

Оптимизация запросов Соединение хешированием



Авторские права

© Postgres Professional, 2019–2022

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

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.

Последовательное соединение хешированием:
одно- и двухпроходное

Группировка с помощью хеширования

Вычислительная сложность

Параллельное соединение хешированием:
одно- и двухпроходное

Применяется, когда для хеш-таблицы достаточно оперативной памяти

Построение хеш-таблицы

```
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

внутренний набор



	хеш-код	id	title
00	01000100	4	The Beatles
01	10100001	3	Let It Be
	10001001	6	Abbey Road
10			
11	11000111	1	Yellow Submarine

$work_mem \times hash_mem_multiplier$

4

участвует
в условии
соединения

используется
в запросе

номер
корзины

Первым этапом в памяти строится хеш-таблица.

Идея хеширования состоит в том, что функция хеширования *равномерно* распределяет значения по ограниченному числу корзин хеш-таблицы. В таком случае разные значения как правило будут попадать в разные корзины. Если равномерности не будет, в одну корзину может попасть много значений. В таком случае они выстраиваются в список, и по мере увеличения длины списка эффективность поиска по хеш-таблице будет падать.

Итак, строки первого набора читаются последовательно, и для каждой из них вычисляется хеш-функция от значения полей, входящих в условие соединения (в нашем примере — числовые идентификаторы).

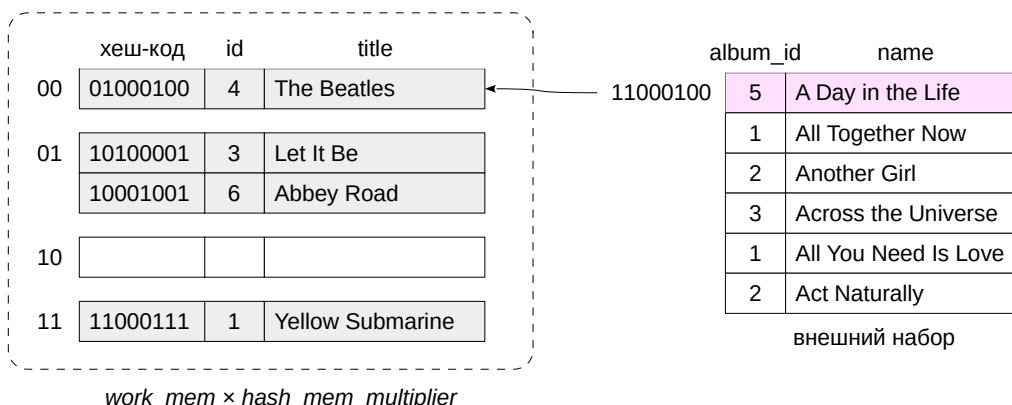
По значению хеш-функции определяется номер корзины. Например, если используется 4 корзины, то в качестве номера корзины можно взять два младших бита.

В корзину хеш-таблицы помещаются вычисленный хеш-код и все поля, которые входят в условие соединения или используются в запросе.

Размер хеш-таблицы в памяти ограничен значением $work_mem \times hash_mem_multiplier$. Наилучшая эффективность достигается, если вся хеш-таблица помещается в этот объем памяти целиком. (Это еще одна причина не использовать в запросе лишние поля, в том числе, «звездочку».)

Исходный код алгоритма можно найти в файле [src/backend/executor/nodeHashjoin.c](#).

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



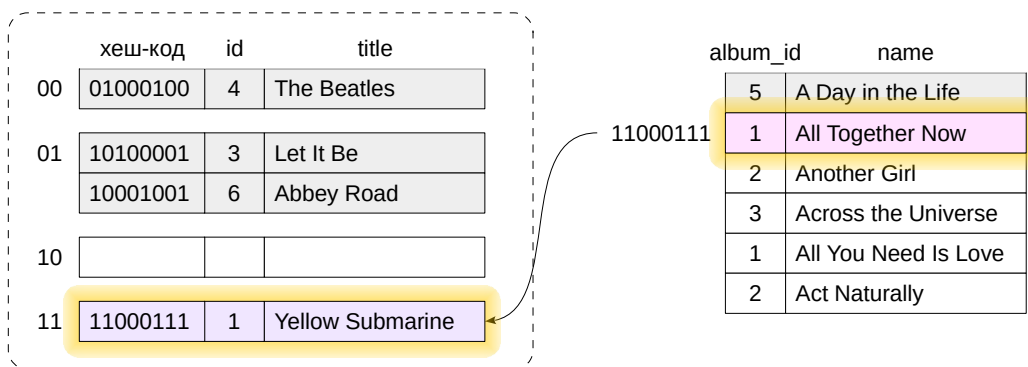
На втором этапе мы последовательно читаем второй набор строк. По мере чтения мы вычисляем хеш-функцию от значения полей, участвующих в условии соединения. Если в соответствующей корзине хеш-таблицы обнаруживается строка

- с таким же хеш-кодом,
 - и со значениями полей, подходящими под условие соединения,
- то мы нашли пару.

Проверки одного только хеш-кода недостаточно. Во-первых, не все условия соединения, перечисленные в запросе, могут быть учтены при выполнении соединения хешированием (поддерживаются только эквисоединения). Во-вторых, возможны коллизии, при которых разные значения получают одинаковые хеш-коды (вероятность этого мала, но тем не менее она есть).

В нашем примере для первой строки соответствия нет.

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



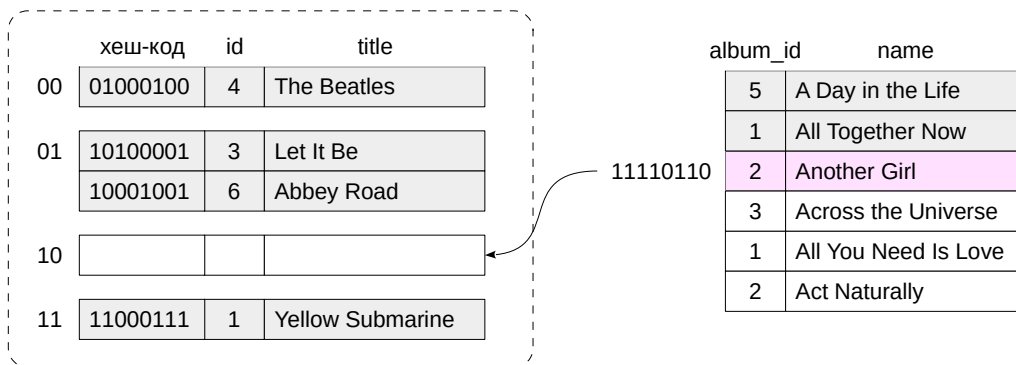
Вторая строка второго набора дает соответствие, которое уже можно вернуть вышестоящему узлу плана: («Yellow Submarine», «All Together Now»).

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

	хеш-код	id	title
00	01000100	4	The Beatles
01	10100001	3	Let It Be
	10001001	6	Abbey Road
10			
11	11000111	1	Yellow Submarine

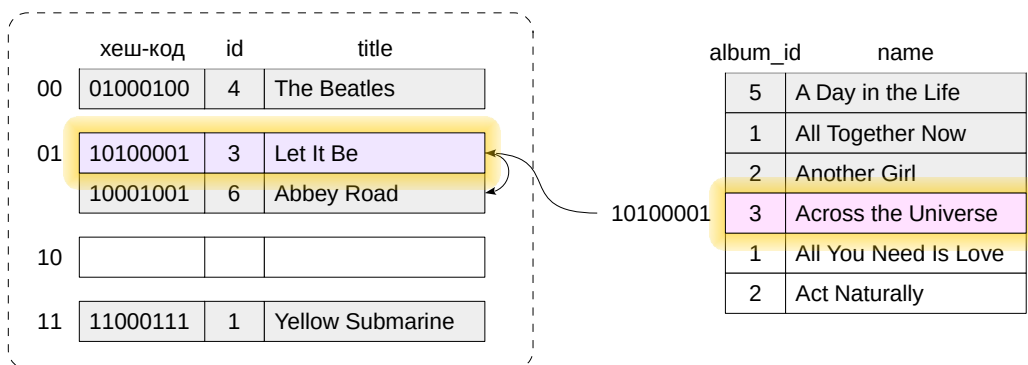
album_id	name
5	A Day in the Life
1	All Together Now
2	Another Girl
3	Across the Universe
1	All You Need Is Love
2	Act Naturally

11110110



Для третьей строки соответствия нет (соответствующая корзина хеш-таблицы пуста).

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



Для четвертой получаем соответствие («Let It Be», «Across the Universe»).

Заметим, что в корзине хеш-таблицы оказалось две строки первого набора, и в общем случае их придется просмотреть обе.


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

	хеш-код	id	title
00	01000100	4	The Beatles
01	10100001	3	Let It Be
	10001001	6	Abbey Road
10			
11	11000111	1	Yellow Submarine

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

11000111

Для пятой строки получаем соответствие («Yellow Submarine», «All You Need Is Love»).

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

	хеш-код	id	title
00	01000100	4	The Beatles
01	10100001	3	Let It Be
	10001001	6	Abbey Road
10			
11	11000111	1	Yellow Submarine

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

11110110

10

Для шестой строки соответствия нет. На этом работа соединения завершена.

Однопроходное соединение хешированием

Для большой выборки оптимизатор предпочитает соединение хешированием:

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

QUERY PLAN

```
-----
Hash Join
  Hash Cond: (tf.ticket_no = t.ticket_no)
    -> Seq Scan on ticket_flights tf
    -> Hash
        -> Seq Scan on tickets t
(5 rows)
```

Узел Hash Join начинает работу с того, что обращается к дочернему узлу Hash. Тот получает от своего дочернего узла (здесь — Seq Scan) весь набор строк и строит хеш-таблицу.

Затем Hash Join обращается ко второму дочернему узлу и соединяет строки, постепенно возвращая полученные результаты.

Модификации Hash Join включают уже рассмотренные ранее Left (Right), Semi и Anti, а также Full для полного соединения:

```
=> EXPLAIN SELECT *
FROM aircrafts a
FULL JOIN seats s ON a.aircraft_code = s.aircraft_code;
```

QUERY PLAN

```
-----
Hash Full Join (cost=3.47..30.04 rows=1339 width=55)
  Hash Cond: (s.aircraft_code = ml.aircraft_code)
    -> Seq Scan on seats s (cost=0.00..21.39 rows=1339 width=15)
    -> Hash (cost=3.36..3.36 rows=9 width=40)
        -> Seq Scan on aircrafts_data ml (cost=0.00..3.36 rows=9 width=40)
(5 rows)
```

Группировка и уникальные значения

Для группировки (GROUP BY) и устранения дубликатов (DISTINCT и операции со множествами без слова ALL) используются методы, схожие с методами соединения. Один из способов выполнения состоит в том, чтобы построить хеш-таблицу по нужным полям и получить из нее уникальные значения.

```
=> EXPLAIN SELECT fare_conditions, count(*)
FROM seats
GROUP BY fare_conditions;
```

QUERY PLAN

```
-----
HashAggregate (cost=28.09..28.12 rows=3 width=16)
  Group Key: fare_conditions
    -> Seq Scan on seats (cost=0.00..21.39 rows=1339 width=8)
(3 rows)
```

То же самое и с DISTINCT:

```
=> EXPLAIN SELECT DISTINCT fare_conditions
FROM seats;
```

QUERY PLAN

```
-----
HashAggregate (cost=24.74..24.77 rows=3 width=8)
  Group Key: fare_conditions
    -> Seq Scan on seats (cost=0.00..21.39 rows=1339 width=8)
(3 rows)
```

Использование памяти

Увеличим размер памяти, отведенной под хеш-таблицу:

```
=> SET work_mem = '64MB';
```

SET

```
=> SET hash_mem_multiplier = 3;
```

SET

Теперь размер ограничен значением:

```
=> SELECT wm.setting work_mem, wm.unit,
       hmm.setting hash_mem_multiplier,
       wm.setting::numeric * hmm.setting::numeric total
FROM pg_settings wm, pg_settings hmm
WHERE wm.name = 'work_mem' AND hmm.name = 'hash_mem_multiplier';
```

work_mem	unit	hash_mem_multiplier	total
65536	kB	3	196608

(1 row)

Команда EXPLAIN показывает нестандартные значения параметров при указании settings:

```
=> EXPLAIN (analyze, settings, costs off, timing off, summary off)
SELECT *
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
```

QUERY PLAN

```
Hash Join (actual rows=2949857 loops=1)
  Hash Cond: (t.book_ref = b.book_ref)
    -> Seq Scan on tickets t (actual rows=2949857 loops=1)
    -> Hash (actual rows=2111110 loops=1)
        Buckets: 4194304 Batches: 1 Memory Usage: 145986kB
        -> Seq Scan on bookings b (actual rows=2111110 loops=1)
  Settings: hash_mem_multiplier = '3', jit = 'off', search_path = 'bookings, public', work_mem = '64MB'
(7 rows)
```

Хеш-таблица поместилась в память (Batches: 1). Параметр Buckets показывает число корзин в хеш-таблице, а Memory Usage — использованную оперативную память.

Обратите внимание, что хеш-таблица строилась по меньшему набору строк.

Сравним с таким же запросом, который выводит только одно поле:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT b.book_ref
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
```

QUERY PLAN

```
Hash Join (actual rows=2949857 loops=1)
  Hash Cond: (t.book_ref = b.book_ref)
    -> Seq Scan on tickets t (actual rows=2949857 loops=1)
    -> Hash (actual rows=2111110 loops=1)
        Buckets: 4194304 Batches: 1 Memory Usage: 113172kB
        -> Seq Scan on bookings b (actual rows=2111110 loops=1)
(6 rows)
```

Расход памяти уменьшился, так как в хеш-таблице теперь только одно поле (вместо трех).

Обратите внимание на строку Hash Cond: она содержит предикаты, участвующие в соединении. Условие может включать и такие предикаты, которые не могут использоваться механизмом соединения, но должны учитываться. Они отображаются в отдельной строке Join Filter, и нужные для их вычисления поля тоже попадают в хеш-таблицу (сравните объем памяти):

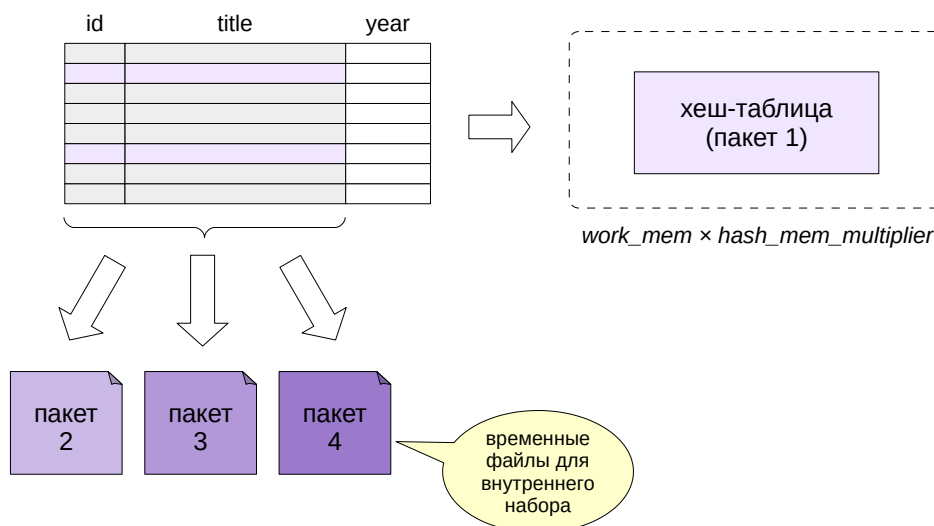
```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT b.book_ref
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref
               AND b.total_amount::text > t.passenger_id;
```

QUERY PLAN

```
-----  
Hash Join (actual rows=1198320 loops=1)  
  Hash Cond: (t.book_ref = b.book_ref)  
  Join Filter: ((b.total_amount)::text > (t.passenger_id)::text)  
  Rows Removed by Join Filter: 1751537  
  -> Seq Scan on tickets t (actual rows=2949857 loops=1)  
  -> Hash (actual rows=2111110 loops=1)  
        Buckets: 4194304  Batches: 1  Memory Usage: 127431kB  
        -> Seq Scan on bookings b (actual rows=2111110 loops=1)  
(8 rows)
```

Применяется, когда хеш-таблица не помещается в оперативную память: наборы данных разбиваются на пакеты и последовательно соединяются

Построение хеш-таблицы



13

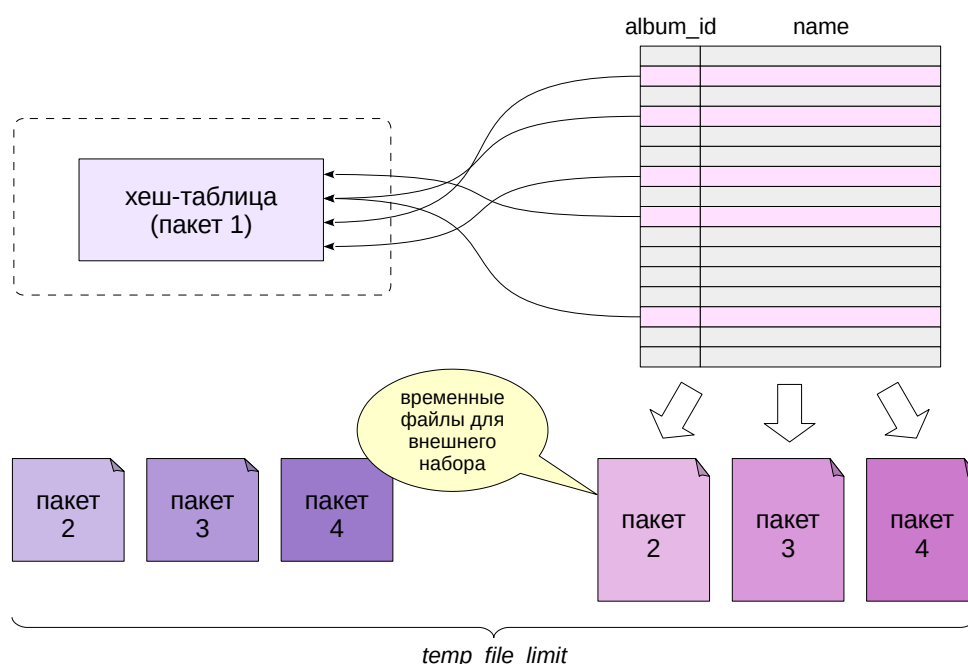
Если хеш-таблица не помещается в объем памяти, ограниченный $work_mem \times hash_mem_multiplier$, первый (внутренний) набор строк разбивается на отдельные пакеты. Для распределения по пакетам используется некоторое количество битов хеш-кода, поэтому число пакетов всегда кратно двум. В идеале в каждый пакет попадает примерно одинаковое количество строк, но, если значения в строках повторяются, возможен перекос.

При планировании запроса заранее вычисляется минимально необходимое число пакетов так, чтобы хеш-таблица для каждого пакета помещалась в памяти. Это число не уменьшается, даже если оптимизатор ошибся с оценками, но при необходимости может динамически увеличиваться.

Хеш-таблица для первого пакета остается в памяти, а строки, принадлежащие другим пакетам, сбрасываются на диск во временные файлы — каждый пакет в свой файл.

На рисунке показано четыре пакета.

Сопоставление – пакет 1



14

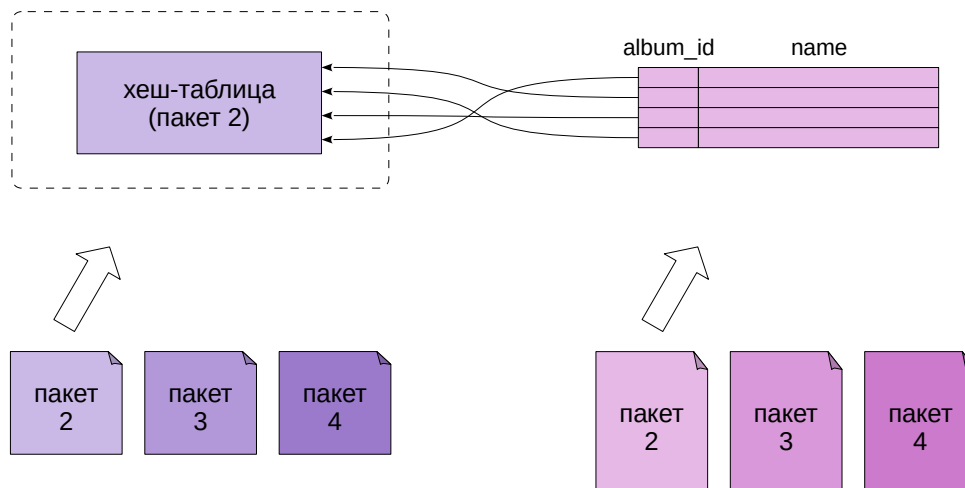
Далее читается второй (внешний) набор строк. Если строка принадлежит первому пакету, она сопоставляется с хеш-таблицей, которая как раз содержит первый пакет. С другими пакетами строку сопоставлять не надо — в них не может найтись соответствие, поскольку хеш-коды заведомо будут отличаться.

Если строка принадлежит другому пакету, она сбрасывается на диск — опять же, каждый пакет в свой временный файл.

Таким образом, при N пакетах используются $2(N-1)$ файлов.

Следует учитывать, что использование временных файлов на диске ограничивается параметром *temp_file_limit*, который определяет общий предел дисковой памяти для сеанса. (Буферы временных таблиц в это ограничение не входят.)

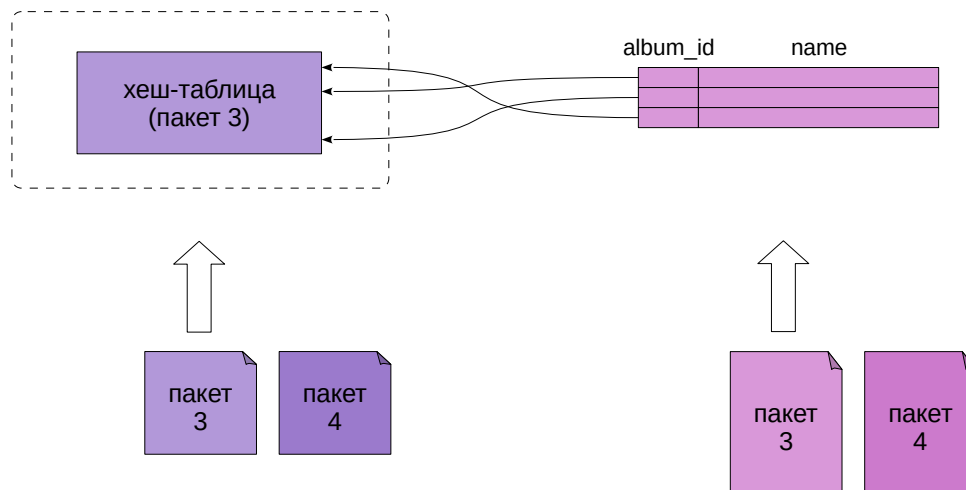
Сопоставление – пакет 2



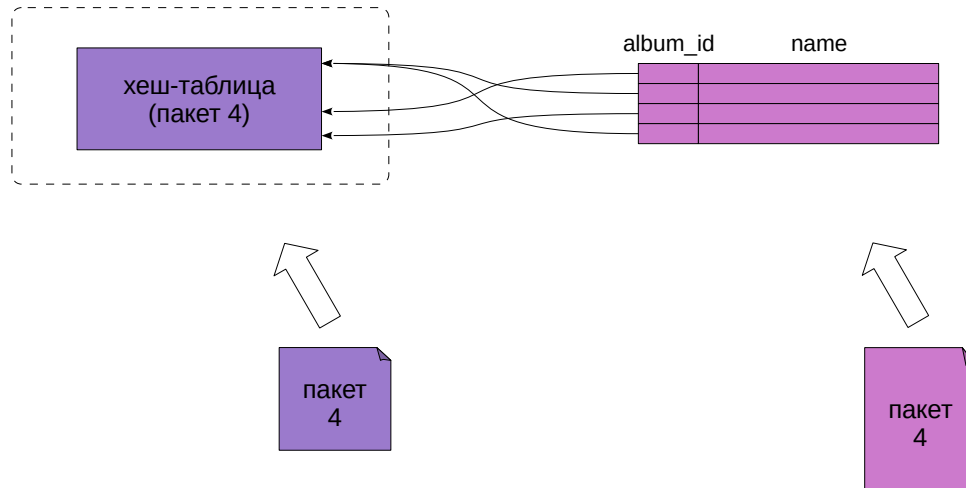
Далее по очереди обрабатываются все пакеты, начиная со второго. Из временного файла в хеш-таблицу считываются строки внутреннего набора, затем из другого временного файла считываются строки внешнего набора и сопоставляются с хеш-таблицей.

Процедура повторяется для всех оставшихся $N-1$ пакетов. На рисунке показано соединение для второго пакета.

Сопоставление – пакет 3



На рисунке показано соединение для третьего пакета.



После обработки последнего пакета соединение завершено и временные файлы освобождены.

Таким образом, при нехватке оперативной памяти алгоритм соединения становится двухпроходным: каждый пакет (кроме первого) требуется записать на диск и затем прочитать повторно. Разумеется, это сказывается на эффективности соединения. Поэтому важно, чтобы:

- в хеш-таблицу попадали только действительно нужные поля (обязанность автора запроса),
- хеш-таблица строилась по меньшему набору строк (обязанность планировщика).

Двухпроходное соединение хешированием

Теперь уменьшим ограничение памяти так, чтобы хеш-таблица не поместилась, и выведем статистику использования буферного кеша:

```
=> SET work_mem = '32MB';
```

```
SET
```

```
=> SET hash_mem_multiplier = 1;
```

```
SET
```

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
```

```
SELECT b.book_ref
```

```
FROM bookings b
```

```
JOIN tickets t ON b.book_ref = t.book_ref;
```

QUERY PLAN

```
-----
Hash Join (actual rows=2949857 loops=1)
  Hash Cond: (t.book_ref = b.book_ref)
  Buffers: shared hit=192 read=62670, temp read=12515 written=12515
  -> Seq Scan on tickets t (actual rows=2949857 loops=1)
        Buffers: shared hit=96 read=49319
  -> Hash (actual rows=2111110 loops=1)
        Buckets: 1048576 Batches: 4 Memory Usage: 28291kB
        Buffers: shared hit=96 read=13351, temp written=5217
        -> Seq Scan on bookings b (actual rows=2111110 loops=1)
              Buffers: shared hit=96 read=13351
Planning:
  Buffers: shared hit=8
(12 rows)
```

Теперь потребовалось четыре пакета (Batches: 4).

Видно, что узел Hash записывает пакеты во временные файлы (temp written), а узел Hash Join и записывает, и читает (temp read и written).

$\sim N + M$, где

N и M — число строк в первом и втором наборах данных

Начальные затраты на построение хеш-таблицы

Эффективно для большого числа строк

Общая сложность соединения хешированием пропорциональна сумме числа строк в одном и другом наборах данных. Поэтому метод соединения хешированием гораздо эффективнее вложенного цикла при большом числе строк.

Однако, чтобы начать соединение, требуется заплатить накладные расходы на построение хеш-таблицы: из-за этого при небольшом числе строк вложенный цикл более эффективен.

Соединение хешированием (в сочетании с полным сканированием таблиц) характерно для OLAP-запросов, в которых надо обработать большое число строк, причем общая пропускная способность важнее времени отклика.

Стоимость хеш-соединения

```
=> EXPLAIN SELECT *  
FROM tickets t JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no;
```

QUERY PLAN

```
-----  
Hash Join  (cost=161874.35..498578.52 rows=8391852 width=136)  
  Hash Cond: (tf.ticket_no = t.ticket_no)  
    -> Seq Scan on ticket_flights tf  (cost=0.00..153851.52 rows=8391852 width=32)  
    -> Hash  (cost=78912.49..78912.49 rows=2949749 width=104)  
      -> Seq Scan on tickets t  (cost=0.00..78912.49 rows=2949749 width=104)  
(5 rows)
```

Начальная стоимость узла Hash Join складывается из стоимостей:

- получения всего первого набора данных (здесь — билеты);
- построения хеш-таблицы — пока таблица не готова, соединение не может начаться.

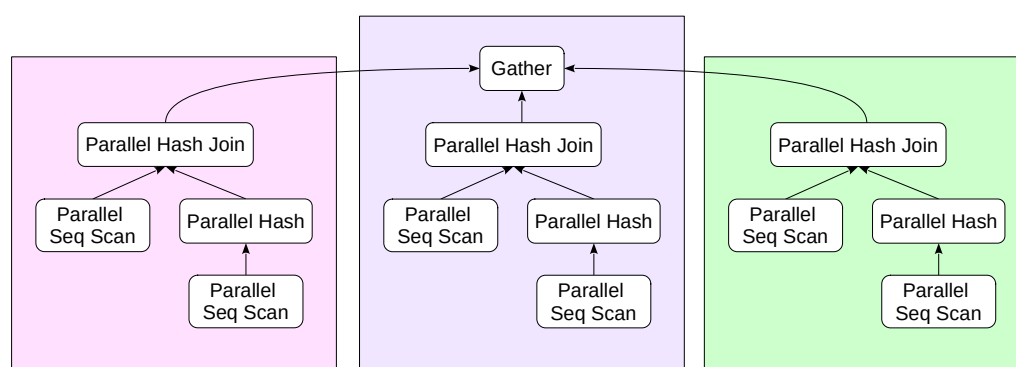
Можно обратить внимание на то, что в узле Hash стоимость построения хеш-таблицы не отражена.

Полная стоимость складывается из стоимостей:

- получения всего второго набора данных (здесь — перелеты);
- проверки по хеш-таблице;
- обращения к диску в случае, когда предполагается использование более одного пакета.

Главный вывод: стоимость хеш-соединения пропорциональна $N + M$, где N и M — число строк в соединяемых наборах данных. При больших N и M это значительно выгоднее, чем произведение в случае соединения внешним циклом.

Процессы используют общую хеш-таблицу



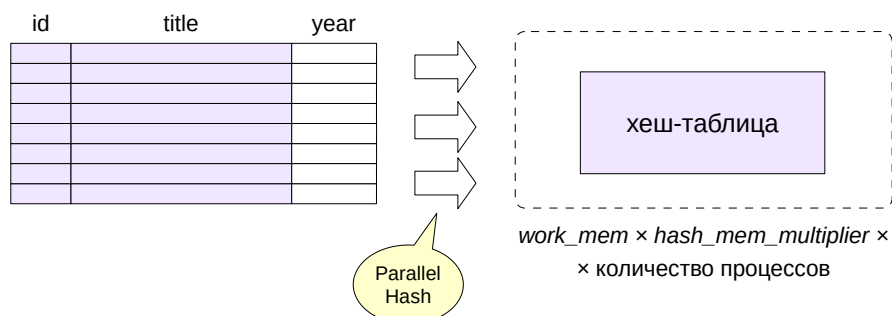
В отличие от других способов соединения, хеш-соединение не только может участвовать в параллельных планах, но и имеет отдельный эффективный алгоритм работы. Этот алгоритм позволяет параллельно выполнять оба этапа соединения: и построение хеш-таблицы по первому (внутреннему) набору строк, и сопоставление с ней строк второго (внешнего) набора.

Возможность параллельного хеш-соединения управляется параметром *enable_parallel_hash*; по умолчанию параметр включен.

Как и последовательного алгоритма, у параллельного есть два варианта: однопроходный при достаточном количестве оперативной памяти и двухпроходный.

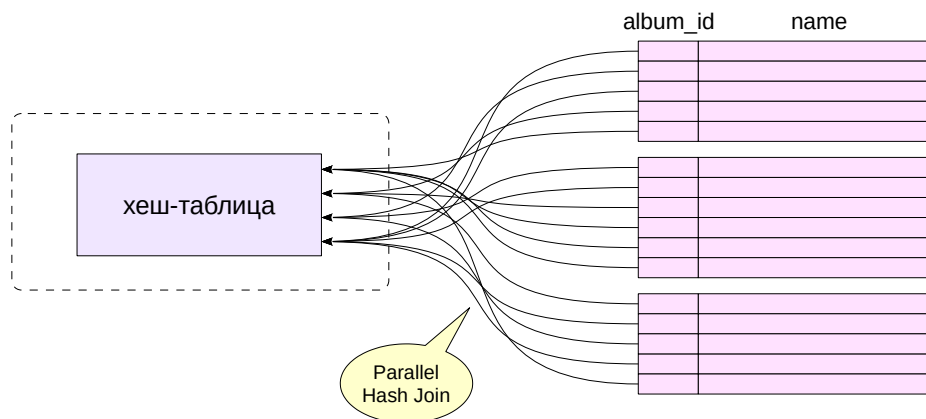
Начнем с однопроходного варианта.

Построение хеш-таблицы



Однопроходный алгоритм используется, если хеш-таблица помещается в *суммарный* объем памяти, выделенный всем участвующим в соединении процессам, то есть размер хеш-таблицы ограничен значением $work_mem \times hash_mem_multiplier \times \text{количество процессов}$.

Процессы параллельно читают первый набор строк (например, используя узел **Parallel Seq Scan**) и строят общую хеш-таблицу в разделяемой памяти, где каждый из них имеет к ней доступ.



После того, как хеш-таблица полностью построена, рабочие процессы приступают к параллельному чтению второго набора и сопоставляют прочитанные ими строки с общей-хеш-таблицей. Таким образом, каждый из процессов проверяет по хеш-таблице только часть данных.

Однопроходное параллельное хеш-соединение

```
=> SET work_mem = '64MB';
```

SET

Выполним запрос с агрегацией:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*)
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
```

QUERY PLAN

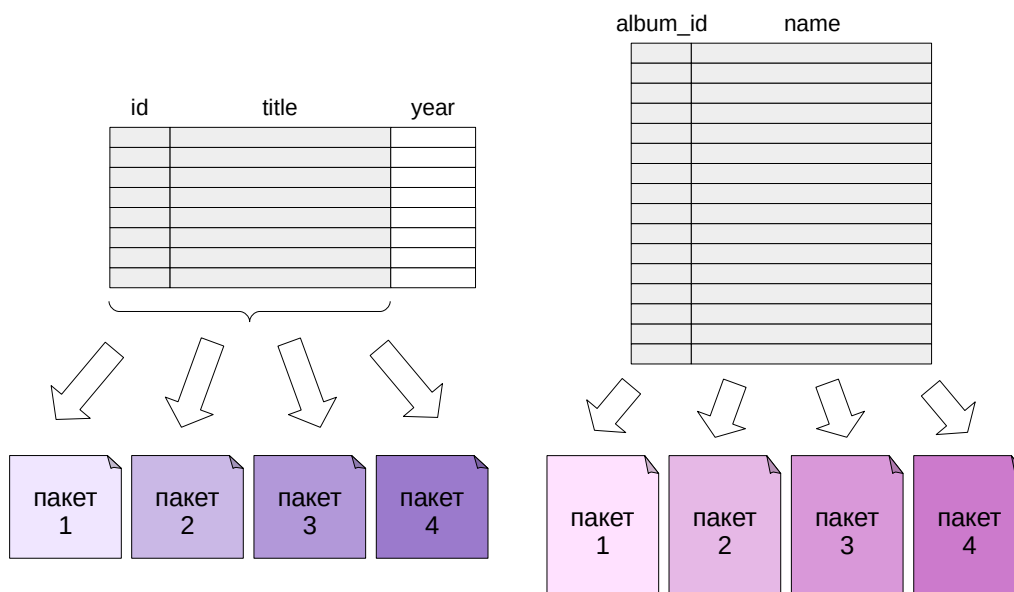
```
-----
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (actual rows=1 loops=3)
              -> Parallel Hash Join (actual rows=983286 loops=3)
                    Hash Cond: (t.book_ref = b.book_ref)
                    -> Parallel Seq Scan on tickets t (actual rows=983286 loops=3)
                    -> Parallel Hash (actual rows=703703 loops=3)
                          Buckets: 4194304 Batches: 1 Memory Usage: 115392kB
                          -> Parallel Seq Scan on bookings b (actual rows=703703 loops=3)

(11 rows)
```

Обратите внимание на использование памяти (Memory Usage): объем превышает ограничение, установленное для одного рабочего процесса, но в общую память трех процессов хеш-таблица помещается. Поэтому выполняется однопроходное соединение (Batches: 1).

Наборы строк разбиваются на пакеты, которые затем параллельно обрабатываются рабочими процессами

Разбиение на пакеты



27

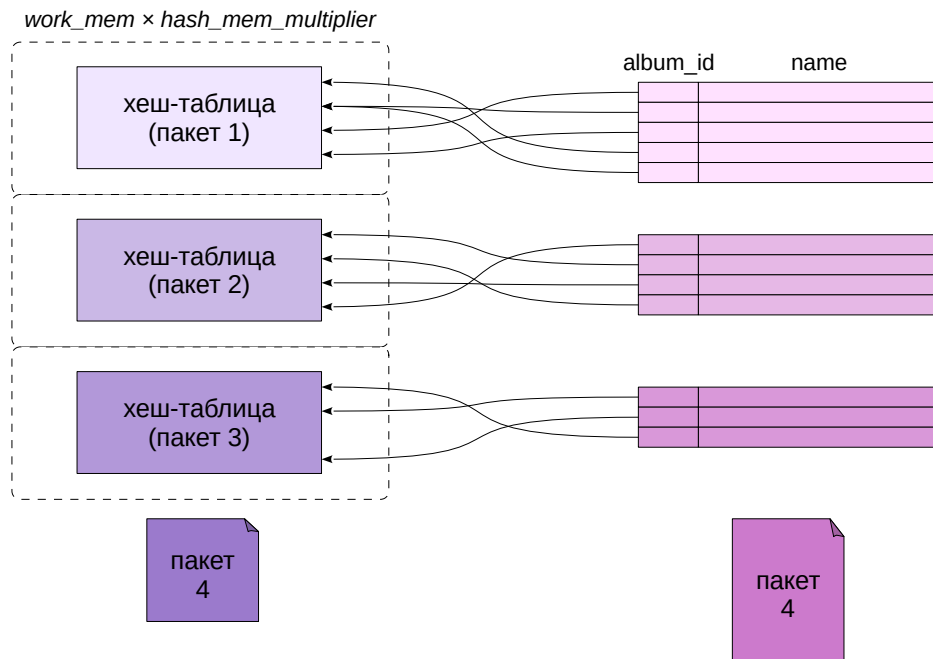
Хеш-таблица может не поместиться в объем памяти, ограниченный $work_mem \times hash_mem_multiplier \times \text{количество процессов}$, причем это может выясниться и на этапе выполнения соединения. В этом случае используется двухпроходный алгоритм, который существенно отличается и от двухпроходного последовательного, и от однопроходного параллельного.

Сначала рабочие процессы параллельно читают первый набор данных, разбивают его на пакеты и записывают пакеты во временные файлы. Первый пакет тоже попадает в файл; хеш-таблица в памяти не строится.

Обратите внимание, что каждый процесс записывает строки в каждый временный файл; запись синхронизируется.

Затем рабочие процессы параллельно читают второй набор данных и также разбивают его на пакеты и записывают во временные файлы.

Таким образом, при N пакетах на диск записываются $2N$ файлов.

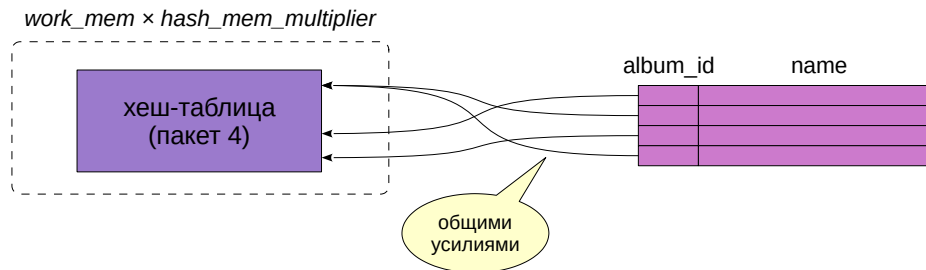


Затем каждый рабочий процесс выбирает себе по одному пакету.

Процесс загружает первый набор выбранного пакета в хеш-таблицу в памяти. В этом алгоритме у каждого процесса своя хеш-таблица размером $work_mem \times hash_mem_multiplier$, но располагаются они в общей памяти, то есть доступ к каждой таблице есть у всех рабочих процессов.

После заполнения хеш-таблицы процесс читает второй набор выбранного пакета и сопоставляет строки.

Когда процесс завершает обработку одного пакета, он выбирает следующий, еще не обработанный.



Когда необработанные пакеты заканчиваются, освободившийся процесс подключается к обработке одного из еще не завершенных пакетов, пользуясь тем, что все хеш-таблицы находятся в разделяемой памяти.

Несколько хеш-таблиц работают лучше, чем одна большая: в этом случае проще организовать совместную работу и меньше ресурсов тратится на синхронизацию.

Двухпроходное параллельное хеш-соединение

Еще уменьшим объем памяти, и соединение станет двухпроходным с четырьмя пакетами:

```
=> SET work_mem = '32MB';
```

SET

Выполним запрос с агрегацией:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*)
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
```

QUERY PLAN

```
-----
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (actual rows=1 loops=3)
              -> Parallel Hash Join (actual rows=983286 loops=3)
                    Hash Cond: (t.book_ref = b.book_ref)
                    -> Parallel Seq Scan on tickets t (actual rows=983286 loops=3)
                    -> Parallel Hash (actual rows=703703 loops=3)
                          Buckets: 1048576 Batches: 4 Memory Usage: 28896kB
                    -> Parallel Seq Scan on bookings b (actual rows=703703 loops=3)

(11 rows)
```


Соединение хешированием требует подготовки

надо построить хеш-таблицу

Эффективно для больших выборок

в том числе есть возможность параллельного соединения

Зависит от порядка соединения

внутренний набор должен быть меньше внешнего,
чтобы минимизировать хеш-таблицу

Поддерживает только эквисоединения

для хеш-кодов операторы «больше» и «меньше» не имеют смысла

В отличие от соединения вложенным циклом, хеш-соединение требует подготовки: построения хеш-таблицы. Пока таблица не построена, ни одна результирующая строка не может быть получена.

Зато соединение хешированием эффективно работает на больших объемах данных. Оба набора строк читаются последовательно и только один раз (два раза в случае нехватки оперативной памяти).

Ограничением соединения хеширования является поддержка только эквисоединений. Дело в том, что хеш-значения можно сравнивать только на равенство, операции «больше» и «меньше» просто не имеют смысла.

1. Напишите запрос, показывающий занятые места в салоне для всех рейсов.
Какой способ соединения выбрал планировщик? Проверьте, хватило ли оперативной памяти для размещения хеш-таблиц.
2. Измените запрос, чтобы он выводил только общее количество занятых мест.
Как изменился план запроса? Почему планировщик не использовал аналогичный план для предыдущего запроса?
3. Напишите запрос, показывающий имена пассажиров и номера рейсов, которыми они следуют.
Разберитесь по плану запроса, в какой последовательности выполняются операции.

1. Для этого достаточно соединить рейсы (flights) с посадочными талонами (boarding_passes).

3. Такой запрос должен соединять три таблицы: билеты (tickets), перелеты (ticket_flights) и рейсы (flights).

1. Список занятых мест

```
=> EXPLAIN SELECT f.flight_id, bp.seat_no
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id;
```

QUERY PLAN

```
-----
Hash Join (cost=8298.51..229398.38 rows=7925581 width=7)
  Hash Cond: (bp.flight_id = f.flight_id)
    -> Seq Scan on boarding_passes bp (cost=0.00..137534.81 rows=7925581 width=7)
    -> Hash (cost=4772.67..4772.67 rows=214867 width=4)
        -> Seq Scan on flights f (cost=0.00..4772.67 rows=214867 width=4)
(5 rows)
```

Использовано соединение хешированием.

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT f.flight_id, bp.seat_no
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id;
```

QUERY PLAN

```
-----
Hash Join (actual rows=7925812 loops=1)
  Hash Cond: (bp.flight_id = f.flight_id)
    -> Seq Scan on boarding_passes bp (actual rows=7925812 loops=1)
    -> Hash (actual rows=214867 loops=1)
        Buckets: 131072 Batches: 4 Memory Usage: 2917kB
        -> Seq Scan on flights f (actual rows=214867 loops=1)
(6 rows)
```

Хеш-таблица не поместилась целиком в память, потребовалось четыре пакета.

2. Количество занятых мест

```
=> EXPLAIN SELECT count(*)
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id;
```

QUERY PLAN

```
-----
Finalize Aggregate (cost=114694.88..114694.89 rows=1 width=8)
  -> Gather (cost=114694.66..114694.87 rows=2 width=8)
      Workers Planned: 2
      -> Partial Aggregate (cost=113694.66..113694.67 rows=1 width=8)
          -> Parallel Hash Join (cost=5467.82..105438.85 rows=3302325 width=0)
              Hash Cond: (bp.flight_id = f.flight_id)
              -> Parallel Seq Scan on boarding_passes bp (cost=0.00..91302.25 rows=3302325 width=4)
              -> Parallel Hash (cost=3887.92..3887.92 rows=126392 width=4)
                  -> Parallel Seq Scan on flights f (cost=0.00..3887.92 rows=126392 width=4)
(9 rows)
```

Здесь планировщик использовал параллельный план. В предыдущем запросе это не было оправдано из-за высокой стоимости пересылки данных между процессами, а в данном случае передается только одно число.

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*)
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id;
```

QUERY PLAN

```
-----
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
  -> Partial Aggregate (actual rows=1 loops=3)
        -> Parallel Hash Join (actual rows=2641937 loops=3)
              Hash Cond: (bp.flight_id = f.flight_id)
              -> Parallel Seq Scan on boarding_passes bp (actual rows=2641937 loops=3)
              -> Parallel Hash (actual rows=71622 loops=3)
                    Buckets: 262144  Batches: 1  Memory Usage: 10528kB
              -> Parallel Seq Scan on flights f (actual rows=71622 loops=3)

(11 rows)
```

Обратите внимание на поле loops в узлах выше и ниже Gather — оно соответствует реальному числу процессов, работавших над запросом.

3. Пассажиры и номера рейсов

```
=> EXPLAIN (costs off)
SELECT t.passenger_name, f.flight_no
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
  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

(9 rows)
```

Сначала выполняется соединение билетов (tickets) с перелетами (ticket_flights), причем хеш-таблица строится по таблице билетов.

Затем рейсы (flights) соединяются с результатом первого соединения; хеш-таблица строится по таблице рейсов.