Big Data Infrastructures
Fall 2018

---

Lab 02 : Greenplum lab

Author : Thomas Schaller, Sylvain Julmy

---

Professor : Philippe Cudré-Mauroux

---

## Table creation

We create additional tables for the exercises as the following :

```
1  CREATE TABLE nation2 WITH (appendonly=true, orientation=row) AS select * from nation;
2
3  CREATE TABLE customer2 WITH (appendonly=true, orientation=row) AS select * from customer;
4
5  CREATE TABLE orders2 WITH (appendonly=true, orientation=row) AS select * from orders;
6
7  CREATE TABLE supplier2 WITH (appendonly=true, orientation=row) AS select * from supplier;
```

## Better column queries

We use the two following query in order to demonstrate that column orientation has benefits and drawbacks.

```
1  SELECT c_name, o_orderstatus, count(*) FROM customer
2      INNER JOIN orders on c_custkey = o_custkey
3      WHERE o_totalprice > 100000.0
4      GROUP BY o_custkey, o_orderstatus, c_name;
5
6  -- the same query which use row orientated table
7  SELECT c_name, o_orderstatus, count(*) FROM customer2
8      INNER JOIN orders2 on c_custkey = o_custkey
9      WHERE o_totalprice > 100000.0
10     GROUP BY o_custkey, o_orderstatus, c_name;
```

Listing 1: This query work better for column oriented tables

### Time measurement

```
tpch=# select c_name, o_orderstatus, count(*) from customer
tpch-#     inner join orders on c_custkey = o_custkey
tpch-#     where o_totalprice > 100000.0
tpch-#     group by o_custkey, o_orderstatus, c_name
tpch-#     ;
Time: 5668.110 ms

tpch=# select c_name, o_orderstatus, count(*) from customer2
tpch-#     inner join orders2 on c_custkey = o_custkey
tpch-#     where o_totalprice > 100000.0
tpch-#     group by o_custkey, o_orderstatus, c_name
tpch-#     ;
Time: 15719.907 ms
```

Listing 2: Time measurement for column oriented tables.

### Argumentation

The query is faster using a column representation since the result obtained juste before the `Group By` don't need to be sorted, because it is already done at the column creation. When we use a row representation, a sort is performed before a group by. This sort is needed, before the row oriented table is not already sorted. Since we only need one or two field among many from an entry, many jump are performed in a row oriented table to retrieve all the needed information.

In addition, if we run the query using the `EXPLAIN ANALYZE` prefix, we can see, from appendix E (line 12) that the data are externaly sorted on the disk, which is way slower than in memory.

Figure 1 show the execution plan for the query using the column oriented table while figure 2 show the execution plan for the query using the row oriented table.

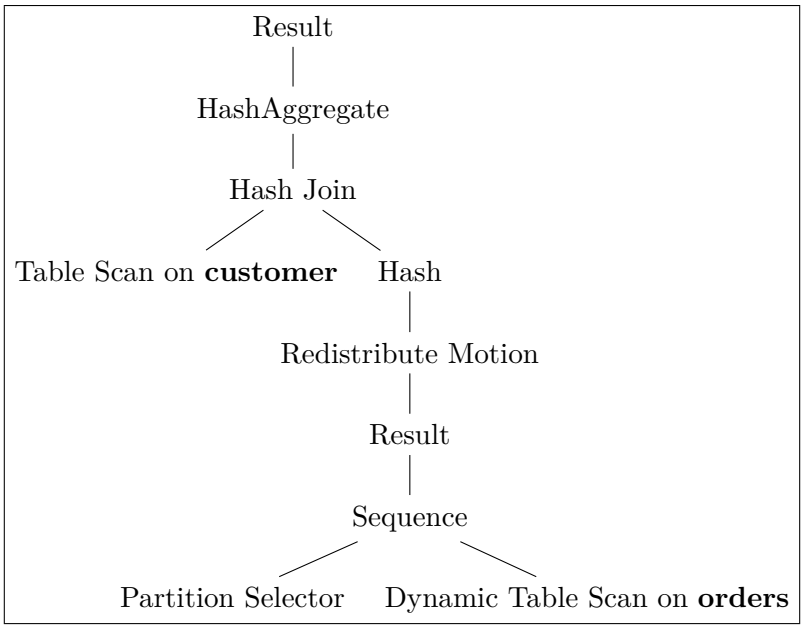


Figure 1: Execution plan in a tree representation from appendix A

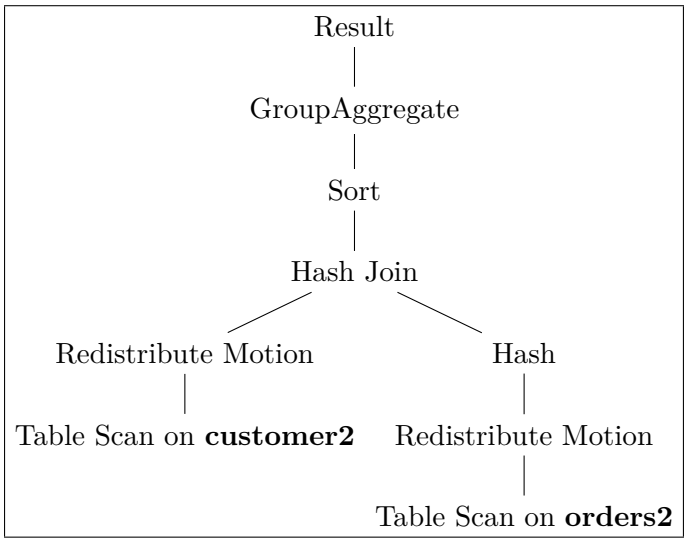


Figure 2: Execution plan in a tree representation from appendix B

## Better row queries

We use the two following query in order to demonstrate that row orientation has benefits and drawbacks as well.

```
SELECT customer2.*, supplier2.*, nation2.* FROM supplier2
    INNER JOIN customer2 on s_nationkey = c_nationkey
    INNER JOIN nation2 on n_nationkey = c_nationkey
    INNER JOIN orders2 on o_custkey = c_custkey
    WHERE o_totalprice > 100000.0
    LIMIT 600000;

SELECT customer.*, supplier.*, nation.* FROM supplier
    INNER JOIN customer on s_nationkey = c_nationkey
    INNER JOIN nation on n_nationkey = c_nationkey
    INNER JOIN orders on o_custkey = c_custkey
    WHERE o_totalprice > 100000.0
    LIMIT 600000;
```

Listing 3: This query work better for row orientated table

### Time measurement

```
tpch=# select customer2.*, supplier2.* from customer2
tpch#     inner join supplier2 on s_nationkey = c_nationkey
tpch#     limit 400000;
Time: 800.947 ms

tpch=# select customer.*, supplier.* from customer
tpch#     inner join supplier on s_nationkey = c_nationkey
tpch#     limit 400000;
Time: 1606.780 ms
```

Listing 4: Time measurement for row oriented tables

### Argumentation

To realize a local join on the segment instance, matching rows must be located together on the same segment instance. In this case, query number 3, where the tables are oriented in columns, a dynamic redistribution of the needed rows from one of the tables to another segment instance must be performed and it is thanks to the Redistribute Motion. So we have a table scan on supplier and then the rows are reditributed to another segment to perform the hash join with the table customer.

After that, the limit is done and all the resulting rows are send to the master host which will do also a limit (don't know why?). For the query number 4, we just have to do a table scan on supplier2 and send to data to the master host. Same for the table customer2. Then the master host can realize the hashjoin between these two tables and finally do the limit to show only the first 400'000 rows. Compared to the query number 3, it is of course more expensive in time since we have to do two redistribute motion because of the orientation in column.

Both query execution plan are available in appendix C (query optimized for the row orientation) and appendix D (query not-optimized for the row orientation).

# Appendix A

# Query Plan for a column optimized query using a column oriented table

```
                                                    QUERY PLAN
------------------------------------------------------------------------------
 Gather Motion 2:1  (slice2; segments: 2)  (cost=0.00..4003.53 rows=4979305 width=29)
   ->  Result  (cost=0.00..3355.17 rows=2489653 width=29)
         ->  HashAggregate  (cost=0.00..3355.17 rows=2489653 width=29)
               Group By: orders.o_custkey, orders.o_orderstatus, customer.c_name
               ->  Hash Join  (cost=0.00..2394.13 rows=2489653 width=25)
                     Hash Cond: customer.c_custkey = orders.o_custkey
                     ->  Table Scan on customer  (cost=0.00..467.09 rows=375000 width=23)
                     ->  Hash  (cost=1034.67..1034.67 rows=2489653 width=6)
                           ->  Redistribute Motion 2:2  (slice1; segments: 2)  (cost=0.00..1034.67
                           ↪  rows=2489653 width=6)
                                 Hash Key: orders.o_custkey
                                 ->  Result  (cost=0.00..987.91 rows=2489653 width=6)
                                       ->  Sequence  (cost=0.00..987.91 rows=2489653 width=16)
                                             ->  Partition Selector for orders (dynamic scan id: 1)
                                             ↪  (cost=10.00..100.00 rows=50 width=4)
                                                   Partitions selected: 87 (out of 87)
                                             ->  Dynamic Table Scan on orders (dynamic scan id: 1)
                                             ↪  (cost=0.00..987.91 rows=2489653 width=16)
                                                   Filter: o_totalprice > 100000.0
 Optimizer status: PQO version 2.56.3
(17 rows)
```

# Appendix B

# Query Plan for a column optimized query using a row oriented table

```
                                                    QUERY PLAN
------------------------------------------------------------------------------
 Gather Motion 2:1  (slice3; segments: 2)  (cost=0.00..862.00 rows=1 width=24)
   ->  Result  (cost=0.00..862.00 rows=1 width=24)
         ->  GroupAggregate  (cost=0.00..862.00 rows=1 width=24)
               Group By: orders2.o_custkey, orders2.o_orderstatus, customer2.c_name
               ->  Sort  (cost=0.00..862.00 rows=1 width=20)
                     Sort Key: orders2.o_custkey, orders2.o_orderstatus, customer2.c_name
                     ->  Hash Join  (cost=0.00..862.00 rows=1 width=20)
                           Hash Cond: customer2.c_custkey = orders2.o_custkey
                           ->  Redistribute Motion 2:2  (slice1; segments: 2)  (cost=0.00..431.00
                           ↪  rows=1 width=12)
                                 Hash Key: customer2.c_custkey
                                 ->  Table Scan on customer2  (cost=0.00..431.00 rows=1 width=12)
                           ->  Hash  (cost=431.00..431.00 rows=1 width=12)
                                 ->  Redistribute Motion 2:2  (slice2; segments: 2)  (cost=0.00..431.00
                                 ↪  rows=1 width=12)
                                       Hash Key: orders2.o_custkey
                                       ->  Table Scan on orders2  (cost=0.00..431.00 rows=1 width=12)
                                             Filter: o_totalprice > 100000.0
 Optimizer status: PQO version 2.56.3
(17 rows)
```

# Appendix C

# Query Plan for a row optimized query using a row oriented table

```
                                                    QUERY PLAN
------------------------------------------------------------------------------
 Limit  (cost=0.00..862.01 rows=1 width=420)
   ->  Hash Join  (cost=0.00..862.01 rows=1 width=420)
         Hash Cond: customer2.c_nationkey = supplier2.s_nationkey
         ->  Gather Motion 2:1  (slice1; segments: 2)  (cost=0.00..431.00 rows=1 width=223)
               ->  Table Scan on customer2  (cost=0.00..431.00 rows=1 width=223)
         ->  Hash  (cost=431.00..431.00 rows=1 width=197)
               ->  Gather Motion 2:1  (slice2; segments: 2)  (cost=0.00..431.00 rows=1 width=197)
                     ->  Table Scan on supplier2  (cost=0.00..431.00 rows=1 width=197)
 Optimizer status: PQO version 2.56.3
(9 rows)
```

# Appendix D

# Query Plan for a row optimized query using a column oriented table

```
                                                    QUERY PLAN
------------------------------------------------------------------------------
 Limit  (cost=0.00..808017.65 rows=200000 width=307)
   ->  Gather Motion 2:1  (slice3; segments: 2)  (cost=0.00..807894.85 rows=400000 width=307)
         ->  Limit  (cost=0.00..807343.48 rows=200000 width=307)
               ->  Hash Join  (cost=0.00..807282.08 rows=750000000 width=307)
                     Hash Cond: customer.c_nationkey = supplier.s_nationkey
                     ->  Redistribute Motion 2:2  (slice1; segments: 2)  (cost=0.00..768.37 rows=375000
                     ↪  width=161)
                           Hash Key: customer.c_nationkey
                           ->  Table Scan on customer  (cost=0.00..467.09 rows=375000 width=161)
                     ->  Hash  (cost=451.41..451.41 rows=25000 width=146)
                           ->  Redistribute Motion 2:2  (slice2; segments: 2)  (cost=0.00..451.41
                           ↪  rows=25000 width=146)
                                 Hash Key: supplier.s_nationkey
                                 ->  Table Scan on supplier  (cost=0.00..433.20 rows=25000 width=146)
 Optimizer status: PQO version 2.56.3
(13 rows)
```

# Appendix E

# Query analyze for a column optimized query using a column oriented table

```
                                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Gather Motion 2:1  (slice3; segments: 2)  (cost=0.00..862.00 rows=1 width=24)
   Rows out:  1114907 rows at destination with 11632 ms to first row, 14786 ms to end.
   ->  Result  (cost=0.00..862.00 rows=1 width=24)
         Rows out:  Avg 557453.5 rows x 2 workers.  Max 557536 rows (seg1) with 11693 ms to first row,
         ↪  14503 ms to end.
         ->  GroupAggregate  (cost=0.00..862.00 rows=1 width=24)
               Group By: orders2.o_custkey, orders2.o_orderstatus, customer2.c_name
               Rows out:  Avg 557453.5 rows x 2 workers.  Max 557536 rows (seg1) with 11693 ms to first
               ↪  row, 14433 ms to end.
               ->  Sort  (cost=0.00..862.00 rows=1 width=20)
                     Sort Key: orders2.o_custkey, orders2.o_orderstatus, customer2.c_name
                     Sort Method:  external merge  Max Disk: 97696KB  Avg Disk: 97696KB (2 segments)
                     Rows out:  Avg 2498144.0 rows x 2 workers.  Max 2498435 rows (seg1) with 11693 ms
                     ↪  to first row, 13818 ms to end.
                     Executor memory:  65740K bytes avg, 65740K bytes max (seg0).
                     Work_mem used:  65740K bytes avg, 65740K bytes max (seg0). Workfile: (2 spilling)
                     Work_mem wanted: 459139K bytes avg, 459192K bytes max (seg1) to lessen workfile
                     ↪  I/O affecting 2 workers.
                     ->  Hash Join  (cost=0.00..862.00 rows=1 width=20)
                           Hash Cond: customer2.c_custkey = orders2.o_custkey
                           Rows out:  Avg 2498144.0 rows x 2 workers.  Max 2498435 rows (seg1) with
                           ↪  4748 ms to first row, 6502 ms to end.
                           Executor memory:  63851K bytes avg, 63851K bytes max (seg0).
                           Work_mem used:  63851K bytes avg, 63851K bytes max (seg0). Workfile: (2
                           ↪  spilling)
                           Work_mem wanted: 78067K bytes avg, 78077K bytes max (seg1) to lessen
                           ↪  workfile I/O affecting 2 workers.
                           (seg1)   Initial batch 0:
                           (seg1)     Wrote 24288K bytes to inner workfile.
                           (seg1)     Wrote 6560K bytes to outer workfile.
                           (seg1)   Overflow batch 1:
                           (seg1)     Read 24286K bytes from inner workfile.
                           (seg1)     Read 6568K bytes from outer workfile.
                           (seg1)   Hash chain length 12.6 avg, 73 max, using 198785 of 524288 buckets.
                           ->  Redistribute Motion 2:2  (slice1; segments: 2)  (cost=0.00..431.00
                           ↪  rows=1 width=12)
                                 Hash Key: customer2.c_custkey
                                 Rows out:  Avg 375000.0 rows x 2 workers at destination.  Max 375000
                                 ↪  rows (seg0) with 0.024 ms to first row, 483 ms to end.
                                 ->  Table Scan on customer2  (cost=0.00..431.00 rows=1 width=12)
                                       Rows out:  Avg 375000.0 rows x 2 workers.  Max 375000 rows
                                       ↪  (seg0) with 72 ms to first row, 536 ms to end.
```

```
                           ->  Hash  (cost=431.00..431.00 rows=1 width=12)
                                 Rows in:  Avg 1251804.5 rows x 2 workers.  Max 1254998 rows (seg1)
                                 ↪  with 4743 ms to end, start offset by 87 ms.
                                 ->  Redistribute Motion 2:2  (slice2; segments: 2)  (cost=0.00..431.00
                                 ↪  rows=1 width=12)
                                       Hash Key: orders2.o_custkey
                                       Rows out:  Avg 2498144.0 rows x 2 workers at destination.  Max
                                       ↪  2498435 rows (seg1) with 22 ms to first row, 3524 ms to end.
                                       ->  Table Scan on orders2  (cost=0.00..431.00 rows=1 width=12)
                                             Filter: o_totalprice > 100000.0
                                             Rows out:  Avg 2498144.0 rows x 2 workers.  Max 2498569
                                             ↪  rows (seg0) with 10 ms to first row, 3477 ms to end.
 Slice statistics:
   (slice0)    Executor memory: 386K bytes.
   (slice1)    Executor memory: 333K bytes avg x 2 workers, 333K bytes max (seg0).
   (slice2)    Executor memory: 373K bytes avg x 2 workers, 373K bytes max (seg0).
   (slice3)  * Executor memory: 168427K bytes avg x 2 workers, 168427K bytes max (seg0).  Work_mem:
   ↪  65740K bytes max, 459192K bytes wanted.
 Statement statistics:
   Memory used: 128000K bytes
   Memory wanted: 919182K bytes
 Optimizer status: PQO version 2.56.3
 Total runtime: 14913.732 ms
(50 rows)
```