

Сортировка и группировка

Группировка



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.

Применение группировки
Группировка хешированием
Группировка сортировкой
Комбинированная группировка
Группировка в параллельных планах
Оконные функции

Явное указание

GROUP BY

Исключение дубликатов

DISTINCT

UNION, INTERSECT, EXCEPT

В SQL-запросе набор строк можно сгруппировать с помощью предложения GROUP BY, в сочетании с которым обычно используются агрегатные функции, обрабатывающие все строки группы. Похожую возможность для оконных функций дает предложение PARTITION BY.

Группировка может выполняться и неявно при устранении дубликатов в запросе со словом DISTINCT или с командами обработки двух наборов строк (UNION, INTERSECT, EXCEPT без слова ALL), которые удаляют дубликаты.

Сервер выбирает один из доступных способов группировки, учитывая имеющиеся ресурсы, возможность получить отсортированные данные для ключа группировки и другие факторы.

Применение группировки

Выполним запрос, явно группирующий строки таблицы seats по классам обслуживания:

```
=> EXPLAIN (costs off)
SELECT fare_conditions
FROM seats
GROUP BY fare_conditions;
```

```
          QUERY PLAN
-----
HashAggregate
  Group Key: fare_conditions
    -> Seq Scan on seats
(3 rows)
```

Теперь получим все различные классы обслуживания:

```
=> EXPLAIN (costs off)
SELECT DISTINCT fare_conditions
FROM seats;
```

```
          QUERY PLAN
-----
HashAggregate
  Group Key: fare_conditions
    -> Seq Scan on seats
(3 rows)
```

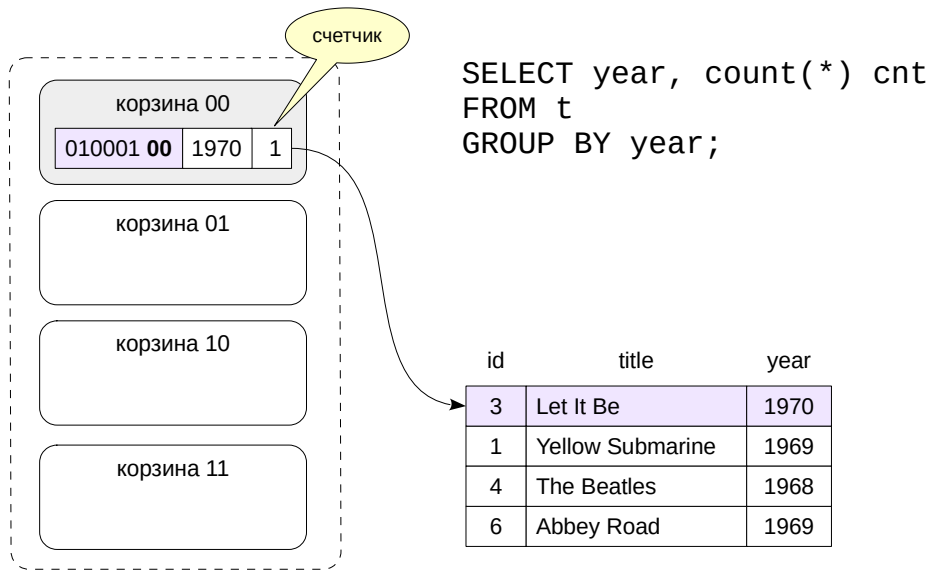
В третьем (исключительно учебном) примере дважды получим все строки таблицы seats и объединим их с помощью команды UNION:

```
=> EXPLAIN (costs off)
SELECT fare_conditions
FROM seats
UNION
SELECT fare_conditions
FROM seats;
```

```
          QUERY PLAN
-----
HashAggregate
  Group Key: seats.fare_conditions
    -> Append
      -> Seq Scan on seats
      -> Seq Scan on seats seats_1
(5 rows)
```

Результаты всех трех запросов одинаковы, и во всех планах есть узел HashAggregate.

Группировка хешированием



5

Про идею хеширования было рассказано в теме «Типы индексов». При группировке значений может использоваться похожий принцип.

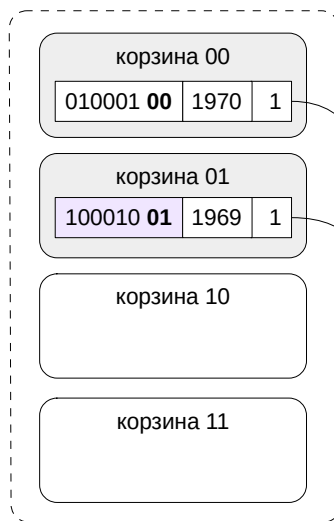
В примере на слайде показано, как вычисляется значение агрегатной функции count при группировке по полю year. Сервер получает и по очереди обрабатывает строки таблицы.

Предварительно рассчитывается количество корзин — степень числа 2, и определяется число разрядов хеш-кода, содержащих номер корзины. В примере на слайде это два последних разряда.

Для строки вычисляется хеш-функция от значения year. По хеш-коду определяется номер корзины и в хеш-таблицу добавляется запись с хеш-кодом, если его еще нет в таблице.

Поскольку в нашем примере вычисляется значение агрегатной функции count, в каждой записи хеш-таблицы дополнительно хранится счетчик, который увеличивается при обработке очередной строки.

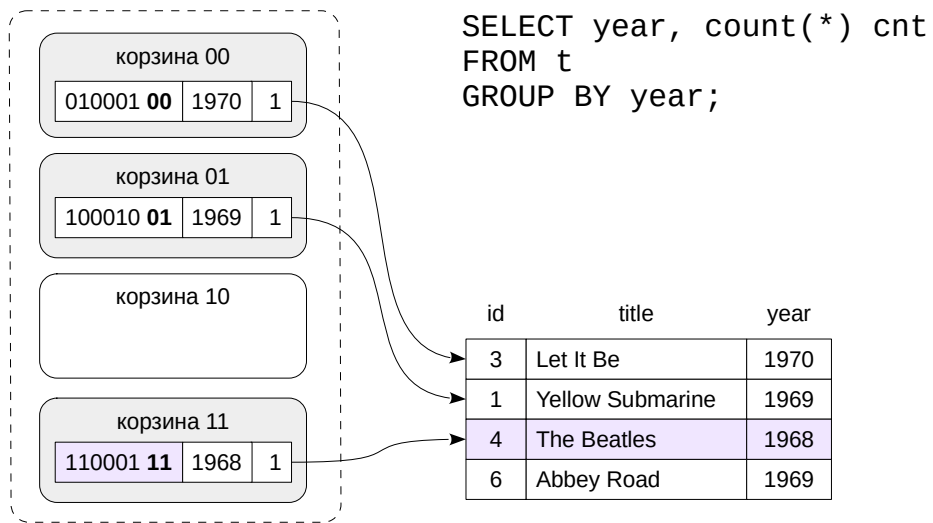
В нашем примере первая строка порождает новую запись в корзине 00.



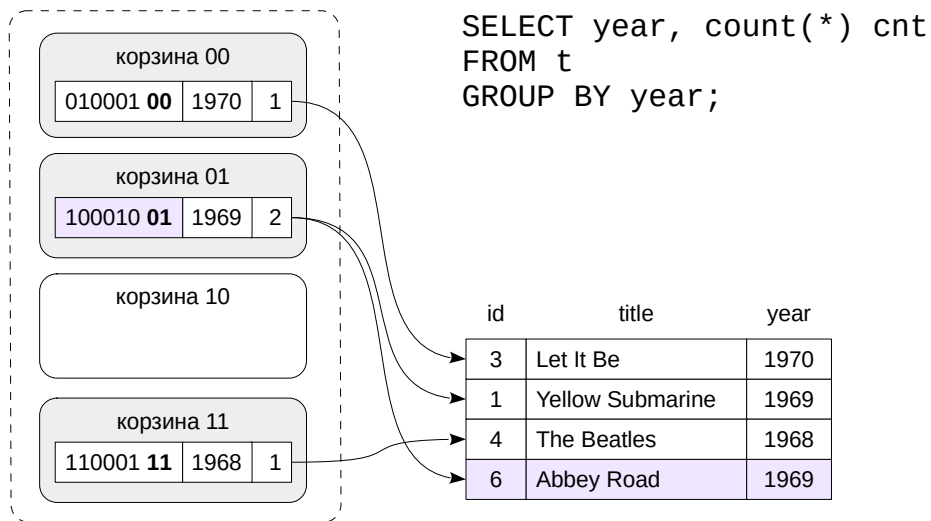
```
SELECT year, count(*) cnt  
FROM t  
GROUP BY year;
```

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

Вторая строка попадает в корзину 01 и порождает в ней новую запись.



Третья строка порождает новую запись в корзине 11.



Хеш-код последней строки попадает в корзину с номером 01. В корзине уже есть запись с таким значением ключа, поэтому сервер увеличивает значение ее счетчика на единицу. На этом обработка таблицы завершается: в хеш-таблице собрались все уникальные значения. Алгоритм возвращает сами значения и их количество.

Такой же алгоритм (с незначительными модификациями) используется и для получения уникальных записей без вычисления агрегатной функции, и при вычислении других агрегатных функций.

Группировка хешированием

Узел, который отвечает за агрегацию методом хеширования, обозначается в плане выполнения как HashAggregate:

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

-----
QUERY PLAN
-----
HashAggregate (cost=28.09..28.18 rows=9 width=12)
  Group Key: aircraft_code
    -> Seq Scan on seats (cost=0.00..21.39 rows=1339 width=4)
(3 rows)
```

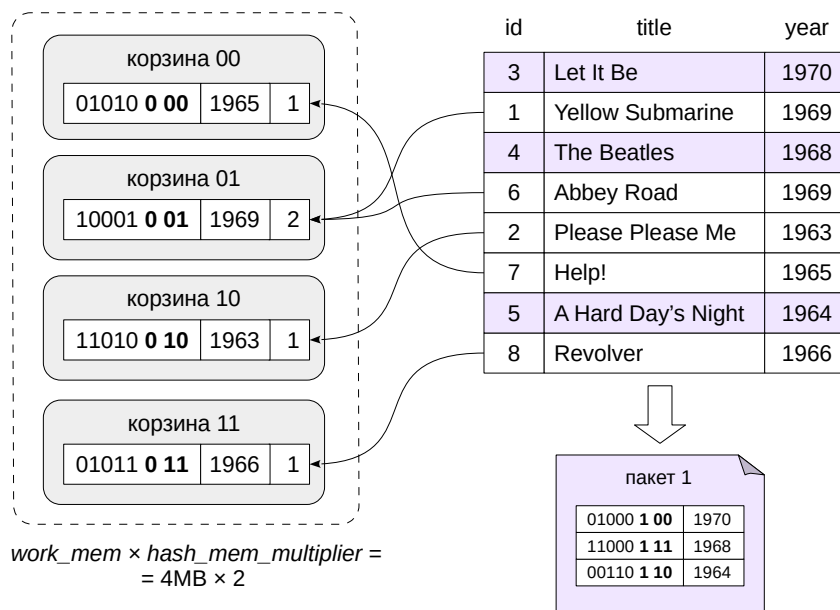
Обратите внимание на начальную стоимость: результаты начинают выдаваться только после построения хеш-таблицы по всем строкам.

Выполнив запрос, можно увидеть объем памяти, выделенной под хеш-таблицу:

```
=> EXPLAIN (analyze, timing off, costs off)
SELECT aircraft_code, count(*)
FROM seats
GROUP BY aircraft_code;

-----
QUERY PLAN
-----
HashAggregate (actual rows=9 loops=1)
  Group Key: aircraft_code
    Batches: 1 Memory Usage: 24kB
    -> Seq Scan on seats (actual rows=1339 loops=1)
Planning Time: 0.066 ms
Execution Time: 0.338 ms
(6 rows)
```

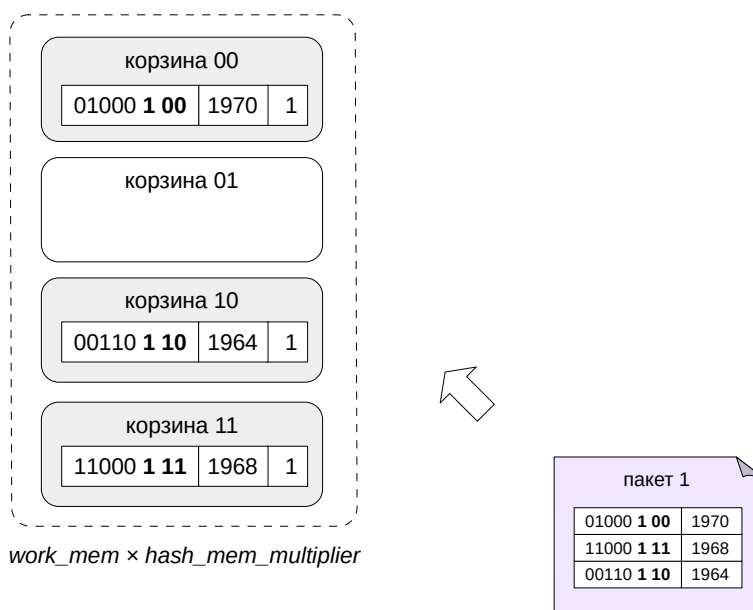
Хеш-группировка и пакеты



10

Если выделенной памяти не хватает, исходный набор разбивается на части (пакеты), так чтобы хеш-таблица для каждого пакета умещалась в оперативной памяти. Расчетное количество пакетов — степень числа 2; в хеш-коде выделяется нужное количество битов, которые определяют принадлежность записи пакету (на слайде два пакета, поэтому достаточно одного бита). Хеш-таблица для пакета 0 располагается в оперативной памяти, строки остальных пакетов помещаются во временные файлы — каждый пакет в свой файл.

Использование временных файлов значительно снижает производительность, и при хешировании этот эффект проявляется сильнее, чем для других алгоритмов. Поэтому при группировке хешированием выделяется $work_mem \times hash_mem_multiplier$ оперативной памяти — по умолчанию вдвое больше, чем обычно.



Затем каждый из пакетов, записанных во временные файлы, читается в память и формирует новую хеш-таблицу.

После обработки всех исходных строк, а также каждого из дополнительных пакетов сервер может возвращать частичные результаты (строки для каждой группы всегда находятся в одном пакете).

Хеш-группировка и пакеты

Рассмотрим план другого запроса, с группировкой по большой таблице:

```
=> EXPLAIN
SELECT book_ref, count(*)
FROM tickets
GROUP BY book_ref;
```

QUERY PLAN

```
-----
HashAggregate (cost=244842.76..283065.11 rows=1517662 width=15)
  Group Key: book_ref
  Planned Partitions: 32
  -> Seq Scan on tickets (cost=0.00..78913.53 rows=2949853 width=7)
(4 rows)
```

В плане появилось предложение Planned Partitions — оптимизатор предполагает, что хеш-таблица не поместится в выделенную память и придется разбивать ее на пакеты.

Выполним запрос:

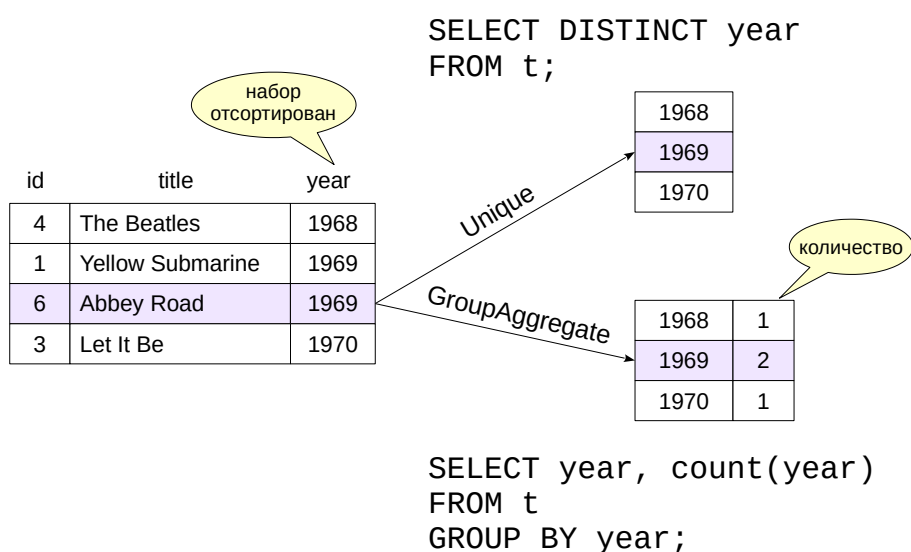
```
=> EXPLAIN (analyze, timing off, costs off)
SELECT book_ref, count(*)
FROM tickets
GROUP BY book_ref;
```

QUERY PLAN

```
-----
HashAggregate (actual rows=2111110 loops=1)
  Group Key: book_ref
  Batches: 33  Memory Usage: 8209kB  Disk Usage: 95584kB
  -> Seq Scan on tickets (actual rows=2949857 loops=1)
Planning Time: 0.073 ms
Execution Time: 7368.814 ms
(6 rows)
```

При выполнении размер одного из пакетов оказался слишком велик, поэтому он был дополнительно разбит на два более мелких пакета. Общее количество пакетов (Batches) увеличилось по сравнению с расчетным.

Группировка сортировкой



13

Исключение дубликатов и группировка могут выполняться не только путем хеширования, но и с помощью сортировки. В этом случае требуется набор данных, предварительно отсортированный по полям группировки.

Узел Unique устраняет дубликаты, добавляя к результату очередное значение, если оно не совпадает с предыдущим.

Узел GroupAggregate возвращает агрегированные данные. В примере на слайде узел строит список различных значений поля year, подсчитывая, сколько раз встретилось каждое из них в наборе данных.

Как было рассмотрено в предыдущей теме, упорядоченный набор строк может быть получен с помощью индексного доступа или в результате сортировки в узле Sort. В первом случае узлы Unique и GroupAggregate обычно более выгодны, чем HashAggregate, поскольку выполняются быстро и не требуют много памяти. Однако во втором случае построение хеш-таблицы, как правило, эффективнее сортировки.

В отличие от HashAggregate, оба узла: Unique и GroupAggregate – возвращают упорядоченные данные.

Группировка сортировкой

Пример плана с узлом GroupAggregate, в котором происходит группировка отсортированного набора:

```
=> EXPLAIN (costs off)
SELECT ticket_no, count(ticket_no)
FROM ticket_flights
GROUP BY ticket_no;
```

QUERY PLAN

```
-----
GroupAggregate
  Group Key: ticket_no
    -> Index Only Scan using ticket_flights_pkey on ticket_flights
(3 rows)
```

В этом запросе сортировку набора данных обеспечивает индексный доступ.

Запрос с DISTINCT использует для устранения дубликатов узел Unique:

```
=> EXPLAIN (costs off)
SELECT DISTINCT ticket_no
FROM ticket_flights
ORDER BY ticket_no;
```

QUERY PLAN

```
-----
Result
  -> Unique
    -> Index Only Scan using ticket_flights_pkey on ticket_flights
(3 rows)
```

Поскольку узел возвращает результаты, упорядоченные так же, как требует предложение ORDER BY, дополнительная сортировка не требуется.

Комбинированная группировка

При группировке по нескольким наборам полей (которая возникает при использовании конструкций GROUPING SETS, CUBE или ROLLUP в предложении GROUP BY) может оказаться выгодным совместно использовать группировку хешированием и группировку сортировкой. В плане запроса такая комбинированная группировка отображается в виде узла MixedAggregate.

В качестве примера рассмотрим запрос к таблице перелетов, в котором подсчитывается и объединяется количество строк для трех различных вариантов группировки:

```
=> EXPLAIN (costs off)
SELECT fare_conditions, ticket_no, amount, count(*)
FROM ticket_flights
GROUP BY
  GROUPING SETS (fare_conditions, ticket_no, amount);
```

QUERY PLAN

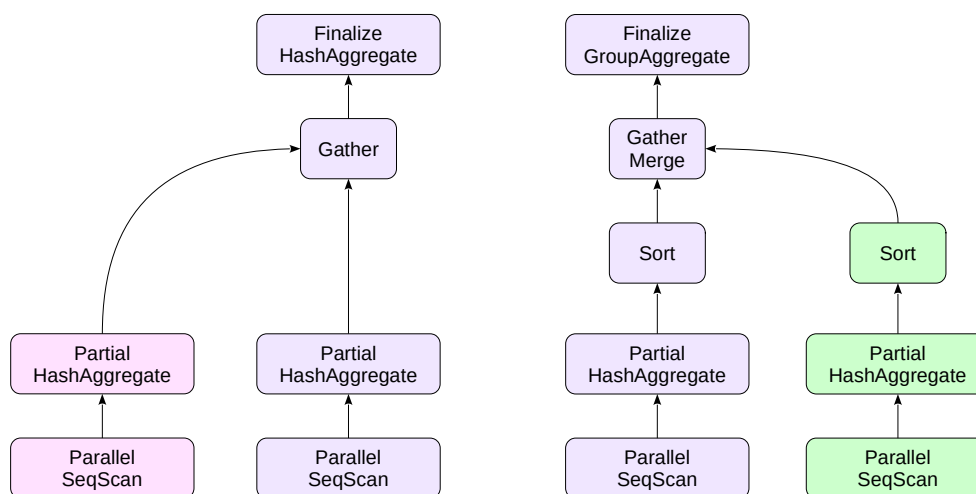
```
-----
MixedAggregate
  Hash Key: amount
  Group Key: fare_conditions
  Sort Key: ticket_no
    Group Key: ticket_no
  -> Sort
    Sort Key: fare_conditions
    -> Seq Scan on ticket_flights
(8 rows)
```

Узел MixedAggregate получает набор данных, отсортированный по столбцу fare_conditions.

На первом этапе в процессе чтения набора выполняется группировка по столбцу fare_conditions (Group Key). По мере чтения строки переупорядочиваются по столбцу ticket_no и одновременно с этим записываются в хеш-таблицу с ключом amount.

На втором этапе сканируется набор строк, отсортированный на предыдущем этапе по столбцу ticket_no, и значения группируются по этому же столбцу (Sort Key и вложенный Group Key).

Наконец, сканируется хеш-таблица, подготовленная на первом этапе, и значения группируются по столбцу amount (Hash Key).



Агрегация не имеет специальной параллельной реализации, но может выполняться в параллельных планах. В этом случае каждый рабочий процесс агрегирует свою часть данных и передает результат узлу Gather, который собирает данные в единый набор. Следующий узел выполняет агрегацию объединенного набора и вычисляет значение агрегатной функции.

Пример такого плана показан на слайде слева. Обычно такой вариант выигрывает, если количество групп достаточно велико.

Если же групп немного, может оказаться выгодным отсортировать сгруппированные строки в каждом процессе. А дальше воспользоваться тем, что наборы упорядочены, и на последнем этапе выполнить более быструю группировку сортировкой.

Пример такого плана показан на слайде справа.

В параллельных планах

Сгруппируем строки таблицы перелетов по рейсам (их, очевидно, много):

```
=> EXPLAIN (analyze, timing off, costs off)
SELECT flight_id, count(*)
FROM ticket_flights
GROUP BY flight_id;
```

QUERY PLAN

```
-----
Finalize HashAggregate (actual rows=150588 loops=1)
  Group Key: flight_id
  Batches: 5  Memory Usage: 8241kB  Disk Usage: 7832kB
  -> Gather (actual rows=439284 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    -> Partial HashAggregate (actual rows=146428 loops=3)
      Group Key: flight_id
      Batches: 5  Memory Usage: 8241kB  Disk Usage: 28072kB
      Worker 0:  Batches: 5  Memory Usage: 8241kB  Disk Usage: 27704kB
      Worker 1:  Batches: 5  Memory Usage: 8241kB  Disk Usage: 26384kB
      -> Parallel Seq Scan on ticket_flights (actual rows=2797284 loops=3)
Planning Time: 0.100 ms
Execution Time: 6304.018 ms
(14 rows)
```

Здесь каждый процесс в узле Partial HashAggregate вычисляет агрегатную функцию по своим данным с помощью собственной хеш-таблицы. Ведущий процесс после сбора данных в узле Gather получает итоговое значение агрегата, объединяя группы в узле Finalize HashAggregate тоже с помощью хеширования.

Теперь сгруппируем строки таблицы перелетов по стоимости. Количество групп будет небольшим:

```
=> SELECT count(DISTINCT amount) FROM ticket_flights;

 count
-----
    338
(1 row)
```

Как будет выполняться такой запрос?

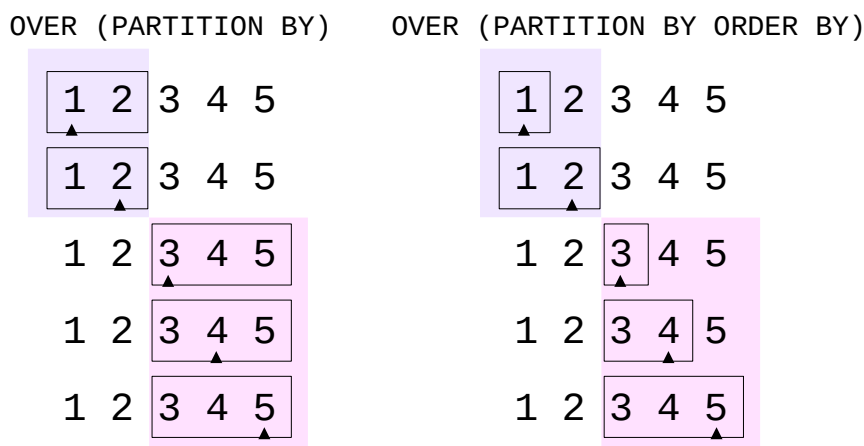
```
=> EXPLAIN (analyze, timing off, costs off)
SELECT amount, count(*)
FROM ticket_flights
GROUP BY amount;
```

QUERY PLAN

```
-----
-
Finalize GroupAggregate (actual rows=338 loops=1)
  Group Key: amount
  -> Gather Merge (actual rows=1010 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    -> Sort (actual rows=337 loops=3)
      Sort Key: amount
      Sort Method: quicksort  Memory: 38kB
      Worker 0:  Sort Method: quicksort  Memory: 38kB
      Worker 1:  Sort Method: quicksort  Memory: 38kB
      -> Partial HashAggregate (actual rows=337 loops=3)
        Group Key: amount
        Batches: 1  Memory Usage: 61kB
        Worker 0:  Batches: 1  Memory Usage: 61kB
        Worker 1:  Batches: 1  Memory Usage: 61kB
        -> Parallel Seq Scan on ticket_flights (actual rows=2797284 loops=3)
Planning Time: 0.107 ms
Execution Time: 2876.060 ms
(18 rows)
```

Поскольку групп немного, оказывается выгодным отсортировать результаты в каждом процессе (Sort), собрать их в ведущем процессе в один отсортированный набор (Gather Merge), а затем, объединяя группы, вычислить агрегатную функцию (Finalize GroupAggregate).

Группировка всегда использует сортировку



17

Группировка используется и в оконных функциях.

Если определение окна включает конструкцию `PARTITION BY`, оконная функция вычисляется на основе групп строк (аналогично группировке `GROUP BY`).

Для каждой группы строк границы рамки меняются в своих пределах. Если при этом нет других указаний, для всех строк одной группы будет получено одинаковое значение функции.

Как и в примере из темы «Сортировка», указание `ORDER BY` упорядочивает строки в каждой группе; в этом случае предполагается, что рамка охватывает строки от первой до текущей.

В отличие от обычной группировки `GROUP BY`, группировка в определении окна всегда реализуется с помощью сортировки: план строится так, чтобы на вход узлу `WindowAgg` пришел набор строк, отсортированный сначала по ключам группировки `PARTITION BY`, а затем — по ключам сортировки `ORDER BY`. Хеширование не применяется даже в отсутствие конструкции `ORDER BY`.

Группировка происходит не только при агрегации,
но и при устранении дубликатов

Выполняется с помощью хеширования или сортировки

1. Как выполняется запрос, вычисляющий количество мест каждой категории в таблице seats?
Попробуйте со значениями параметров по умолчанию, а затем запретите группировку хешированием.
2. Изучите запрос с оконной функцией и предложением PARTITION BY:

```
SELECT status, count(*) OVER (PARTITION BY status)
FROM flights
WHERE flight_no = 'PG0007'
AND departure_airport = 'VKO'
AND flight_id BETWEEN 24104 AND 24115;
```


Добавьте вызов функции row_number с тем же окном и сравните результаты и планы выполнения.

19

1. Запрос:

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

Для запрета группировки хешированием установите параметр *enable_hashagg* в off.

1. Разные способы группировки

Сначала выполним запрос со значениями параметров по умолчанию:

```
=> 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)
```

Для группировки используется узел HashAggregate.

Запретим группировку хешированием:

```
=> SET enable_hashagg = off;
```

SET

Повторно выполним запрос и сравним планы выполнения:

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

-----
QUERY PLAN
-----
GroupAggregate (cost=90.93..101.00 rows=3 width=16)
  Group Key: fare_conditions
    -> Sort (cost=90.93..94.28 rows=1339 width=8)
        Sort Key: fare_conditions
        -> Seq Scan on seats  (cost=0.00..21.39 rows=1339 width=8)
(5 rows)
```

Теперь остается только вариант с узлом GroupAggregate, а для него приходится сортировать данные в узле Sort.

Вернем значение по умолчанию:

```
=> RESET enable_hashagg;
```

RESET

2. Оконные функции и PARTITION BY

Приведенный запрос возвращает количество элементов в каждой группе:

```
=> SELECT status, count(*) OVER (PARTITION BY status)
FROM flights
WHERE flight_no = 'PG0007'
AND departure_airport = 'VKO'
AND flight_id BETWEEN 24104 AND 24115;
```

status	count
Arrived	4
Arrived	4
Arrived	4
Arrived	4
Scheduled	2
Scheduled	2

(6 rows)

В плане выполнения видно, что группировка обеспечивается тем же узлом WindowAgg, который мы видели при использовании конструкции ORDER BY в определении окна:

```
=> EXPLAIN (costs off)
SELECT status, count(*) OVER (PARTITION BY status)
FROM flights
WHERE flight_no = 'PG0007'
AND departure_airport = 'VKO'
AND flight_id BETWEEN 24104 AND 24115;
```

QUERY PLAN

```
-----
WindowAgg
  -> Sort
      Sort Key: status
      -> Index Scan using flights_pkey on flights
          Index Cond: ((flight_id >= 24104) AND (flight_id <= 24115))
          Filter: ((flight_no = 'PG0007'::bpchar) AND (departure_airport =
'VKO'::bpchar))
(6 rows)
```

Добавим номер текущей строки в ее группе с помощью функции row_number():

```
=> SELECT status,
       count(*) OVER (PARTITION BY status),
       row_number() OVER (PARTITION BY status)
FROM flights
WHERE flight_no = 'PG0007'
AND departure_airport = 'VKO'
AND flight_id BETWEEN 24104 AND 24115;
```

status	count	row_number
Arrived	4	1
Arrived	4	2
Arrived	4	3
Arrived	4	4
Scheduled	2	1
Scheduled	2	2

(6 rows)

Добавление новой функции с совпадающим окном не порождает новые узлы в плане:

```
=> EXPLAIN (costs off)
SELECT status,
       count(*) OVER (PARTITION BY status),
       row_number() OVER (PARTITION BY status)
FROM flights
WHERE flight_no = 'PG0007'
AND departure_airport = 'VKO'
AND flight_id BETWEEN 24104 AND 24115;
```

QUERY PLAN

```
-----
WindowAgg
  -> Sort
      Sort Key: status
      -> Index Scan using flights_pkey on flights
          Index Cond: ((flight_id >= 24104) AND (flight_id <= 24115))
          Filter: ((flight_no = 'PG0007'::bpchar) AND (departure_airport =
'VKO'::bpchar))
(6 rows)
```