

Оптимизация запросов Материализация



Авторские права

© Postgres Professional, 2019–2024

Авторы: Егор Рогов, Павел Лузанов, Павел Толмачев, Илья Баштанов

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

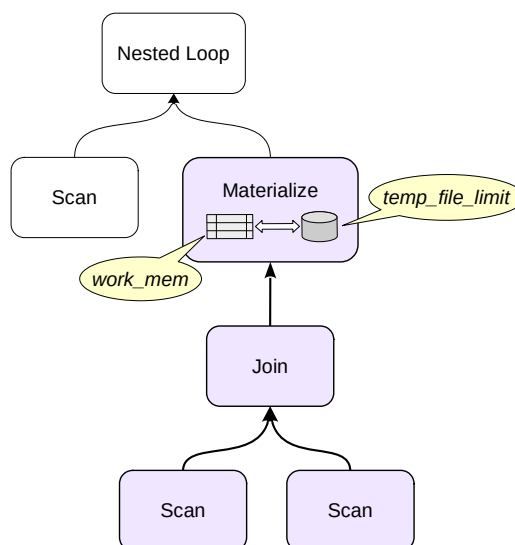
Материализация в запросах

Временные таблицы

Управление порядком соединений

Материализованные представления

Материализация —
сохранение промежуточного
набора строк
для последующего
многократного
использования



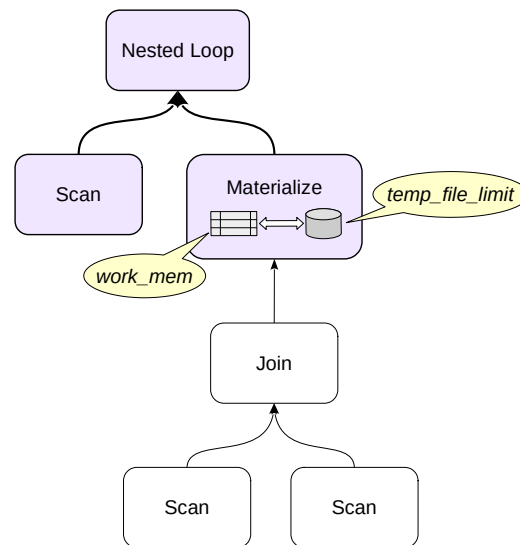
Материализацией называется сохранение промежуточного набора строк с целью повторного использования, как правило, многократного. Сохранять набор можно на разных уровнях: для конкретного запроса, на уровне сеанса или на уровне базы данных.

Обычно узлы запроса передают друг другу данные по принципу конвейера: когда алгоритму, выполняющемуся в узле плана, требуется очередная строка набора, узел обращается к одному из дочерних узлов за следующей порцией. Однако в некоторых случаях исполнителю запроса имеет смысл (а иногда и необходимо) сразу получить все строки, сохранить их и иметь возможность обращаться к сохраненному результату повторно. Такое сохранение выполняет узел Materialize.

Строки сохраняются в оперативной памяти, пока их объем укладывается в ограничение *work_mem*. При превышении этого ограничения все строки сбрасываются во временный файл и при необходимости читаются оттуда. Объем всех временных файлов одного сеанса ограничен значением параметра *temp_file_limit*.

Например, на слайде верхний узел Nested Loop выполняет соединение вложенным циклом, но к внутреннему набору данных нет эффективного доступа: он вычисляется с помощью другого соединения. Чтобы не повторять многократно вложенное соединение, его результат можно материализовать.

Материализация —
сохранение промежуточного
набора строк
для последующего
многократного
использования



После того как результат вложенного соединения материализован, узел Nested Loop будет обращаться к уже готовому набору строк.

Узел Materialize

Если операция требует существенных ресурсов, а к ее результату обращаются многократно, планировщик может выбрать план с узлом Materialize, в котором полученные строки накапливаются для повторного использования:

```
=> EXPLAIN (costs off)
SELECT a1.city, a2.city
FROM airports a1, airports a2
WHERE a1.timezone = 'Europe/Moscow'
      AND abs(a2.coordinates[1]) > 66.652; -- за полярным кругом
```

QUERY PLAN

```
-----
Nested Loop
  -> Seq Scan on airports_data ml
      Filter: (timezone = 'Europe/Moscow'::text)
  -> Materialize
      -> Seq Scan on airports_data ml_1
          Filter: (abs(coordinates[1]) > '66.652'::double precision)
(6 rows)
```

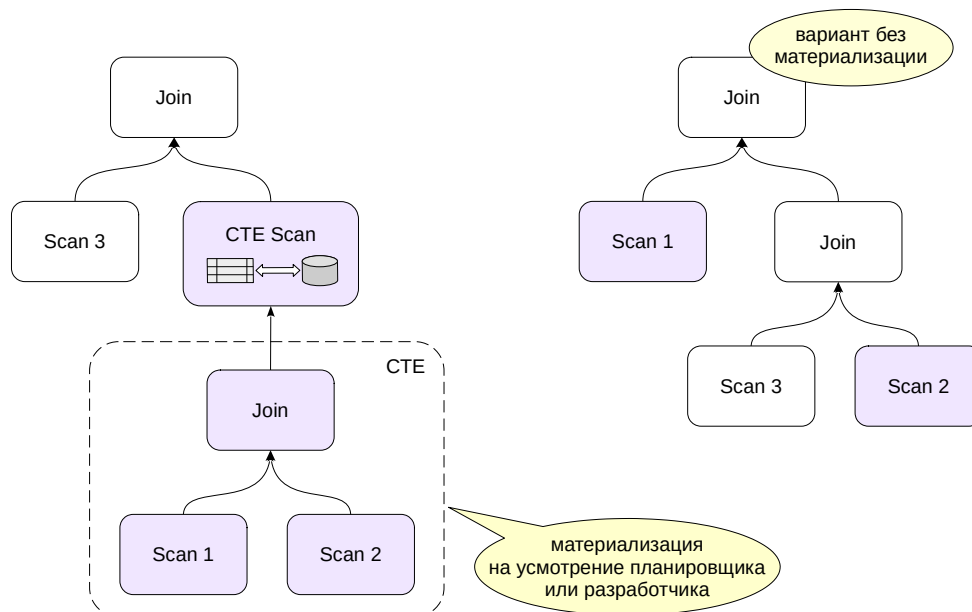
Здесь для предиката внутреннего набора строк нет подходящего индекса, поэтому план с материализацией оказывается выгодным.

В некоторых случаях планировщик использует материализацию внутреннего набора данных и при соединении слиянием, чтобы иметь возможность перечитать часть строк при повторяющихся значениях во внешнем наборе:

```
=> EXPLAIN (costs off)
SELECT * FROM
  (SELECT * FROM tickets ORDER BY ticket_no) AS t
JOIN
  (SELECT * FROM ticket_flights ORDER BY ticket_no) AS tf
ON tf.ticket_no = t.ticket_no;
```

QUERY PLAN

```
-----
Merge Join
  Merge Cond: (tickets.ticket_no = ticket_flights.ticket_no)
  -> Index Scan using tickets_pkey on tickets
  -> Materialize
      -> Index Scan using ticket_flights_pkey on ticket_flights
(5 rows)
```



Общие табличные выражения (common table expression, CTE), они же подзапросы WITH, — прекрасный способ структурировать запрос и сделать его более понятным. В отличие от обычных подзапросов, использование CTE не приводит к большой вложенности.

По возможности планировщик раскрывает (не материализует) подзапросы CTE. Это позволяет ему выбирать оптимальный порядок соединений. По умолчанию подзапрос материализуется в следующих ситуациях:

- Основной запрос обращается к подзапросу несколько раз — чтобы не повторять вычисления.
В этом случае материализацию можно отменить, указав предложение AS NOT MATERIALIZED.
- Подзапрос имеет побочные эффекты (изменяет данные) — чтобы изменение произошло ровно один раз. (К побочным эффектам относится также обращение к изменчивым функциям, см. тему «Функции».)
В этом случае материализацию отменить невозможно.

Материализацию всегда можно принудительно включить, указав предложение AS MATERIALIZED.

<https://postgrespro.ru/docs/postgresql/16/queries-with#QUERIES-WITH-CTE-MATERIALIZATION>

Материализация CTE

Оптимизатор старается не материализовать подзапросы в WITH без надобности:

```
=> EXPLAIN (costs off)
WITH q AS (
  SELECT f.flight_id, a.aircraft_code
  FROM flights f
  JOIN aircrafts a ON a.aircraft_code = f.aircraft_code
)
SELECT *
FROM q
JOIN seats s ON s.aircraft_code = q.aircraft_code
WHERE s.seat_no = '1A';
```

QUERY PLAN

```
-----
Hash Join
  Hash Cond: (f.aircraft_code = ml.aircraft_code)
    -> Hash Join
        Hash Cond: (f.aircraft_code = s.aircraft_code)
        -> Seq Scan on flights f
        -> Hash
            -> Seq Scan on seats s
            Filter: ((seat_no)::text = '1A'::text)
    -> Hash
        -> Seq Scan on aircrafts_data ml
(10 rows)
```

Но явное указание заставляет оптимизатор планировать подзапрос отдельно от основного запроса:

```
=> EXPLAIN (costs off)
WITH q AS MATERIALIZED (
  SELECT f.flight_id, a.aircraft_code
  FROM flights f
  JOIN aircrafts a ON a.aircraft_code = f.aircraft_code
)
SELECT *
FROM q
JOIN seats s ON s.aircraft_code = q.aircraft_code
WHERE s.seat_no = '1A';
```

QUERY PLAN

```
-----
Hash Join
  Hash Cond: (q.aircraft_code = s.aircraft_code)
  CTE q
    -> Hash Join
        Hash Cond: (f.aircraft_code = ml.aircraft_code)
        -> Seq Scan on flights f
        -> Hash
            -> Seq Scan on aircrafts_data ml
    -> CTE Scan on q
  -> Hash
      -> Seq Scan on seats s
      Filter: ((seat_no)::text = '1A'::text)
(12 rows)
```

Если подзапрос используется в запросе несколько раз, планировщик выбирает материализацию, чтобы не выполнять одни и те же действия многократно:

```
=> EXPLAIN (analyze, costs off, buffers)
WITH b AS (
  SELECT * FROM bookings
)
SELECT *
FROM b AS b1
JOIN b AS b2 ON b1.book_ref = b2.book_ref
WHERE b2.book_ref = '000112';
```

QUERY PLAN

```
-----
Nested Loop (actual time=0.033..714.518 rows=1 loops=1)
  Buffers: shared read=13447, temp read=8483 written=8483
  CTE b
    -> Seq Scan on bookings (actual time=0.018..150.006 rows=2111110 loops=1)
        Buffers: shared read=13447
    -> CTE Scan on b b1 (actual time=0.028..196.997 rows=1 loops=1)
        Filter: (book_ref = '000112'::bpchar)
        Rows Removed by Filter: 2111109
        Buffers: shared read=1, temp read=8483 written=1
    -> CTE Scan on b b2 (actual time=0.003..517.517 rows=1 loops=1)
        Filter: (book_ref = '000112'::bpchar)
        Rows Removed by Filter: 2111109
        Buffers: shared read=13446, temp written=8482
Planning:
  Buffers: shared hit=26 read=1
Planning Time: 0.190 ms
Execution Time: 721.972 ms
(17 rows)
```

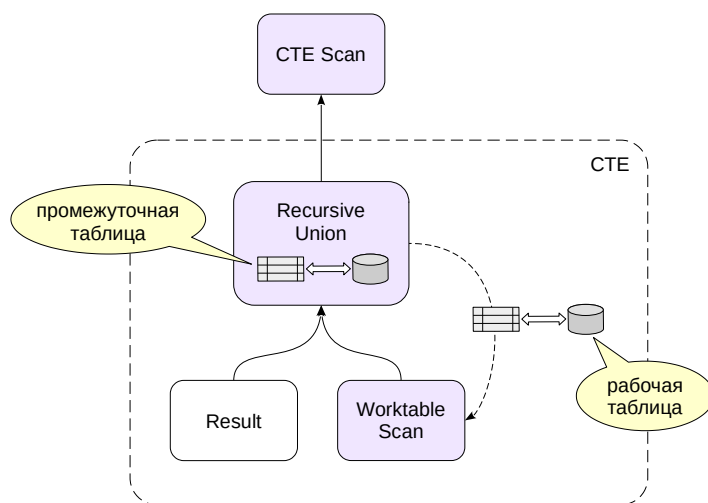
Обычно это оправдано, особенно если в подзапросе происходят затратные вычисления. Но в некоторых случаях (как в этом примере) план без материализации может оказаться эффективнее, и тогда ее можно отменить. Сравните значение buffers с предыдущим вариантом:

```
=> EXPLAIN (analyze, costs off, buffers)
WITH b AS NOT MATERIALIZED (
  SELECT * FROM bookings
)
SELECT *
FROM b AS b1
JOIN b AS b2 ON b1.book_ref = b2.book_ref
WHERE b2.book_ref = '000112';
```

QUERY PLAN

```
-----
Nested Loop (actual time=0.172..0.175 rows=1 loops=1)
  Buffers: shared hit=4 read=4
  -> Index Scan using bookings_pkey on bookings (actual time=0.153..0.154 rows=1
loops=1)
      Index Cond: (book_ref = '000112'::bpchar)
      Buffers: shared read=4
  -> Index Scan using bookings_pkey on bookings bookings_1 (actual time=0.011..0.012
rows=1 loops=1)
      Index Cond: (book_ref = '000112'::bpchar)
      Buffers: shared hit=4
Planning:
  Buffers: shared hit=2
Planning Time: 0.475 ms
Execution Time: 0.203 ms
(12 rows)
```

Однако если подзапрос изменяет данные, материализация будет выполнена в любом случае: изменение обязано произойти только один раз.



Рекурсивные запросы строятся на базе общих табличных выражений.

Узел Recursive Union использует рабочую и промежуточную таблицы: рабочая таблица хранит строки, получаемые на текущей итерации, а промежуточная — накапливает результат. Содержимое рабочей таблицы считывается в рекурсивной части запроса узлом Worktable Scan.

Каждая из этих двух таблиц материализуется по тем же правилам: пока содержимое таблицы помещается в *work_mem*, она хранится в памяти, а затем все строки сбрасываются на диск.

После выполнения рекурсивного запроса накопленное содержимое промежуточной таблицы передается узлу CTE Scan, а рабочая таблица отбрасывается.

<https://postgrespro.ru/docs/postgresql/16/queries-with#QUERIES-WITH-RECURSIVE>

Рекурсивные запросы

Убедимся в том, что рекурсивный запрос материализует промежуточные данные. Для этого намеренно напишем его так, чтобы рабочая и промежуточная таблицы не поместились в память:

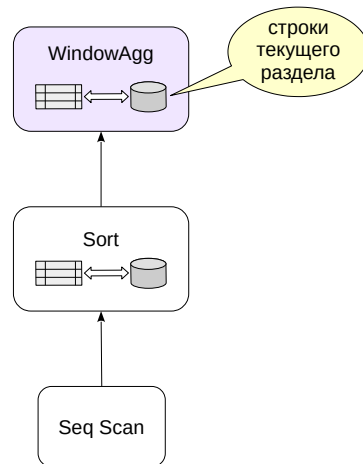
```
=> EXPLAIN (analyze, buffers, costs off, timing off)
WITH RECURSIVE r(n, airport_code) AS (
  SELECT 1, a.airport_code
  FROM airports a
  UNION ALL
  SELECT r.n+1, f.arrival_airport
  FROM r
  JOIN flights f ON f.departure_airport = r.airport_code
  WHERE r.n < 2
)
SELECT * FROM r;
```

QUERY PLAN

```
-----
CTE Scan on r (actual rows=214971 loops=1)
  Buffers: shared read=2627, temp read=473 written=945
  CTE r
    -> Recursive Union (actual rows=214971 loops=1)
      Buffers: shared read=2627, temp read=473 written=473
      -> Seq Scan on airports_data ml (actual rows=104 loops=1)
        Buffers: shared read=3
      -> Hash Join (actual rows=107434 loops=2)
        Hash Cond: (f.departure_airport = r_1.airport_code)
        Buffers: shared read=2624, temp read=473 written=1
        -> Seq Scan on flights f (actual rows=214867 loops=1)
          Buffers: shared read=2624
        -> Hash (actual rows=52 loops=2)
          Buckets: 1024 Batches: 1 Memory Usage: 13kB
          Buffers: temp read=473 written=1
          -> WorkTable Scan on r r_1 (actual rows=52 loops=2)
            Filter: (n < 2)
            Rows Removed by Filter: 107434
            Buffers: temp read=473 written=1

Planning:
  Buffers: shared hit=12 read=1 dirtied=1
Planning Time: 0.189 ms
Execution Time: 161.316 ms
(23 rows)
```

Обратите внимание на строки temp read/written в узлах WorkTable Scan и Recursive Union, специфичных для рекурсивных запросов.



При вычислении оконных функций в узле WindowAgg (см. тему «Сортировка») тоже применяется материализация: строки текущего раздела (PARTITION BY) могут неоднократно попадать в рамку, и поэтому материализуются.

В этом случае материализованными строками пользуется только сам узел WindowAgg: родительскому узлу передается результат вычислений, а не эти промежуточные данные.

<https://postgrespro.ru/docs/postgresql/16/tutorial-window>

Таблица, доступная в пределах одного сеанса

- регистрируется в системном каталоге
- не журналируется
- создаются файлы на диске
- кеширование в локальной памяти сеанса (*temp_buffers*)

Очистка и анализ

- только вручную

Для получения промежуточных данных может использоваться сложный алгоритм, например, написанный на процедурном языке. В таком случае данные не получится посчитать в СТЕ, но их можно поместить во временную таблицу и использовать в нескольких запросах.

Временные таблицы удобнее для промежуточных данных, чем обычные, поскольку они существуют только в пределах сеанса или транзакции (в зависимости от указанного при создании режима) и автоматически удаляются вместе с данными и зависимыми объектами (представлениями и индексами). К тому же временные таблицы не журналируются и кешируются в локальной памяти процесса, обслуживающего сеанс — это делает работу с ними более быстрой.

Локальная память процесса недоступна процессам автоочистки, поэтому очистку и анализ следует проводить вручную. Память для кеша выделяется по необходимости и ограничена для сеанса параметром *temp_buffers* (после первого обращения к временной таблице изменить ограничение уже нельзя).

<https://postgrespro.ru/docs/postgresql/16/sql-createtable>

Однако для временных таблиц создаются записи в системном каталоге и файлы на диске. Поэтому при массовой работе с временными таблицами — например, в системе 1С — будет разрастаться системный каталог и увеличится нагрузка на файловую систему. Поэтому для 1С используют специальные патчи, нивелирующие эти нежелательные эффекты.

Временные таблицы

Создадим временную таблицу:

```
=> CREATE TEMP TABLE airports_msk
ON COMMIT PRESERVE ROWS -- по умолчанию
AS SELECT *
FROM airports
WHERE timezone = 'Europe/Moscow';
```

SELECT 44

Временная таблица создается во временной схеме pg_temp.

По умолчанию таблица существует до конца сеанса; можно указать, чтобы при завершении транзакции удалялись строки (ON COMMIT DELETE ROWS) или сама таблица (ON COMMIT DROP).

Зачастую есть смысл проанализировать только что заполненную временную таблицу, прежде чем использовать ее в запросах. Сравните оценки кардинальности при выборе из обычной таблицы и из временной:

```
=> EXPLAIN
SELECT *
FROM airports
WHERE timezone = 'Europe/Moscow';
```

QUERY PLAN

```
Seq Scan on airports_data ml (cost=0.00..26.52 rows=44 width=99)
  Filter: (timezone = 'Europe/Moscow'::text)
(2 rows)
```

```
=> EXPLAIN
SELECT *
FROM airports_msk;
```

QUERY PLAN

```
Seq Scan on airports_msk (cost=0.00..15.20 rows=520 width=128)
(1 row)
```

В последнем случае оптимизатор, не имея статистики, предполагает, что таблица занимает 10 страниц, на которых умещается 520 строк, и стоимость получается завышенной:

```
=> SELECT relpages, reltuples,
       10 * current_setting('seq_page_cost')::float +
       520 * current_setting('cpu_tuple_cost')::float AS cost
FROM pg_class
WHERE relname = 'airports_msk'
AND relpersistence = 't'; -- временная
```

```
relpages | reltuples | cost
-----+-----+-----
      0 |         -1 | 15.2
(1 row)
```

Если проанализировать таблицу, оценка станет точной:

```
=> ANALYZE airports_msk;
```

ANALYZE

```
=> EXPLAIN
SELECT *
FROM airports_msk;
```

QUERY PLAN

```
Seq Scan on airports_msk (cost=0.00..1.44 rows=44 width=67)
(1 row)
```

Количество вариантов соединений растет экспоненциально с ростом количества таблиц в запросе

Соединение JOIN

полный перебор при числе соединений не более *join_collapse_limit* = 8
дальше — группами по *join_collapse_limit* таблиц

Соединение через запятую

полный перебор при числе соединений не более *geqo_threshold* = 12
дальше — генетический алгоритм

Материализация — подсказка планировщику

С ростом количества таблиц в запросе число вариантов соединений и, как следствие, затраты на выбор плана растут экспоненциально.

Если число соединений (записанных в виде JOIN) превышает значение *join_collapse_limit* (по умолчанию 8), планировщик рассматривает возможные соединения в группах по *join_collapse_limit* таблиц, а затем соединяет эти группы между собой.

(Для подзапросов в предложении FROM есть похожий параметр — *from_collapse_limit*.)

Если таблицы соединены запятой (без ключевого слова JOIN), планировщик рассматривает все варианты соединений, но переключается на генетический алгоритм оптимизации, когда количество соединений превышает *geqo_threshold* (12 по умолчанию).

Все это может приводить к неоптимальным планам. Подробности можно прочитать в статье Павла Толмачева:

<https://habr.com/ru/company/postgrespro/blog/662021/>

Вручную управляя материализацией — либо с помощью CTE, либо разбивая запрос на части и задействуя временные таблицы, — разработчик может разделить таблицы на группы, каждую из которых оптимизатор будет планировать отдельно. Обычно CTE — более простой и эффективный способ, но второй вариант позволяет проанализировать временную таблицу и сообщить таким образом планировщику больше информации о данных.

<https://postgrespro.ru/docs/postgresql/16/explicit-joins>

Управление порядком соединений

Значение по умолчанию параметра `join_collapse_limit` выбрано так, чтобы планирование соединения такого количества таблиц не требовало чрезмерных ресурсов:

```
=> SHOW join_collapse_limit;
```

```
join_collapse_limit
-----
8
(1 row)
```

Выполним запрос, в котором используется явное соединение таблиц с помощью ключевого слова `JOIN`:

```
=> EXPLAIN (costs on)
SELECT *
FROM tickets t
JOIN ticket_flights tf ON (tf.ticket_no = t.ticket_no)
JOIN flights f ON (f.flight_id = tf.flight_id);
```

QUERY PLAN

```
-----
Hash Join (cost=171631.56..778504.23 rows=8391030 width=199)
  Hash Cond: (tf.ticket_no = t.ticket_no)
    -> Hash Join (cost=9767.51..302695.70 rows=8391030 width=95)
      Hash Cond: (tf.flight_id = f.flight_id)
        -> Seq Scan on ticket_flights tf (cost=0.00..153870.30 rows=8391030 width=32)
        -> Hash (cost=4772.67..4772.67 rows=214867 width=63)
          -> Seq Scan on flights f (cost=0.00..4772.67 rows=214867 width=63)
    -> Hash (cost=78909.69..78909.69 rows=2949469 width=104)
      -> Seq Scan on tickets t (cost=0.00..78909.69 rows=2949469 width=104)
(9 rows)
```

В выбранном плане сначала соединяются таблицы рейсов (`flights`) и перелетов (`ticket_flights`), а затем результат соединяется с билетами (`tickets`).

Установив параметр `join_collapse_limit` в единицу, можно зафиксировать порядок соединений:

```
=> SET join_collapse_limit = 1;
```

SET

```
=> EXPLAIN (costs on)
SELECT *
FROM tickets t
JOIN ticket_flights tf ON (tf.ticket_no = t.ticket_no)
JOIN flights f ON (f.flight_id = tf.flight_id);
```

QUERY PLAN

```
-----
Hash Join (cost=171631.56..860448.23 rows=8391030 width=199)
  Hash Cond: (tf.flight_id = f.flight_id)
    -> Hash Join (cost=161864.05..498568.84 rows=8391030 width=136)
      Hash Cond: (tf.ticket_no = t.ticket_no)
        -> Seq Scan on ticket_flights tf (cost=0.00..153870.30 rows=8391030 width=32)
        -> Hash (cost=78909.69..78909.69 rows=2949469 width=104)
          -> Seq Scan on tickets t (cost=0.00..78909.69 rows=2949469 width=104)
    -> Hash (cost=4772.67..4772.67 rows=214867 width=63)
      -> Seq Scan on flights f (cost=0.00..4772.67 rows=214867 width=63)
(9 rows)
```

Теперь таблицы соединяются друг с другом в том порядке, в котором они перечислены в запросе, несмотря на то, что итоговая стоимость запроса выше.

Однако такой способ перекладывает на автора запроса слишком много ответственности. Планировщик сможет выбирать только то, какой из наборов строк поставить в соединении внешним, а какой — внутренним.

```
=> RESET join_collapse_limit;
```

RESET

С похожим параметром `from_collapse_limit` предлагается познакомиться в практике.

Материализация результата запроса

- только для чтения
- возможно создание индексов

Обновление данных

- ручная синхронизация
- инкрементальное обновление — расширение `pg_ivm`

Очистка и анализ

- вручную и автоматически

Материализованное представление — это именованный результат запроса, сохраненный на уровне базы данных. К материализованному представлению можно обращаться как к обычной таблице, доступной только для чтения.

По материализованному представлению можно построить индексы (но нельзя добавить ограничение целостности — ограничения должны проверяться в базовых таблицах). Для материализованного представления собирается такая же статистика, как и для обычных таблиц.

В отличие от обычных представлений, строки материализованного представления не изменяются при изменении базовых таблиц. Синхронизация выполняется вручную.

<https://postgrespro.ru/docs/postgresql/16/sql-creatematerializedview>

Полная синхронизация материализованного представления может быть слишком затратной. Штатное инкрементальное обновление (отдельных строк по мере изменения базовых таблиц) не доступно, но такую возможность дает расширение `pg_ivm` (автор — Юго Харата, https://github.com/sraoss/pg_ivm).

Материализованные представления

Создадим материализованное представление:

```
=> CREATE MATERIALIZED VIEW airports_msk AS
SELECT *
FROM airports
WHERE timezone = 'Europe/Moscow';
```

SELECT 44

```
=> \dt airports_msk
```

```
          List of relations
Schema |      Name      | Type | Owner
-----+-----+-----+-----
pg_temp_3 | airports_msk | table | postgres
(1 row)
```

Схема pg_temp при поиске проверяется первой, теперь придется обращаться к материализованному представлению по полному имени. Это неудобно, лучше удалить временную таблицу:

```
=> DROP TABLE pg_temp.airports_msk;
```

DROP TABLE

Материализованные представления можно индексировать:

```
=> CREATE UNIQUE INDEX ON airports_msk (airport_code);
```

CREATE INDEX

```
=> EXPLAIN (costs off) SELECT *
FROM airports_msk
ORDER BY airport_code
LIMIT 3;
```

```
          QUERY PLAN
-----
Limit
->  Index Scan using airports_msk_airport_code_idx on airports_msk
(2 rows)
```

При анализе и автоанализе для материализованных представлений собирается та же статистика, что и для таблиц.

При изменении содержимого базовых таблиц строки материализованного представления не изменяются:

```
=> INSERT INTO airports_data (airport_code, airport_name, city, coordinates, timezone)
VALUES ('ZIA', '{"ru": "Жуковский"}', '{}', point(38.1517, 55.5533), 'Europe/Moscow');
```

INSERT 0 1

```
=> SELECT count(*)
FROM airports_msk
WHERE airport_code = 'ZIA';
```

```
count
-----
0
(1 row)
```

Синхронизацию нужно проводить явно. Команда REFRESH полностью блокирует материализованное представление на время перестроения, что может быть нежелательным. В данном случае этого можно избежать, указав CONCURRENTLY, поскольку на материализованном представлении создан уникальный индекс:

```
=> REFRESH MATERIALIZED VIEW CONCURRENTLY airports_msk;
```

REFRESH MATERIALIZED VIEW

```
=> SELECT count(*)
FROM airports_msk
WHERE airport_code = 'ZIA';
```

```
count
-----
1
(1 row)
```


Оптимизатор может материализовать строки, полученные узлами плана, для повторного использования

Можно управлять материализацией с помощью CTE, временных таблиц и материализованных представлений

Материализация позволяет управлять порядком соединений

1. В начале демонстрации приводились примеры двух запросов с узлом Materialize. Выдайте планировщику указание не использовать этот узел и проверьте, в каком случае он сможет обойтись без материализации, а в каком — нет.
2. Проверьте план выполнения запроса при значениях параметра *from_collapse_limit* 8 (по умолчанию) и 1:

```
SELECT *
FROM
(
  SELECT *
  FROM ticket_flights tf, tickets t
  WHERE tf.ticket_no = t.ticket_no
) ttf,
flights f
WHERE f.flight_id = ttf.flight_id;
```

18

1. Воспользуйтесь параметром *enable_material*:
<https://postgrespro.ru/docs/postgresql/16/runtime-config-query#GUC-ENABLE-MATERIAL>

2. Прочитайте про параметр в документации:
<https://postgrespro.ru/docs/postgresql/16/explicit-joins>

1. Отключение узла Materialize

Проверим план первого запроса из демонстрации:

```
=> EXPLAIN SELECT a1.city, a2.city
FROM airports a1, airports a2
WHERE a1.timezone = 'Europe/Moscow'
AND abs(a2.coordinates[1]) > 66.652;
```

QUERY PLAN

```
-----
Nested Loop (cost=0.00..805.90 rows=1540 width=64)
-> Seq Scan on airports_data ml (cost=0.00..4.30 rows=44 width=49)
    Filter: (timezone = 'Europe/Moscow'::text)
-> Materialize (cost=0.00..4.74 rows=35 width=49)
    -> Seq Scan on airports_data ml_1 (cost=0.00..4.56 rows=35 width=49)
        Filter: (abs(coordinates[1]) > '66.652'::double precision)
(6 rows)
```

Попросим планировщик не использовать материализацию:

```
=> SET enable_material = off;
```

SET

```
=> EXPLAIN SELECT a1.city, a2.city
FROM airports a1, airports a2
WHERE a1.timezone = 'Europe/Moscow'
AND abs(a2.coordinates[1]) > 66.652;
```

QUERY PLAN

```
-----
Nested Loop (cost=0.00..948.16 rows=1540 width=64)
-> Seq Scan on airports_data ml_1 (cost=0.00..4.56 rows=35 width=49)
    Filter: (abs(coordinates[1]) > '66.652'::double precision)
-> Seq Scan on airports_data ml (cost=0.00..4.30 rows=44 width=49)
    Filter: (timezone = 'Europe/Moscow'::text)
(5 rows)
```

Планировщик обошелся без материализации, но стоимость плана увеличилась.

Проверим второй запрос:

```
=> RESET enable_material;
```

RESET

```
=> EXPLAIN (costs off)
SELECT * FROM
  (SELECT * FROM tickets ORDER BY ticket_no) AS t
JOIN
  (SELECT * FROM ticket_flights ORDER BY ticket_no) AS tf
ON tf.ticket_no = t.ticket_no;
```

QUERY PLAN

```
-----
Merge Join
  Merge Cond: (tickets.ticket_no = ticket_flights.ticket_no)
-> Index Scan using tickets_pkey on tickets
-> Materialize
    -> Index Scan using ticket_flights_pkey on ticket_flights
(5 rows)
```

```
=> SET enable_material = off;
```

SET

```
=> EXPLAIN (costs off)
SELECT * FROM
  (SELECT * FROM tickets ORDER BY ticket_no) AS t
JOIN
  (SELECT * FROM ticket_flights ORDER BY ticket_no) AS tf
ON tf.ticket_no = t.ticket_no;
```

QUERY PLAN

```
-----
Merge Join
  Merge Cond: (tickets.ticket_no = ticket_flights.ticket_no)
    -> Index Scan using tickets_pkey on tickets
    -> Materialize
        -> Index Scan using ticket_flights_pkey on ticket_flights
(5 rows)
```

В данном случае планировщик не может не использовать материализацию, поскольку для соединения слиянием требуется не только перемещаться по набору данных вперед, но и возвращаться назад.

```
=> RESET enable_material;
```

RESET

2. Параметр from_collapse_limit

Сначала посмотрим, как работает запрос со значением from_collapse_limit по умолчанию:

```
=> SHOW from_collapse_limit;
```

```
from_collapse_limit
-----
8
(1 row)
```

```
=> EXPLAIN (costs off, summary off, settings)
```

```
SELECT *
FROM
(
  SELECT *
  FROM ticket_flights tf, tickets t
  WHERE tf.ticket_no = t.ticket_no
) ttf,
flights f
WHERE f.flight_id = ttf.flight_id;
```

QUERY PLAN

```
-----
Hash Join
  Hash Cond: (tf.ticket_no = t.ticket_no)
    -> Hash Join
        Hash Cond: (tf.flight_id = f.flight_id)
        -> Seq Scan on ticket_flights tf
        -> Hash
            -> Seq Scan on flights f
    -> Hash
        -> Seq Scan on tickets t
Settings: search_path = 'bookings, public', jit = 'off'
(10 rows)
```

В подзапросе соединяются две таблицы, но оптимизатор раскрывает подзапрос и выбирает порядок соединения для всех трех таблиц. Первыми здесь соединяются таблицы перелетов (ticket_flights) и рейсов (flights).

Теперь уменьшим значение from_collapse_limit до единицы и посмотрим на новый план того же запроса:

```
=> SET from_collapse_limit = 1;
```

SET

```
=> EXPLAIN (costs off, summary off, settings)
```

```
SELECT *
FROM
(
  SELECT *
  FROM ticket_flights tf, tickets t
  WHERE tf.ticket_no = t.ticket_no
) ttf,
flights f
WHERE f.flight_id = ttf.flight_id;
```

QUERY PLAN

```
-----  
Hash Join  
  Hash Cond: (tf.flight_id = f.flight_id)  
    -> Hash Join  
      Hash Cond: (tf.ticket_no = t.ticket_no)  
        -> Seq Scan on ticket_flights tf  
        -> Hash  
          -> Seq Scan on tickets t  
    -> Hash  
      -> Seq Scan on flights f  
Settings: search_path = 'bookings, public', jit = 'off', from_collapse_limit = '1'  
(10 rows)
```

План запроса поменялся — сначала соединяются две таблицы из подзапроса, который теперь не раскрывается.

Если во FROM вместо списка таблиц использовать конструкцию JOIN, оптимизатор сначала преобразует каждую такую конструкцию в список таблиц (при этом обрабатывая таблицы группами не более чем по `join_collapse_limit`), а потом уже раскрывает подзапросы (на глубину не более чем `from_collapse_limit`). Поэтому имеет смысл устанавливать обоим параметрам равные значения.