

Способы соединения

Соединение вложенным циклом



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.

Общие соображения о соединениях

Соединение вложенным циклом

Модификации: левые, полу- и анти- соединения

Вычислительная сложность

Вложенный цикл в параллельных планах

Способы соединения — не соединения SQL

`inner/left/right/full/cross join`, `in`, `exists` — логические операции
способы соединения — механизм реализации

Соединяются не таблицы, а наборы строк

могут быть получены от любого узла дерева плана

Наборы строк соединяются попарно

порядок соединений важен с точки зрения производительности
обычно важен и порядок внутри пары

Мало получать данные с помощью рассмотренных методов доступа, надо еще и уметь соединять их. Для этого PostgreSQL предоставляет несколько способов.

Способы соединения представляют собой алгоритмы для соединения двух наборов строк. С их помощью реализуются и другие конструкции языка SQL, например, `EXISTS`. Не стоит путать одно с другим: SQL-соединения — это логические операции над двумя множествами; способы соединения PostgreSQL — это возможные реализации таких соединений, учитывающие производительность.

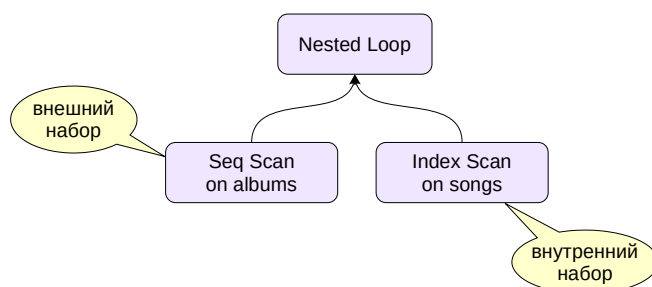
Часто можно услышать, что соединяются *таблицы*. Это удобное упрощение, но на самом деле в общем случае соединяются *наборы строк*. Эти наборы действительно могут быть получены непосредственно из таблицы (с помощью одного из методов доступа), но с тем же успехом могут, например, являться результатом соединения других наборов строк.

Наконец, наборы строк всегда соединяются попарно. Порядок, в котором соединяются таблицы, не важен с точки зрения логики запроса (например, `(a join b) join c` или `(b join c) join a`), но очень важен с точки зрения производительности. Как мы увидим дальше, важен и порядок, в котором соединяются два набора строк (`a join b` или `b join a`).

Nested Loop

Для каждой строки одного набора
перебираем подходящие строки другого набора

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```



Начнем с соединения вложенными циклами (nested loop), как с самого простого. Его алгоритм таков: для каждой строки одного из наборов перебираем и возвращаем соответствующие ему строки второго набора. По сути, это два вложенных цикла, отсюда и название способа.

Заметим, что ко второму (внутреннему) набору мы будем обращаться столько раз, сколько строк в первом (внешнем) наборе. Если нет эффективного метода доступа для поиска соответствующих строк во втором наборе (то есть, попросту говоря, индекса на таблице), придется неоднократно перебирать большое количество строк, не относящихся к делу. Очевидно, это будет не лучший выбор, хотя для небольших наборов алгоритм и в этом случае может оказаться достаточно эффективным.

В плане запроса мы будем видеть узел Nested Loop с двумя дочерними узлами (которые могут представлять не только методы доступа, но и другие операции, например, соединения или агрегации).

Nested Loop

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```

id	title	year
3	Let It Be	1970
1	Yellow Submarine	1969
6	Abbey Road	1969
4	The Beatles	1968

album_id	name
5	A Day in the Life
1	All Together Now
2	Another Girl
3	Across the Universe
1	All You Need Is Love
2	Act Naturally

Рисунки иллюстрируют этот способ соединения. На них:

- строки, к которым ранее уже был доступ, показаны серым;
- строки, доступ к которым выполняется на текущем шаге, выделены цветом;
- оранжевым контуром выделены строки, составляющие пару, подходящую по условию соединения (в данном примере — по равенству числовых идентификаторов).

Сначала мы читаем первую строку первого набора и находим ее пару во втором наборе. Соответствие нашлось, и у нас уже есть первая строка результата, которую можно вернуть вышестоящему узлу плана: («Let It Be», «Across the Universe»).

Nested Loop

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```

id	title	year	album_id	name
3	Let It Be	1970	5	A Day in the Life
1	Yellow Submarine	1969	1	All Together Now
6	Abbey Road	1969	2	Another Girl
4	The Beatles	1968	3	Across the Universe
			1	All You Need Is Love
			2	Act Naturally

Читаем вторую строку первого набора.

Для нее тоже перебираем пары из второго набора. Сначала возвращаем («Yellow Submarine», «All Together Now»)...

Nested Loop

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```

id	title	year	album_id	name
3	Let It Be	1970	5	A Day in the Life
1	Yellow Submarine	1969	1	All Together Now
6	Abbey Road	1969	2	Another Girl
4	The Beatles	1968	3	Across the Universe
			1	All You Need Is Love
			2	Act Naturally

...затем вторую пару («Yellow Submarine», «All You Need Is Love»).

Nested Loop

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```

id	title	year
3	Let It Be	1970
1	Yellow Submarine	1969
6	Abbey Road	1969
4	The Beatles	1968

album_id	name
5	A Day in the Life
1	All Together Now
2	Another Girl
3	Across the Universe
1	All You Need Is Love
2	Act Naturally

Переходим к третьей строке первого набора. Для нее соответствий нет.

Nested Loop

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```

id	title	year
3	Let It Be	1970
1	Yellow Submarine	1969
6	Abbey Road	1969
4	The Beatles	1968

album_id	name
5	A Day in the Life
1	All Together Now
2	Another Girl
3	Across the Universe
1	All You Need Is Love
2	Act Naturally

не все строки
внутреннего набора
были прочитаны

Для четвертой строки тоже нет соответствий. На этом работа соединения заканчивается.

Часть строк второго набора мы вообще не рассматривали — на рисунке они остались белыми.

(С исходным кодом алгоритма можно ознакомиться в файле <src/backend/executor/nodeNestloop.c>.)

Соединение вложенным циклом

Так выглядит план для соединения вложенным циклом, которое оптимизатор обычно предпочитает для небольших выборок (смотрим перелеты, включенные в два билета):

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no IN ('0005432312163','0005432312164');
```

QUERY PLAN

```
-----
Nested Loop
-> Index Scan using tickets_pkey on tickets t
    Index Cond: (ticket_no = ANY ('{0005432312163,0005432312164}'::bpchar[]))
-> Index Scan using ticket_flights_pkey on ticket_flights tf
    Index Cond: (ticket_no = t.ticket_no)
(5 rows)
```

Планировщик использует параметризованное соединение. Для каждой строки внешнего набора (узел Index Scan по таблице билетов) выбираются строки из внутреннего набора (узел Index Scan по таблице перелетов), отвечающие условию соединения. На каждой итерации внешнего цикла условие индексного доступа для внутреннего набора имеет вид `ticket_no = константа`.

Процесс повторяется до тех пор, пока внешний набор не исчерпает все строки.

Команда EXPLAIN ANALYZE позволяет узнать, сколько раз на самом деле выполнялся вложенный цикл (loops), сколько в среднем было выбрано строк (rows) и сколько потрачено времени (time) за один раз. Видно, что планировщик немного ошибся — в итоге получилось 8 строк вместо 6:

```
=> EXPLAIN (analyze, summary off)
SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no IN ('0005432312163','0005432312164');
```

QUERY PLAN

```
-----
Nested Loop (cost=0.99..46.10 rows=6 width=136) (actual time=2.744..6.852 rows=8
loops=1)
-> Index Scan using tickets_pkey on tickets t (cost=0.43..12.90 rows=2 width=104)
(actual time=1.603..1.621 rows=2 loops=1)
    Index Cond: (ticket_no = ANY ('{0005432312163,0005432312164}'::bpchar[]))
-> Index Scan using ticket_flights_pkey on ticket_flights tf (cost=0.56..16.57
rows=3 width=32) (actual time=0.724..2.603 rows=4 loops=2)
    Index Cond: (ticket_no = t.ticket_no)
(5 rows)
```

Вот еще один пример соединения вложенным циклом. Здесь мы выводим самолеты, способные обслуживать перелеты заданной протяженности:

```
=> EXPLAIN (costs off) SELECT *
FROM ( VALUES (1000), (10000) ) d(range)
JOIN aircrafts a ON a.range >= d.range;
```

QUERY PLAN

```
-----
Nested Loop
Join Filter: (ml.range >= "VALUES".column1)
-> Seq Scan on aircrafts_data ml
-> Materialize
    -> Values Scan on "VALUES*"
(5 rows)
```

Это непараметризованное соединение: для каждой строки внешнего набора придется читать весь внутренний набор (про узел Materialize мы будем говорить позже; в данном случае это просто значения из VALUES) и для полученных пар строк проверять условие Join Filter. По сути, это декартово произведение двух наборов строк с фильтрацией.

Обратите внимание, что соединение вложенным циклом позволяет соединять строки по любому условию, не обязательно по равенству значений.

Модификации

Существует несколько модификаций алгоритма. Для левого соединения:

```
=> EXPLAIN (costs off) SELECT *
FROM aircrafts a
LEFT JOIN seats s ON (a.aircraft_code = s.aircraft_code)
WHERE a.model LIKE 'Аэробус%';
```

QUERY PLAN

```
-----
Nested Loop Left Join
-> Seq Scan on aircrafts_data ml
    Filter: ((model ->> lang()) ~~ 'Аэробус% '::text)
-> Bitmap Heap Scan on seats s
    Recheck Cond: (ml.aircraft_code = aircraft_code)
    -> Bitmap Index Scan on seats_pkey
        Index Cond: (aircraft_code = ml.aircraft_code)
(7 rows)
```

Эта модификация возвращает строки, даже если для левого (a) набора строк не нашлось соответствия в правом (s) наборе.

Антисоединение возвращает те строки одного набора, для которых не нашлось соответствия в другом наборе. Такая модификация может использоваться для обработки предиката NOT EXISTS:

```
=> EXPLAIN (costs off) SELECT *
FROM aircrafts a
WHERE a.model LIKE 'Аэробус%'
AND NOT EXISTS (
    SELECT * FROM seats s WHERE s.aircraft_code = a.aircraft_code
);
```

QUERY PLAN

```
-----
Nested Loop Anti Join
-> Seq Scan on aircrafts_data ml
    Filter: ((model ->> lang()) ~~ 'Аэробус% '::text)
-> Index Only Scan using seats_pkey on seats s
    Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

Та же операция антисоединения используется и для аналогичного запроса, записанного иначе:

```
=> EXPLAIN (costs off) SELECT *
FROM aircrafts a
LEFT JOIN seats s ON (a.aircraft_code = s.aircraft_code)
WHERE a.model LIKE 'Аэробус%'
AND s.aircraft_code IS NULL;
```

QUERY PLAN

```
-----
Nested Loop Anti Join
-> Seq Scan on aircrafts_data ml
    Filter: ((model ->> lang()) ~~ 'Аэробус% '::text)
-> Index Scan using seats_pkey on seats s
    Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

Для предиката EXISTS может использоваться полусоединение, которое возвращает строки одного набора, для которых нашлось хотя бы одно соответствие в другом наборе:

```
=> EXPLAIN SELECT *
FROM aircrafts a
WHERE a.model LIKE 'Аэробус%'
AND EXISTS (
    SELECT * FROM seats s WHERE s.aircraft_code = a.aircraft_code
);
```

QUERY PLAN

```
-----  
Nested Loop Semi Join (cost=0.28..4.02 rows=1 width=40)  
  -> Seq Scan on aircrafts_data ml (cost=0.00..3.39 rows=1 width=72)  
      Filter: ((model ->> lang()) ~~ 'Аэробус% '::text)  
  -> Index Only Scan using seats_pkey on seats s (cost=0.28..6.88 rows=149 width=4)  
      Index Cond: (aircraft_code = ml.aircraft_code)  
(5 rows)
```

Обратите внимание: хотя в плане для таблицы s указано rows=149, на самом деле достаточно получить всего одну строку, чтобы понять значение предиката EXISTS.

PostgreSQL так и делает (actual rows=1):

```
=> EXPLAIN (analyze, costs off, timing off, summary off)  
SELECT *  
FROM aircrafts a  
WHERE a.model LIKE 'Аэробус%'  
AND EXISTS (  
    SELECT * FROM seats s WHERE s.aircraft_code = a.aircraft_code  
);
```

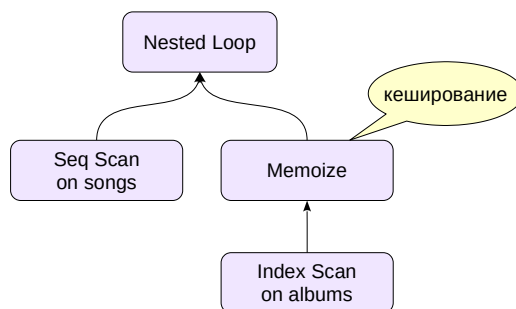
QUERY PLAN

```
-----  
Nested Loop Semi Join (actual rows=3 loops=1)  
  -> Seq Scan on aircrafts_data ml (actual rows=3 loops=1)  
      Filter: ((model ->> lang()) ~~ 'Аэробус% '::text)  
      Rows Removed by Filter: 6  
  -> Index Only Scan using seats_pkey on seats s (actual rows=1 loops=3)  
      Index Cond: (aircraft_code = ml.aircraft_code)  
      Heap Fetches: 0  
(7 rows)
```

Модификаций алгоритма вложенного цикла для правого (RIGHT) и полного (FULL) соединений не существует. Это связано с тем, что полный проход по второму набору строк может не выполняться.

Кеширование повторяющихся данных внутреннего набора

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```

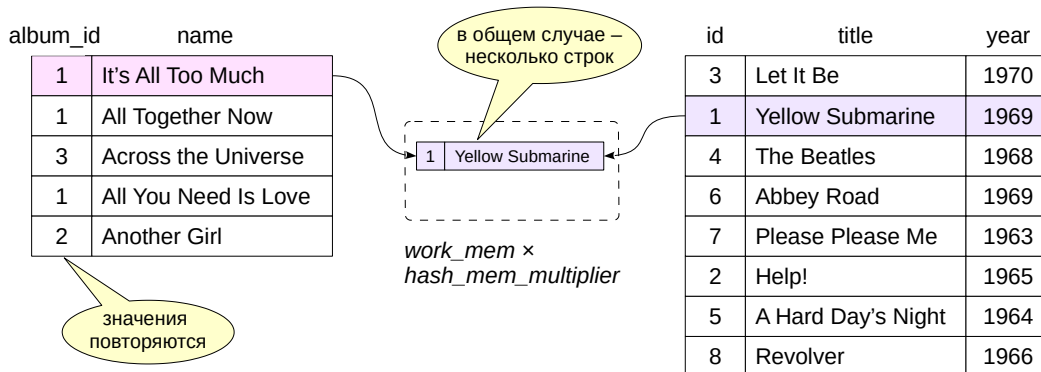


Если внутренний набор сканируется много раз с повторяющимися значениями параметров, иногда имеет смысл закешировать результат, чтобы не читать много раз одни и те же данные. Эта операция называется *мемоизацией*. Она выполняется в узле Memoize, который встраивается между Nested Loop и узлом, поставляющим данные.

(Если планировщик ошибается в своих расчетах, мемоизацию можно отключить, установив параметр *enable_memoize* в значение off.)

Для кеширования используется хеш-таблица. Ключом хеширования служит параметр или несколько параметров, с которыми выполняется обращение к внутреннему набору.

```
SELECT a.title, s.name
FROM albums a JOIN songs s ON a.id = s.album_id;
```

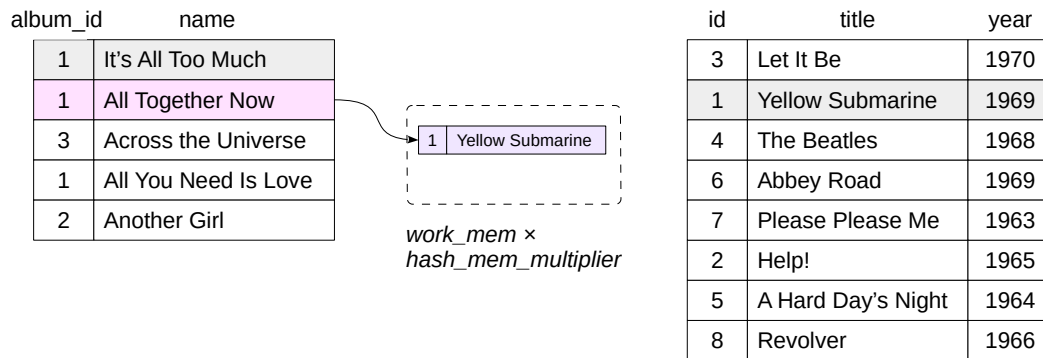


Рассмотрим пример. В отличие от предыдущего, здесь песен меньше, чем альбомов; к тому же, почти все они относятся к одному и тому же альбому.

Если необходимой строки нет в хеш-таблице, узел Memoize читает ее из внутреннего набора, сохраняет в кеше и возвращает родительскому узлу Nested Loop.

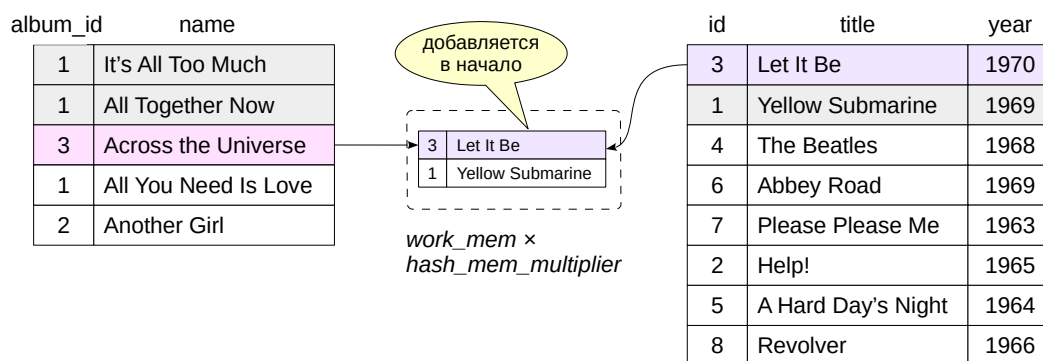
В общем случае значению параметра могут соответствовать несколько строк из внутреннего набора — будут закешированы все эти строки. Если все они не помещаются в память (которая ограничена размером $work_mem \times hash_mem_multiplier$), значение параметра игнорируется, поскольку кешировать только часть строк нет смысла. В плане запроса количество таких ситуаций будет показано как overflow.

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```



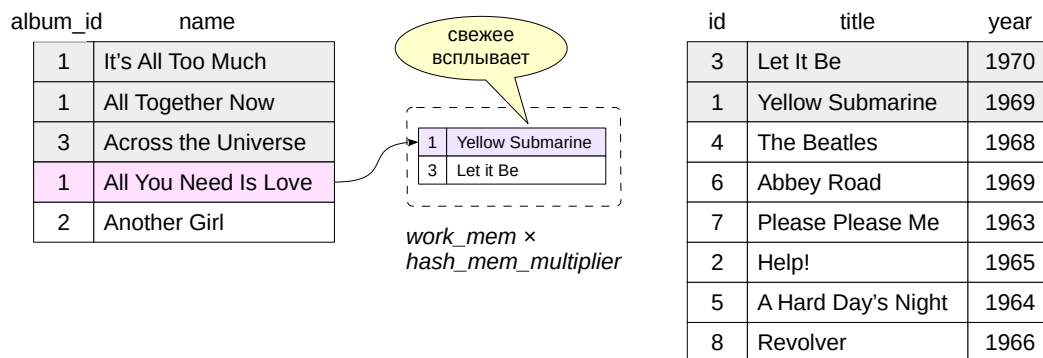
Если необходимая строка уже есть в кеше, узел Memoize сразу возвращает ее узлу Nested Loop. Обращения к внутреннему набору не происходит.

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```



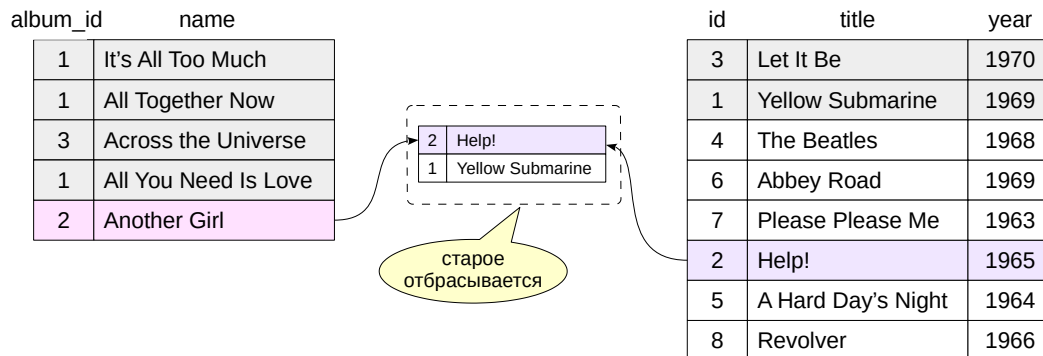
Пока в кеше хватает места, новые значения продолжают кешироваться. При этом новое значение добавляется в начало кеша.

```
SELECT a.title, s.name
FROM albums a JOIN songs s ON a.id = s.album_id;
```



Уже закешированное значение «всплывает наверх», когда к нему обращаются, а остальные, соответственно, опускаются вниз.

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```



Если память заканчивается, из кеша удаляются «нижние» строки, к которым дольше всего не было обращений. Таким образом реализуется алгоритм вытеснения LRU.

Мемоизация

Посмотрим план запроса, в котором соединяются таблицы перелетов и модели самолетов.

Пример подобран таким образом, что из внутреннего набора (таблица `aircrafts_data`) будет получена только одна строка по ключу `f.aircraft_code` (Cache Key), она и будет закеширована в узле Memoize:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM flights f
      JOIN aircrafts_data a ON f.aircraft_code = a.aircraft_code
WHERE f.flight_no = 'PG0003';
```

QUERY PLAN

```
-----
-----
Nested Loop (actual rows=113 loops=1)
  -> Bitmap Heap Scan on flights f (actual rows=113 loops=1)
      Recheck Cond: (flight_no = 'PG0003'::bpchar)
      Heap Blocks: exact=2
      -> Bitmap Index Scan on flights_flight_no_scheduled_departure_key (actual
rows=113 loops=1)
          Index Cond: (flight_no = 'PG0003'::bpchar)
  -> Memoize (actual rows=1 loops=113)
      Cache Key: f.aircraft_code
      Cache Mode: logical
      Hits: 112 Misses: 1 Evictions: 0 Overflows: 0 Memory Usage: 1kB
      -> Index Scan using aircrafts_pkey on aircrafts_data a (actual rows=1 loops=1)
          Index Cond: (aircraft_code = f.aircraft_code)
(12 rows)
```

Обратите внимание:

- в первый раз за нужной строкой приходится сходить в таблицу (Misses: 1);
- все повторные обращения обслуживаются кешем (Hits: 112), для этого хватило одного килобайта памяти;
- вытеснений из кеша не было (Evictions: 0);
- строки, выбранные из внутреннего набора, всегда умещались в выделенную память (Overflows: 0).

$\sim N \times M$, где

N — число строк во внешнем наборе данных,

M — среднее число строк внутреннего набора,
приходящееся на одну итерацию

Соединение эффективно только для небольшого числа строк

Если принять за N число строк во внешнем наборе данных, а за M — среднее число строк внутреннего набора, приходящееся на одну итерацию, то общая сложность соединения будет пропорциональна произведению $N \times M$.

Для непараметризованного соединения M будет в точности равно числу строк внутреннего набора; для параметризованного — M может быть значительно меньше.

Метод соединения вложенным циклом эффективен только для небольшого числа строк. В частности, такой метод (в сочетании с индексным доступом) характерен для OLTP-запросов, в которых надо очень быстро вернуть мало строк.

Стоимость соединения вложенным циклом

Посмотрим на стоимость следующего плана выполнения:

```
=> EXPLAIN SELECT *  
FROM tickets t  
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no  
WHERE t.ticket_no IN ('0005432312163','0005432312164');
```

QUERY PLAN

```
-----  
-----  
Nested Loop (cost=0.99..46.10 rows=6 width=136)  
-> Index Scan using tickets_pkey on tickets t (cost=0.43..12.90 rows=2 width=104)  
    Index Cond: (ticket_no = ANY ('{0005432312163,0005432312164}'::bpchar[]))  
-> Index Scan using ticket_flights_pkey on ticket_flights tf (cost=0.56..16.57  
rows=3 width=32)  
    Index Cond: (ticket_no = t.ticket_no)  
(5 rows)
```

Первый результат выдается сразу, без предварительных действий, поэтому начальная стоимость узла Nested Loop равна сумме начальных стоимостей дочерних узлов Index Scan.

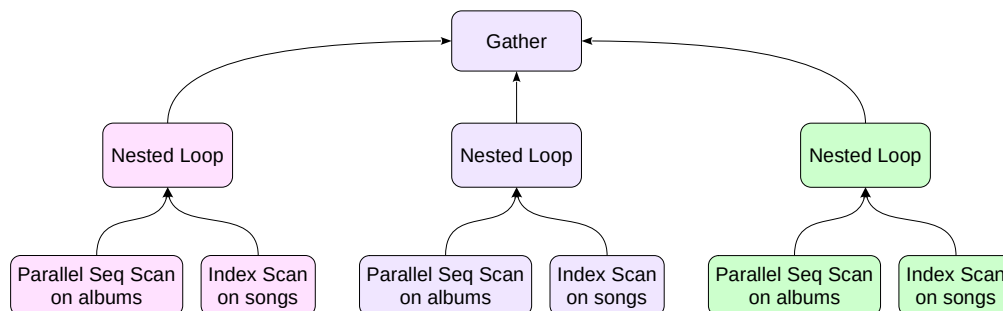
Полная стоимость узла Nested Loop складывается из:

- стоимости получения данных от внешнего набора (полная стоимость узла Index Scan по билетам);
- стоимости получения данных от внутреннего набора (полная стоимость узла Index Scan по перелетам), умноженной на расчетное число строк внешнего набора (2);
- стоимости процессорной обработки строк.

В общем случае формула более сложная, но основной вывод: стоимость пропорциональна $N \times M$, где N — число строк во внешнем наборе данных, а M — среднее число строк, внутреннего набора, читаемое за одну итерацию. В худшем случае стоимость получается квадратичной.

Внешний набор строк сканируется параллельно,
внутренний — последовательно каждым процессом

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```



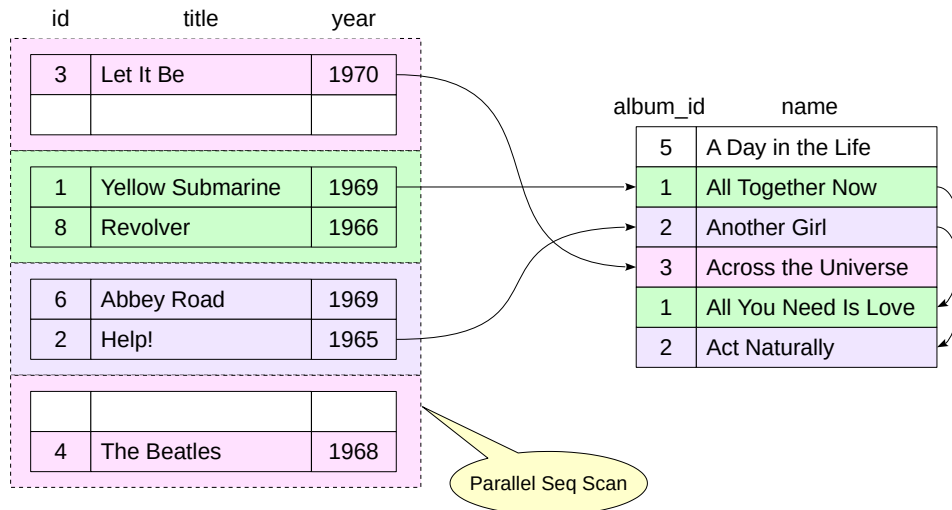
20

Соединение вложенным циклом может использоваться в параллельных планах.

Внешний набор строк сканируется несколькими рабочими процессами параллельно. Получив строку из внешнего набора, процесс затем перебирает соответствующие ему строки внутреннего набора последовательно.

В параллельных планах

```
SELECT a.title, s.name  
FROM albums a JOIN songs s ON a.id = s.album_id;
```



Чтобы получить очередную страницу из внешнего набора, процессы синхронизируются. К внутреннему набору данных каждый процесс обращается независимо от остальных.

Вложенный цикл в параллельных планах

Найдем всех пассажиров, купивших билеты на определенный рейс:

```
=> EXPLAIN (costs off)
SELECT t.passenger_name
FROM tickets t
      JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
      JOIN flights f ON f.flight_id = tf.flight_id
WHERE f.flight_id = 12345;
```

QUERY PLAN

```
-----
Nested Loop
->  Index Only Scan using flights_pkey on flights f
     Index Cond: (flight_id = 12345)
->  Gather
     Workers Planned: 2
     -> Nested Loop
         -> Parallel Seq Scan on ticket_flights tf
              Filter: (flight_id = 12345)
         -> Index Scan using tickets_pkey on tickets t
              Index Cond: (ticket_no = tf.ticket_no)

(10 rows)
```

На верхнем уровне используется соединение вложенным циклом. Внешний набор данных состоит из одной строки, полученной из таблицы рейсов (flights) по уникальному индексу.

Для получения внутреннего набора используется параллельный план. Каждый из процессов читает свою часть таблицы перелетов (ticket_flights) и соединяет ее с билетами (tickets) с помощью другого вложенного цикла.

Вложенный цикл не требует подготовительных действий

может отдавать результат соединения без задержек

Эффективен для небольших выборок

внешний набор строк не очень велик

к внутреннему есть эффективный доступ (обычно по индексу)

Зависит от порядка соединения

обычно лучше, если внешний набор меньше внутреннего

Поддерживает соединение по любому условию

как эквисоединения, так и любые другие

Сильной стороной способа соединения вложенными циклами является его простота: не требуется никаких подготовительных действий, мы можем начать возвращать результат практически моментально.

Обратная сторона состоит в том, что этот способ крайне неэффективен для больших объемов данных. Ситуация та же, что и с индексами: чем больше выборка, тем больше накладных расходов.

Таким образом, соединение вложенными циклами имеет смысл применять, если:

- один из наборов строк небольшой;
- к другому набору есть эффективный доступ по условию соединения;
- общее количество строк результата невелико.

Это обычная ситуация для OLTP-запросов (например, запросов от пользовательского интерфейса, где веб-страница или экранная форма должны открыться быстро и не выводят большой объем информации).

Еще одна особенность, которую стоит отметить: соединение вложенными циклами может работать для любого условия соединения. Подходит как эквисоединение (по условию равенства, как в примере), так и любое другое.

1. Создайте индекс на таблице рейсов (flights) по аэропортам отправления (departure_airport).
Найдите все рейсы из Ульяновска и проверьте план выполнения запроса.
2. Постройте таблицу расстояний между всеми аэропортами (так, чтобы каждая пара встречалась только один раз).
Какой способ соединения используется в таком запросе?

2. Используйте оператор <@> из расширения earthdistance.

1. Рейсы из Ульяновска

Индекс на таблице рейсов:

```
=> CREATE INDEX ON flights(departure_airport);
```

CREATE INDEX

План запроса:

```
=> EXPLAIN SELECT *
FROM flights f JOIN airports a ON a.airport_code = f.departure_airport
WHERE a.city = 'Ульяновск';
```

QUERY PLAN

```
-----
Nested Loop (cost=24.31..3732.03 rows=2066 width=162)
-> Seq Scan on airports_data ml (cost=0.00..30.56 rows=1 width=145)
    Filter: ((city ->> lang()) = 'Ульяновск'::text)
-> Bitmap Heap Scan on flights f (cost=24.31..2637.48 rows=2066 width=63)
    Recheck Cond: (departure_airport = ml.airport_code)
-> Bitmap Index Scan on flights_departure_airport_idx (cost=0.00..23.79
rows=2066 width=0)
    Index Cond: (departure_airport = ml.airport_code)
(7 rows)
```

Планировщик использовал соединение вложенным циклом.

Заметим, что в данном случае соединение необходимо, так как в Ульяновске два аэропорта:

```
=> SELECT airport_code, airport_name FROM airports WHERE city = 'Ульяновск';
```

airport_code	airport_name
ULY	Ульяновск-Восточный
ULV	Баратаевка

(2 rows)

2. Таблица расстояний между аэропортами

```
=> CREATE EXTENSION earthdistance CASCADE;
```

NOTICE: installing required extension "cube"

CREATE EXTENSION

Чтобы отсеять повторяющиеся пары, можно соединить таблицы по условию «больше»:

```
=> EXPLAIN SELECT a1.airport_code "from",
a2.airport_code "to",
a1.coordinates <@> a2.coordinates "distance, miles"
FROM airports a1 JOIN airports a2 ON a1.airport_code > a2.airport_code;
```

QUERY PLAN

```
-----
Nested Loop (cost=0.14..147.97 rows=3605 width=16)
-> Seq Scan on airports_data ml (cost=0.00..4.04 rows=104 width=20)
-> Index Scan using airports_data_pkey on airports_data ml_1 (cost=0.14..0.95
rows=35 width=20)
    Index Cond: (airport_code < ml.airport_code)
(4 rows)
```

Вложенный цикл — единственный способ соединения для такого условия.