

Сортировка и группировка

Сортировка



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.

Получение отсортированных данных

Сортировка в памяти

Внешняя сортировка

Инкрементальная сортировка

Сортировка в параллельных планах

Сортировка при построении индекса

Оконные функции с сортировкой

Индексный доступ (В-дерево)

возвращает отсортированный набор строк

Последовательный доступ (Seq Scan)

возвращает неупорядоченный набор строк

необходима дополнительная операция сортировки

Как уже было сказано ранее, индексный доступ (при использовании В-дерева) позволяет сразу же получить набор строк, отсортированный по ключам индексирования.

Но что делать, если сортировка требуется по полям, для которых нет индекса? В этом случае серверу приходится сначала получить данные из таблицы с помощью последовательного сканирования, а затем выполнить дополнительную операцию сортировки.

Получение отсортированных данных

Индексный доступ автоматически возвращает строки, отсортированные по проиндексированному столбцу:

```
=> EXPLAIN (costs off)
SELECT * FROM flights
ORDER BY flight_id;
```

QUERY PLAN

```
-----
Index Scan using flights_pkey on flights
(1 row)
```

Но если попросить сервер отсортировать данные по столбцу без индекса, потребуется два отдельных шага: получение данных и сортировка.

```
=> EXPLAIN (costs off)
SELECT * FROM flights
ORDER BY status;
```

QUERY PLAN

```
-----
Sort
  Sort Key: status
  -> Seq Scan on flights
(3 rows)
```

Дальше мы будем разбираться, как устроена сортировка в узле Sort.

Быстрая сортировка (quick sort)

Частичная пирамидальная сортировка (top-N heapsort)

когда нужна только часть значений

В идеальном случае набор строк, подлежащий сортировке, целиком помещается в память, ограниченную параметром *work_mem*. В этом случае все строки просто сортируются алгоритмом *быстрой сортировки* (quick sort) и результат возвращается родительскому узлу.

Если нужно получить лишь несколько первых строк отсортированного набора (при использовании предложения LIMIT), сортировать весь набор не обязательно. В этом случае сервер может применить *частичную пирамидальную сортировку* (top-N heapsort), которая также выполняется в оперативной памяти.

Сортировка в памяти

В распоряжении планировщика имеется несколько методов сортировки. В следующем примере используется быстрая сортировка (Sort Method: quicksort). В той же строке указан объем использованной памяти:

```
=> EXPLAIN (analyze, timing off, summary off)
SELECT *
FROM seats
ORDER BY seat_no;
```

QUERY PLAN

```
-----
-
Sort  (cost=90.93..94.28 rows=1339 width=15) (actual rows=1339 loops=1)
  Sort Key: seat_no
  Sort Method: quicksort  Memory: 101kB
    -> Seq Scan on seats  (cost=0.00..21.39 rows=1339 width=15) (actual rows=1339 loops=1)
(4 rows)
```

Чтобы начать выдавать данные, узлу Sort нужно полностью отсортировать набор строк. Поэтому начальная стоимость узла включает в себя полную стоимость чтения таблицы. В общем случае сложность быстрой сортировки равна $O(M \log M)$, где M — число строк в исходном наборе.

Если набор строк ограничен, планировщик может переключиться на частичную сортировку (грубо говоря, вместо полной сортировки здесь 100 раз находится минимальное значение):

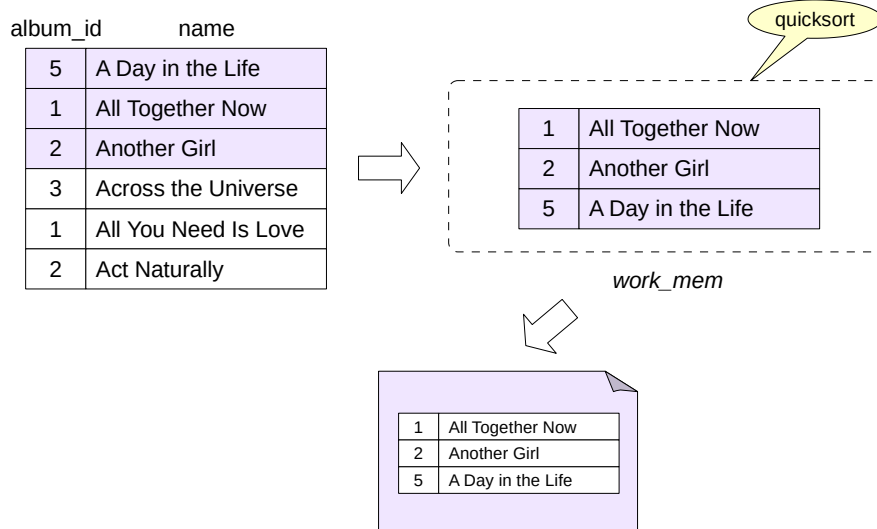
```
=> EXPLAIN (analyze, timing off, summary off)
SELECT *
FROM seats
ORDER BY seat_no
LIMIT 100;
```

QUERY PLAN

```
-----
-----
Limit  (cost=72.57..72.82 rows=100 width=15) (actual rows=100 loops=1)
  -> Sort  (cost=72.57..75.91 rows=1339 width=15) (actual rows=100 loops=1)
    Sort Key: seat_no
    Sort Method: top-N heapsort  Memory: 32kB
      -> Seq Scan on seats  (cost=0.00..21.39 rows=1339 width=15) (actual rows=1339
loops=1)
(5 rows)
```

Обратите внимание, что стоимость запроса снизилась и для сортировки потребовалось меньше памяти. В общем случае сложность алгоритма top-N heapsort ниже, чем для быстрой сортировки — она равна $O(M \log N)$.

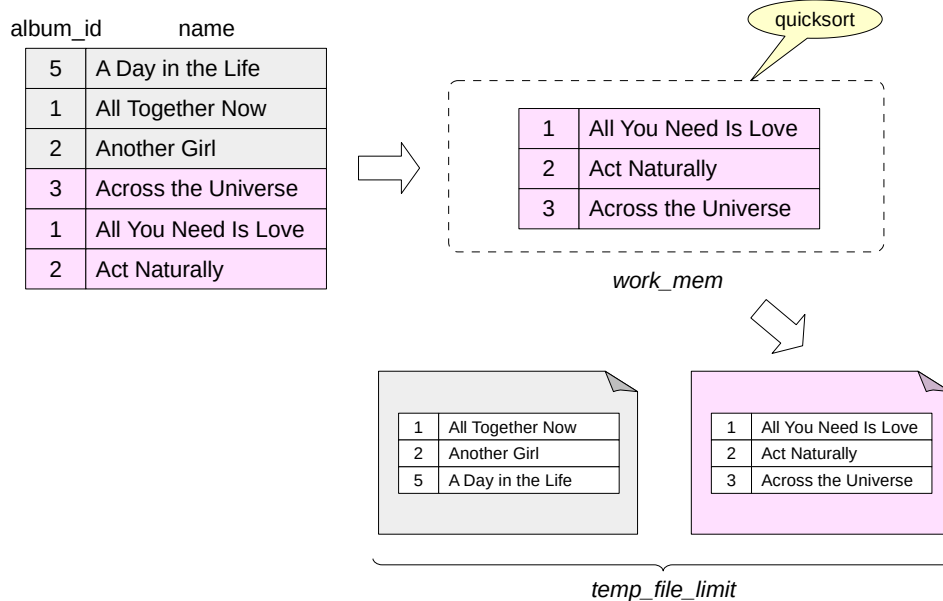
Внешняя сортировка



Если набор строк велик, он не поместится в память целиком. В таком случае используется алгоритм внешней сортировки.

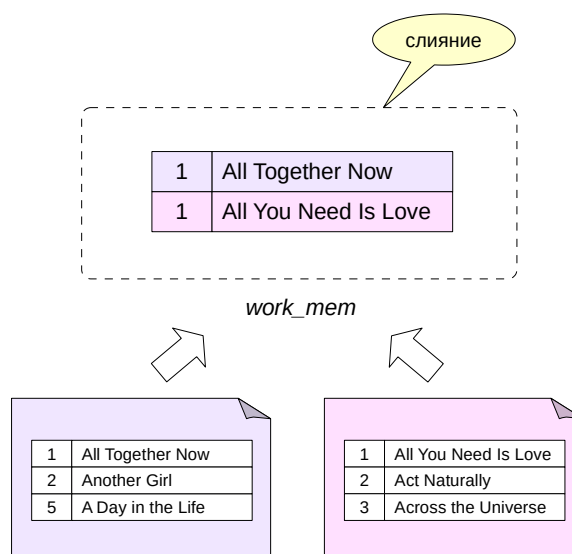
Набор строк читается в память, пока есть возможность, затем сортируется и записывается во временный файл.

Внешняя сортировка



Эта процедура повторяется столько раз, сколько необходимо, чтобы записать все данные в файлы, каждый из которых по отдельности отсортирован.

Напомним, что общий размер временных файлов сеанса (не включая временные таблицы) ограничен значением параметра *temp_file_limit*.



Далее несколько файлов (два или более) соединяются с сохранением порядка строк.

Для соединения не требуется много места в памяти, достаточно разместить по одной строке из каждого файла (как в примере на слайде). Среди этих строк выбирается минимальная (максимальная) и возвращается как часть результата, а на ее место читается новая строка из того же файла. На практике строки читаются не по одной, а пакетами, чтобы ускорить ввод-вывод.

Если оперативной памяти недостаточно, чтобы соединить сразу все файлы, сначала соединяется часть файлов и результат записывается в новый временный файл, затем происходит соединение этих новых файлов и так далее.

Мы не обсуждаем здесь все детали реализации сортировки, с ними можно ознакомиться в файле [src/backend/utils/sort/tuplesort.c](https://github.com/postgres/postgres/blob/master/src/backend/utils/sort/tuplesort.c).

История развития способов сортировки в PostgreSQL хорошо описана в презентации Грегори Старка «Sorting Through The Ages»: https://wiki.postgresql.org/images/5/59/Sorting_through_the_ages.pdf. синхронный перевод доклада на русский язык доступен на сайте <https://pgconf.ru/talk/1587768>.

Внешняя сортировка

Пример плана с внешней сортировкой (Sort Method: external merge):

```
=> EXPLAIN (analyze, buffers, timing off, summary off)
SELECT *
FROM flights
ORDER BY scheduled_departure;
```

QUERY PLAN

```
-----
Sort  (cost=31883.96..32421.12 rows=214867 width=63) (actual rows=214867 loops=1)
  Sort Key: scheduled_departure
  Sort Method: external merge  Disk: 17112kB
  Buffers: shared hit=3 read=2624, temp read=2139 written=2145
    -> Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63) (actual rows=214867
loops=1)
      Buffers: shared read=2624
Planning:
  Buffers: shared hit=10 read=1
(8 rows)
```

Обратите внимание на то, что узел Sort записывает и читает временные данные (temp read и written).

Увеличим значение work_mem:

```
=> SET work_mem = '48MB';
```

SET

```
=> EXPLAIN (analyze, buffers, timing off, summary off)
SELECT *
FROM flights
ORDER BY scheduled_departure;
```

QUERY PLAN

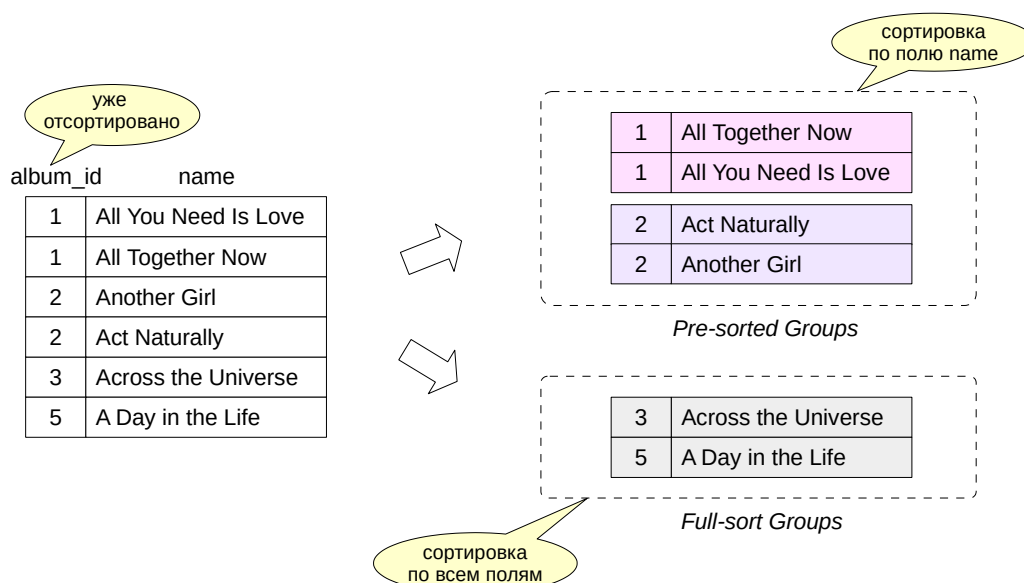
```
-----
Sort  (cost=23802.46..24339.62 rows=214867 width=63) (actual rows=214867 loops=1)
  Sort Key: scheduled_departure
  Sort Method: quicksort  Memory: 26161kB
  Buffers: shared hit=2624
    -> Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63) (actual rows=214867
loops=1)
      Buffers: shared hit=2624
(6 rows)
```

Теперь все строки поместились в память, и планировщик выбрал более дешевую быструю сортировку.

```
=> RESET work_mem;
```

RESET

Incremental Sort



11

Если набор данных требуется отсортировать по ключам $K_1 \dots K_m \dots K_n$ и при этом известно, что набор уже отсортирован по первым m ключам, можно не пересортировывать все данные заново. Достаточно разбить набор данных на группы, имеющие одинаковые значения начальных ключей $K_1 \dots K_m$ (значения таких групп следуют друг за другом), и затем отсортировать отдельно каждую из групп по оставшимся ключам $K_{m+1} \dots K_n$. Такой способ называется *инкрементальной сортировкой*.

Алгоритм обрабатывает относительно крупные группы строк отдельно, а небольшие объединяет и сортирует полностью. Поскольку сортируемые наборы данных уменьшаются, уменьшается и требование к доступному объему памяти. Но увеличение количества наборов приводит к росту накладных расходов.

Наборы могут обрабатываться как в оперативной памяти, так и на диске — если не хватает объема *work_mem*.

Инкрементальная сортировка позволяет выдавать результаты уже после обработки первой группы, не дожидаясь сортировки всего набора.

Отключить использование инкрементальной сортировки можно с помощью параметра *enable_incremental_sort*.

Инкрементальная сортировка

Инкрементальная сортировка может использовать как сортировку в памяти, так и внешнюю сортировку.

Для иллюстрации создадим индекс на таблице bookings:

```
=> CREATE INDEX ON bookings(total_amount);
```

CREATE INDEX

Посмотрим на пример инкрементальной сортировки:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM bookings
ORDER BY total_amount, book_date;
```

QUERY PLAN

```
-----
Incremental Sort (actual rows=2111110 loops=1)
  Sort Key: total_amount, book_date
  Presorted Key: total_amount
  Full-sort Groups: 2823  Sort Method: quicksort  Average Memory: 28kB  Peak Memory: 28kB
  Pre-sorted Groups: 2624  Sort Method: quicksort  Average Memory: 2193kB  Peak Memory:
2263kB
  -> Index Scan using bookings_total_amount_idx on bookings (actual rows=2111110
loops=1)
(6 rows)
```

Здесь данные, полученные из таблицы bookings по только что созданному индексу bookings_total_amount_idx, уже отсортированы по столбцу total_amount (Presorted Key), поэтому остается доупорядочить строки по столбцу book_date.

Строка Pre-sorted Groups относится к крупным группам, которые досортировывались по столбцу book_date, а строка Full-sort Groups — к небольшим группам, которые были объединены и отсортированы полностью. В примере все группы поместились в выделенную память и применялась быстрая сортировка.

Уменьшим work_mem и повторим запрос:

```
=> SET work_mem = '128 kB';
```

SET

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM bookings
ORDER BY total_amount, book_date;
```

QUERY PLAN

```
-----
Incremental Sort (actual rows=2111110 loops=1)
  Sort Key: total_amount, book_date
  Presorted Key: total_amount
  Full-sort Groups: 2823  Sort Method: quicksort  Average Memory: 28kB  Peak Memory: 28kB
  Pre-sorted Groups: 2624  Sort Methods: quicksort, external merge  Average Memory: 0kB
Peak Memory: 45kB  Average Disk: 1047kB  Peak Disk: 1088kB
  -> Index Scan using bookings_total_amount_idx on bookings (actual rows=2111110
loops=1)
(6 rows)
```

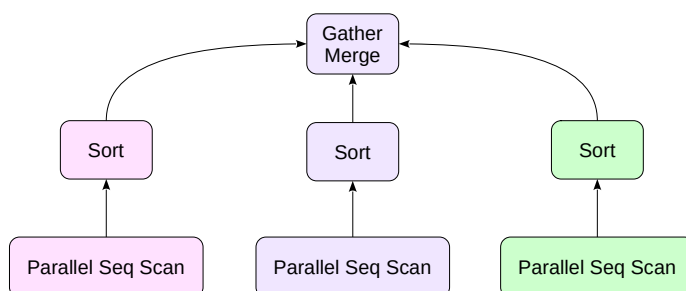
Теперь, при нехватке памяти, для некоторых крупных групп пришлось применить внешнюю сортировку с использованием временных файлов (Pre-sorted Groups ... external merge).

```
=> RESET work_mem;
```

RESET

Узел Gather Merge сохраняет порядок сортировки

выполняет слияние данных, поступающих от дочерних узлов



13

Сортировка может участвовать в параллельных планах. Каждый рабочий процесс сортирует свою часть данных и передает результат вышестоящему узлу, который собирает данные в единый набор.

Но узел Gather для этого не годится, поскольку он выдает результат в том порядке, в котором строки поступают от рабочих процессов.

Поэтому в таких планах используется узел Gather Merge, сохраняющий порядок сортировки поступающих строк. Для этого он реализует алгоритм слияния, объединяя несколько отсортированных наборов в один.

В параллельных планах

Запрос, сортирующий большую таблицу по неиндексированному полю, может выполняться параллельно:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM bookings
ORDER BY book_date;
```

QUERY PLAN

```
-----
Gather Merge (actual rows=2111110 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Sort (actual rows=703703 loops=3)
        Sort Key: book_date
        Sort Method: external merge  Disk: 25288kB
        Worker 0:  Sort Method: external merge  Disk: 21216kB
        Worker 1:  Sort Method: external merge  Disk: 21720kB
        -> Parallel Seq Scan on bookings (actual rows=703703 loops=3)
(9 rows)
```

Здесь каждый процесс читает и сортирует свою часть таблицы, а затем отсортированные наборы сливаются при передаче их ведущему процессу с сохранением порядка.

Используется сортировка

сначала все строки сортируются
затем строки собираются в листовые индексные страницы
ссылки на них собираются в страницы следующего уровня
и так далее, пока не дойдем до корня

Может выполняться параллельно

max_parallel_maintenance_workers

Ограничение

maintenance_work_mem, так как операция не частая

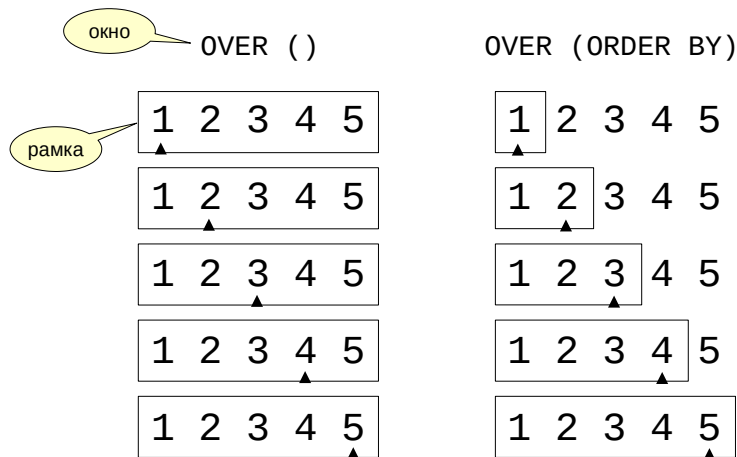
При построении индекса (речь идет про В-дерево) сервер мог бы добавлять записи в пустой индекс по одной, обрабатывая последовательно строки таблицы. Но такой способ крайне неэффективен.

Поэтому при создании и перестроении индекса используется сортировка: все строки таблицы сортируются, и формируются записи, которые раскладываются по листовым индексным страницам. Затем достраиваются верхние уровни дерева, состоящие из ссылок на элементы страниц нижележащего уровня, до тех пор, пока на очередном уровне не получится одна страница — она и будет корнем дерева.

Сортировка устроена точно так же, как рассматривалось выше. Однако размер памяти ограничен не *work_mem*, а *maintenance_work_mem*, поскольку операция создания индекса не слишком частая и имеет смысл выделить для нее больше памяти.

Построение индекса может выполняться параллельно. Количество рабочих процессов выбирается так же, как и при выполнении параллельных запросов (в зависимости от размера таблицы), но ограничено значением параметра *max_parallel_maintenance_workers*.

Окно определяет агрегируемую выборку для каждой строки
В отсортированном наборе границы рамки могут двигаться



16

Использование сортировки в оконных функциях имеет особенности.

В отличие от обычных агрегатных функций, оконные функции вычисляются для каждой строки набора, не уменьшая его размер. После имени функции в предложении `OVER` указывается *окно*, определяющее набор строк, которую должна обработать оконная функция в каждой строке набора. Такая выборка называется *рамкой*.

Если окно указано как `OVER()`, то рамка одинакова для каждой строки и состоит из всех строк набора.

Оконные функции

Посмотрим план запроса, вычисляющего сумму бронирований нарастающим итогом:

```
=> EXPLAIN SELECT *, sum(total_amount) OVER (ORDER BY book_date)
FROM bookings;
```

QUERY PLAN

```
-----
WindowAgg (cost=342915.67..379860.10 rows=2111110 width=53)
  -> Sort (cost=342915.67..348193.45 rows=2111110 width=21)
      Sort Key: book_date
      -> Seq Scan on bookings (cost=0.00..34558.10 rows=2111110 width=21)
(4 rows)
```

Оконная функция вычисляется в узле WindowAgg, который получает отсортированные данные от дочернего узла Sort.

Добавление к запросу других оконных функций, использующих тот же порядок строк (не обязательно с совпадающим окном), а также предложения ORDER BY не приводит к появлению лишних сортировок:

```
=> EXPLAIN SELECT *,
    sum(total_amount) OVER (ORDER BY book_date),
    avg(total_amount) OVER (ORDER BY book_date ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING)
FROM bookings
ORDER BY book_date;
```

QUERY PLAN

```
-----
WindowAgg (cost=342915.67..411526.75 rows=2111110 width=85)
  -> WindowAgg (cost=342915.67..379860.10 rows=2111110 width=53)
      -> Sort (cost=342915.67..348193.45 rows=2111110 width=21)
          Sort Key: book_date
          -> Seq Scan on bookings (cost=0.00..34558.10 rows=2111110 width=21)
(5 rows)
```

Здесь два узла WindowAgg (каждый для своего окна), но общий узел Sort. Стоимость запроса увеличилась, но лишь немного.

Конечно, оконные функции, требующие разного порядка строк, вынуждают сервер пересортировывать данные:

```
=> EXPLAIN SELECT *,
    sum(total_amount) OVER (ORDER BY book_date),
    count(*) OVER (ORDER BY book_ref)
FROM bookings;
```

QUERY PLAN

```
-----
WindowAgg (cost=422743.30..459687.73 rows=2111110 width=61)
  -> Sort (cost=422743.30..428021.08 rows=2111110 width=29)
      Sort Key: book_date
      -> WindowAgg (cost=0.43..99951.73 rows=2111110 width=29)
          -> Index Scan using bookings_pkey on bookings (cost=0.43..68285.08
rows=2111110 width=21)
(5 rows)
```

В этом примере нижний узел WindowAgg (соответствующий функции count) получает упорядоченный набор строк по индексу, а для верхнего узла WindowAgg (соответствующего функции sum) строки переупорядочивает узел Sort.

Сортировка в оконных функциях может использовать любые методы, рассмотренные выше. Например, быструю сортировку:

```
=> EXPLAIN (analyze, buffers, timing off, summary off)
SELECT *, count(*) OVER (ORDER BY seat_no)
FROM seats;
```

QUERY PLAN

```
-----  
-----  
WindowAgg (cost=90.93..114.36 rows=1339 width=23) (actual rows=1339 loops=1)  
  Buffers: shared read=8  
    -> Sort (cost=90.93..94.28 rows=1339 width=15) (actual rows=1339 loops=1)  
        Sort Key: seat_no  
        Sort Method: quicksort Memory: 101kB  
        Buffers: shared read=8  
        -> Seq Scan on seats (cost=0.00..21.39 rows=1339 width=15) (actual rows=1339  
loops=1)  
            Buffers: shared read=8  
(8 rows)
```

Или внешнюю:

```
=> EXPLAIN (analyze, buffers, timing off, summary off)  
SELECT *, sum(total_amount) OVER (ORDER BY book_date)  
FROM bookings;
```

QUERY PLAN

```
-----  
-----  
WindowAgg (cost=342915.67..379860.10 rows=2111110 width=53) (actual rows=2111110  
loops=1)  
  Buffers: shared hit=13447, temp read=16491 written=16528  
    -> Sort (cost=342915.67..348193.45 rows=2111110 width=21) (actual rows=2111110  
loops=1)  
        Sort Key: book_date  
        Sort Method: external merge Disk: 65984kB  
        Buffers: shared hit=13447, temp read=16491 written=16528  
        -> Seq Scan on bookings (cost=0.00..34558.10 rows=2111110 width=21) (actual  
rows=2111110 loops=1)  
            Buffers: shared hit=13447  
(8 rows)
```

Сортировка используется при выполнении запросов
и для построения В-деревьев

Существуют разные способы реализации сортировки

- в памяти

- внешняя (создаются временные файлы)

- инкрементальная

Наличие индексов может позволить избежать сортировки

1. Какой план будет выбран для следующего запроса:

```
SELECT *  
FROM flights  
ORDER BY scheduled_departure;
```

Изменится ли план выполнения, если увеличить значение *work_mem* до 32 Мбайт?
2. Создайте индекс по столбцам *passenger_name* и *passenger_id* таблицы билетов (*tickets*). Потребовался ли временный файл для выполнения этой операции?

2. Включите журналирование использования временных файлов, установив значение параметра *log_temp_files* в ноль.

1. Выполнение запроса с сортировкой

Выполним запрос со значениями параметров по умолчанию:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT *
FROM flights
ORDER BY scheduled_departure;

-----
QUERY PLAN
-----
Sort (actual rows=214867 loops=1)
  Sort Key: scheduled_departure
  Sort Method: external merge  Disk: 17112kB
  Buffers: shared hit=3 read=2624, temp read=2139 written=2145
  -> Seq Scan on flights (actual rows=214867 loops=1)
      Buffers: shared read=2624
Planning:
  Buffers: shared hit=112 read=25 dirtied=6
Planning Time: 4.511 ms
Execution Time: 131.380 ms
(10 rows)
```

Сервер выбрал внешнюю сортировку (Sort Method: external merge).

Обратите внимание на количество страниц и тип ввода-вывода (Buffers): значения temp read и written говорят о том, что сервер использовал временные файлы.

Выполните запрос повторно несколько раз — видно, что серверу всегда не хватает оперативной памяти.

Увеличим значение параметра work_mem до 32 Мбайт:

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

SET

Увеличив work_mem, мы позволяем серверу использовать больше оперативной памяти для сортировки.

Повторно выполним запрос и сравним планы выполнения:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT *
FROM flights
ORDER BY scheduled_departure;

-----
QUERY PLAN
-----
Sort (actual rows=214867 loops=1)
  Sort Key: scheduled_departure
  Sort Method: quicksort  Memory: 26161kB
  Buffers: shared hit=2624
  -> Seq Scan on flights (actual rows=214867 loops=1)
      Buffers: shared hit=2624
Planning Time: 0.069 ms
Execution Time: 92.015 ms
(8 rows)
```

Теперь сервер использует сортировку в памяти (Sort Method: quicksort), в строке Buffers исчезли поля temp read и written.

2. Построение индекса

Включим журналирование временных файлов:

```
=> SET log_temp_files = 0;
```

SET

Текущее значение maintenance_work_mem:

```
=> SHOW maintenance_work_mem;
```

```
maintenance_work_mem
-----
64MB
(1 row)
```

Создаем индекс:

```
=> \timing on
```

Timing is on.

```
=> CREATE INDEX ON tickets(passenger_name, passenger_id);
```

CREATE INDEX

Time: 22746,443 ms (00:22,746)

Временный файл понадобился:

```
student$ tail -n 2 /var/log/postgresql/postgresql-16-main.log
```

```
2025-02-05 11:32:22.476 MSK [80891] postgres@demo LOG:  temporary file: path  
"base/pgsql_tmp/pgsql_tmp80891.0.fileset/0.0", size 64790528
```

```
2025-02-05 11:32:22.476 MSK [80891] postgres@demo STATEMENT:  CREATE INDEX ON  
tickets(passenger_name, passenger_id);
```