.aircraft_code;
```

Запрос 1: Количество мест в самолёте, с учётом класса обслуживания.

```
GroupAggregate  (cost=32.34..33.49 rows=27 width=36)
  Group Key: s.aircraft_code
  -> Sort  (cost=32.34..32.41 rows=27 width=20)
      Sort Key: s.aircraft_code, s.fare_conditions
      -> HashAggregate  (cost=31.43..31.70 rows=27 width=20)
          Group Key: s.aircraft_code, s.fare_conditions
          -> Seq Scan on seats s  (cost=0.00..21.39 rows=1339 width=12)
```

План 1: План к Запросу 1.

```
WITH s2 AS (
  SELECT    s.aircraft_code, s.fare_conditions, count(*) as num
  FROM      seats s
  GROUP BY s.aircraft_code, s.fare_conditions
  ORDER BY s.aircraft_code, s.fare_conditions
)
SELECT    s2.aircraft_code,
          string_agg (s2.fare_conditions || '(' || s2.num::text || ')',
                      ', ') as fare_conditions
FROM s2
GROUP BY s2.aircraft_code
ORDER BY s2.aircraft_code;
```

Запрос 2: Запрос с ОТВ, эквивалентный по результату Запросу 1.

```
GroupAggregate  (cost=33.59..34.47 rows=27 width=48)
  Group Key: s2.aircraft_code
  CTE s2
    -> Sort  (cost=32.34..32.41 rows=27 width=20)
        Sort Key: s.aircraft_code, s.fare_conditions
        -> HashAggregate  (cost=31.43..31.70 rows=27 width=20)
            Group Key: s.aircraft_code, s.fare_conditions
            -> Seq Scan on seats s  (cost=0.00..21.39 rows=1339 width=12)
  -> Sort  (cost=1.18..1.25 rows=27 width=62)
      Sort Key: s2.aircraft_code
      -> CTE Scan on s2  (cost=0.00..0.54 rows=27 width=62)
```

План 2: План к Запросу 2.

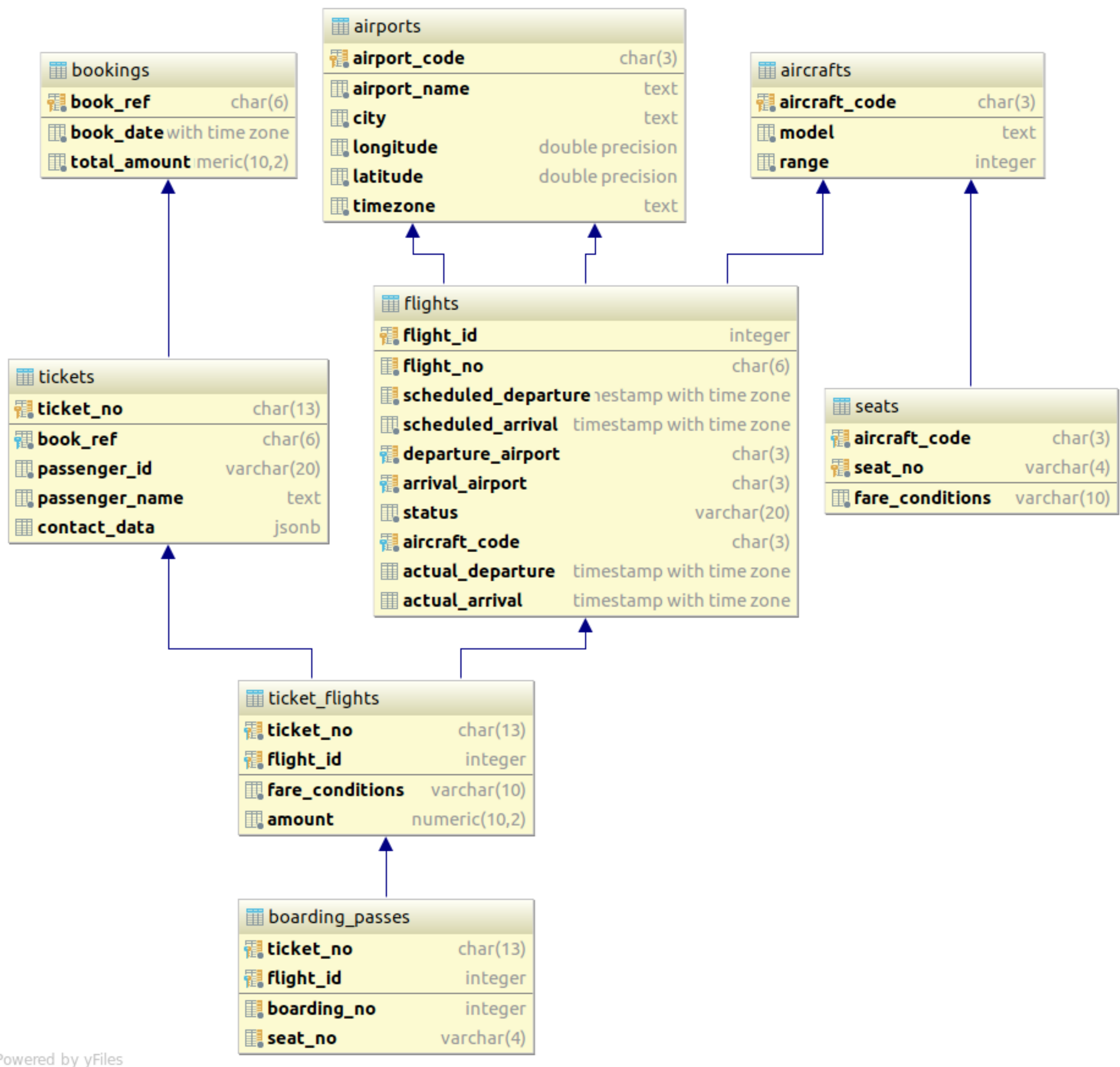

Рис. 1: Схема тестовой базы данных.

3.2. «Проталкивание» предикатов

Рассмотрим запрос, который находит все стыковочные рейсы из Домодедово в Пулково без ограничений по времени вылета и продолжительности пересадки. В этот раз мы начнём с запроса, использующего ОТВ. Так нам будет легче понять его структуру и проследить влияние оптимизации на план, при поэтапном встраивании ОТВ.

В исходном Запросе 3 используются три ОТВ:

- `FlightSeatsCount` подсчитывает количество мест на каждом судне.
- `FlightSeatsOccupied` подсчитывает количество занятых мест на

```sql
WITH FlightSeatsCount as (
  SELECT aircraft_code, fare_conditions, count(seat_no) AS total
  FROM seats
  GROUP BY (aircraft_code, fare_conditions)
), FlightSeatsOccupied as (
  SELECT
    fl.flight_id, fl.departure_airport, fl.arrival_airport,
    fl.scheduled_departure, fl.scheduled_arrival, fl.aircraft_code,
    tf.fare_conditions, count(ticket_no) AS occupied
  FROM flights fl
  JOIN ticket_flights tf ON fl.flight_id = tf.flight_id
  GROUP BY (fl.flight_id, tf.fare_conditions)
), FlightSeatsAvailable as (
  SELECT fo.flight_id, fo.departure_airport, fo.arrival_airport,
    fo.scheduled_departure, fo.scheduled_arrival,
    fc.fare_conditions, (fc.total - fo.occupied) AS available
  FROM FlightSeatsCount fc
  JOIN FlightSeatsOccupied fo
      ON fc.aircraft_code = fo.aircraft_code AND
         fc.fare_conditions = fo.fare_conditions
  WHERE fc.total - fo.occupied > 0
)
SELECT
  flo.flight_id, flo.departure_airport, flo.arrival_airport,
  flo.available, flo.fare_conditions,
  flo.scheduled_departure, flo.scheduled_arrival,
  fli.flight_id, fli.departure_airport, fli.arrival_airport,
  fli.available, fli.fare_conditions,
  fli.scheduled_departure, fli.scheduled_arrival
FROM FlightSeatsAvailable flo
JOIN FlightSeatsAvailable fli ON
                   flo.arrival_airport = fli.departure_airport AND
                   flo.scheduled_arrival < fli.scheduled_departure
WHERE flo.departure_airport = 'DME' AND fli.arrival_airport = 'LED'
```

Запрос 3: Нахождение стыковочных рейсов из Домодедово в Пулково.

полёте.

- `FlightSeatsAvailable` использует результаты предыдущих вычислений, чтобы посчитать количество свободных мест на полёте.

Выполнение данного запроса занимает значительной время — около двух минут. Большая часть времени уходит на самое внешнее соединение, при котором отфильтровывается 42465913 строк. Также, мы можем заметить, что при выполнении `FlightSeatsOccupied` происходит внешняя сортировка. Мы можем сильно сократить количество обрабатываемых данных, если сразу отфильтруем ненужные полёты.

Встраивание `FlightSeatsAvailable` и `FlightSeatsOccupied` (Запрос 4)

```
Nested Loop  (cost=296683.39..296685.43 rows=1 width=196) (actual time=5956.278..232121.631 rows=112448 loops=1)
  Join Filter: ((flo.scheduled_arrival < fli.scheduled_departure) AND (flo.arrival_airport = fli.departure_airport))
  Rows Removed by Join Filter: 42465913
  CTE flightseatscount
    ->  HashAggregate  (cost=31.43..31.70 rows=27 width=20)
          Group Key: seats.aircraft_code, seats.fare_conditions
          ->  Seq Scan on seats  (cost=0.00..21.39 rows=1339 width=15)
  CTE flightseatsoccupied
    ->  Finalize GroupAggregate  (cost=227984.02..291146.85 rows=200064 width=48) (actual time=3975.011..5183.306 rows=74697 loops=1)
          Group Key: fl.flight_id, tf.fare_conditions
          ->  Gather Merge  (cost=227984.02..286145.25 rows=400128 width=48)
                Workers Planned: 2
                ->  Partial GroupAggregate  (cost=226983.99..238960.52 rows=200064 width=48)
                      Group Key: fl.flight_id, tf.fare_conditions
                      ->  Sort  (cost=226983.99..229477.97 rows=997589 width=54)
                            Sort Key: fl.flight_id, tf.fare_conditions
                            Sort Method: external merge  Disk: 62472kB
                            ->  Hash Join  (cost=2764.48..59382.80 rows=997589 width=54)
                                  Hash Cond: (tf.flight_id = fl.flight_id)
                                  ->  Parallel Seq Scan on ticket_flights tf  (cost=0.00..29927.89 rows=997589 width=26)
                                  ->  Hash  (cost=1474.88..1474.88 rows=66688 width=32)
                                        ->  Seq Scan on flights fl  (cost=0.00..1474.88 rows=66688 width=32)
  CTE flightseatsavailable
    ->  Hash Join  (cost=0.95..5504.84 rows=45 width=98) (actual time=3976.147..5383.548 rows=71223 loops=1)
          Hash Cond: ((fo.aircraft_code = fc.aircraft_code) AND ((fo.fare_conditions)::text = (fc.fare_conditions)::text))
          Join Filter: ((fc.total - fo.occupied) > 0)
          Rows Removed by Join Filter: 3474
          ->  CTE Scan on flightseatsoccupied fo  (cost=0.00..4001.28 rows=200064 width=114) (actual time=3975.015..5284.268 rows=74697 loops=1)
          ->  Hash  (cost=0.54..0.54 rows=27 width=62)
                ->  CTE Scan on flightseatscount fc  (cost=0.00..0.54 rows=27 width=62)
  ->  CTE Scan on flightseatsavailable flo  (cost=0.00..1.01 rows=1 width=98)
        Filter: (departure_airport = 'DME'::bpchar)
        Rows Removed by Filter: 62424
  ->  CTE Scan on flightseatsavailable fli  (cost=0.00..1.01 rows=1 width=98)
        Filter: (arrival_airport = 'LED'::bpchar)
        Rows Removed by Filter: 66384
```

План 3: План выполнения Запроса 3.

может сократить число полётов для которых подсчитываются свободный места, так как позволяет оптимизатору протолкнуть предикаты к листьям плана (План 4), к сканированию таблицы `flights`.

После проведённого рефакторинга, появилось много дублирующегося кода, запрос стал значительно менее понятным, однако время его выполнения, благодаря «проталкиванию» предикатов, уменьшилось радикально — с 2 минут до 2 секунд. Как видно из данного примера, возможность проталкивание предикатов может кардинально сократить время выполнения запроса.

```sql
WITH FlightSeatsCount as (
  SELECT aircraft_code, fare_conditions, count(seat_no) AS total
  FROM seats
  GROUP BY (aircraft_code, fare_conditions)
)
SELECT
  flo.flight_id, flo.departure_airport, flo.arrival_airport,
  flo.available, flo.fare_conditions,
  flo.scheduled_departure, flo.scheduled_arrival,
  fli.flight_id, fli.departure_airport, fli.arrival_airport,
  fli.available, fli.fare_conditions,
  fli.scheduled_departure, fli.scheduled_arrival
FROM (
  SELECT fo.flight_id, fo.departure_airport, fo.arrival_airport,
    fo.scheduled_departure, fo.scheduled_arrival,
    fc.fare_conditions, (fc.total - fo.occupied) AS available
  FROM FlightSeatsCount fc
  JOIN (
    SELECT
      fl.flight_id, fl.departure_airport, fl.arrival_airport,
      fl.scheduled_departure, fl.scheduled_arrival, fl.aircraft_code,
      tf.fare_conditions, count(ticket_no) AS occupied
    FROM flights fl
    JOIN ticket_flights tf ON fl.flight_id = tf.flight_id
    GROUP BY (fl.flight_id, tf.fare_conditions)
  ) fo
      ON fc.aircraft_code = fo.aircraft_code AND
      fc.fare_conditions = fo.fare_conditions
  WHERE fc.total - fo.occupied > 0
) flo
JOIN (
  SELECT fo.flight_id, fo.departure_airport, fo.arrival_airport,
    fo.scheduled_departure, fo.scheduled_arrival,
    fc.fare_conditions, (fc.total - fo.occupied) AS available
  FROM FlightSeatsCount fc
  JOIN (
    SELECT
      fl.flight_id, fl.departure_airport, fl.arrival_airport,
      fl.scheduled_departure, fl.scheduled_arrival, fl.aircraft_code,
      tf.fare_conditions, count(ticket_no) AS occupied
    FROM flights fl
    JOIN ticket_flights tf ON fl.flight_id = tf.flight_id
    GROUP BY (fl.flight_id, tf.fare_conditions)
  ) fo
      ON fc.aircraft_code = fo.aircraft_code AND
      fc.fare_conditions = fo.fare_conditions
  WHERE fc.total - fo.occupied > 0
) fli ON
                  flo.arrival_airport = fli.departure_airport AND
                  flo.scheduled_arrival < fli.scheduled_departure
WHERE flo.departure_airport = 'DME' AND fli.arrival_airport = 'LED';
```

Запрос 4: Нахождение стыковочных рейсов, со встроенными `FlightSeatsAvailable` и `FlightSeatsOccupied`.

```
1  Hash Join  (cost=98883.72..99342.67 rows=1 width=148)
2    Hash Cond: ((fl.aircraft_code = fc.aircraft_code) AND ((tf.fare_conditions)
       ::text = (fc.fare_conditions)::text))
3    Join Filter: ((fc.total - (count(tf.ticket_no))) > 0)
4    CTE flightseatscount
5      ->  HashAggregate  (cost=31.43..31.70 rows=27 width=20)
6            Group Key: seats.aircraft_code, seats.fare_conditions
7            ->  Seq Scan on seats  (cost=0.00..21.39 rows=1339 width=15)
8    ->  Hash Join  (cost=98851.07..99309.38 rows=83 width=130)
9          Hash Cond: (fl.arrival_airport = fl_1.departure_airport)
10         Join Filter: (fl.scheduled_arrival < fl_1.scheduled_departure)
11         ->  Finalize HashAggregate  (cost=50267.56..50459.02 rows=19146 width
               =48)
12               Group Key: fl.flight_id, tf.fare_conditions
13               ->  Gather  (cost=45959.71..49980.37 rows=38292 width=48)
14                     Workers Planned: 2
15                     ->  Partial HashAggregate  (cost=44959.71..45151.17 rows
                             =19146 width=48)
16                           Group Key: fl.flight_id, tf.fare_conditions
17                           ->  Hash Join  (cost=1721.38..44243.69 rows=95469
                                   width=54)
18                                 Hash Cond: (tf.flight_id = fl.flight_id)
19                                 ->  Parallel Seq Scan on ticket_flights tf  (
                                         cost=0.00..29927.89 rows=997589 width=26)
20                                 ->  Hash  (cost=1641.60..1641.60 rows=6382
                                         width=32)
21                                       ->  Seq Scan on flights fl  (cost
                                               =0.00..1641.60 rows=6382 width=32)
22                                             Filter: (departure_airport = 'DME
                                                   '::bpchar)
```

План 4: План выполнения Запроса 4. Часть 1.

```
23          ->  Hash  (cost=48583.48..48583.48 rows=3 width=82)
24                ->  Hash Join  (cost=48263.67..48583.48 rows=3 width=82)
25                      Hash Cond: ((fl_1.aircraft_code = fc_1.aircraft_code) AND
                            ((tf_1.fare_conditions)::text = (fc_1.fare_conditions)
                            ::text))
26                      Join Filter: ((fc_1.total - (count(tf_1.ticket_no))) > 0)
27                      ->  Finalize HashAggregate  (cost=48262.72..48378.97 rows
                            =11625 width=48)
28                            Group Key: fl_1.flight_id, tf_1.fare_conditions
29                            ->  Gather  (cost=45647.10..48088.35 rows=23250
                                  width=48)
30                                  Workers Planned: 2
31                                  ->  Partial HashAggregate  (cost
                                        =44647.10..44763.35 rows=11625 width=48)
32                                        Group Key: fl_1.flight_id, tf_1.
                                              fare_conditions
33                                        ->  Hash Join  (cost=1690.04..44212.35
                                              rows=57966 width=54)
34                                              Hash Cond: (tf_1.flight_id = fl_1.
                                                    flight_id)
35                                              ->  Parallel Seq Scan on
                                                    ticket_flights tf_1  (cost
                                                    =0.00..29927.89 rows=997589
                                                    width=26)
36                                              ->  Hash  (cost=1641.60..1641.60
                                                    rows=3875 width=32)
37                                                    ->  Seq Scan on flights fl_1
                                                            (cost=0.00..1641.60
                                                          rows=3875 width=32)
38                                                            Filter: (
                                                                  arrival_airport =
                                                                  'LED'::bpchar)
39                      ->  Hash  (cost=0.54..0.54 rows=27 width=62)
40                            ->  CTE Scan on flightseatscount fc_1  (cost
                                  =0.00..0.54 rows=27 width=62)
41    ->  Hash  (cost=0.54..0.54 rows=27 width=62)
42          ->  CTE Scan on flightseatscount fc  (cost=0.00..0.54 rows=27 width
                =62)
```

План 4: План выполнения Запроса 4. Часть 2.

4. Оптимизатор/планировщик PostgreSQL

В PostgreSQL нет чёткого разделения на оптимизатор и планировщик. По сути, это один компонент, который переписывает запрос и тут же строит план его выполнения. В дальнейшем, мы будем называть этот компонент оптимизатором.

На вход оптимизатору приходит дерево запроса, прошедшее семантический анализ и представляющее собой логический план выполнения запроса. На выходе получается физический план, который также является деревом.

Процесс планирования выполняется одним потоком. Оптимизатор рекурсивно обрабатывает подзапросы, строя план «снизу-вверх». Упрощённая последовательность действий, выполняемая оптимизатором, представлена в Таблице 2. Также в ней указаны имена некоторых важных функций, на которые мы будем ссылаться при описании алгоритма в Разделе 5.

При нахождении планов для каждого отношения, как исходного, так и того, которое встречается в качестве промежуточного результата при выполнении запроса, оптимизатор сохраняет несколько планов. Планы идентифицируются ключами (структура `PathKey`), которые содержат информацию о порядке записей, получаемом при выполнении данного плана. При добавлении очередного плана в список планов для текущего отношения, он сравнивается с другим планом с таким же ключём. Если такой план имеется и его стоимость ниже, то новый план отбрасывается, а иначе новый план заменяет старый.

1. Построение планов для всех ОТВ.
   - Вызов `subquery_planner` для каждого ОТВ.
2. Преобразование `ANY` и `EXISTS` в JOIN.
3. Встраивание функций.
4. «Поднятие» подзапросов — рассматривается возможность объединени подзапроса с его родителем.
5. Предобработка, упрощение выражений в разных частях запроса.
   - Вызов `subquery_planner` для подзапросов в `EXISTS` и `ANY`.
6. Перемещение условий из `HAVING` в `WHERE`.
7. Вызов `grouping_planner`
   7.1. Вызов `query_planner` для нахождения планов выполнения соединений.
      - Вызов `subquery_planner` для подзапросов в `FROM`
   7.2. Нахождение планов группировки.
   7.3. Нахождение планов для оконных функций.
   7.4. Нахождение планов для `DISTINCT`
   7.5. Нахождение планов сортировки. (Обработка `ORDER BY`)
   7.6. Добавление к планам узла для `LIMIT`
8. Выбор самого дешёвого плана.

Таблица 2: Упрощённая последовательность действий при планировании запроса. Содержание вызова `query_planner`.

5. Возможность реализации алгоритма встраивания ОТВ в PostgreSQL

Рассмотрим, как можно реализовать способ оптимизации, основанный на встраивании, который был описан в Разделе 2, в оптимизаторе PostgreSQL.

Перед тем как встраивать ОТВ, нужно убедиться, что это не изменит семантику запроса. В случае PostgreSQL, это означает, что подзапрос

- не является «модифицирующим», то есть иметь вид `SELECT ...`,
- не использует `VOLATILE` или `STABLE` функции.

Если запрос, удовлетворяет этим условиям, то мы можем рассматривать его встраивание.

Начнём с простого случая, когда имеется только одна ссылка на ОТВ. Тогда мы можем встроить запрос без проверки стоимости. Это можно сделать в самом начале функции `subquery_planner` (Таблица 2) перед построением планов для ОТВ (пункт 1). Так как все оптимизации выполняются после, то нам не нужно ничего делать дополнительно, только удалить его из списка ОТВ, которые лежит в структуре `PlannerInfo` для корневого запроса.

Теперь рассмотрим общий случай. Для его реализации нам нужно понять

1. как добавить в рассмотрение альтернативные планы, включающие встраивание ОТВ,
2. как отложить сравнение стоимости планов до наименьшего общего предка?

Разберём представленные вопросы по-порядку. Начнём с краткого описание решения первой проблемы в оптимизаторе Orca [9]. В нём добавление альтернативных планов осуществляется путём применения правил трансформации к элементам, содержащимся в структуре `Memo` [6].

`Memo` — это ациклический граф из групп. Каждая группа представляет собой логический оператор и содержит все возможные его реализации (физический оператор или их последовательность), которые генерируются правилами трансформации.

В PostgreSQL единицей, содержащей информацию о логически эквивалентных операциях, является структура `RelOptInfo`. Она представляет собой абстракцию некоторого отношения для оптимизатора. Кроме прочей информации, `RelOptInfo` содержит список путей (`Path`) с помощью которого это отношение можно получить. Структура `Path` и её производные, по сути, описывают различные физические операторы. Дерево из структур `Path` мы будем называть путём или планом.

Таким образом, чтобы добавить альтернативные планы, нам нужно в `RelOptInfo`, которое содержит информацию об отношении, получаемом по ссылки на ОТВ, добавить пути `SubqueryScanPath`, которые описывают получение этого отношения, как результат обращения к подзапросу. Однако мы не можем так поступить, так как `RelOptInfo` содержит информацию для оптимизатора, которая будет различна для ссылки на ОТВ и подзапроса.

Возможным решением может быть создание специальной структуры, аналогичной `AlternativeSubPlan`, которая представляла бы альтернативу между несколькими `RelOptInfo`. Это позволило бы применять при оптимизации методы, которые требуют сравнения альтернатив, связанных с возможной трансформацией запроса.

Перейдём ко второму вопросу. Как уже упоминалось в Разделе 4, при выборе путей, оптимизатор сохраняет самый дешёвый путь с учётом получающегося порядка записей, а также отдельно сохраняет самый дешёвый путь без учёта порядка и путь с минимальной стартовой стоимостью (стоимость до получения первого кортежа ответа). Однако, как обсуждалось в Разделе 2, мы не можем сразу учесть полную стоимость для путей, в которых используются встроенные ОТВ. Для этого нам нужно сохранить их до наименьшего общего предка ссылок на ОТВ. По всей видимости, мы должны выделить отдельный список для таких путей и сравнивать их отдельно между собой, если их ключи

равны.

Таким образом, для реализации оптимизации нетривиального случая, потребуется довольно серьёзный рефакториг оптимизатора и, скорей всего, добавление новых структур данных.

5.1. Сравнение планов со встроенными ОТВ

Обсудим подробнее как можно сравнивать пути со встроенными ОТВ. Мы рассматриваем пути сгенерированные для получения одного и того же отношения, имеющие равные ключи путей (`PathKey`). Это означает, что с точки зрения планировщика они взаимозаменяемы.

Более формально отношение взаимозаменяемости путей можно описать следующим образом. Пусть существуют пути p и q. Каждый путь представляет собой дерево из операций. Будем обозначать $p \leftarrow q$, если путь q является поддеревом p. Тогда пути p_i и p_j называются взаимозаменяемыми, если для любого пути $p \leftarrow p_i$, существует путь $p \leftarrow p_j$, и наоборот — для любого пути $p \leftarrow p_j$ существует путь $p \leftarrow p_i$.

Пусть мы хотим сравнить два взаимозаменяемых плана, которые имеют операции обращения к материализованному или встроенному ОТВ. Пусть всего в запросе есть M различных ОТВ. При этом нв ОТВ M_i в запросе m_i ссылок. Обозначим также

- n_i^p — количество сканирований материализованного M_i, использующееся в плане p,
- r_i^p — количество сканирований материализованного M_i, использующееся в некотором *родительском* плане p,
- R — количество ссылок на ОТВ в остальной части запроса (часть, которая не покрывается рассматриваемыми путями),
- C_i — цена материализации M_i,
- C^p — цена плана p без учёта стоимости материализации используемых в нём ОТВ.

Предположим мы хотим сравнить два взаимозаменяемых плана p_1 и p_2. Стоимость плана p_j с учётом материализации, при некотором родительском пути можно выразить как:

$$\texttt{cost}(p_j; r_1, \dots, r_M) = C^{p_j} + \sum_{i=1}^{M} \texttt{addition_cost}(p_j, r_i) \tag{1}$$

где $\texttt{addition_cost}(p_j, r_i)$ — доля стоимости материализации M_i, которая распределяется на путь p_j. При этом, r_i задаются некоторым родительским планом p.

$$\texttt{addition_cost}(p_j, r_i) = \begin{cases} \frac{C_i \cdot n_i^{p_j}}{n_i^{p_j} + r_i}, & \text{если } n_i^{p_j} > 0 \\ 0, & \text{иначе} \end{cases} \tag{2}$$

Тогда мы можем сравнить стоимость планов с помощью неравенства, получающегося из равенств 1 и 2:

$$C^{p_1} - C^{p_2} < \sum_{i=1}^{M} \frac{C_i \cdot r_i \cdot (n_i^{p_2} - n_i^{p_1})}{(n_i^{p_1} + r_i)(n_i^{p_2} + r_i)} \tag{3}$$

Ясно, что если неравенство будет выполняться при минимальном значении суммы справа, то мы должны оставить только план p_1. В противном случае, какой путь выгоднее будет зависеть от пути, выбранного для остальной части запроса, поэтому мы должны оставить оба кандидата.

Рассмотрим, при каких r_i правая сумма принимает минимальное значения. Каждое слагаемое суммы мы можем рассматривать независимо. Для удобства обозначим его как:

$$\texttt{cost_diff}(r_i) = \frac{C_i \cdot r_i \cdot (n_i^{p_2} - n_i^{p_1})}{(n_i^{p_1} + r_i)(n_i^{p_2} + r_i)} \tag{4}$$

Будем по-отдельности рассматривать случаи различных значений $n_i^{p_2}$ и $n_i^{p_1}$.

$n_i^{p_2} = n_i^{p_1}$ Это тривиальный случай, так как тогда $\texttt{cost_diff}(r_i) \equiv 0$.

$n_i^{p_1} = 0$ В этом случае

$$\texttt{cost_diff}(r_i) = \frac{C_i \cdot n_i^{p_2}}{n_i^{p_2} + r_i}$$

и минимум достигается при $r_i = R$. Таким образом

$$\texttt{min_cost_diff}(r_i) = \frac{C_i \cdot n_i^{p_2}}{n_i^{p_2} + R} \tag{5}$$

$n_i^{p_2} = 0$ В этом случае

$$\texttt{cost_diff}(r_i) = -\frac{C_i \cdot n_i^{p_1}}{n_i^{p_1} + r_i}$$

и минимум достигается при $r_i = 0$. Таким образом

$$\texttt{min_cost_diff}(r_i) = -C_i \tag{6}$$

Перед тем как рассмотреть оставшиеся случаи отметим, что $\texttt{cost_diff}(r_i) = 0$, при $r_i = 0$, а также, что она непрерывно дифференцируема по r_i на $[0, \infty)$. Её производная представлена в Равенстве 7.

$$\begin{aligned} \texttt{cost_diff}'(r) &= \\ &= \frac{C(n^{p_2} - n^{p_1})((n^{p_2} + r)(n^{p_1} + r) - r((n^{p_2} + n^1 p_1) + 2r^2))}{(n^{p_2} + r)^2 (n^{p_1} + r)^2} \\ &= \frac{C(n^{p_2} - n^{p_1})(n^{p_1} n^{p_2} + r^2 - 2r^3)}{(n^{p_2} + r)^2 (n^{p_1} + r)^2} \end{aligned} \tag{7}$$

$n_i^{p_2} > n_i^{p_1} > 0$ Тогда $\texttt{cost_diff}(r_i) > 0$ на интервале $(0, \infty)$, причём $\texttt{cost_diff}(r_i) \xrightarrow[r_i \to \infty]{} +0$. Учитывая непрерывность производной, мы можем сказать, что у неё должен быть минимум один корень на этом интервале.

Проанализируем как меняется знак производной на $[0, \infty)$. Очевидно, что при фиксированных n^{p_1} и n^{p_2}, знак будет зависеть только от члена $(n^{p_1} n^{p_2} + r^2 - 2r^3)$. Запишем соответствующее неравенство $n^{p_1} n^{p_2} + r^2 -$

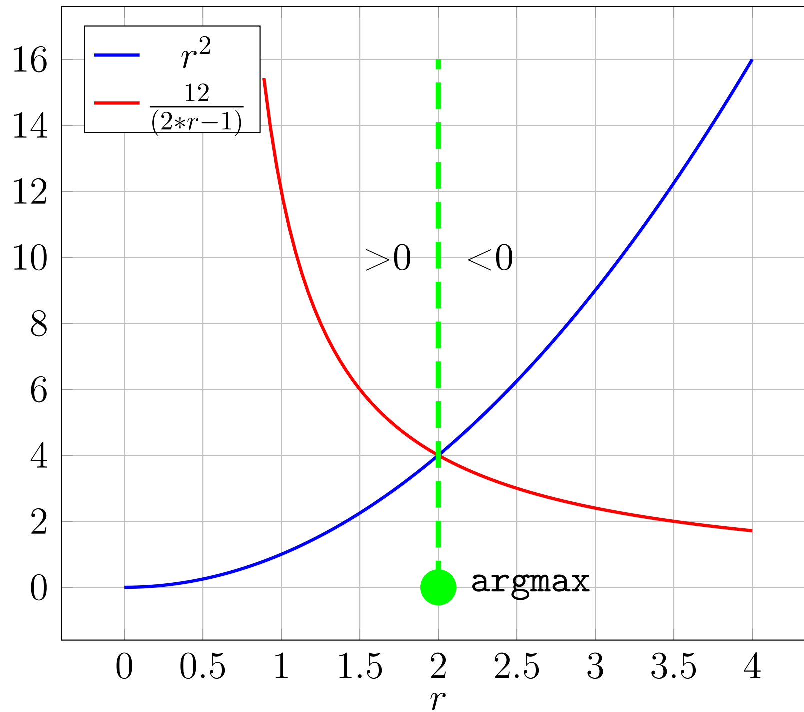

Рис. 2: Иллюстрация смены знака для $r^2 < \frac{n^{p_1} n^{p_2}}{2r-1}$ при $n^{p_1} n^{p_2} = 12$

$2r^3 > 0$ и перепишем его в виде $r^2 < \frac{n^{p_1} n^{p_2}}{2r-1}$. Тогда по графику (Рис. 2) легко понять, что производная на рассматриваемом интервале будет менять знак только один раз. Точка в которой производная равна нулю, в данном случае, будет точкой максимума функции $\texttt{cost_diff}(r_i)$.

Таким образом, при $n_i^{p_2} > n_i^{p_1} > 0$, минимум будет достигаться в точке $r = 0$, так как в остальных точках $\texttt{cost_diff}(r_i) > 0$.

$n_i^{p_1} > n_i^{p_2} > 0$ В этом случае $\texttt{cost_diff}(r_i) \leq 0$ на рассматриваемом отрезке, а значит минимум достигается в точке $\operatorname{argmax} \texttt{cost_diff}'(r)$. Данное значение легко предварительно подсчитать с помощью приведённого выше неравенства для всех разумных (т.е. не слишком больших) значений $n^{p_1} n^{p_2}$.

Сводная информация о значениях на которых достигается минимум

$\texttt{cost_diff}(r_i)$ представлена в равенстве 8.

$$\operatorname{argmin} \texttt{cost_diff}(r_i) = \begin{cases} 0, & \text{если } n_i^{p_2} > n_i^{p_1} > 0 \\ \operatorname{argmax} \texttt{cost_diff}'(r), & \text{если } n_i^{p_1} > n_i^{p_2} > 0 \\ m_i - R, & \text{если } n_i^{p_1} = 0 \\ 0, & \text{если } n_i^{p_2} = 0 \end{cases} \tag{8}$$

В результате, чтобы сравнить планы, в которых есть обращение к ОТВ, мы можем посчитать минимально возможное значение суммы справа в неравенстве 3. Если при этом неравенство выполняется, то мы оставляем план с меньшей стоимостью. В противном случае мы не можем принять окончательного решения и должны оставить оба плана для дальнейшего рассмотрения.

6. Мнение сообщества разработчиков PostgreSQL о встраивании ОТВ

Предложение о реализации алгоритма встраивания ОТВ было высказано автором в рассылке для разработчиков PostgreSQL [11] в теме «CTE inlining». Проблемы, возникающие из-за того, что оптимизатор считает ОТВ «оптимизационной решёткой», несколько раз поднимались раньше [2] [3] [4], однако широкого обсуждения и принятие какого-либо решения не последовало.

В этот раз в обсуждении участвовало 3 core разработчика из 5, 5 major разработчиков, и ещё несколько контрибьюторов. Участники дискуссии сошлись во мнении, что «оптимизационная решётка» должна быть убрана по-умолчанию, а для возможности использования текущего поведения, необходимо расширить стандартный синтаксис объявления обобщённых выражений опциональным словом `MATERIALIZE`, которое можно будет добавить после `WITH` в объявлении ОТВ.

[2] https://www.postgresql.org/message-id/flat/4EA6E252.6030002%40linos.es#4EA6E252.6030002@linos.es
[3] https://www.postgresql.org/message-id/flat/29918.1320244719%40sss.pgh.pa.us#29918.1320244719@sss.pgh.pa.u
[4] https://www.postgresql.org/message-id/201209191305.44674.db@kavod.com

Заключение

В данной работе были рассмотрены методы оптимизации, применяемые к обобщённым табличным выражениям в различных СУБД. На основе существующих реализаций был предложен подход к реализации в PostgreSQL встраивания ОТВ, как основного метода оптимизации. Также был описан способ сравнения стоимости путей со встроенными ОТВ при построении плана выполнения запроса. Кроме того, были отмечены недостатки работы оптимизатора PostgreSQL с ОТВ в текущей версии СУБД и приведены примеры запросов, иллюстрирующие их. Предлагаемая инициатива по реализации механизма встраивания ОТВ получила поддержку сообщества разработчиков PostgreSQL.

Список литературы

[1] Cost-based Query Transformation in Oracle / Rafi Ahmed, Allison Lee, Andrew Witkowski et al. // Proceedings of the 32Nd International Conference on Very Large Data Bases. — VLDB '06. — VLDB Endowment, 2006. — P. 1026–1036. — URL: http://dl.acm.org/citation.cfm?id=1182635.1164215.

[2] Dayal Umeshwar. Of Nests aud Trees : A Untied Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers // Vldb. — 1987. — P. 197–208.

[3] Efficient exploitation of similar subexpressions for query processing / Jingren Zhou, Per Ake Larson, Johann Christoph Freytag, Wolfgang Lehner // SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data. — 2007. — no. April. — P. 533–544. — URL: http://portal.acm.org/citation.cfm?id=1247480.1247540.

[4] Enhanced subquery optimizations in Oracle / Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski et al. // Proceedings of the VLDB Endowment. — 2009. — aug. — Vol. 2, no. 2. — P. 1366–1377. — URL: http://dl.acm.org/citation.cfm?doid=1687553.1687563.

[5] Execution strategies for SQL subqueries / Mostafa Elhemali, César a. Galindo-Legaria, Torsten Grabs, Milind M. Joshi // Proceedings of the 2007 ACM SIGMOD international conference on Management of data - SIGMOD '07. — 2007. — P. 993. — URL: http://portal.acm.org/citation.cfm?doid=1247480.1247598.

[6] Graefe Goetz. The Cascades framework for query optimization // Data Engineering Bulletin. — 1995. — Vol. 18, no. 3. — P. 19–29. — URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.9460{&}rep=rep1{&}type=pdf.

[7] Kim Won. On optimizing an SQL-like nested query // ACM

Transactions on Database Systems. — 1982. — Vol. 7, no. 3. — P. 443–469.

[8] Optimization of common table expressions in MPP database systems / Amr El-Helw, Venkatesh Raghavan, Mohamed A. Soliman et al. // Proceedings of the VLDB Endowment. — 2015. — Vol. 8, no. 12. — P. 1704–1715. — URL: http://dl.acm.org/citation.cfm?id=2824032.2824068.

[9] Orca: A modular query optimizer architectur for big data // Sigmod. — 2014. — P. 337–348. — URL: http://dl.acm.org/citation.cfm?doid=2588555.2595637.

[10] PostgreSQL repository // The PostgreSQL Licence (PostgreSQL), The PostgreSQL Global Development Group. — URL: https://github.com/postgres/postgres (дата обращения: 02.05.2017).

[11] pgsql-hackers mailing list, The PostgreSQL Global Development Group. — URL: https://www.postgresql.org/list/pgsql-hackers/ (дата обращения: 22.05.2017).

[12] Демонстрационна база данных // The PostgreSQL Licence (PostgreSQL), Postgres Professional. — URL: https://postgrespro.ru/education/demodb (дата обращения: 02.05.2017).

Список Листингов

1. Количество мест в самолёте, с учётом класса обслуживания. 9
1. План к Запросу 1. 9
2. Запрос с ОТВ, эквивалентный по результату Запросу 1. 9
2. План к Запросу 2. 9
3. Нахождение стыковочных рейсов из Домодедово в Пулково. 11
3. План выполнения Запроса 3. 12
4. Нахождение стыковочных рейсов, со встроенными `FlightSeatsAvailable` и `FlightSeatsOccupied`. 14
4. План выполнения Запроса 4. Часть 1. 15
4. План выполнения Запроса 4. Часть 2. 16