



# > Конспект > 4 урок > Обработка запросов в обычной СУБД и в MPP СУБД: Практика

## > Оглавление

- > [Оглавление](#)
- > [Первая итерация explain и explain analyze](#)
  - [Подготовка данных](#)
  - [План выполнения запроса в Greenplum](#)
  - [План выполнения запроса в PostgreSQL](#)
- > [Вторая итерация explain и explain analyze](#)
  - [Подготовка данных](#)
  - [План запроса в Greenplum](#)
  - [Актуальность статистики](#)
  - [План запроса в PostgreSQL](#)
- > [Использование ключей распределения](#)
  - [Изменим ключ распределения](#)
- > [Использование индексов](#)
  - [Индексирование по ключу соединения](#)

[План выполнения запроса в Greenplum](#)

[План выполнения запроса в PostgreSQL](#)

> [Изменение параметров оптимизатора](#)

> [Использование внешних таблиц](#)

> [Explain с другими запросами](#)

[План запроса в Greenplum](#)

[План запроса в PostgreSQL](#)

> [Сжатие таблиц](#)

[Размер таблицы](#)

[Сжатие с построчным хранением](#)

[Сжатие с построчным хранением](#)

## > Первая итерация explain и explain analyze

### Подготовка данных

```
create table tpch1.orders_1 (like tpch1.orders) distributed randomly;  
create table tpch1.lineitem_1 (like tpch1.lineitem) distributed randomly;
```

Из [документации](#): «Предложение **LIKE** определяет таблицу, из которой в новую таблицу будут автоматически скопированы все имена столбцов, их типы данных и их ограничения на NULL»

**Distributed randomly** применяется здесь к **Greenplum** и означает, что записи будут распределены по сегментам случайно. Для PostgreSQL этого не требуется.

Вставим в таблицы по 100 записей из изначальных таблиц.

```
insert into tpch1.orders_1 select * from tpch.orders limit 100;  
insert into tpch1.lineitem_1 select * from tpch.lineitem limit 100;
```

### План выполнения запроса в Greenplum

Посмотрим на план выполнения запроса в **Greenplum**, указав перед ним оператор **explain**.

```
explain
select * from tpch1.lineitem_1 l join tpch1.orders_1 o on o.o_orderkey=l.l_orderkey;
```

	ABC QUERY PLAN
1	Gather Motion 4:1 (slice3; segments: 4) (cost=0.00..862.20 rows=101 width=226)
2	-> Hash Join (cost=0.00..862.12 rows=26 width=226)
3	Hash Cond: (lineitem_1.l_orderkey = orders_1.o_orderkey)
4	-> Redistribute Motion 4:4 (slice1; segments: 4) (cost=0.00..431.02 rows=25 width=120)
5	Hash Key: lineitem_1.l_orderkey
6	-> Seq Scan on lineitem_1 (cost=0.00..431.00 rows=25 width=120)
7	-> Hash (cost=431.02..431.02 rows=25 width=106)
8	-> Redistribute Motion 4:4 (slice2; segments: 4) (cost=0.00..431.02 rows=25 width=106)
9	Hash Key: orders_1.o_orderkey
10	-> Seq Scan on orders_1 (cost=0.00..431.00 rows=25 width=106)
11	Optimizer: Pivotal Optimizer (GPORCA)

Explain в Greenplum

1. Чтобы уменьшить операции чтения, **Greenplum** ставит в начало меньшую таблицу `orders_1`

```
-> Seq Scan on orders_1 (cost=0.00..431.00 rows=25 width=106)
```

2. Затем создает хэш-таблицу с ключами по полю `o_orderkey` для дальнейшего соединения с таблицей `lineitem_1`.

```
Hash Key: orders_1.o_orderkey
```

3. Производится перераспределение всей таблицы `orders_1` между всеми сегментами **Greenplum**, чтобы на каждом из них была полная копия таблицы

```
> Redistribute Motion 4:4 (slice2; segments: 4) (cost=0.00..431.02 rows=25 width=106)
```

4. Для таблицы `lineitem_1` производится ровно то же самое

```
> Redistribute Motion 4:4 (slice1; segments: 4) (cost=0.00..431.02 rows=25 width=120)
```

```
Hash Key: lineitem_1.l_orderkey
```

```
-> Seq Scan on lineitem_1 (cost=0.00..431.00 rows=25 width=120)
```

5. Выполняется объединение и получение результата

```
> Hash Join (cost=0.00..862.12 rows=26 width 226)
```

```
Hash Cond: (lineitem_1.l_orderkey=orders_1.o_orderkey)
```

6. Оператор **Gather Motion** собирает все результаты на мастер-сегменте и отдает их клиенту

```
Gather Motion 4:1 (slice3, segments: 4) (cost=0.00..862.20 rows=101 width=226)
```

Теперь применим оператор **explain analyze** и посмотрим сколько данных могло бы вернуться клиенту, если бы запрос выполнялся (запрос отработает фактически, но клиент не получит результат).

```
explain analyze
select * from tpch1.lineitem_1 l join tpch1.orders_1 o on o.o_orderkey=l.l_orderkey;
```

ABC QUERY PLAN
Gather Motion 4:1 (slice3; segments: 4) (cost=0.00..862.20 rows=101 width=226) (actual time=3.385..3.818 rows=28 loops=1)
-> Hash Join (cost=0.00..862.12 rows=26 width=226) (actual time=2.138..2.465 rows=11 loops=1)
Hash Cond: (lineitem_1.l_orderkey = orders_1.o_orderkey)
Extra Text: (seg2) Hash chain length 1.0 avg, 1 max, using 19 of 131072 buckets.
Extra Text: (seg3) Hash chain length 1.0 avg, 1 max, using 29 of 131072 buckets.
-> Redistribute Motion 4:4 (slice1; segments: 4) (cost=0.00..431.02 rows=25 width=120) (actual time=0.003..0.010 rows=37 loops=1)
Hash Key: lineitem_1.l_orderkey
-> Seq Scan on lineitem_1 (cost=0.00..431.00 rows=25 width=120) (actual time=0.018..0.021 rows=28 loops=1)
-> Hash (cost=431.02..431.02 rows=25 width=106) (actual time=0.058..0.058 rows=29 loops=1)
-> Redistribute Motion 4:4 (slice2; segments: 4) (cost=0.00..431.02 rows=25 width=106) (actual time=0.013..0.029 rows=29 loops=1)
Hash Key: orders_1.o_orderkey
-> Seq Scan on orders_1 (cost=0.00..431.00 rows=25 width=106) (actual time=0.022..0.025 rows=29 loops=1)
Planning time: 10.107 ms
(slice0) Executor memory: 161K bytes.
(slice1) Executor memory: 60K bytes avg x 4 workers, 60K bytes max (seg0).
(slice2) Executor memory: 60K bytes avg x 4 workers, 60K bytes max (seg0).
(slice3) Executor memory: 1128K bytes avg x 4 workers, 1128K bytes max (seg0). Work_mem: 4K bytes max.
Memory used: 128000kB
Optimizer: Pivotal Optimizer (GPORCA)
Execution time: 27.881 ms

Explain analyze в Greenplum

Здесь появляется реальная информация о результатах каждого шага: затраченное время, количество строк.

**Memory used: 120000kB** - используемая память

**Optimizer: Pivotal Optimizer (GPORCA)** - используемый оптимизатор

**Execution time: 27.881 ms** - время выполнения всего запроса

## План выполнения запроса в PostgreSQL

```

              QUERY PLAN
-----
Hash Join (cost=4.25..8.62 rows=100 width=231)
  Hash Cond: (l.l_orderkey = o.o_orderkey)
    -> Seq Scan on lineitem_1 l (cost=0.00..3.00 rows=100 width=119)
    -> Hash (cost=3.00..3.00 rows=100 width=112)
          -> Seq Scan on orders_1 o (cost=0.00..3.00 rows=100 width=112)
(5 rows)

```

Explain в PostgreSQL

Также как и в **Greenplum**, здесь обработка запроса начинается с последовательного сканирования таблицы **orders\_1**. Так как у нас всего один инстанс, то ничего не распределяется по сегментам и ожидаемое количество строк - 100.

После строится хэш-таблица по ключу соединения, сканируется **lineitem\_1** и таблицы объединяются по ключу.

В результате ожидаем получить 100 строк.

```

              QUERY PLAN
-----
Hash Join (cost=4.25..8.62 rows=100 width=231) (actual time=0.161..0.162 rows=0 loops=1)
  Hash Cond: (l.l_orderkey = o.o_orderkey)
    -> Seq Scan on lineitem_1 l (cost=0.00..3.00 rows=100 width=119) (actual time=0.014..0.024 rows=100 loops=1)
    -> Hash (cost=3.00..3.00 rows=100 width=112) (actual time=0.038..0.038 rows=100 loops=1)
          Buckets: 1024 Batches: 1 Memory Usage: 23kB
          -> Seq Scan on orders_1 o (cost=0.00..3.00 rows=100 width=112) (actual time=0.006..0.016 rows=100 loops=1)
Planning Time: 0.141 ms
Execution Time: 0.202 ms
(8 rows)

```

Explain analyze в PostgreSQL

Запрос выполнен гораздо быстрее, чем в **Greenplum**, так как нет объединений результатов сегментов, все хранится в одном месте. А еще **PostgreSQL** может кэшировать данные в памяти, что, вероятно, произошло при наполнении таблицы данными.

Количество строк в результате - 0. Такое вполне могло произойти, так как наполнение таблиц происходило по взятию 100 случайных строк из первоначальных таблиц и вовсе необязательно, что подтянутся одинаковые ключи с обеих таблиц.

## > Вторая итерация explain и explain analyze

### Подготовка данных

Удалим данные из таблиц `orders_1` и `lineitem_1` и наполним их большим количеством строк

```
delete from tpch1.orders_1;
delete from tpch1.lineitem_1;

insert into tpch1.orders_1 select * from tpch1.orders limit 100000;
insert into tpch1.lineitem_1 select * from tpch1.lineitem limit 1000000;
```

### План запроса в Greenplum

ABC QUERY PLAN
Gather Motion 4:1 (slice3; segments: 4) (cost=0.00..862.20 rows=101 width=226)
-> Hash Join (cost=0.00..862.12 rows=26 width=226)
Hash Cond: (lineitem_1.l_orderkey = orders_1.o_orderkey)
-> Redistribute Motion 4:4 (slice1; segments: 4) (cost=0.00..431.02 rows=25 width=120)
Hash Key: lineitem_1.l_orderkey
-> Seq Scan on lineitem_1 (cost=0.00..431.00 rows=25 width=120)
-> Hash (cost=431.02..431.02 rows=25 width=106)
-> Redistribute Motion 4:4 (slice2; segments: 4) (cost=0.00..431.02 rows=25 width=106)
Hash Key: orders_1.o_orderkey
-> Seq Scan on orders_1 (cost=0.00..431.00 rows=25 width=106)
Optimizer: Pivotal Optimizer (GPORCA)

Explain в Greenplum

Как видно, **Greenplum** все еще думает, что на каждом сегменте находится по 25 строк из таблиц. Но мы знаем, что это уже не так. Всему виной неактуальная статистика.

ABC QUERY PLAN	
Gather Motion 4:1 (slice3; segments: 4) (cost=0.00..862.20 rows=101 width=226) (actual time=65.755..1925.098 rows=1603316)	
-> Hash Join (cost=0.00..862.12 rows=26 width=226) (actual time=63.486..1015.128 rows=404894 loops=1)	
Hash Cond: (lineitem_1.l_orderkey = orders_1.o_orderkey)	
Extra Text: (seg0) Hash chain length 1.9 avg, 7 max, using 13096 of 131072 buckets.	
-> Redistribute Motion 4:4 (slice1; segments: 4) (cost=0.00..431.02 rows=25 width=120) (actual time=0.043..481.488 rows=)	
Hash Key: lineitem_1.l_orderkey	
-> Seq Scan on lineitem_1 (cost=0.00..431.00 rows=25 width=120) (actual time=0.146..131.360 rows=500591 loops=1)	
-> Hash (cost=431.02..431.02 rows=25 width=106) (actual time=55.887..55.887 rows=25214 loops=1)	
-> Redistribute Motion 4:4 (slice2; segments: 4) (cost=0.00..431.02 rows=25 width=106) (actual time=0.062..22.407 row=)	
Hash Key: orders_1.o_orderkey	
-> Seq Scan on orders_1 (cost=0.00..431.00 rows=25 width=106) (actual time=0.622..6.437 rows=25105 loops=1)	
Planning time: 10.363 ms	
(slice0) Executor memory: 193K bytes.	
(slice1) Executor memory: 60K bytes avg x 4 workers, 60K bytes max (seg0).	
(slice2) Executor memory: 60K bytes avg x 4 workers, 60K bytes max (seg0).	

Explain analyze в Greenplum

Запрос `explain analyze` тоже отработал достаточно быстро, благодаря кэшированию данных со стороны операционной системы.

Основной порядок действий сохранился с предыдущего шага, увеличилось лишь количество строк и время обработки.

## Актуальность статистики

Для того, чтобы оптимизатор понимал, с чем ему придется работать, важно поддерживать актуальную статистику. Выполним следующий код:

```
analyze tpch1.orders_1;
analyze tpch1.lineitem_1;
```

Выполним `explain` снова:



ABC QUERY PLAN
Gather Motion 4:1 (slice3; segments: 4) (cost=0.00..3254.06 rows=1966337 width=224)
-> Hash Join (cost=0.00..1780.72 rows=491585 width=224)
Hash Cond: (lineitem_1.l_orderkey = orders_1.o_orderkey)
-> Redistribute Motion 4:4 (slice1; segments: 4) (cost=0.00..763.34 rows=500000 width=117)
Hash Key: lineitem_1.l_orderkey
-> Seq Scan on lineitem_1 (cost=0.00..471.43 rows=500000 width=117)
-> Hash (cost=446.23..446.23 rows=25000 width=107)
-> Redistribute Motion 4:4 (slice2; segments: 4) (cost=0.00..446.23 rows=25000 width=107)
Hash Key: orders_1.o_orderkey
-> Seq Scan on orders_1 (cost=0.00..432.88 rows=25000 width=107)
Optimizer: Pivotal Optimizer (GPORCA)

Explain в Greenplum после обновления статистики

Получаем расчет, основанный на актуальных данных с более правильными параметрами и более высоким cost.

## План запроса в PostgreSQL

```

                                QUERY PLAN
-----
Hash Join  (cost=5689.00..79967.00 rows=456700 width=230)
  Hash Cond: (l.l_orderkey = o.o_orderkey)
    -> Seq Scan on lineitem_1 l  (cost=0.00..29142.00 rows=1000000 width=120)
    -> Hash  (cost=2778.00..2778.00 rows=100000 width=110)
      -> Seq Scan on orders_1 o  (cost=0.00..2778.00 rows=100000 width=110)
(5 rows)

```

Explain в PostgreSQL

PostgreSQL самостоятельно обновил статистику и знает, с чем ему придется работать.

```

                                QUERY PLAN
-----
Hash Join  (cost=5689.00..79967.00 rows=456700 width=230) (actual time=987.374..987.380 rows=0 loops=1)
  Hash Cond: (l.l_orderkey = o.o_orderkey)
    -> Seq Scan on lineitem_1 l  (cost=0.00..29142.00 rows=1000000 width=120) (actual time=0.010..0.84741 rows=1000000 loops=1)
    -> Hash  (cost=2778.00..2778.00 rows=100000 width=110) (actual time=78.360..78.362 rows=100000 loops=1)
      Buckets: 32768  Batches: 4  Memory Usage: 3764kB
      -> Seq Scan on orders_1 o  (cost=0.00..2778.00 rows=100000 width=110) (actual time=0.004..0.8369 rows=100000 loops=1)
Planning Time: 0.137 ms
Execution Time: 987.962 ms
(8 rows)

```

Explain analyze в PostgreSQL



Также, как и с меньшими версиями таблиц, здесь ничего не удалось объединить. Результат вывода - 0 строк.

## > Использование ключей распределения

### Изменим ключ распределения

Следующий код перераспределит наши таблицы по ключу соединения:

```
alter table tpch1.lineitem_1 set with (reorganize=True) distributed by (l_orderkey);
alter table tpch1.orders_1 set with (reorganize=True) distributed by (o_orderkey);
```

#### QUERY PLAN

```
Gather Motion 4:1 (slice1; segments: 4) (cost=0.00..3062.58 rows=1966337 width=224)
-> Hash Join (cost=0.00..1589.25 rows=491585 width=224)
    Hash Cond: (lineitem_1.l_orderkey = orders_1.o_orderkey)
    -> Seq Scan on lineitem_1 (cost=0.00..471.43 rows=500000 width=117)
    -> Hash (cost=432.88..432.88 rows=25000 width=107)
        -> Seq Scan on orders_1 (cost=0.00..432.88 rows=25000 width=107)
Optimizer: Pivotal Optimizer (GPORCA)
(7 rows)
```

Explain в Greenplum

Сразу замечаем, что план уменьшился в размере. Нет шага с **Redistribute Motion**, потому что все объединение происходит непосредственно на сегментах. Это уменьшает расчетное время выполнения.

#### QUERY PLAN

```
Gather Motion 4:1 (slice1; segments: 4) (cost=0.00..3062.58 rows=1966337 width=224) (actual time=17.514..1283.314 rows=1603316 loops=1)
-> Hash Join (cost=0.00..1589.25 rows=491585 width=224) (actual time=16.874..694.860 rows=404894 loops=1)
    Hash Cond: (lineitem_1.l_orderkey = orders_1.o_orderkey)
    Extra Text: (seg0) Hash chain length 1.9 avg, 7 max, using 130% of 131072 buckets.
    -> Seq Scan on lineitem_1 (cost=0.00..471.43 rows=500000 width=117) (actual time=0.095..185.164 rows=501797 loops=1)
    -> Hash (cost=432.88..432.88 rows=25000 width=107) (actual time=15.971..15.971 rows=25214 loops=1)
        -> Seq Scan on orders_1 (cost=0.00..432.88 rows=25000 width=107) (actual time=0.035..3.449 rows=25214 loops=1)
Planning time: 17.232 ms
(slice0) Executor memory: 157K bytes.
(slice1) Executor memory: 9362K bytes avg x 4 workers, 9362K bytes max (seg0). Work_mem: 3497K bytes max.
Memory used: 128000kB
Optimizer: Pivotal Optimizer (GPORCA)
Execution time: 1417.869 ms
(13 rows)
```

Explain analyze в Greenplum

Благодаря тому, что мы распределили таблицы по ключу джойна, фактическое время объединения у нас также сократилось до 1,5 секунд.

## > Использование индексов

### Индексирование по ключу соединения

Построим индекс по ключу соединения с помощью следующего кода:

```
create index li_ok on tpch1.lineitem_1 (l_orderkey);
create index or_ok on tpch1.orders_1 (o_orderkey);
```

### План выполнения запроса в Greenplum

Проверим, будет ли теперь оптимизатор использовать индексы в построении плана.

```
QUERY PLAN
-----
Gather Motion 4:1 (slice1; segments: 4) (cost=0.00..3062.58 rows=1966337 width=224)
-> Hash Join (cost=0.00..1589.25 rows=491585 width=224)
    Hash Cond: (lineitem_1.l_orderkey = orders_1.o_orderkey)
    -> Seq Scan on lineitem_1 (cost=0.00..471.43 rows=500000 width=117)
    -> Hash (cost=432.88..432.88 rows=25000 width=107)
        -> Seq Scan on orders_1 (cost=0.00..432.88 rows=25000 width=107)
Optimizer: Pivotal Optimizer (GPORCA)
(7 rows)
```

Explain в Greenplum

Ожидаемо, оптимизатор решил не пользоваться индексами, поскольку **Greenplum** рассчитывает, что на каждом сегменте находится лишь небольшая порция данных и быстрее будет ее считать напрямую в память и работать уже с ней, чем сначала читать файл с индексом, а затем по ссылкам из файла индексов читать данные из таблицы. В 90% случаев читать придется все и выигрыша никакого не получится. Поэтому **Greenplum** идет сразу в файл с данными.

### План выполнения запроса в PostgreSQL

```
QUERY PLAN
-----
Merge Join (cost=0.72..46599.72 rows=456700 width=230)
  Merge Cond: (o.o_orderkey = l.l_orderkey)
  -> Index Scan using or_ok on orders_1 o (cost=0.29..3554.29 rows=100000 width=110)
  -> Index Scan using li_ok on lineitem_1 l (cost=0.42..35728.43 rows=1000000 width=120)
(4 rows)
```

Explain в PostgreSQL

А вот PostgreSQL уже решает использовать индексы при выполнении запроса. Таким образом, он понимает, какие строки можно сразу объединять, не сканируя весь файл данных

## > Изменение параметров оптимизатора

Так как Greenplum в данный момент используют оптимизатор Pivotal Optimizer, который не воспринимает подсказки, отключим его.

```
set optimizer off;
```

Посмотрим на стоимость разных типов чтения страниц:

```
show seq_page_cost;  
show random_page_cost;
```

Выдача означает, что для оптимизатора дешевле будет использовать последовательное чтение страниц.

Попробуем перевернуть ситуацию и установить для выборочного чтения стоимость = 1 и для последовательного = 100.

```
set seq_page_cost=100;  
set random_page_cost=1;
```

Посмотрим теперь на план запроса:

```
QUERY PLAN  
-----  
Gather Motion 4:1 (slice1; segments: 4) (cost=5318.31..144577.28 rows=1966205 width=224)  
-> Hash Join (cost=5318.31..144577.28 rows=491552 width=224)  
    Hash Cond: (l.l_orderkey = o.o_orderkey)  
    -> Index Scan using li_ok on lineitem_1 l (cost=0.18..97930.43 rows=500000 width=117)  
    -> Hash (cost=4068.14..4068.14 rows=25000 width=107)  
        -> Index Scan using or_ok on orders_1 o (cost=0.17..4068.14 rows=25000 width=107)  
Optimizer: Postgres query optimizer  
(7 rows)
```

Explain в Greenplum с измененными параметрами оптимизатора

Видим, что оптимизатор начал использовать индексы для чтения данных.

Теперь вернём значения обратно и убедимся, что оптимизатор снова стал использовать последовательное чтение.

```
QUERY PLAN
-----
Gather Motion 4:1 (slice1; segments: 4) (cost=2685.00..73309.71 rows=1966205 width=224)
-> Hash Join (cost=2685.00..73309.71 rows=491552 width=224)
    Hash Cond: (l.l_orderkey = o.o_orderkey)
    -> Seq Scan on lineitem_1 l (cost=0.00..29296.00 rows=500000 width=117)
    -> Hash (cost=1435.00..1435.00 rows=25000 width=107)
        -> Seq Scan on orders_1 o (cost=0.00..1435.00 rows=25000 width=107)
Optimizer: Postgres query optimizer
(7 rows)
```

Explain в Greenplum с параметрами по умолчанию

## > Использование внешних таблиц

В **Greenplum** можно использовать внешние таблицы для работы. Давайте попробуем сделать объединение по таким таблицам.

`tpch1.orders_ext` и `tpch1.lineitem_ext` представляют собой обращение к внешней программе, которая создаёт данные и представляет их в виде таблицы. В данном случае, оптимизатор не знает статистики этих таблиц и все значения выдаёт навскидку.

```
QUERY PLAN
-----
Gather Motion 4:1 (slice3; segments: 4) (cost=55219.00..135125342.00 rows=1000000000 width=740)
-> Hash Join (cost=55219.00..135125342.00 rows=250000000 width=740)
    Hash Cond: (l.l_orderkey = o.o_orderkey)
    -> Redistribute Motion 4:4 (slice1; segments: 4) (cost=0.00..31000.00 rows=250000 width=382)
        Hash Key: l.l_orderkey
        -> External Scan on lineitem_ext l (cost=0.00..11000.00 rows=250000 width=382)
    -> Hash (cost=31000.00..31000.00 rows=250000 width=358)
        -> Redistribute Motion 4:4 (slice2; segments: 4) (cost=0.00..31000.00 rows=250000 width=358)
            Hash Key: o.o_orderkey
            -> External Scan on orders_ext o (cost=0.00..11000.00 rows=250000 width=358)
Optimizer: Postgres query optimizer
(11 rows)
```

Explain при джойне внешних таблиц

## > Explain с другими запросами

Посмотрим, как будет вести себя оптимизатор, когда получит более развернутый запрос, где будут встречаться агрегирующие функции, расчеты и сортировка.

```
select
    l_returnflag,
```

```

    l_linestatus,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_cty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    tpch1.lineitem_1
where
    l_shipdate <= date '1998-12-01' - interval '100 days'
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus;

```

## План запроса в Greenplum

```

-----
QUERY PLAN
-----
Gather Motion 4:1 (slice2; segments: 4) (cost=112880.02..112880.04 rows=6 width=248)
  Merge Key: lineitem_1.l_returnflag, lineitem_1.l_linestatus
    -> Sort (cost=112880.02..112880.04 rows=2 width=248)
      Sort Key: lineitem_1.l_returnflag, lineitem_1.l_linestatus
      -> HashAggregate (cost=112879.78..112879.94 rows=2 width=248)
        Group Key: lineitem_1.l_returnflag, lineitem_1.l_linestatus
        -> Redistribute Motion 4:4 (slice1; segments: 4) (cost=112879.36..112879.48 rows=2 width=248)
          Hash Key: lineitem_1.l_returnflag, lineitem_1.l_linestatus
          -> HashAggregate (cost=112879.36..112879.36 rows=2 width=248)
            Group Key: lineitem_1.l_returnflag, lineitem_1.l_linestatus
            -> Seq Scan on lineitem_1 (cost=0.00..34296.00 rows=491146 width=25)
              Filter: (l_shipdate <= '1998-08-23 00:00:00'::timestamp without time zone)
Optimizer: Postgres query optimizer
(13 rows)

```

Explain в Greenplum

Несмотря на то, что таблица распределена по полю `l_orderkey` и это удобно при джойне с использованием данного ключа, здесь все равно есть шаг с **Redistribute Motion**

## План запроса в PostgreSQL

```

QUERY PLAN
-----
Finalize GroupAggregate (cost=41725.04..41727.11 rows=6 width=236)
  Group Key: l_returnflag, l_linestatus
  -> Gather Merge (cost=41725.04..41726.44 rows=12 width=236)
      Workers Planned: 2
      -> Sort (cost=40725.02..40725.03 rows=6 width=236)
          Sort Key: l_returnflag, l_linestatus
          -> Partial HashAggregate (cost=40724.77..40724.94 rows=6 width=236)
              Group Key: l_returnflag, l_linestatus
              -> Parallel Seq Scan on lineitem_1 (cost=0.00..24350.33 rows=409361 width=25)
                  Filter: (l_shipdate <= '1998-08-23 00:00:00'::timestamp without time zone)
(10 rows)

```

Explain в PostgreSQL

В PostgreSQL план очень похож на тот, что выдает оптимизатор Greenplum, но здесь уже нет Redistribute Motion и Gather Motion.

## > Сжатие таблиц

### Размер таблицы

Посмотрим на размер таблицы `tpch1.lineitem`

```
select pg_size_pretty(pg_total_relation_size('tpch1.lineitem'));
```

Применив запрос выше, получаем размер таблицы 34 GB.

Greenplum умеет сжимать таблицы. У него есть разные алгоритмы и уровни сжатия.

### Сжатие с построчным хранением

Создадим таблицу `tpch1.lineitem_compressed`, которая будет сжиматься алгоритмом сжатия `zstd` с уровнем сжатия 7. В ней обычное построчное хранение, но блоки данных сжаты. Также посмотрим на размер этой таблицы.

```

create table tpch1.lineitem_compressed (like tpch1.lineitem) with
  (appendoptimized=true, compressedtype=zstd, compresslevel=7, orientation=row)
distributed randomly;

insert into tpch1.lineitem_compressed select * from tpch1.lineitem;

select pg_size_pretty(pg_total_relation_size('tpch1.lineitem_compressed'));

```

`tpch1.lineitem_compressed` получилась почти в 3,5 раза меньше, и занимает всего 11 GB.

При обращении к такой таблице, у нас будет меньше операций ввода-вывода, а они самые дорогие для Greenplum. Мы быстро читаем блоки данных, поднимаем их в память, процессор их разжимает и может использовать. Для такой таблицы может быть выгодно использовать индексы, потому что в случае дискретной выборки это позволит поднимать меньше блоков данных и меньше разжимать.

## Сжатие с поколоночным хранением

Создадим еще одну таблицу `tpch1.lineitem_compressed_columnar`, которая будет сжиматься тем же алгоритмом сжатия `zstd` с уровнем сжатия 7. Это будет сжатая таблица с поколоночным хранением. Каждая из колонок на диске представляет собой отдельный файл, который также сжимается. Посмотрим сколько места будет занимать такая таблица.

```
create table tpch1.lineitem_compressed_columnar (like tpch1.lineitem) with
  (appendoptimized=true, compressedtype=zstd, compresslevel=7, orientation=column)
distributed randomly;

insert into tpch1.lineitem_compressed_columnar select * from tpch1.lineitem;

select pg_size_pretty(pg_total_relation_size('tpch1.lineitem_compressed_columnar'));
```

Она занимает всего **7,5 GB**, что почти в 5 раз меньше, чем исходная несжатая таблица.