

Оптимизация запросов Соединение слиянием



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.

Соединение слиянием

Сортировка

Алгоритм соединения слиянием

Вычислительная сложность

Соединение слиянием в параллельных планах

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

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

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

либо сортировка,
либо получение отсортированных данных от нижестоящего узла плана

Третий, и последний, способ соединения — соединение слиянием.

Подготовительным этапом для него служит сортировка обоих наборов строк. Сортировка — дорогая операция, она имеет сложность $O(N \log N)$.

Но иногда этого этапа удастся избежать, если можно сразу получить отсортированные наборы строк, например, за счет индексного доступа к таблице.

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

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

результат соединения автоматически отсортирован

Само слияние устроено просто. Сначала берем первые строки обоих наборов и сравниваем их между собой. В данном случае мы сразу нашли соответствие и можем вернуть первую строку результата: («Yellow Submarine», «All Together Now»).

Общий алгоритм таков: читаем следующую строку того набора, для которого значение поля, по которому происходит соединение, меньше (один набор «догоняет» другой). Если же значения одинаковы, как в нашем примере, то читаем следующую строку второго набора.

(На самом деле алгоритм сложнее — что, если и в первом наборе строк может быть несколько одинаковых значений? Но мы не будем загромождать общую картину деталями. Псевдокод алгоритма можно посмотреть в файле `src/backend/executor/nodeMergejoin.c`.)

Важно, что алгоритм слияния возвращает результат соединения в отсортированном виде. В частности, полученный набор строк можно использовать для следующего соединения слиянием без дополнительной сортировки.

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

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

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

Вновь соответствие: («Yellow Submarine», «All You Need Is Love»).

Снова читаем следующую строку второго набора.

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


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

В данном случае соответствия нет.

Поскольку $1 < 2$, читаем следующую строку первого набора.

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

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



Соответствия нет.

3 > 2, поэтому читаем следующую строку второго набора.


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

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

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

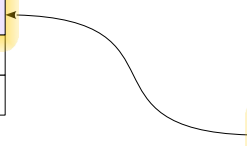


Снова нет соответствия, снова $3 > 2$, снова читаем строку второго набора.

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

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

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



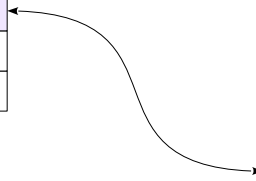
Есть соответствие: («Let It Be», «Across the Universe»).

3 = 3, читаем следующую строку второго набора.

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

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

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



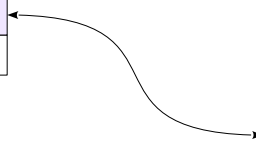
Соответствия нет.

3 < 5, читаем строку первого набора.

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

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

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



Соответствия нет.

4 < 5, читаем строку первого набора.

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

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

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



И последний шаг: снова нет соответствия.
На этом соединение слиянием окончено.

Соединение слиянием

Если результат необходим в отсортированном виде, оптимизатор может предпочесть соединение слиянием. Особенно, если данные от дочерних узлов можно получить уже отсортированными — как в этом примере:

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

QUERY PLAN

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

Вот еще один пример с двумя соединениями слиянием, в котором один узел Merge Join получает отсортированный набор от другого узла Merge Join:

```
=> EXPLAIN (costs off) SELECT t.ticket_no, bp.flight_id, bp.seat_no
FROM tickets t
     JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
     JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no
     AND bp.flight_id = tf.flight_id
ORDER BY t.ticket_no;
```

QUERY PLAN

```
-----
Merge Join
  Merge Cond: (tf.ticket_no = t.ticket_no)
    -> Merge Join
          Merge Cond: ((tf.ticket_no = bp.ticket_no) AND (tf.flight_id = bp.flight_id))
            -> Index Only Scan using ticket_flights_pkey on ticket_flights tf
            -> Index Scan using boarding_passes_pkey on boarding_passes bp
          -> Index Only Scan using tickets_pkey on tickets t
(7 rows)
```

Здесь соединяются перелеты (ticket_flights) и посадочные талоны (boarding_passes), и с этим, уже отсортированным по номерам билетов, набором строк соединяются билеты (tickets).

$\sim N + M$, где

N и M — число строк в первом и втором наборах данных,
если не требуется сортировка

$\sim N \log N + M \log M$,

если сортировка нужна

Возможные начальные затраты на сортировку

Эффективно для большого числа строк

В случае, когда не требуется сортировать данные, общая сложность соединения слиянием пропорциональна сумме числа строк в обоих наборах данных. Но, в отличие от соединения хешированием, здесь не требуются накладные расходы на построение хеш-таблицы.

Поэтому соединение слиянием может успешно применяться как в OLTP-, так и в OLAP-запросах.

Однако если сортировка требуется, то стоимость становится пропорциональной произведению количества строк на логарифм этого количества, и на больших объемах такой способ скорее всего проиграет соединению хешированием.

Вычислительная сложность

Посмотрим на стоимость соединения слиянием:

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

QUERY PLAN

```
-----
Merge Join  (cost=1.04..822385.86 rows=8391852 width=136)
  Merge Cond: (t.ticket_no = tf.ticket_no)
    -> Index Scan using tickets_pkey on tickets t  (cost=0.43..139108.66 rows=2949749 width=104)
    -> Index Scan using ticket_flights_pkey on ticket_flights tf  (cost=0.56..571004.68 rows=8391852 width=32)
(4 rows)
```

Начальная стоимость включает:

- сумму начальных стоимостей дочерних узлов (включает стоимость сортировки, если она необходима);
- стоимость получения первой пары строк, соответствующих друг другу.

Полная стоимость складывается из:

- суммы стоимостей получения обоих наборов данных;
- стоимости сравнения строк.

Общий вывод: стоимость соединения слиянием пропорциональна $N + M$ (где N и M — число соединяемых строк), если не требуется отдельная сортировка. Сортировка набора из K строк добавляет к оценке как минимум $K * \log(K)$.

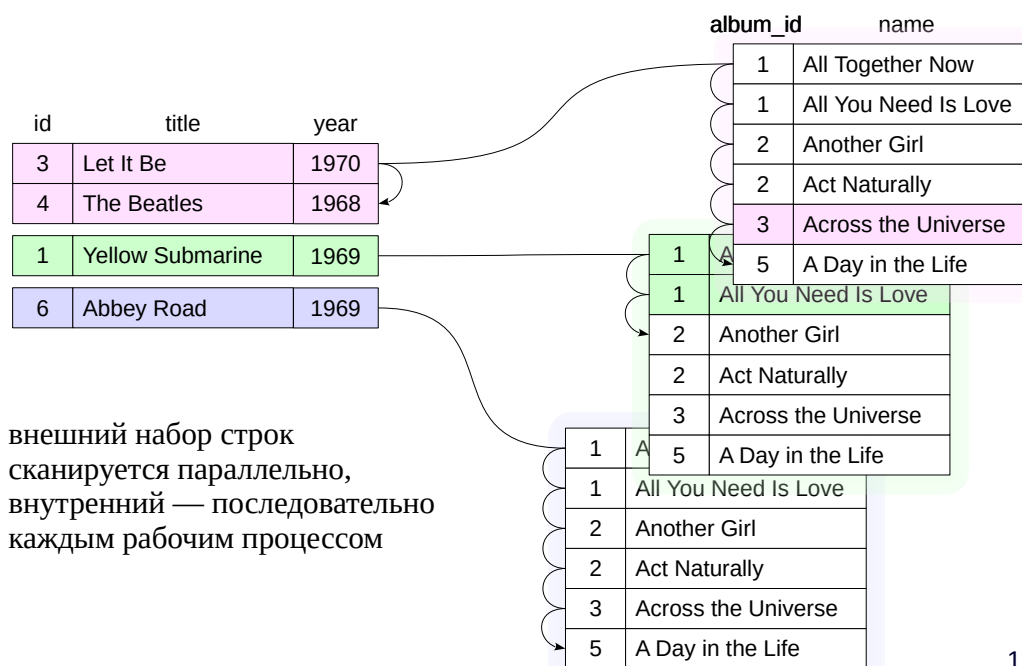
В отличие от соединения хешированием, слияние без сортировки хорошо подходит для случая, когда надо быстро получить первые строки.

```
=> EXPLAIN SELECT *
FROM tickets t
     JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no
LIMIT 1000;
```

QUERY PLAN

```
-----
Limit  (cost=1.04..99.04 rows=1000 width=136)
  -> Merge Join  (cost=1.04..822385.86 rows=8391852 width=136)
    Merge Cond: (t.ticket_no = tf.ticket_no)
      -> Index Scan using tickets_pkey on tickets t  (cost=0.43..139108.66 rows=2949749 width=104)
      -> Index Scan using ticket_flights_pkey on ticket_flights tf  (cost=0.56..571004.68 rows=8391852 width=32)
(5 rows)
```

Обратите внимание и на то, как уменьшилась общая стоимость.



Соединение слиянием может использоваться в параллельных планах.

Так же, как и при соединении вложенным циклом, сканирование одного набора строк выполняется рабочими процессами параллельно, но другой набор строк каждый рабочий процесс читает полностью самостоятельно. Поэтому при соединении больших объемов строк в параллельных планах гораздо чаще используется соединение хешированием, имеющее эффективный параллельный алгоритм.

Сортировка в памяти

Внешняя сортировка

Группировка с помощью сортировки

Сортировка в параллельных планах

Быстрая сортировка (quick sort)

Частичная пирамидальная сортировка (top-N heapsort)

когда нужна только часть значений

Инкрементальная сортировка

когда данные уже отсортированы, но не по всем ключам

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 вышестоящему узлу.

Если нужно отсортировать не весь набор данных, а только его часть (при использовании предложения LIMIT), может применяться *частичная пирамидальная сортировка* (top-N heapsort).

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*. Поскольку сортируемые наборы данных уменьшаются, уменьшается и требование к доступному объему памяти.

Сортировка в памяти

Сортировка выполняется в узле Sort. Чтобы начать выдавать данные вышестоящему узлу, сортировка должна быть полностью завершена.

Вот пример плана с соединением слиянием, использующим сортировку (здесь явная сортировка выбрана из-за небольшого размера таблицы):

```
=> EXPLAIN (costs off) SELECT *
FROM aircrafts a
JOIN seats s ON a.aircraft_code = s.aircraft_code
ORDER BY a.aircraft_code;
```

QUERY PLAN

```
-----
Merge Join
  Merge Cond: (s.aircraft_code = ml.aircraft_code)
    -> Index Scan using seats_pkey on seats s
    -> Sort
          Sort Key: ml.aircraft_code
          -> Seq Scan on aircrafts_data ml
(6 rows)
```

В следующем примере для сортировки используется быстрая сортировка (Sort Method). В той же строке указан использованный объем 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: 111kB
  -> Seq Scan on seats (cost=0.00..21.39 rows=1339 width=15) (actual rows=1339 loops=1)
(4 rows)
```

Если набор строк ограничен, планировщик может переключиться на частичную сортировку (грубо говоря, вместо полной сортировки здесь 100 раз находится максимальное 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: 33kB
        -> 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.

Пример инкрементальной сортировки, доступной с 13-й версии PostgreSQL:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM tickets
ORDER BY ticket_no, passenger_id;
```

QUERY PLAN

```
-----
Incremental Sort (actual rows=2949857 loops=1)
  Sort Key: ticket_no, passenger_id
  Presorted Key: ticket_no
  Full-sort Groups: 92184  Sort Method: quicksort  Average Memory: 31kB  Peak Memory: 31kB
  -> Index Scan using tickets_pkey on tickets (actual rows=2949857 loops=1)
(5 rows)
```

Здесь данные, полученные из таблицы tickets по индексу tickets_pkey, уже отсортированы по столбцу ticket_no (Presorted Key), поэтому остается доупорядочить строки по столбцу passenger_id. Для сортировки отдельных групп использовалась быстрая сортировка.

Группировка и уникальные значения

Как мы видели в предыдущей теме, для устранения дубликатов может использоваться хеширование. Другой способ — сортировка значений:

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

QUERY PLAN

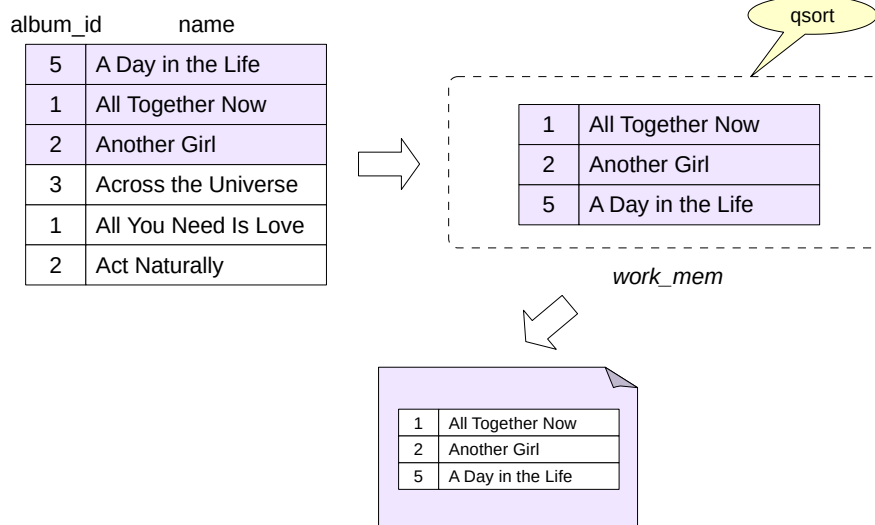
```
-----
Result
  -> Unique
    -> Index Only Scan using ticket_flights_pkey on ticket_flights
(3 rows)
```

Такой способ особенно выгоден, если требуется получить отсортированный результат (как в данном случае).

Устранение дубликатов происходит в узле Unique. Он получает отсортированный набор строк (от узла Sort или, как в этом примере, от индексного сканирования) и убирает повторяющиеся значения.

Для группировки будет использоваться похожий по смыслу узел GroupAggregate.

Внешняя сортировка

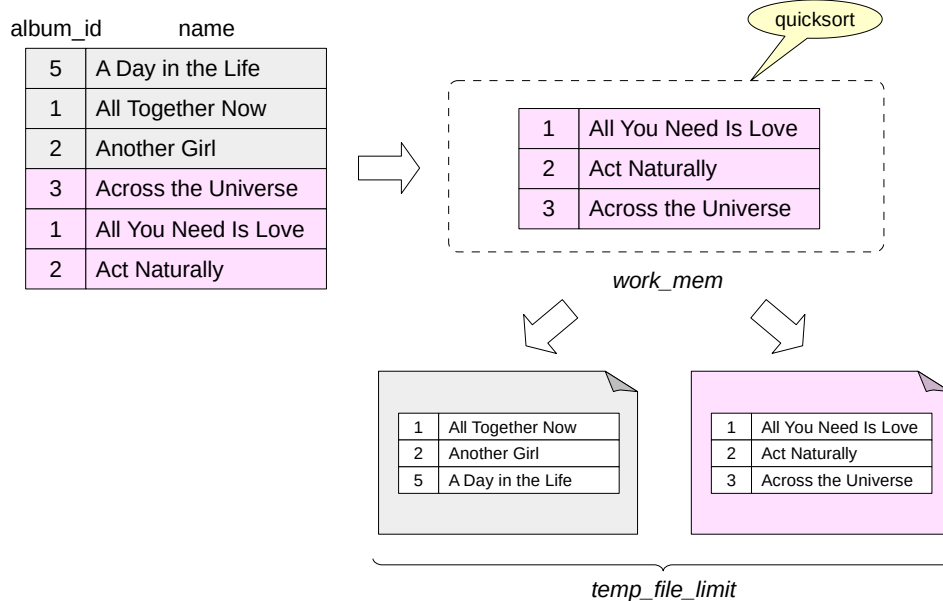


В идеальном случае набор строк, подлежащий сортировке, целиком помещается в память, ограниченную параметром `work_mem`. В этом случае все строки просто сортируются (используется алгоритм быстрой сортировки `qsort`) и возвращается результат.

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.

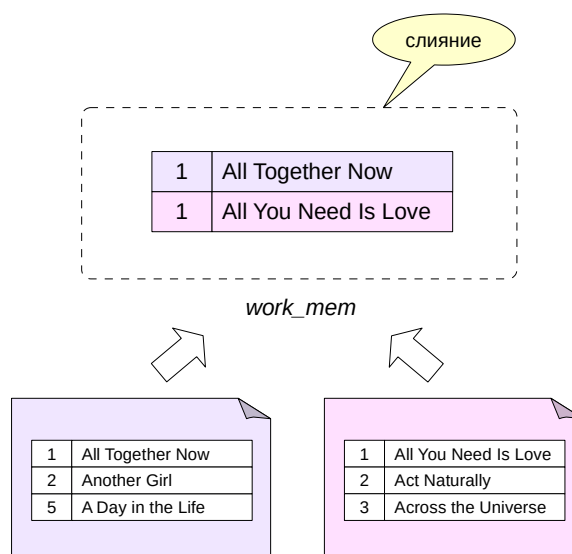
Внешняя сортировка



22

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 сливаются аналогично тому, как работает соединение слиянием. Основное отличие состоит в том, что сливаться могут более двух файлов одновременно.

Для слияния 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 порциями, чтобы оптимизировать ввод-вывод.

Если оперативной памяти недостаточно, чтобы слить сразу все файлы, процесс повторяется многократно: сливаются по несколько файлов за раз и результаты записываются в новые временные файлы, затем происходит слияние этих новых файлов и так далее.

На самом деле, конечно, сортировка устроена сложнее. С деталями реализации можно познакомиться в файле `src/backend/utils/sort/tuplesort.c`.

История развития способов сортировки в PostgreSQL хорошо описана в презентации Грегори Стапка «Sorting Through The Ages»:
https://wiki.postgresql.org/images/5/59/Sorting_through_the_ages.pdf.

Внешняя сортировка

Если набор строк для сортировки не помещается целиком в оперативную память размером `work_mem`, применяется внешняя сортировка с использованием временных файлов. Вот пример такого плана (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: 17136kB
  Buffers: shared hit=3 read=2624, temp read=2142 written=2150
  -> 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=61 read=14
(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: 36360kB
  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)
```

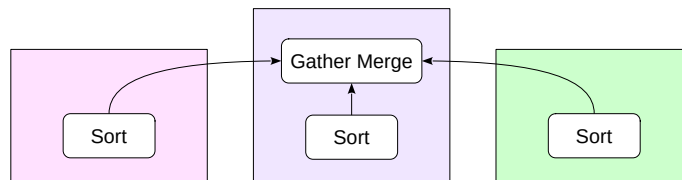
Теперь все строки поместились в память, и планировщик выбрал более дешевую быструю сортировку.

```
=> RESET work_mem;
```

RESET

Узел Gather Merge сохраняет порядок сортировки

выполняет слияние данных, поступающих от нижестоящих узлов



25

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.

Параллельная сортировка

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

QUERY PLAN

```
-----  
Finalize GroupAggregate  
  Group Key: amount  
    -> Gather Merge  
      Workers Planned: 2  
        -> Sort  
          Sort Key: amount  
            -> Partial HashAggregate  
              Group Key: amount  
                -> Parallel Seq Scan on ticket_flights  
(9 rows)
```

В этом плане выполнения рабочие процессы параллельно читают таблицу перелетов, выполняют группировку с помощью хеширования (Partial HashAggregate) и сортируют полученный результат (Sort).

Узел Gather Merge собирает данные в один отсортированный набор, который затем окончательно группируется узлом Finalize GroupAggregate.

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

Индекс (речь идет про B-дерево) можно строить, добавляя последовательно в пустой индекс по одной строке из таблицы. Но такой способ крайне неэффективен.

Поэтому при создании индекса используется сортировка: все строки таблицы сортируются и 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. Ограничение на количество процессов накладывается значением параметра *max_parallel_maintenance_workers*, хотя реальное количество может выбрано и меньше. Ограничение на память действует для всех запущенных процессов, а не для каждого из них по отдельности.

Соединение слиянием может потребовать подготовки

- надо отсортировать наборы строк
- или получить их заранее отсортированными

Эффективно для больших выборок

- хорошо, если наборы уже отсортированы
- хорошо, если нужен отсортированный результат

Не зависит от порядка соединения

Поддерживает только эквисоединения

- другие не реализованы, но принципиальных ограничений нет

Чтобы начать соединение слиянием, оба набора строк должны быть отсортированы. Хорошо, если удастся получить данные уже в нужном порядке; если нет — требуется выполнить сортировку.

Само слияние выполняется очень эффективно даже для больших наборов строк. В качестве приятного бонуса результирующая выборка тоже упорядочена, поэтому такой способ соединения привлекателен, если вышестоящим узлам плана также требуется сортировка (например, запрос с фразой ORDER BY или еще одна сортировка слиянием).

В настоящее время соединение слиянием поддерживает только эквисоединения, соединения по операциям «больше» или «меньше» не реализованы.

Итак, в распоряжении планировщика есть три способа соединения: вложенный цикл, хеширование и слияние (не считая различных модификаций). Есть ситуации, в которых каждый из способов оказывается более эффективным, чем остальные. Это позволяет планировщику выбрать именно тот способ, который — как предполагается — лучше подойдет в каждом конкретном случае. А точность предположений напрямую зависит от имеющейся статистики.

1. Создайте индекс по столбцам `passenger_name` и `passenger_id` таблицы билетов (`tickets`).
Потребовался ли временный файл для выполнения этой операции?
2. Проверьте план выполнения запроса из демонстрации, показывающего все места в салонах в порядке кодов самолетов, но оформленного в виде курсора.
Уменьшите значение параметра `cursor_tuple_fraction` в десять раз. Как при этом изменился план выполнения?

1. Включите журналирование использования временных файлов, установив значение параметра `log_temp_files` в ноль.

2. Речь идет о запросе

```
SELECT *  
FROM aircrafts a  
  JOIN seats s ON a.aircraft_code = s.aircraft_code  
ORDER BY a.aircraft_code;
```

1. Индекс

Включим журналирование временных файлов.

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

SET

```
=> \timing on
```

Timing is on.

Текущее значение maintenance_work_mem:

```
=> SHOW maintenance_work_mem;
```

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

Time: 0,739 ms

Создаем индекс:

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

CREATE INDEX

Time: 14421,124 ms (00:14,421)

Временный файл понадобился:

```
postgres$ tail -n 2 /var/log/postgresql/postgresql-13-main.log
```

```
2022-07-25 22:59:00.937 MSK [67955] postgres@demo LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp67955.0.sharedfilesset/0.0", size 64634880
2022-07-25 22:59:00.937 MSK [67955] postgres@demo STATEMENT:  CREATE INDEX ON tickets(passenger_name, passenger_id);
```

2. Параметр cursor_tuple_fraction

План выполнения курсора:

```
=> EXPLAIN DECLARE c CURSOR FOR SELECT *
FROM aircrafts a
JOIN seats s ON a.aircraft_code = s.aircraft_code
ORDER BY a.aircraft_code;
```

QUERY PLAN

```
-----
Merge Join  (cost=1.51..420.71 rows=1339 width=55)
  Merge Cond: (s.aircraft_code = ml.aircraft_code)
    -> Index Scan using seats_pkey on seats s  (cost=0.28..64.60 rows=1339 width=15)
    -> Sort  (cost=1.23..1.26 rows=9 width=72)
          Sort Key: ml.aircraft_code
          -> Seq Scan on aircrafts_data ml  (cost=0.00..1.09 rows=9 width=72)
(6 rows)
```

Time: 38,077 ms

Текущее значение cursor_tuple_fraction:

```
=> SHOW cursor_tuple_fraction;
```

```
 cursor_tuple_fraction
-----
          0.1
(1 row)
```

Time: 0,108 ms

Уменьшим его:

```
=> SET cursor_tuple_fraction = 0.01;
```

SET

Time: 0,104 ms

```
=> EXPLAIN DECLARE c CURSOR FOR SELECT *
FROM aircrafts a
JOIN seats s ON a.aircraft_code = s.aircraft_code
ORDER BY a.aircraft_code;
```

QUERY PLAN

```
-----
Merge Join  (cost=0.41..431.73 rows=1339 width=55)
  Merge Cond: (ml.aircraft_code = s.aircraft_code)
    -> Index Scan using aircrafts_pkey on aircrafts_data ml  (cost=0.14..12.27 rows=9 width=72)
    -> Index Scan using seats_pkey on seats s  (cost=0.28..64.60 rows=1339 width=15)
(4 rows)
```

Time: 0,388 ms

Теперь планировщик выбирает другой план: его начальная стоимость ниже (хотя общая стоимость, наоборот, выше).