GreenPlum简单性能测试与分析

如今，多样的交易模式以及大众消费观念的改变使得数据库应用领域不断扩大，现代的大型分布式应用系统的数据膨胀也对数据库的海量数据处理能力和并行处理能力提出了更高的要求，如何在数据呈现海量扩张的同时提高处理速度和应用系统的可用性，使客户能同时得到更高的处理速度、更高的数据可用性和更大的数据集，是数据库系统面临的一个挑战。
通过TPC-H基准测试，可获得数据库单位时间内的性能处理能力，为评估数据库系统的现有性能服务水平提供有效依据，通过横向对比促进数据库系统的整体质量提升，能更好地在重大信息化工程中实现推广。

一.TPC-H原理简介

TPC-H是由TPC(Transaction Processing Performance Council)事务处理性能委员会公布的一套针对数据库决策支持能力的测试基准，通过模拟数据库中与业务相关的复杂查询和并行的数据修改操作考察数据库的综合处理能力，获取数据库操作的响应时间和每小时执行的查询数指标(QphH@Size)。
TPC-H基准模型中定义了一个数据库模型，容量可以在1GB~10000GB的8个级别中进行选择。数据库模型包括CUSTOMER、LINEITEM、NATION、ORDERS、PART、PARTSUPP 、REGION和SUPPLIER 8张数据表，涉及22条复杂的select查询流语句和2条带有insert和delete程序段的更新流语句。

二.目的

1.比较在同等资源条件下具有分布式属性的GreenPlum与单机版mysql在进行TPC-H类测试的性能区别。
2.分析两种DB造成性能区别的原因。

三.测试环境与配置信息

测试环境：腾讯云
测试对象：GreenPlum、Mysql，两者的配置信息统计如下：

表 1　GreenPlum 集群服务器

Host / 指标	Master Host	Segment Host1	Segment Host2
操作系统	CentOS 6.7 64 位	CentOS 6.7 64 位	CentOS 6.7 64 位
cpu	Intel(R) Xeon(R) CPU E5-26xx v3　2 核	Intel(R) Xeon(R) CPU E5-26xx v3　2 核	Intel(R) Xeon(R) CPU E5-26xx v3　2 核
内存	8GB	8GB	8GB
公网带宽	100Mbps	100Mbps	100Mbps
IP	123.207.228.40	123.207.228.21	123.207.85.105
Segment 数量	0	2	2
版本	greenplum-db-4.3.8.1-build-1-RHEL5-x86_64		

指标	参数
文本1	文本2
操作系统	CentOS 6.7 64位
cpu	Intel(R) Xeon(R) CPU E5-26xx v3 8核
内存	24GB
公网带宽	100Mbps
IP	123.207.228.51
版本	MySql5.6

表2 Mysql服务器

四.测试数据量统计

表名称	数据条数
customer	150000
lineitem	6001215
nation	25
orders	1500000
part	200000
partsupp	800000
region	5
supplier	10000

表3 各测试表数据量统计

五.执行时间统计

执行的sql	GeenPlum执行时间（单位：秒）	Mysql执行时间（单位：秒）
Q1	4.01	12.66
Q2	0.50	3.27
Q3	1.35	5.06
Q4	0.11	0.01
Q5	0.19	27.29
Q6	0.01	2.50
Q7	6.06	10.79
Q8	1.46	39.78
Q9	4.00	>12小时
Q10	0.14	4.74
Q11	0.30	7.90
Q12	0.08	2.35
Q13	1.04	>12小时
Q14	0.04	9.37

执行的sql	GreenPlum执行时间（单位：秒）	Mysql执行时间（单位：秒）
Q15	0.07	4.76
Q16	0.51	2.90
Q17	3.21	48697.95
Q18	14.23	>12小时
Q19	0.95	23.12
Q20	0.16	>12小时
Q21	7.23	>12小时
Q22	0.96	8540.22

表4 22条sql执行时间统计

六.性能对比分析

根据执行时间的统计，我们可以看出两种数据库在进行TPC-H类测试有着较大差异，下面我们将选取两个典型的事例SQL，分析GreenPlum与Mysql在执行该类SQL的性能差异原因。

示例一

我们选取Q3，从执行时间统计可以看出GreenPlum的执行速度大概是Mysql的4倍左右。首先，查看下Q3语句，如下图1所示。

```
select
        l_orderkey,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        o_orderdate,
        o_shippriority
from
        customer,
        orders,
        lineitem
where
        c_mktsegment = 'HOUSEHOLD'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate < date '1995-03-03'
        and l_shipdate > date '1995-03-03'
group by
        l_orderkey,
        o_orderdate,
        o_shippriority
order by
        revenue desc,
        o_orderdate
LIMIT 10;[root@VM_0_21_centos queries]#
```

图1 Q3语句

然后，explain下Q3，得到结果分别如图2和图3。

```
                                                     QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=216169.01..216169.23 rows=10 width=48)
   ->  Gather Motion 4:1  (slice4; segments: 4)  (cost=216169.01..216169.23 rows=10 width=48)
         Merge Key: revenue, partial_aggregation.o_orderdate
         ->  Limit  (cost=216169.01..216169.03 rows=3 width=48)
               ->  Sort  (cost=216169.01..216980.96 rows=81196 width=48)
                     Sort Key (Limit): revenue, partial_aggregation.o_orderdate
                     ->  HashAggregate  (cost=182376.99..186436.75 rows=81196 width=48)
                           Group By: lineitem.l_orderkey, orders.o_orderdate, orders.o_shippriority
                           ->  Redistribute Motion 4:4  (slice3; segments: 4)  (cost=167761.85..175881.37 rows=81196 width=48)
                                 Hash Key: lineitem.l_orderkey, orders.o_orderdate, orders.o_shippriority
                                 ->  HashAggregate  (cost=167761.85..169385.75 rows=81196 width=48)
                                       Group By: lineitem.l_orderkey, orders.o_orderdate, orders.o_shippriority
                                       ->  Hash Join  (cost=48961.76..164514.03 rows=81196 width=32)
                                             Hash Cond: lineitem.l_orderkey = orders.o_orderkey
                                             ->  Append-only Columnar Scan on lineitem  (cost=0.00..100617.19 rows=843946 width=24)
                                                   Filter: l_shipdate > '1995-03-03'::date
                                             ->  Hash  (cost=41746.06..41746.06 rows=144314 width=16)
                                                   ->  Broadcast Motion 4:4  (slice2; segments: 4)  (cost=5675.79..41746.06 rows=144314 width=16)
                                                         ->  Hash Join  (cost=5675.79..34530.36 rows=36079 width=16)
                                                               Hash Cond: orders.o_custkey = customer.c_custkey
                                                               ->  Append-only Columnar Scan on orders  (cost=0.00..24175.00 rows=179330 width=24)
                                                                     Filter: o_orderdate < '1995-03-03'::date
                                                               ->  Hash  (cost=4166.89..4166.89 rows=30178 width=8)
                                                                     ->  Broadcast Motion 4:4  (slice1; segments: 4)  (cost=0.00..4166.89 rows=30178 width=8)
                                                                           ->  Append-only Columnar Scan on customer  (cost=0.00..2658.00 rows=7545 width=8)
                                                                                 Filter: c_mktsegment = 'HOUSEHOLD'::bpchar
 Optimizer status: legacy query optimizer
(27 rows)
```

图2 GreenPlum执行explain Q3的结果

```
+----+-------------+----------+--------+---------------+------------+---------+--------------------------+---------+----------------------------------------------+
| id | select_type | table    | type   | possible_keys | key        | key_len | ref                      | rows    | Extra                                        |
+----+-------------+----------+--------+---------------+------------+---------+--------------------------+---------+----------------------------------------------+
|  1 | SIMPLE      | lineitem | ALL    | NULL          | NULL       | NULL    | NULL                     | 6001215 | Using where; Using temporary; Using filesort |
|  1 | SIMPLE      | orders   | eq_ref | O_ORDERKEY    | O_ORDERKEY | 8       | test.lineitem.L_ORDERKEY |       1 | Using where                                  |
|  1 | SIMPLE      | customer | eq_ref | C_CUSTKEY     | C_CUSTKEY  | 8       | test.orders.O_CUSTKEY    |       1 | Using where                                  |
+----+-------------+----------+--------+---------------+------------+---------+--------------------------+---------+----------------------------------------------+
3 rows in set (0.00 sec)
```

图3 Mysql执行explain Q3的结果

从以上的执行过程解释可以看出，GreenPlum上的执行步骤主要有：

1. 在所有segment（这里为4个）同时进行条件查询Filter；
2. 两表做关联时，会进行数据广播，每个segment将查询到的结果广播到其他所有segment，每个segment得到该表Filter后的所有结果（全量数据），后会进行一次hash；
3. 在所有segment上同时做hash join，因为还要和其他表做join，会继续将结果广播到所有segment上；
4. 进行group by聚合操作。首先在所有segment上根据group by条件进行一次HashAggregate聚合（目的是减少重分布的数据量），然后将结果数据按group by字段进行重分布，最后，每个segment再按条件聚合一次得到最终结果；
5. 根据order by条件，在所有segment上同时进行sort，根据Limit条件选取数据，这里是Limit 10，每个segment都选取10条数据汇总到master上，由master再选取前10条；
6. 进行Merge，所有segment将结果发给master，由master进行一次归并，根据Limit条件选取结果的前10条数据，返回。

整个过程耗时的点主要有：

1. 做了两次广播，总量为（30178+144314=174492）17万条；
2. 根据group by的条件Redistribute一次，数量约为8万条；
3. hash join两次，都是在两个表之间进行的hash join，在单个segment上，两表之间的hash join量分别大约是18万与3万、84万与14万；
4. sort一次，单个segment的sort从8万条数据中取出前10条记录。

Mysql的执行过程比较简单，首先是在lineitem表做一次where过滤，获取结果计算出revenue值，由于order by的值是revenue，因此，需要一次非关键字（revenue）排序，排序的量为3271974（约320万），这里非常耗时。然后在order表和customer表做一些where过滤。

从以上执行过程可以看出，主要的耗时点应该在sort操作上，GreenPlum是在所有segment上同时进行一次8万条记录的sort，而Mysql则是直接进行一次320万记录的sort。由于Mysql是在单个服务器上搭建的，该服务器的性能（8核CPU、24GB内存）远高于GreenPlum的单个segment（1核CPU、4GB内存），因此，如果充分利用服务器的性能，两者的sort时间应该相差不大，可是事实如此吗？接下来我们查看下Mysql所在服务器的CPU使用情况，得到执行Q3前后的结果如图4所示：

```
[root@VM_116_171_centos dss]# cat cpustat.before.txt
cpu  377932 12 560694 53701024 35220 0 37 1858 0
cpu0 39170 4 52649 6709652 30790 0 15 267 0
cpu1 1656 0 2851 6829805 758 0 4 163 0
cpu2 1390 0 1456 6831269 614 0 2 254 0
cpu3 3608 1 1859 6827612 1712 0 3 229 0
cpu4 1848 3 1616 6830839 286 0 1 155 0
cpu5 30271 0 46827 6757130 548 0 2 373 0
cpu6 298857 0 452035 6083775 227 0 5 200 0
cpu7 1130 0 1398 6830938 282 0 2 214 0
intr 20265316 179 6 0 0 90 0 0 0 0 0 0 0 104 0 0 0 0 0 0 0 0 0 0 0 0 75108 0 17
6 0 241033 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
ctxt 9349155
btime 1471525992
processes 77275
procs_running 1
procs_blocked 0
softirq 26659597 0 11948436 0 778398 0 0 90 1158281 7853 12766539
[root@VM_116_171_centos dss]# cat cpustat.after.txt
cpu  378360 12 560768 53704520 35222 0 37 1858 0
cpu0 39170 4 52650 6710149 30792 0 15 267 0
cpu1 1657 0 2851 6830304 758 0 4 163 0
cpu2 1390 0 1456 6831769 614 0 2 254 0
cpu3 3608 1 1859 6828112 1712 0 3 229 0
cpu4 1848 3 1616 6831339 286 0 1 155 0
cpu5 30271 0 46827 6757630 548 0 2 373 0
cpu6 299283 0 452108 6083776 227 0 5 200 0
cpu7 1130 0 1398 6831438 282 0 2 214 0
intr 20270805 179 6 0 0 90 0 0 0 0 0 0 0 104 0 0 0 0 0 0 0 0 0 0 0 0 75115 0 176
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ctxt 9349778
btime 1471525992
processes 77278
procs_running 1
procs_blocked 0
softirq 26670063 0 11953614 0 778401 0 0 90 1158385 7853 12771720
```

图4 Mysql执行Q3前后其所在服务器的CPU使用时间情况

可以看出，执行Q3前后，只有CPU6的使用时间有较大变化，变化时间大概为500jiffies即5秒，与总的sql执行时间（5.06秒）基本吻合，因此，执行Q3 过程中，mysql所在的服务器只使用了一个CPU来进行计算。

综上，Mysql和GreenPlum的耗时区别主要体现在sort操作上，Mysql对320万条记录做了一次sort，但只能使用单个CPU计算，没有发挥服务器本身多核CPU的性能优势，整体执行时间较长。而GreenPlum由于采用了分布式的结构，每个segment对应一个CPU，数据均匀分布到各个segment，在各节点并行执行Filter、hash join、group by，sort等操作，充分利用了每个CPU的计算能力，然后将结果进行广播，或对整体数据进行重分布再进行计算，最后由master归并各segment的结果数据。在进行广播或者重分布时，会在segment节点间进行数据传输，消耗了一定的时间，但由于GreenPlum对sql的优化更好，以及并行计算的能力，因此，相比于Mysql，总的执行时间更短。

示例二

我们再选取一个典型的事例——Q17，根据执行时间统计，Mysql的执行时间是GreenPlum的1.5万倍，这是一个相当大的差距！究竟是什么原因会导致如此大的区别，我们首先查看Q17的sql语句如下图5所示。

```
select
        sum(l_extendedprice) / 7.0 as avg_yearly
from
        lineitem,
        part,
        (SELECT l_partkey AS agg_partkey, 0.2 * avg(l_quantity) AS avg_quantity FROM lineitem GROUP BY l_partkey) part_agg
where
        p_partkey = l_partkey
        and agg_partkey = l_partkey
        and p_brand = 'Brand#21'
        and p_container = 'JUMBO JAR'
        and l_quantity < avg_quantity
LIMIT 1;
```

图5 Q17语句

与Q3不同的是Q17涉及到了子查询，依旧，我们在mysql和GreenPlum上explain下sql，得到的结果如图6、图7所示。

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------
 Limit  (cost=232764.86..232764.88 rows=1 width=32)
   ->  Aggregate  (cost=232764.86..232764.88 rows=1 width=32)
         ->  Gather Motion 4:1  (slice3; segments: 4)  (cost=232764.79..232764.85 rows=1 width=32)
               ->  Aggregate  (cost=232764.79..232764.80 rows=1 width=32)
                     ->  Hash Join  (cost=131427.54..232738.81 rows=2598 width=10)
                           Hash Cond: public.lineitem.l_partkey = part.p_partkey
                           Join Filter: public.lineitem.l_quantity < part_agg.avg_quantity
                           ->  Append-only Columnar Scan on lineitem  (cost=0.00..85614.15 rows=1500304 width=25)
                           ->  Hash  (cost=131378.36..131378.36 rows=984 width=48)
                                 ->  Broadcast Motion 4:4  (slice2; segments: 4)  (cost=126109.19..131378.36 rows=984 width=48)
                                       ->  Hash Join  (cost=126109.19..131329.19 rows=246 width=48)
                                             Hash Cond: part_agg.agg_partkey = part.p_partkey
                                             ->  HashAggregate  (cost=122248.21..125088.78 rows=47343 width=40)
                                                   Group By: public.lineitem.l_partkey
                                                   ->  Redistribute Motion 4:4  (slice1; segments: 4)  (cost=115620.23..119407.65 rows=47343 width=40)
                                                         Hash Key: public.lineitem.l_partkey
                                                         ->  HashAggregate  (cost=115620.23..115620.23 rows=47343 width=40)
                                                               Group By: public.lineitem.l_partkey
                                                               ->  Append-only Columnar Scan on lineitem  (cost=0.00..85614.15 rows=1500304 width=15)
                                             ->  Hash  (cost=3848.00..3848.00 rows=260 width=8)
                                                   ->  Append-only Columnar Scan on part  (cost=0.00..3848.00 rows=260 width=8)
                                                         Filter: p_brand = 'Brand#21'::bpchar AND p_container = 'JUMBO JAR'::bpchar
 Optimizer status: legacy query optimizer
(23 rows)
```

图6 GreenPlum执行explain Q17的结果

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	200000	
1	PRIMARY	lineitem	ALL	NULL	NULL	NULL	NULL	6001215	Using where; Using join buffer
1	PRIMARY	part	eq_ref	P_PARTKEY	P_PARTKEY	8	test.lineitem.L_PARTKEY	1	Using where
2	DERIVED	lineitem	ALL	NULL	NULL	NULL	NULL	6001215	Using temporary; Using filesort

4 rows in set (6.09 sec)

图7 Mysql执行explain Q17的结果

子查询sql（select l_partkey as agg_partkey, 0.2 * avg(l_quantity) as avg_quantity from lineitem group by l_partkey）里面涉及group by，我们来看一下两者在聚合上的区别。

Mysql：由于group by的是非索引关键字，所以直接进行了filesort lineitem(600万条记录)。

GreenPlum：首先在每个segment（有该表150万条记录）做一次group by l_partkey，采用了更高效的HashAggregate聚合方式。为了所有segment可以并行做join，会将lineitem表的数据做一次重分布（5万条记录），每个segment得到的是hash分布到自身的记录。

可以看出，Mysql在聚合上的效率要明显低于GreenPlum。
然后，子查询结果会与现表做join操作，我们来继续看下两者在join上的区别：
Mysql：把子查询结果作为临时表（20万条记录）与现表lineitem(600万条记录）直接做了join，将产生600万×20万=1.2万亿的数据量.......

GreenPlum：首先对sql进行了优化，先执行了where条件，减少了part表的数据到260条（单个segment的量，总量为4×260条，接下来的数据量都为单个segment的）。

采用了更高效的join方式hash join。
如果使用临时表与lineitem表直接hash join，会产生50万左右的数据量，但GreenPlum并没有这么做，而是利用part表来进行join，因为part表经过where过滤后数据量非常小，和part表做hash join，数据量也相对较小。总共做了两次hash join：

- part表与临时表part_agg，产生数据量246条；

- part表与lineitem表，产生数据量2598条；

两者一对比，GreenPlum做join的数据量为(246+2598)×4=11376条，远小于Mysql的1.2万亿条，两者的性能不言而喻。
综上，在执行Q17时，Mysql和GreenPlum的效率差别除了GreenPlum具有并行计算能力外，还体现在聚合和关联这两个操作的优化上面。

七.总结

根据以上的统计结果和性能对比分析，可以看出，GreenPlum在TPC-H类的测试性能会远远高于单机版的Mysql，说明具有分布式属性的GreenPlum在关于复杂语句（涉及到多表属性查询、group by、order by 、join、子查询等）的查询效率较高，且可以充分利用系统CPU资源来提升查询速度，更适用于OLAP。

八.其他事项

1. 由于原生的TPC-H的测试用例不直接支持GreenPlum和Mysql，因此需要修改测试脚本，生成新的建表语句如《附录一》所示，测试sql如《附录二》。
2. GreenPlum各节点间的带宽要尽可能大，一般查询中会涉及到广播或者重分布动作，需要在节点间传输数据，如果带宽过小，易在此造成性能瓶颈。
3. 测试语句不要过于简单，并且测试数据量不要太少，否则GreenPlum在做数据传输的时间会远高于计算时间。