第六章 查询执行

Greenplum 提供全面的 SQL-92 和 SQL-99 语言支持以及 SQL 2003 OLAP 扩展支持，例如 window function、rollup、cube 和很多其他表达型的函数。所有的查询都是并行的在整个系统上执行。标准的数据库接口（PostgreSQL、SQL、ODBC、JDBC、OLEDB 等）都被完全支持，还支持商业智能（BI）认证和提取/转换/加载（ETL）工具。这使得现有的分析工具和应用仅需要很少的重组代价即可在 Greenplum 上使用。

查询执行和优化自数据库诞生开始就是一个经久不衰的研究话题，上世纪 70 年代，关系数据诞生不久，Selinger 等人就提出基于路径代价模型的查询优化算法[1]，将查询语句转化为树状结构，每一个执行

策略为从根节点到叶节点的一条路径，然后建立一套路径代价模型用于估计每一条路径的执行代价，选择最优的路径作为最终执行策略。文献[2]也是关于查询优化的早期的重要工作，采用查询图模型对查询语句进行重写，定义了 8 条重写规则，将 SQL 查询转化为更具表达性的等价语句，为查询优化提供更多选择性。除了直接对查询进行直接优化，文献[3]则是通过详细的实验，探索查询执行过程中，时间在各个阶段的消耗细节，探索出数据库系统是否充分利用当前诸如 cpu、缓存等硬件性能，为查询优化工作提供指导。

关系数据诞生以来，对其性能优化的研究不曾停滞，文献[8]提出一种近似的优化策略，通过降低优化器的搜索空间以加快优化策略生成时间，同时又要保证所选的策略与未近似时的误差不至于太大。文献[9]则优化关系查询的分组处理，提出查询分割的理论方法，对于复杂查询 group 划分有着不错的效果。

随着数据系统不断发展，分布式数据库和数据仓库应运而生，传统关系数据库的查询优化工作不再完全适应这种新的变化，一些针对分布式系统的查询优化工作成为研究的热点。文献[4]提出将 hadoop 系统与数据库相结合的方式，充分发挥两者的优势。文献[5]则介绍 SWQserver 查询优化器 PDW 的优化机制，将数据分配到各个子节点，充分利用分布式优势，采用自底向上的策略枚举方式，使用特有的代价消耗模型选取最优策略。文献[6]则是从数据倾斜的角度，采用改进的 Binary Space Partitioning 算法，将并行 JOIN 问题化简为易于处理的子问题，从而达到高效并行化 JOIN 的目的。文献[7]则提出一种基

于 share-nothing 架构的查询优化算法，通常的分布式系统查询优化工作只由管理节点独立完成，在多并发高并行的系统中很容易成为瓶颈，文章首次提出将查询优化工作下放到子节点，充分发挥整个系统的处理性能，达到近似线性的可扩展性增长曲线。

下文将从 Greenplum 的并行查询计划生成和查询执行两个方面进行介绍。

6.1 Greenplum 中的查询计划生成

该部分重点介绍 Greenplum 的并行查询计划是如何生成的以及一个可以和 Greenplum 搭配使用的查询优化器 ORCA。

6.1.1 查询计划优化实例

相比于普通数据库查询计划只能在单节点上执行，Greenplum 需要将查询计划并行化，以充分发挥 MPP 的优势。Greenplum 引入 Motion 算子（操作符）实现查询计划的并行化。Motion 算子实现了数据在不同节点间的传输，它为其他算子隐藏了 MPP 架构和单机的不同，使得其他大多数算子不用关心是在集群上执行还是在单机上执行。每个 Motion 算子都有发送方和接收方。此外 Greenplum 还对某些算子进行了分布式优化，譬如聚集。

下面的例子中创建了 2 张表 t1 和 t2，它们都有两个列 c1, c2，都是以 c1 为分布键。

```
CREATE table t1 AS SELECT g c1, g + 1 as c2 FROM generate_
```

series(1, 10) g DISTRIBUTED BY (c1);

CREATE table t2 AS SELECT g c1, g + 1 as c2 FROM generate_series(5, 15) g DISTRIBUTED BY (c1);

SQL1:

```
SELECT * from t1, t2 where t1.c1 = t2.c1;
 c1 | c2 | c1 | c2
----+----+----+----
  5 |  6 |  5 |  6
  6 |  7 |  6 |  7
  7 |  8 |  7 |  8
  8 |  9 |  8 |  9
  9 | 10 |  9 | 10
 10 | 11 | 10 | 11
(6 rows)
```

SQL1 的查询计划为如下所示，因为关联键是两个表的分布键，所以关联可以在本地执行，HashJoin 算子的子树不需要数据移动，最后 GatherMotion 在 master 上做汇总即可。

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=3.23..6.48 rows=10 width=16)
```

-> Hash Join (cost=3.23..6.48 rows=4 width=16)

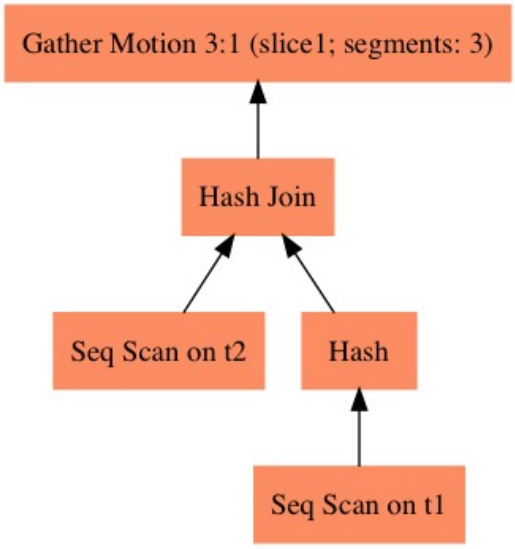
Hash Cond: t2.c1 = t1.c1

-> Seq Scan on t2 (cost=0.00..3.11 rows=4 width=8)

-> Hash (cost=3.10..3.10 rows=4 width=8)

-> Seq Scan on t1 (cost=0.00..3.10 rows=4 width=8)

Optimizer: legacy query optimizer



SQL2:

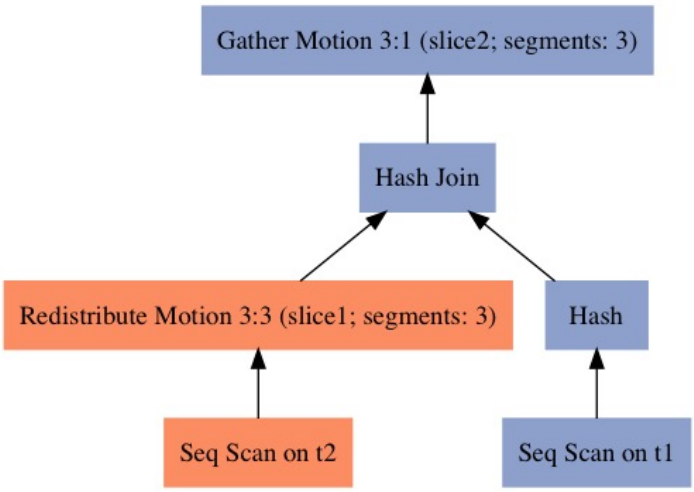
SELECT * from t1, t2 where t1.c1 = t2.c2;

```
 c1 | c2 | c1 | c2
----+----+----+----
  9 | 10 |  8 |  9
 10 | 11 |  9 | 10
```

```
 8 |  9 |  7 |  8
 6 |  7 |  5 |  6
 7 |  8 |  6 |  7
(5 rows)
```

SQL2 的查询计划如下所示，t1 表的关联键 c1 也是其分布键，t2 表的关联键 c2 不是分布键，所以数据需要根据 t2.c2 重分布，以便所有 t1.c1 = t2.c2 的行都在同一个 segment 上执行关联操作。

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Gather Motion 3:1  (slice2; segments: 3)  (cost=3.23..6.70 rows=10 width=16)
   ->  Hash Join  (cost=3.23..6.70 rows=4 width=16)
         Hash Cond: t2.c2 = t1.c1
       ->  Redistribute Motion 3:3  (slice1; segments: 3)  (cost=0.00..3.33 rows=4 width=8)
             Hash Key: t2.c2
         ->  Seq Scan on t2  (cost=0.00..3.11 rows=4 width=8)
       ->  Hash  (cost=3.10..3.10 rows=4 width=8)
         ->  Seq Scan on t1  (cost=0.00..3.10 rows=4 width=8)
 Optimizer: legacy query optimizer
```



SQL3:

```
SELECT * from t1, t2 where t1.c2 = t2.c2;
 c1 | c2 | c1 | c2
----+----+----+----
  8 |  9 |  8 |  9
  9 | 10 |  9 | 10
 10 | 11 | 10 | 11
  5 |  6 |  5 |  6
  6 |  7 |  6 |  7
  7 |  8 |  7 |  8
(6 rows)
```

SQL3 的查询计划如下所示，t1 的关联键 c2 不是分布键，t2 的关联键 c2 也不是分布键，所以采用广播 Motion，使得其中一个表的数据可以广播到所有节点上，以保证关联的正确性。

QUERY PLAN

Gather Motion 3:1 (slice2; segments: 3) (cost=3.25..6.96 rows=10 width=16)

-> Hash Join (cost=3.25..6.96 rows=4 width=16)

Hash Cond: t1.c2 = t2.c2

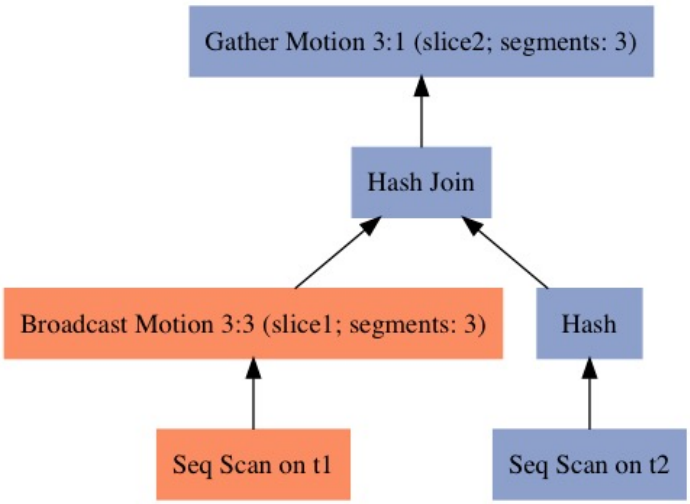
-> Broadcast Motion 3:3 (slice1; segments: 3) (cost=0.00..3.50 rows=10 width=8)

-> Seq Scan on t1 (cost=0.00..3.10 rows=4 width=8)

-> Hash (cost=3.11..3.11 rows=4 width=8)

-> Seq Scan on t2 (cost=0.00..3.11 rows=4 width=8)

Optimizer: legacy query optimizer



SQL4:

SELECT * from t1 LEFT JOIN t2 on t1.c2 = t2.c2 ;

```
 c1 | c2 | c1 | c2
----+----+----+----
  1 |  2 |    |
  2 |  3 |    |
  3 |  4 |    |
  4 |  5 |    |
  5 |  6 |  5 |  6
  6 |  7 |  6 |  7
  7 |  8 |  7 |  8
  8 |  9 |  8 |  9
  9 | 10 |  9 | 10
 10 | 11 | 10 | 11
(10 rows)
```

SQL4 的查询计划如下所示，尽管关联键和 SQL3 一样，然而由于采用了 left join，所以不能使用广播 t1 的方法，否则数据会有重复，因而这个查询的计划对两张表都进行了重分布。根据路径代价的不同，对于 SQL4 优化器也可能选择广播 t2 的方法。（如果数据量一样，单表广播代价要高于双表重分布，对于双表重分布，每个表的每个元组传输一次，相当于单表每个元组传输两次，而广播则需要单表的每个元组传输 nSegments 次。）

QUERY PLAN

Gather Motion 3:1 (slice3; segments: 3) (cost=3.47..6.91 rows=10 width=16)

-> Hash Left Join (cost=3.47..6.91 rows=4 width=16)

Hash Cond: t1.c2 = t2.c2

-> Redistribute Motion 3:3 (slice1; segments: 3) (cost=0.00..3.30 rows=4 width=8)

Hash Key: t1.c2

-> Seq Scan on t1 (cost=0.00..3.10 rows=4 width=8)

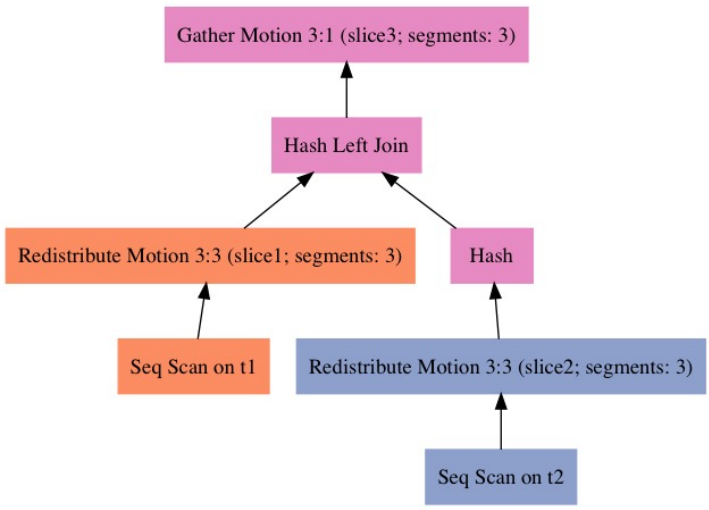
-> Hash (cost=3.33..3.33 rows=4 width=8)

-> Redistribute Motion 3:3 (slice2; segments: 3) (cost=0.00..3.33 ...

Hash Key: t2.c2

-> Seq Scan on t2 (cost=0.00..3.11 rows=4 width=8)

Optimizer: legacy query optimizer

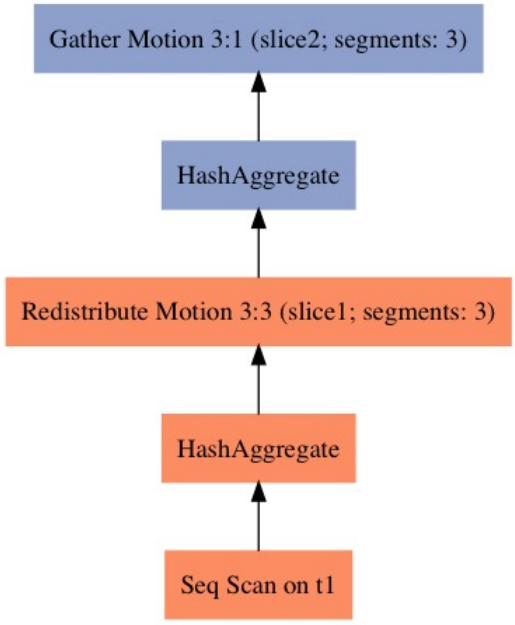


SQL5:

```
SELECT  c2, count(1) from t1 group by c2;
 c2  |  count
-----+-------
   5 |      1
   6 |      1
   7 |      1
   4 |      1
   3 |      1
  10 |      1
  11 |      1
   8 |      1
   9 |      1
   2 |      1
(10  rows)
```

上面四个 SQL 都是不同类型的 JOIN 对数据移动类型(Motion)的影响。SQL5 演示了 Greenplum 对聚集的优化：两阶段聚集。第一阶段聚集在每个 Segment 上对本地数据执行，然后通过重分布到每个 segment 上执行第二阶段聚集。最后由 Master 通过 Gather Motion 进行汇总。 Greenplum 对某些 SQL 譬如 DISTINCT GROUP BY 也会采用三阶段聚集。

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Gather Motion 3:1  (slice2; segments: 3)  (cost=3.55..3.70 rows=10 width=12)
   ->  HashAggregate  (cost=3.55..3.70 rows=4 width=12)
         Group Key: t1.c2
         ->  Redistribute Motion 3:3  (slice1; segments: 3)  (cost=3.17..3.38 rows=4 width=12)
               Hash Key: t1.c2
           ->  HashAggregate  (cost=3.17..3.17 rows=4 width=12)
                 Group Key: t1.c2
                 ->  Seq Scan on t1  (cost=0.00..3.10 rows=4 width=4)
 Optimizer: legacy query optimizer
(9 rows)
```



Greenplum 引入的新数据结构和概念

6.1.2 查询计划的内部机制

前面几个直观的例子展示了 Greenplum 对不同 SQL 生成的各种分布式查询计划。下面介绍其主要内部机制。

为了把单机查询计划变成并行计划，Greenplum 引入了一些新的概念，分别对 PostgreSQL 的 Node、Path 和 Plan 结构体进行了增强：

- 新增一个新的节点类型：Flow
- 新增一种路径类型：CdbMotionPath
- 新增一个新的查询计划算子：Motion（Motion 的第一个字段是 Plan，Plan 结构体的第一个字段是 NodeTag type。Flow 的第一个节点也是 NodeTag type，和 RangeVar、IntoClause、Expr、

RangeTableRef 是一个级别的概念)

- 为 Path 结构体添加了 CdbPathLocus locus 这个字段，以表示结果元组在这个路径下的重分布策略
- 为 Plan 结构体增加 Flow 字段，以表示这个算子的元组流向；

Node

新节点类型 Flow 描述了并行计划中元组的流向。 每个查询计划节点（Plan 结构体）都有一个 Flow 字段，以表示当前节点的输出元组的流向。 Flow 是一个新的节点类型，但不是一个查询计划节点。此外 Flow 结构体还包括一些用于计划并行化的成员字段。

Flow 有三个主要字段：

- FlowType，表示 Flow 的类型
    - UNDEFINED: 未定义 Flow
    - SINGLETON：表示的是 GatherMotion
    - REPLICATED：表示的是广播 Motion
    - PARTITIONED: 表示的是重分布 Motion。
- Movement，确定当前计划节点的输出，该使用什么样的 motion。主要用于把子查询的计划进行处理以适应分布式环境。
    - None：不需要 motion
    - FOCUS：聚焦到单个 segment，相当于 GatherMotion
    - BROADCAST: 广播 motion
    - REPARTITION: 哈希重分布

    - EXPLICIT：定向移动元组到 segid 字段标记的 segments
- CdbLocusType： Locus 的类型，优化器使用这个信息以选择最合适的节点进行最合适的数据流向处理，确定合适 Motion。
    - CdbLocusType_Null：不用 Locus
    - CdbLocusType_Entry: 表示 entry db （即 master） 上单个 backend 进程，可以是 QD （Query Dispatcher)，也可以是 entrydb 上的 QE（Query Executor）
    - CdbLocusType_SingleQE：任何节点上的单个 backend 进程，可以是 QD 或者任意 QE 进程
    - CdbLocusType_General：和任何 locus 都兼容
    - CdbLocusType_Replicated：在所有 QEs 都有副本
    - CdbLocusType_Hashed：哈希分布到所有 QEs
    - CdbLocusType_Strewn：数据分布存储，但是分布键未知

Path

Path 表示了一种可能的计算路径（譬如顺序扫描或者哈希关联），更复杂的路径会继承 Path 结构体并记录更多信息以用于优化。 Greenplum 为 Path 结构体新加 CdbPathLocus locus 这个字段，用于表示结果元组在当前路径下的重分布和执行策略。

Greenplum 中表的分布键决定了元组存储时的分布情况，影响元组在那个 segment 的磁盘上的存储。CdbPathLocus 决定了在执行时一个元组在不同的进程间（不同 segment 的 QE）的重分布情况，即

一个元组该被那个进程处理。元组可能来自于表，也可能来自于函数。

Greenplum 还引入了一个新的路径： CdbMotionPath， 用以表示子路径的结果如何从发送方进程传送给接收方进程。

如上面所述，Motion 是一种查询计划树节点，它实现了数据的洗牌（Shuffle），使得其父算子可以从其子算子得到需要的数据。Motion 有三种类型：

- MOTIONTYPE_HASH：使用哈希算法根据重分布键对数据进行重分布，把经过算子的每个元组发送到目标 segment，目标 segment 由重分布键的哈希值确定。
- MOTIONTYPE_FIXED：发送元组给固定的 segment 集合，可以是广播 Motion（发送给所有的 segments）或者 Gather Motion（发送给固定的某个 segment）
- MOTIONTYPE_EXPLICIT：发送元组给其 segid 字段指定的 segments，对应于显式重分布 Motion。和 MOTIONTYPE_HASH 的区别是不需要计算哈希值。

前面提到，Greenplum 为 Plan 结构体引入了 Flow *flow 这个字段表示结果元组的流向。此外 Plan 结构体还引入了其他几个与优化和执行相关的字段，譬如表示是否需要 MPP 调度的 DispatchMethod dispatch 字段、是否可以直接调度的 directDispatch 字段（直接调度到某个 segment，通常用于主键查询）、方便 MPP 执行的分布式计划的 sliceTable、用于记录当前计划节点的父 motion 节点的 motion

Node 等。

6.1.3 分布式查询计划生成

下图展示了 Greenplum 中传统优化器（ORCA 优化器于此不同）的优化流程，本节强调与 PostgreSQL 的单机优化器不同的部分。

standard_planner 是 PostgreSQL 缺省的优化器，它主要调用了 subquery_planner 和 set_plan_references。在 Greenplum 中，set_plan_references 之后又调用了 cdbparallelize 以对查询树做最后的并行化处理。

subquery_planner 如名字所示对某个子查询进行优化，生成查询计划树，它主要有两个执行阶段：

- 基本查询特性的优化，由 query_planner() 实现
- 高级查询特性的优化，例如聚集等，由 grouping_planner() 实现，grouping_planner() 会调用 query_planner() 进行基本优化，然后对高级特性进行优化。

Greenplum 对单机计划的分布式处理主要发生在两个地方：

- 单个子查询：Greenplum 的 subquery_planner() 返回的子查询计划树已经进行了某些分布式处理，譬如为 HashJoin 添加 Motion 算子，二阶段聚集等。
- 多个子查询间：Greenplum 需要设置多个子查询间恰当的数据流向，以使得某个子查询的结果可以被上层查询树使用。这个

操作是由函数 cdbparallelize 实现的。

1. 单个子查询的并行化

Greenplum 优化单个子查询的流程和 PostgreSQL 相似，主要区别在于：

- 关联：根据关联算子的左右子表的数据分布情况确定是否添加 Motion 节点、什么类型的 Motion 等。
- 聚集等高级操作的优化，譬如前面提到的两阶段聚集。

下面简要介绍下主要流程：

首先使用 build_simple_rel() 构建简单表的信息。build_simple_rel 获得表的基本信息，譬如表里面有多少元组，占用了多少个页等。其中很重要的一个信息是数据分布信息：GpPolicy 描述了基本表的数据分布类型和分布键。

然后使用 set_base_rel_pathlists() 设置基本表的访问路径。set_base_rel_pathlists 根据表类型的不同，调用不同的函数：

- RTE_FUNCTION: create_functionscan_path()
- RTE_RELATION: create_external_path()/create_aocs_path()/create_seqscan_path()/create_index_paths()
- RTE_VALUES: create_valuesscan_path

这些函数会确定路径节点的 locus 类型，表示数据分布处理相关的一种特性。 这个信息对于子查询并行化非常重要，在后面把 path 转换成 plan 的时候，被用于决定一个计划的 FLOW 类型，而 FLOW 会决定执行器使用什么样类型的 Gang 来执行。

对于普通的堆表（Heap），顺序扫描路径 create_seqscan_path() 使用下面方式确定路径的 locus 信息：

- 如果表是哈希分布，则 locus 类型为 CdbLocusType_Hashed
- 如果是随机分布，则 locus 类型为 CdbLocusType_Strewn
- 如果是系统表，则 locus 类型为 CdbLocusType_Entry

对于函数，则 create_function_path() 使用下面方式确定路径的 locus：

- 如果函数是 immutable 函数，则使用：CdbLocusType_General
- 如果函数是 mutable 函数，则使用：CdbLocusType_Entry
- 如果函数需要在 master 上执行，则使用： CdbLocusType_Entry
- 如果函数需要在所有 segments 上执行，则使用 CdbLocusType_Strewn

如果 SQL 语句中包含关联，则使用 make_rel_from_joinlist() 为关联树生成访问路径。相应的函数有：create_nestloop_path/create_mergejoin_path/create_hashjoin_path。这个过程最重要的一点是确定是否需要添加 Motion 节点以及什么类型的 Motion 节点。 譬如前面 SQL1 关联键是两张表 t1/t2 的分布键，因而不需要添加 Motion；而 SQL2 则需要对 t2 进行重分布，以使得对于任意 t1 的元组，满足关联条件 (t1.c1 = t2.c2) 的所有 t2 的元组都在同一个 segment 上。

如果 SQL 包含聚集、窗口函数等高级特性，则调用 cdb_grouping_planner() 进行优化处理，譬如将聚集转换成两阶段聚集或者三阶段聚集等。

最后一步是从所有可能的路径中选择最廉价的路径，并调用 create_plan() 把最优路径树转换成最优查询树。

在这个阶段， Path 路径的 Locus 影响生成的 Plan 计划的 Flow 类型。Flow 和执行器一节中的 Gang 相关，Flow 使得执行器不用关心数据以什么形式分布、分布键是什么，而只关心数据是在多个 segment 上还是单个 segment 上。 Locus 和 Flow 之间的对应关系：

- FLOW_SINGLETON: Locus_Entry/Locus_SingleQE/Locus_General
- FLOW_PARTITIONED: Locus_Hash/Locus_Strewn/Locus_Replicated

2. 多个子查询之间的并行化

cdbparallelize() 主要目的是解决多个子查询之间的数据流向，生成最终的并行化查询计划。它含有两个主要步骤：prescan 和 apply_motion

- prescan 有两个目的，一个目的是对某些类型的计划节点（譬如 Flow ）做标记以备后面 apply_motion 处理；第二个目的是对子计划节点 （SubPlan）进行标记或者变形。SubPlan 实际上不是查询计划节点，而是表达式节点，它包含一个计划节点及其范围表（Range Table）。 SubPlan 对应于查询树中的 SubLink（SQL 子查询表达式），可能出现在表达式中。prescan 对 SubPlan 包含的计划树做以下处理：
- 如果 Subplan 是个 Initplan，则在查询树的根节点做一个标注，表示需要以后调用 apply_motion 添加一个 motion 节点。
- 如果 Subplan 是不相关的多行子查询，则根据计划节点中包含的 Flow 信息对子查询执行 Gather 或者广播操作。并在查询树之上添加一个新的 materialized （物化）节点，以防止对 Subplan 进行重新扫描。因为避免了每次重新执行子查询，所以效率提高。

- 如果 Subplan 是相关子查询，则转换成可执行的形式。递归扫描直到遇到叶子扫描节点，然后使用下面的形式替换该扫描节点。经过这个转换后，查询树可以并行执行，因为相关子查询已经变成结果节点的一部分，和外层的查询节点在同一个 Slice 中。

```
Result
  \
   \_Material
     \
      \_Broadcast (or Gather)
        \
         \_SeqScan
```

- apply_motion: 根据计划中的 Flow 节点，为顶层查询树添加 motion 节点。根据 SubPlan 类型的不同（譬如 InitPlan、不相关多行子查询、相关子查询）添加不同的 Motion 节点。

譬如 SELECT * FROM tbl WHERE id = 1，prescan() 遍历到查询树的根节点时会在根节点上标注，apply_motion() 时在根节点之上添加一个 GatherMotion。

6.1.4 ORCA 查询优化器

ORCA 是 Pivotal 数据管理产品（包括 GPDB 和 HAWQ）中一个新的查询优化器。它是基于 Cascades 优化框架的一个现代的自上而下的优化器。相比于许多其他 Cascades 优化器会与主机系统紧密耦合，Orca 的一个独一无二的特征就是它作为一个独立的优化器运行在数据库系统之外。这个特性对于具有不同计算架构（例如 MPP 和 Hadoop）但使用一个优化器的产品是至关重要的。同时，它还允许利用在新的查询优化范例中关系型优化的丰富的特性。另外，将优化器作为一个独立的产品运行能够解耦测试，这样就不需要通过数据库的整体架构。

DXL.从数据库中解耦优化器需要建立一个通信机制来处理查询。Orca 包含一个在优化器和数据库系统之间进行信息交换的框架，叫做信息交换框架（DXL）。该框架使用一个基于 XML 的语言去编码必要信息间的通信，比如输入查询、输出计划和元数据。覆盖在 DXL 上的是一个简单的通信协议，用来发送初始查询的结构和检索优化计划。DXL 的一个主要好处在于将 Orca 包装为一个独立的产品。

图 6-6 显示了 Orca 与外部数据库系统之间的相互作用。对 Orca 的输入是一个 DXL 查询，输出则一个 DXL 计划。在优化过程中，数据库系统可以用来查询元数据（比如表的定义）。Orca 通过允许数据库系统去注册一个元数据提供者（MD Provider）可以抽象出元数据访问具体细节，其中元数据提供者负责在送到 Orca 之前将元数据序列化变为 DXL。元数据也可以从包含元数据对象的普通文件中抽取出来并序列化为 DXL 格式。

数据库系统需要包含一个可以将数据转换为 DXL 格式的转换器。Query2DXL 转化器将查询解析树转化为 DXL 查询，而 DXL2Plan 转化器将 DXL 计划转化为执行计划。这个转化器的实现则是由外部的 Orca 实现的，它允许多个系统通过它提供的适当的转化器去使用 Orca。

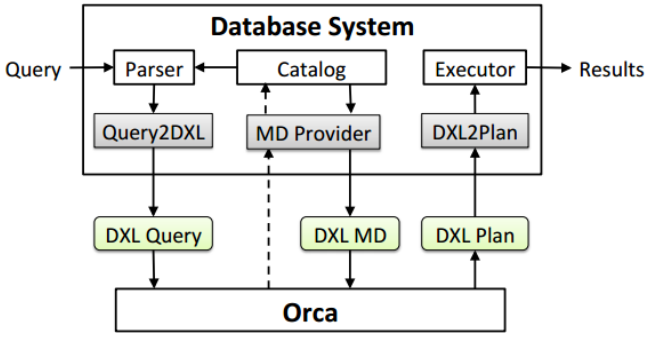


图 6-6 Orca 与数据库系统间的交互

Orca 是一个高度可扩展的架构;所有的组件都可单独更换和配置。图 6-7 显示 Orca 的不同组件，我们将简要的描述下这些组件。

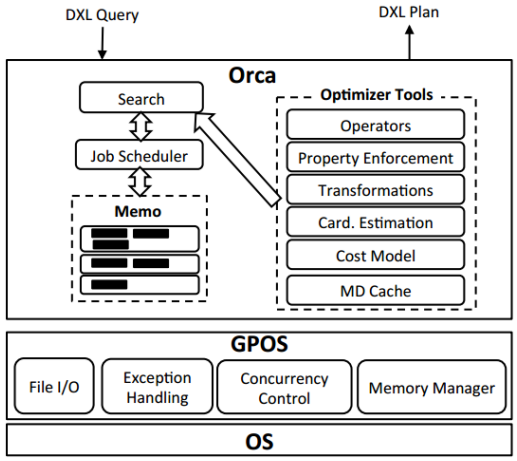


图 6-7 Orca 架构

- 备忘录（Memo）。由优化器生成的存储各替代计划的空间被编码到一个紧凑的内存数据结构中叫做备忘录。备忘录结构由一个称为组（group）的容器集合组成。其中每个组包含一些逻辑上等价表达式。每个备忘录的组集合负责捕获不同查询的子目标（例如对一个表的过滤或两个表得连接）。

组内成员，称为组表达式，以不同的逻辑表达式实现组目标（例如不同的连接顺序）。每个组表达式是一个操作符，而该操作符又有其他的组作为它的孩子。这种递归结构的备忘录能够将一个占巨大空间的所有可能计划编码到一个紧凑的空间中。

- 搜索和作业调度（Search and Job Scheduler）。Orca 通过一个搜索机制去浏览所有可能计划空间并识别出具有最小估计费用的计划。搜索机制由一个专门的作业调度器执行，这个作业调度器能够创建独立的（或多线程的）的工作单元去执行查询优化。这个过程主要包括三步：探测（exploration），这一步等价的逻辑表达式将会生成；实现（implementation），这一步物理计划生成；优化（optimization），一些要求的物理属性（例如排序顺序）被执行并计算计划的代价。
- **转化（Transformations）。**通过应用转换规则生成可能的计划。这些转化规则可以产生等价的逻辑表达式（例如 Join(A,B) → HashJoin(A,B)）。应用转化规则生成的结果将被复制到备忘录中，这个可能会导致创建一个新的组或者（并且）添加一个新的组表达式到存在的组中。每个转换规则在 Orca 的配置中是一个独立的组件，可以被明确地激活/停用。
- **属性执行（Property Enforcement）。**Orca 包含一个可扩展的框架，能够基于标准的属性书名来描述查询要求和计划特

征。属性有不同的类型，包括逻辑属性（例如输出列）、物理属性（例如排序顺序和数据分布）以及范围属性（例如在连接条件中用到的列数）。在查询优化期间，每个操作可能要求它的孩子满足特定的属性。一个优化过的子计划可能满足它自己所要求的属性，同时一个执行（例如排序操作）需要插入到计划中以满足要求的属性。

- **元数据缓存（Metadata Cache）。**由于元数据（例如表定义）很少发生变化，每一个查询调用它都会产生额外的开销。Orca 缓存元数据在优化器这边，并且如果在缓存中是不可用的或者从它加载到缓存的上一次已发生改变的话，那么我们只从目录检索它。元数据缓存同时也能从优化器中抽象出数据库系统的详细描述。这一点在测试和调试中是十分有效的。
- **GPOS。**为了与包含可能不同的 API 的操作系统交互，Orca 使用一个 OS 抽象层，称为 GPOS。GPOS 层为 Orca 提供了广泛的基础设施，包括内存管理、并发控制、异常处理、文件 I/O 管理以及数据结构的同步。

我们使用下面这个例子阐述查询优化工作流：

SELECT T1.a FROM T1, T2 WHERE T1.a = T2.b ORDER BY T1.a;

其中表 T1 的分布是 Hashed(T1.a)，而表 T2 的分布是 Hashed(T2.a)。首先我们将变成 DXL 格式，然后将得到的 DXL 查询转化为逻辑表达式

存到备忘录中。图 4 展示了初始化备忘录中的内容。逻辑表达式创建了三个组，其中两个为表一个为 InnerJoin 操作。为了简洁，我们省略了连接条件。Group 0 称为根组，它对应逻辑表达式的根操作。逻辑表达式的操作符之间依赖关系将作为组之间的引用。例如，Inner Join[1,2]将 Group 1 和 Group 2 作为孩子。下面我们将描述优化步骤。

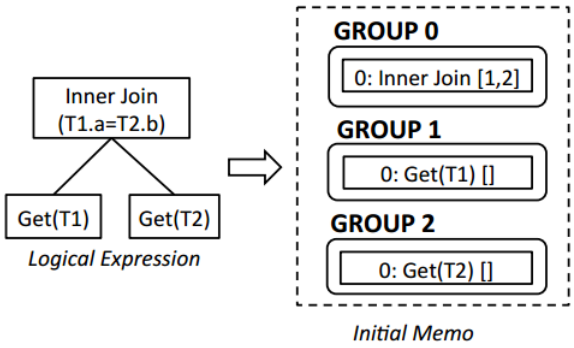


图 6-8 拷贝的初始的逻辑表达式

（1）探测(Exploration).生成等价逻辑表达式的转换规则被触发。例如，一个连接交换规则触发生成 InnerJoin[2,1] 和 InnerJoin[1,2]。探测的结果可能在现有的组中添加新的组表达式或者创建新组。备忘录有一个基于表达式的拓扑结构的内置的重复检测机制，能够检测和消除由不同转换规则创建的重复的表达式。

(2)统计推导（Statistics Derivation)。在探测解释后，备忘录维护了给定查询的完整的逻辑空间。Orca 的统计推导机制被触发，统计对象主要是列直方图集合，用以估算基数以及数据倾斜。统计推导发生在紧凑的备忘录结构中，避免扩大搜索空间。为了对目标组进行统计，Orca 使用最高的承诺度（promise）来表达组表达式的可靠统计。统计承诺度计算是具体的。例如，一个包含少量连接条件的 Inn

erJoin 表达式的承诺度要比包含大量连接条件的 InnerJoin（当生成多个连接顺序时这种情况发生次数将增加）的承诺度要高。理由是因为随着连接条件的增加，预测出错将会被传播和放大的几率将增加。我们目前正在探索几种在备忘录结构中计算置信度分数的方法。

在挑选出最高承诺的组表达式后，Orca 将会在子组递归的触发统计推导。最后目标组的统计对象将组合子组的统计对象。

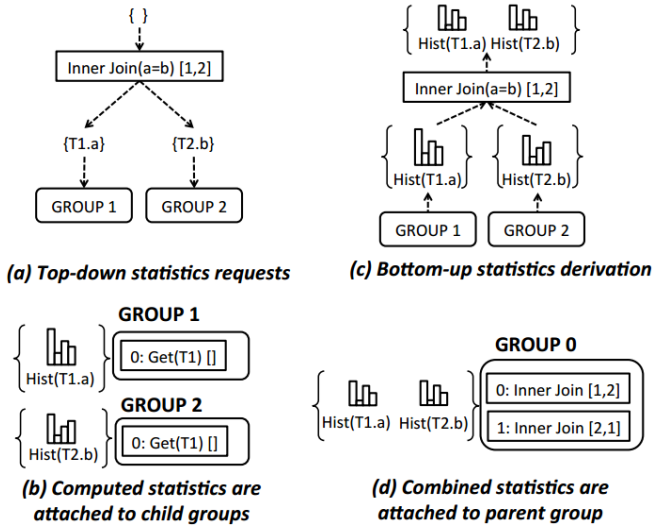


图 6-9 统计推导机制

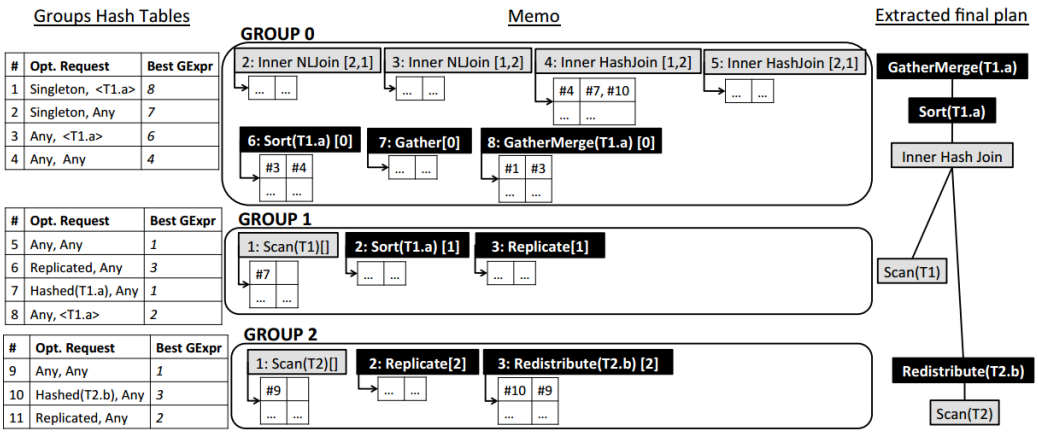
图 6-9 展示了统计推导机制的一个具体例子。首先，从父组表达式到子组请求统计进行自顶向下的传递。例如，InnerJoin(T1,T2) on (a=b)要求在 T1.a 和 T1.b 上的直方图。这个直方图可以从日志中获取。然后一个自顶向上组合子组的统计对象到父组的统计对象中。

（3）实施（Implementation）。创建逻辑表达式的物理实现的转换规则被触发。例如，get2scan 规则触发将逻辑 Get 变成物理表 Scan。

（4）优化（Optimization）。这一步中，属性约束被执行以及可选计划被计算。首先，提交一个初始的优化请求到根组，详细指出查

询要求，例如结果分布和排序次序。对于每个传入的请求，每个物理组表达式会相应传递对应的要求给子组，这个取决于到来的要求以及操作的本地要求。在优化的过程中，许多相同的要求可能会提交到同一个组中，为此，Orca 的缓存会将要求计算到 hash 表中，一个刚到来的要求只有当它在组 hash 表不存在中才会被计算。另外，每个物理组表达式维护一个本地 hash 表，用以将到来的要求映射到对应的只要求中。本地哈希表提供了从备忘录中提取物理计划时所使用的链接结构，这个我们将随后介绍。

图 6-10 显示了在备忘录中优化要求的一个实例。初始的优化要求是 req. #1: {Singleton, <T1.a>}，这就指定结果要按 T1.a 排序然后聚集到 master 节点。同时我们也展示了组 hash 表，其中每一个要求都与一个最佳组表达式（GExpr）相关联，Gexpr 就是满足该要求且代价最小的实现。图中黑框表示的是那些插入到备忘录中用来排序和数据分布的执行操作。Gather 操作表示将所有片段中元组收集到 master 节点。GatherMerge 操作表示将已排好序的元组从片段汇集到 master 节点。而 Redistribute 操作则是将数据按指定要求重分布。



#	Opt. Request	Best GExpr
1	Singleton, <T1.a>	8
2	Singleton, Any	7
3	Any, <T1.a>	6
4	Any, Any	4

#	Opt. Request	Best GExpr
5	Any, Any	1
6	Replicated, Any	3
7	Hashed(T1.a), Any	1
8	Any, <T1.a>	2

#	Opt. Request	Best GExpr
9	Any, Any	1
10	Hashed(T2.b), Any	3
11	Replicated, Any	2

图 6-10 在备忘录中处理优化请求

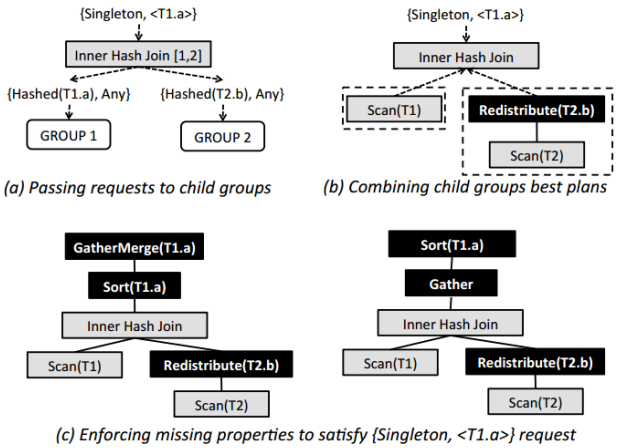


图 6-11 生成 InnerHashJoin 的可选计划

图 6-11 显示了通过 InnerHashJoin[1,2]优化要求 req.#1。对于这个要求，一个可选的计划是基于连接条件矫正孩子分布（当然还有其他实现方法）。这个可以通过 Group 1 的 Hashed(T1.a)分布和 Group 2 的 Hashed(T2.a)分布实现。两个 Group 要求的排序方式均为 Any。可以看到，对 Group 2 最好的计划是对 T2.b 建立 hash 分布，因为原来是对 T2.a 建立 hash 的，而对 Group 1 最好的计划是简单的 Scan，因为 T1 原本就是对 T1.a 建立 hash 的。

当确定传递的属性不满足初始的要求时，不满足的属性必须被执行。在 Orca 中属性执行是一个非常灵活的框架，它允许每个操作基于子计划和操作本地行为所传递的属性来定义执行要求的属性的行为。比如对于一个排序操作 NL Join，如果这个操作早就传递到它的孩子，那么在连接的上面就可能没必要执行排序操作。

图 6-11（c）显示了通过属性执行满足 req.#1 的两种可能计划。左边那个是在片段上对结果排序然后汇集到 master 节点，而右边那个则是先汇集到 master 节点再进行排序。这些不同的可选计划最终在备忘录中执行，而他们的代价则由代价模型决定。

接下来基于给定优化要求的链接结构可以从备忘录中抽取最佳的计划。图 6-10 展示了计划抽取过程。首先我们找到根组最佳表达式是 8，对应实现是 GatherMerge 操作。而 GatherMerge 本地 hash 表中对应的子要求是 req.#3。而 req.#3 的最佳组表达式是 Sort，所以将 GatherMerge 与 Sort 相连接。同理，而 Sort 本地 hash 表中对应的子要求是 req.#4。而 req.#4 的最佳组表达式是 InnerHashJoin，所以将 Sort 与 InnerHashJoin 相连接。继续该过程可以完成计划抽取，最后的结果可在图 6-10 右侧看到。最后抽取出的结果序列化成 DXL 格式并且传送到数据库系统执行。这就是整个计划优化的过程。

6.2 Greenplum 中的分布式查询执行

该部分重点介绍 Greenplum 中的查询执行器以及其 Greenplum 是如何并行执行查询计划

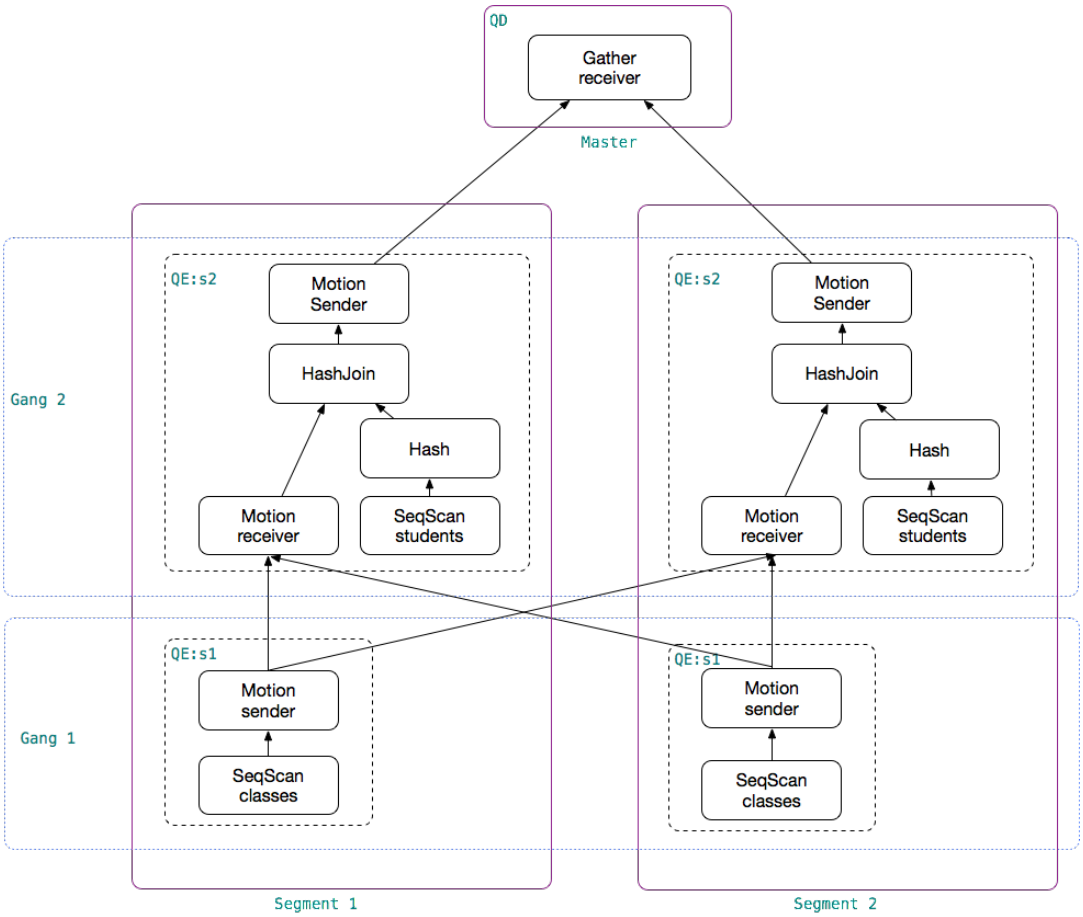
6.2.1 查询执行器相关概念

我们先看一个 SQL 例子及其计划：

```
test=# CREATE TABLE students (id int, name text) DISTRIBUTED BY (id);
test=# CREATE TABLE classes(id int, classname text, student_id int) DISTRIBUTED BY (id);
test=# INSERT INTO students VALUES (1, 'steven'), (2, 'changchang'),
```

```
 (3, 'guoguo');
test=# INSERT INTO classes VALUES (1, 'math', 1), (2, 'math', 2), (3,
 'physics', 3);

test=# explain SELECT s.name student_name, c.classname
test-# FROM students s, classes c
test-# WHERE s.id=c.student_id;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Gather Motion 2:1  (slice2; segments: 2)  (cost=2.07..4.21 rows=4
width=14)
   ->  Hash Join  (cost=2.07..4.21 rows=2 width=14)
         Hash Cond: c.student_id = s.id
         ->  Redistribute Motion 2:2  (slice1; segments: 2)  (cost
=0.00..2.09 rows=2 width=10)
           Hash Key: c.student_id
         ->  Seq Scan on classes c  (cost=0.00..2.03 rows=2 widt
h=10)
         ->  Hash  (cost=2.03..2.03 rows=2 width=12)
         ->  Seq Scan on students s  (cost=0.00..2.03 rows=2 wid
th=12)
 Optimizer status: legacy query optimizer
```



这个图展示了上面例子中的 SQL 在 2 个 segment 的 Greenplum 集群中执行时的示意图。

QD（Query Dispatcher、查询调度器）：Master 节点上负责处理用户查询请求的进程称为 QD（PostgreSQL 中称之为 Backend 进程）。QD 收到用户发来的 SQL 请求后，进行解析、重写和优化，将优化后的并行计划分发给每个 segment 上执行，并将最终结果返回给用户。此外还负责整个 SQL 语句涉及到的所有的 QE 进程间的通讯控制和协调，譬如某个 QE 执行时出现错误时，QD 负责收集错误详细信息，

并取消所有其他 QEs；如果 LIMIT n 语句已经满足，则中止所有 QE 的执行等。QD 的入口是 exec_simple_query()。

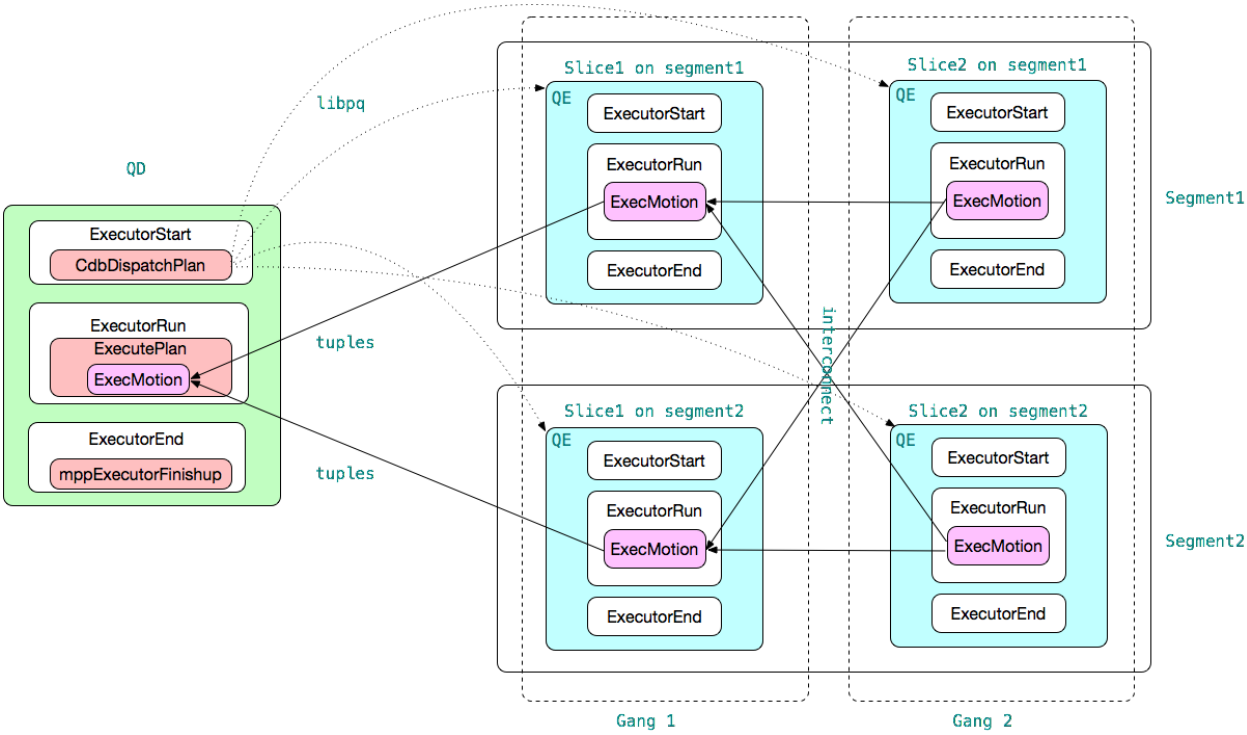
QE（Query Executor、查询执行器）：Segment 上负责执行 QD 分发来的查询任务的进程称为 QE。Segment 实例运行的也是一个 PostgreSQL，所以对于 QE 而言，QD 是一个 PostgreSQL 的客户端，它们之间通过 PostgreSQL 标准的 libpq 协议进行通讯。对于 QD 而言，QE 是负责执行其查询请求的 PostgreSQL Backend 进程。通常 QE 执行整个查询的一部分（称为 Slice）。QE 的入口是 exec_mpp_query()。

Slice：为了提高查询执行并行度和效率，Greenplum 把一个完整的分布式查询计划从下到上分成多个 Slice，每个 Slice 负责计划的一部分。划分 slice 的边界为 Motion，每遇到 Motion 则一刀将 Motion 切成发送方和接收方，得到两颗子树。每个 slice 由一个 QE 进程处理。上面例子中一共有三个 slice。

Gang：在不同 segments 上执行同一个 slice 的所有 QEs 进程称为 Gang。上例中有两组 Gang，第一组 Gang 负责在 2 个 segments 上分别对表 classes 顺序扫描，并把结果数据重分布发送给第二组 Gang；第二组 Gang 在 2 个 segments 上分别对表 students 顺序扫描，与第一组 Gang 发送到本 segment 的 classes 数据进行哈希关联，并将最终结果发送给 Master。

6.2.2 并行执行流程

下图展示了查询在 Greenplum 集群中并行执行的流程。该图假设有 2 个 segments，查询计划有两个 slices，一共有 4 个 QEs，它们之间通过网络进行通讯。



QD 和 QE 都是 PostgreSQL backend 进程，其执行逻辑非常相似。对于数据操作（DML）语句（数据定义语句的执行逻辑更简单），其核心执行逻辑由 ExecutorStart, ExecutorRun, ExecutorEnd 实现。

QD：

- ExecutorStart 负责执行器的初始化和启动。Greenplum 通过 C

dbDispatchPlan 把完整的查询计划发送给每个 Gang 中的每个 QE 进程。Greenplum 有两种发送计划给 QE 的方式：1）异步方式，使用 libpq 的异步 API 以非阻塞方式发送查询计划给 QE；2）同步多线程方式：使用 libpq 的同步 API，使用多个线程同时发送查询计划给 QE。GUC gp_connections_per_thread 控制使用线程数量，缺省值为 0，表示采用异步方式。

- ExecutorRun 启动执行器，执行查询树中每个算子的代码，并以火山模型（volcano）风格返回结果元组给客户端。在 QD 上，ExecutorRun 调用 ExecutePlan 处理查询树，该查询树的最下端的节点是一个 Motion 算子。其对应的函数为 ExecMotion，该函数等待来自于各个 QE 的结果。QD 获得来自于 QE 的元组后，执行某些必要操作（譬如排序）然后返回给最终用户。
- ExecutorEnd 负责执行器的清理工作，包括检查结果，关闭 interconnect 连接等。

QE 上的 ExecutorStart/ExecutorRun/ExecutorEnd 函数和单节点的 PostgreSQL 代码逻辑类似。主要的区别在 QE 执行的是 Greenplum 分布式计划中的一个 slice，因而其查询树的根节点一定是个 Motion 节点。其对应的执行函数为 ExecMotion，该算子从查询树下部获得元组，并根据 Motion 的类型发送给不同的接收方。低一级的 Gang 的 QE 把 Motion 节点的结果元组发送给上一级 Gang 的 QE，最顶层 Gang 的 QE 的 Motion 会把结果元组发送给 QD。Motion 的 Flow 类型确定了数据传输的方式，有两种：广播和重分布。广播方式

将数据发送给上一级 Gang 的每一个 QE；重分布方式将数据根据重分布键计算其对应的 QE 处理节点，并发送给该 QE。

QD 和 QE 之间有两种类型的网络连接：

- libpq：QD 通过 libpq 与各个 QE 间传输控制信息，包括发送查询计划、收集错误信息、处理取消操作等。libpq 是 PostgreSQL 的标准协议，Greenplum 对该协议进行了增强，譬如新增了‘M’消息类型（QD 使用该消息发送查询计划给 QE）。libpq 是基于 TCP 的。
- interconnect：QD 和 QE、QE 和 QE 之间的表元组数据传输通过 interconnect 实现。Greenplum 有两种 interconnect 实现方式，一种基于 TCP，一种基于 UDP。缺省方式为 UDP interconnect 连接方式。

Direct Dispatch 优化

有一类特殊的 SQL，执行时只需要单个 segment 执行即可。譬如主键查询：SELECT * FROM tbl WHERE id = 1;

为了提高资源利用率和效率，Greenplum 对这类 SQL 进行了专门的优化，称为 Direct Dispatch 优化：生成查询计划阶段，优化器根据分布键和 WHERE 子句的条件，判断查询计划是否为 Direct Dispatch 类型查询；在执行阶段，如果计划是 Direct Dispatch，QD 则只会把计划发送给需要执行该计划的单个 segment 执行，而不是发送给所

有的 segments 执行。

6.2.3 执行模式

Greenplum 数据库所有的节点分为 Master 节点和 Segment 节点两种，Master 节点负责全局调度而不负责具体的数据存储和 SQL 语句执行，当查询计划来临，Master 节点将任务分配到 Segment 节点，然后等待各个 Segment 节点执行完毕将结果汇总返回给用户。基于这种结构，Greenplum 中分为三种分布式的执行模式：

1) Gather Motion （N:1）

   聚合操作，故名思议就是 Master 节点将所有 segment 节点的数据聚合起来，这是一种多对一的模式。这时每个节点操作所需数据都在本地，不需要从其他节点获取数据，Master 节点只需接收数据即可，接收顺序是先到先放，也就是说同样的语句多次执行返回的数据顺序可能不同。

2) Broadcast Motion（N：N）

   广播，将每个 segment 节点上某一张表的数据全部发送给其他节点，换句话是每个 segment 节点拥有该表的完整数据，这样与该表有关的所有操作都可以在本地完成，广播通常只会发生在表的关联操作，是一种多对多的执行模式。

3) Redistribute Motion（N：N）

   重分布，与广播一样也是一种解决数据分布的方法，通常当数据不能满足广播条件，或者广播消耗太大时作为一种替代

方案解决数据分布问题。重分布时将一张表重新按照新的属性键进行分布，也是一种多对多的执行模式。

6.2.4 数据广播与重分布

Greenplum 是分布式数据库，采用 share-nothing 架构，也就是各个 segment 节点不能相互对方的内存或者磁盘，但是数据在存储的时候采用分布式策略，同一张表的数据都是按照分布键均匀的分布在所有的节点之中，当我们需要对全表进行复杂 SQL 操作时，每个单独的节点无法包含操作所需要的所有数据。这时为了 SQL 语句的顺利进行需要对操作数据进行迁移，Greenplum 中目前支持两种迁移方式：广播和重分布。

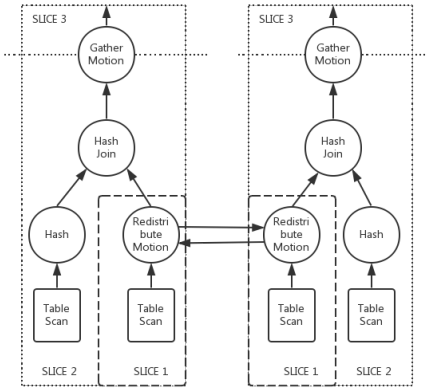


图 6-4 数据分布实例

Greenplum 中涉及广播和重分布时会专门创建一个切片（slice），每一个切片都会在对应的数据节点上发起进程用于处理该切边负责的数据，上层负责读取下层 slice 广播或者重分布的数据，然后进行计算。如图 6-4 展示的是 Hash Join 执行过程的数据分布情况，由于一张表的数据分布与 Join 键值不匹配，需要进行数据的重分布，因而

创建了一个切片 SLICE1 用于处理这部分数据，实际上是将表的所有数据按照 Join 的键值进行重新的分配，这样可以 Join 的数据全部在同一个节点之上，就可以并行的执行 SQL 查询。

为了更好的说明究竟该采用哪种方式迁移数据，首先我们定义两张表 A,B。两张表都包含两个同名字段 id、id2，分布键都选择 id，A 的数据量是 M，B 的数据量是 N，以简单的内连接作为分析实例

1) 如果 SQL 是 select * from A,B where A.id = B.id，此时分布键与关联键相同，不需要进行广播或者重分布，各个节点只需利用本地数据。
2) 如果 SQL 是 select * from A,B where A.id = B.id2，此时 A 表是按照分布键进行关联的不涉及广播和重分布，B 表是按照非分布键关联的，需要进行数据广播或者重分布。可以将 B 表按照 id2 字段将数据重新进行数据分布，然后就可以与 A 表进行关联，重分布的数据量是 N。或者将 A 表广播，这样每一个节点都可以获得 A 的全部数据，然后与 B 关联，此时数据量是 M×节点数。所以当 N>M×节点数，选择 A 表广播，否则选择 B 表重分布。
3) 如果 SQL 是 select * from A,B where A.id2 = B.id2，两张表的分布键与关连键都不相同，此时也有两种做法：一种是 A 表和 B 表都按照 id2 字段进行重分布，重分布的代价是 M+N；另外一种是将一张表广播然后再关联，广播时会选取较小的表，代价是 min（M,N）×节点数。当 M+N>min（M,N）×节点

数，选择小表广播，否则选择重分布。

6.2.5 Greenplum 分析函数执行

Greenplum 不仅完美支持标准 SQL 语言，对于一些分析型语句也提供良好支持，这一章节将介绍窗口函数，窗口函数使得对数据的操作不再局限于单个元组而是一组特定范围的元组，又不同于 GROUP BY 子句只返回一个结果，窗口函数可以针对每一个元组返回它在整个组中的计算结果。窗口函数可以实现诸如排名（rank）、百分比（percentiles）、移动均值（moving average）、最大值（MAX）、最小值（MIN）以及累积求和（cumulative sums）等相对复杂的操作，克服了传统的相关子查询的缺点，而且语法简单，降低了对于数据处理编程能力的要求。窗口函数的通用格式如下所示：

```
Agg_func(expression)OVER(
[PARTITION BY expr_list]
[ORDER BY order_list [fram_clause]])
```

其中，Agg_func 是一个普通的聚集函数（例如 SUM、AVG 等），但在窗口函数中，聚集函数只作用于当前的窗口，并为每一个元组返回聚集的结果。OVER 子句定义了窗口，并通过另外三个子句描述窗口的详细内容。

- PARTITION BY 子句定义了窗口的分区，所谓分区就是指具有相同指定属性值的元组划分在一起。
- ORDER BY 子句定义窗口的重排序，排序的范围被限

定在 PARTITION 所指定的分区内部，每个分区各自排序，互相不影响。

- frame_clause 定义了边框，该子句如果缺省则会默认每一个由 PARTITION 定义的分区即为一个边框。frame_clause 的基本形式是 ROWS/RANGE BETWEEN p_value AND f_value，其中 p_value 定义了边框头与当前元组的位置关系，f_value 定义了边框尾和当前元组的关系。目前主要有三种不同的定义模式 UNBOUNDED、CURRENT 和 value PRECEDING/FOLLOWING。

窗口函数拥有众多不同的计算函数，但是函数的执行流程大致相同。首先要经过 PARTITION 和 ORDER 阶段，将拥有相同 PARTITION 值的元组划分到一起，在每个划分内部按照 ORDER BY 的值进行排序，不同分区的排序过程互不影响，然后是选择相应的执行策略计算窗口函数。

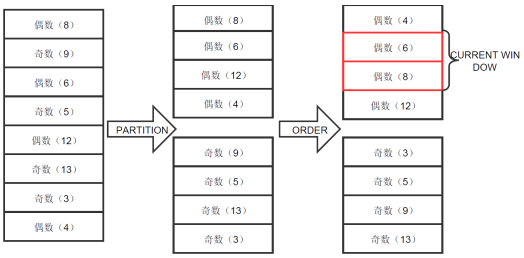


图 6-5 窗口函数的执行过程

如图 6-5 所示，首先整张表按照奇偶属性进行 PARTITION，所有元组被分成两个分区，然后在每个分区内部按照 ORDER BY 属性值的大小排序，最后通过边框定义子句确定当前边框大小，当所有的窗口边界被确定下来以后，便会选择合适的执行策略进行窗口函数的计算。

需要注意的是，ORDER BY 属性值与数据分布可能不一致，这时又会涉及到数据的广播或者重分布，具体策略选择前一小节已经讨论过。

6.3 Greenplum 源码修改实例

该部分我们将详细介绍对 GP 的查询优化器进行改进，增加两种新的功能：

- select-from 子句去除冗余表。
- 新的多表连接算法——贪心算法

6.3.1 select-from 子句优化

GP 对于 select-from 子句中冗余的表，也会做笛卡尔积。所谓冗余的表是指，select 中并不需要从这些的表中获得投影列，并且也不是 where 条件中的 join 条件。但是，这些表却由于用户失误或者粗心导致 from 后面添加了一个或者很多个这样的表。目前，GP 并不能对这种情形进行优化，于是 GP 的处理过程是将没有 join 条件的表生成使用笛卡尔积的 join 算子。这样数据库凭空多消耗一些资源用于处理这部分信息，而我们的实验目标就是要从查询语句中去除这些冗余的表。

select-from 子句之所以会做笛卡尔积是由于多表 join 的时候并不考虑目标列和 where 条件，from 子句后面存在多少个表，就会将多少个表 join 成一张表。因此，我们的方案是在 join 多表前，预先处理 select、from 和 where 子句包含的表，若检查到 from 后面的表

并没有在 select 和 from 后面出现，我们就从 fromlist 中移除掉该表，这样 join 时就不会有该表参与。但是我们仍然保留该表在所有的基表中，因为由于这些范围表的顺序 id 已经确定，并被使用，贸然移除，导致额外的麻烦，因此冗余这些信息。实际处理位置在，在语义分析阶段的 transformSelectStmt()函数中在处理 where 子句函数 transformWhereClause()调用完之后。目前，实现时考虑到 where 子句的的表达式的复杂表示结构，隐含的连接关系，因此我们暂时

```
postgres=# explain select t1.id,t1.t_id,t1.info,t2.t_id,t2.info from t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Gather Motion 4:1  (slice2; segments: 4)  (cost=4.07..11.31 rows=36 width=22)
   ->  Nested Loop  (cost=4.07..11.31 rows=9 width=22)
         ->  Broadcast Motion 4:4  (slice1; segments: 4)  (cost=0.00..4.36 rows=6 width=9)
               ->  Seq Scan on t2  (cost=0.00..4.06 rows=2 width=9)
         ->  Materialize  (cost=4.07..4.13 rows=2 width=13)
               ->  Seq Scan on t1  (cost=0.00..4.06 rows=2 width=13)
(6 rows)
```

不处理含有 where 子句的 select-from 子句。

详细的代码参见附录二，从实验截图中可以发现，select 语句中只会涉及 t1 和 t2 两张表的属性，其他表皆为冗余的表，通过冗余的去除我们的查询计划只会涉及有效表，t3-t11 的表被自动去除。

6.3.2 多表连接——贪心算法

GP 当中对于多表连接使用动态规划的算法生成最优的查询路径，其核心思路如下：

1) 初始状态为每一个待连接关系生成基本的访问路径，选出最优的路径。这些关系被称为第 1 层中间关系。把所有的由 n 个关系连接生成的中间关系成为 n 层关系。
2) 已知第 1~n-1 层的路径，将第 n-1 层的关系与每个第 i 层的关系连

接，计算当前情况下的最优方法，其中 1≤i≤n/2.

3) 生成第 n 层的结果后选出第 n 层的最优路径作为结果。

换句话说，在路径生成的过程当中，每生成一个中间关系就估算其代价，供上层的路径计算代价。具体实现当中，不能精确判断出中间结果的路径当中那个是最优的，所以需要尽量保留可能的最优路径，因而动态规划的算法对于连接规模非常敏感，当涉及的连接表数目增加就会造成优化计划生成时间相对比较漫长。

多表连接的贪心算法，尽量利用了原有的动态规划的代码中代价计算模型。通过改变的多表的 join 构建方式，每次选择代价最小的两个连接方法，将它们连接之后最为一个新的待选表放到列表当中。重复选择代价最小的根据 my_join_search_one_level()生成的 joined 表，然后和之前的不在 joined 表中的表重新生成一个需要 join 的 list，list 长度相比之前会少一。如此循环，最后得到最后的代价最小的连接方式。my_join_search_one_level()的作用是根据需要 join 表的 list 构建两两连接的表供 my_join_search()选择，具体的两个函数的实现参见附录二。 贪心算法最大的优势在表连接规模比较大时能够比较快速的选择出一条相对比较优的查询路径，不需要像动态规划算法那样保留很多中间结果。但是该方法的问题也非常明显，选择出来的路径并非全局最优的，执行时可能需要花费更多的执行时间。

6.4 查询执行实验导引

6.4.1 查询执行分析实例

Greenplum 中提供了用于查看执行计划的 explain 命令，其语法形式如下：

EXPLAIN [ANALYSE][VERBOSE] statement

各个参数含义如下：

- ANALYSE:执行命令并显示实际运行时间。
- VERBOSE：显示规划树完整的内部表现形式，而不仅是一个摘要。常在调试过程中有用。
- statement：查询执行的 sql 语句。

在该部分我们将通过查看执行计划，详细分析在不同数据量和 SQL 查询下，广播或者重分布策略的选择。实验平台是使用三台虚拟机搭建包含一个 Master 节点和两个 Segment 节点的 Greenplum 数据库。

1）内连接

情况 1：select * from A,B where A.id = B.id

分布键与关联键相同，不会造成广播或重分布，这种情况过于简单，故没有进行测试

情况 2：select * from A,B where A.id=B.id2

在下图的测试中，A 表（test005）共有 10000 条数据，有 2 个 segment，如果对 A 表进行广播，代价是 20000 条数据；B 表共有 19000 条数据，选择将 B 表重分布，代价是 19000 条数据，后者代价

较小，故优先后者。

```
testdb=# explain select * from test005,test006 where test005.id = test006.id2;
                                        QUERY PLAN
------------------------------------------------------------------------------
------------------
 Gather Motion 2:1  (slice2; segments: 2)  (cost=237.00..1001.50 rows=10000 widt
h=16)
   ->  Hash Join  (cost=237.00..1001.50 rows=5000 width=16)
         Hash Cond: test006.id2 = test005.id
         ->  Redistribute Motion 2:2  (slice1; segments: 2)  (cost=0.00..592.00
rows=9500 width=8)
               Hash Key: test006.id2
               ->  Seq Scan on test006  (cost=0.00..212.00 rows=9500 width=8)
         ->  Hash  (cost=112.00..112.00 rows=5000 width=8)
               ->  Seq Scan on test005  (cost=0.00..112.00 rows=5000 width=8)
(8 rows)
```

在下图的测试中，A 表（test005）共有 10000 条数据，有 2 个 segment，如果对 A 表进行广播，代价是 20000 条数据；B 表共有 21000 条数据，选择将 B 表重分布，代价是 21000 条数据，前者代价较小，故选择广播。

```
testdb=# explain select * from test005,test007 where test005.id = test007.id2;
                                        QUERY PLAN
------------------------------------------------------------------------------
----------------
 Gather Motion 2:1  (slice2; segments: 2)  (cost=496.50..1108.50 rows=10000 widt
h=16)
   ->  Hash Join  (cost=496.50..1108.50 rows=5000 width=16)
         Hash Cond: test005.id = test007.id2
         ->  Broadcast Motion 2:2  (slice1; segments: 2)  (cost=0.00..412.00 row
s=10000 width=8)
               ->  Seq Scan on test005  (cost=0.00..112.00 rows=5000 width=8)
         ->  Hash  (cost=234.00..234.00 rows=10500 width=8)
               ->  Seq Scan on test007  (cost=0.00..234.00 rows=10500 width=8)
(7 rows)
```

2）左连接

情况 1：select * from A left join B on A.id=B.id

不涉及数据库的跨库关联，较简单，不需要进行数据迁移

情况 2：select * from A left join B on A.id2=B.id

左表的分布键是关联键，左连接的性质要求必须将表 B 重分布。

```
testdb=# Explain select * from test008 left join test005 on test008.id = test005
.id2 ;
                                          QUERY PLAN

------------------------------------------------------------------------------
------------------------
 Gather Motion 2:1  (slice2; segments: 2)  (cost=437.00..911.50 rows=19000 width
=16)
   ->  Hash Left Join  (cost=437.00..911.50 rows=9500 width=16)
         Hash Cond: test008.id = test005.id2
         ->  Seq Scan on test008  (cost=0.00..212.00 rows=9500 width=8)
         ->  Hash  (cost=312.00..312.00 rows=5000 width=8)
               ->  Redistribute Motion 2:2  (slice1; segments: 2)  (cost=0.00..3
12.00 rows=5000 width=8)
                     Hash Key: test005.id2
                     ->  Seq Scan on test005  (cost=0.00..112.00 rows=5000 width
=8)
(8 rows)
```

情况 3：select * from A left join B on A.id=B.id2

左表的分布键不是关联键，由于左连接不能将 A 表广播，有两种方式

1.将表 A 按照 id2 重分布，代价为 M

2.将表 B 广播，代价为 N*节点数

在下图的测试中，A 表（test005）共有 19000 条数据，如果将 A 表重分布，代价是 19000，B 表共有 10000 条数据，2 个 segment，如果将 B 表进行广播，代价是 20000，所以选择将 A 表重分布。

```
testdb=# Explain select * from test008 left join test005 on test008.id2 = test00
5.id ;
                                        QUERY PLAN
------------------------------------------------------------------------------
------------------
 Gather Motion 2:1  (slice2; segments: 2)  (cost=237.00..1091.50 rows=19000 widt
h=16)
   ->  Hash Left Join  (cost=237.00..1091.50 rows=9500 width=16)
         Hash Cond: test008.id2 = test005.id
         ->  Redistribute Motion 2:2  (slice1; segments: 2)  (cost=0.00..592.00
rows=9500 width=8)
               Hash Key: test008.id2
               ->  Seq Scan on test008  (cost=0.00..212.00 rows=9500 width=8)
         ->  Hash  (cost=112.00..112.00 rows=5000 width=8)
               ->  Seq Scan on test005  (cost=0.00..112.00 rows=5000 width=8)
(8 rows)
```

在下图的测试中，A 表（test005）共有 21000 条数据，如果将 A 表重分布，代价是 21000，B 表共有 10000 条数据，2 个 segment，如果将 B 表进行广播，代价是 20000，所以选择将 B 表广播。

```
testdb=# Explain select * from test009 left join test005 on test009.id2 = test00
5.id;
                                          QUERY PLAN
------------------------------------------------------------------------------
----------------------
 Gather Motion 2:1  (slice2; segments: 2)  (cost=662.00..1208.50 rows=21000 widt
h=16)
   ->  Hash Left Join  (cost=662.00..1208.50 rows=10500 width=16)
         Hash Cond: test009.id2 = test005.id
         ->  Seq Scan on test009  (cost=0.00..234.00 rows=10500 width=8)
         ->  Hash  (cost=412.00..412.00 rows=10000 width=8)
               ->  Broadcast Motion 2:2  (slice1; segments: 2)  (cost=0.00..412.
00 rows=10000 width=8)
                     ->  Seq Scan on test005  (cost=0.00..112.00 rows=5000 width
=8)
(7 rows)
```

3）全连接

情况 1：select * from A full outer join B on A.id = B.id

在 greenplum 中，全连接只能采用 merge join 实现

```
testdb=# explain select * from test005 full outer join test006 on test005.id = t
est006.id;
                                    QUERY PLAN
-------------------------------------------------------------------------------
-------
 Gather Motion 2:1  (slice1; segments: 2)  (cost=2338.69..2583.69 rows=19000 wid
th=16)
   ->  Merge Full Join  (cost=2338.69..2583.69 rows=9500 width=16)
         Merge Cond: test005.id = test006.id
         ->  Sort  (cost=776.39..801.39 rows=5000 width=8)
               Sort Key: test005.id
               ->  Seq Scan on test005  (cost=0.00..112.00 rows=5000 width=8)
         ->  Sort  (cost=1562.30..1609.80 rows=9500 width=8)
               Sort Key: test006.id
               ->  Seq Scan on test006  (cost=0.00..212.00 rows=9500 width=8)
(9 rows)
```

情况 2：select * from A full outer join B on A.id = B.id2

为了实现全连接，不能将表广播，只能重分布。

```
testdb=# explain select * from test005 full outer join test006 on test005.id = t
est006.id2;
                                         QUERY PLAN
-------------------------------------------------------------------------------
------------------------
 Gather Motion 2:1  (slice2; segments: 2)  (cost=2718.69..2963.69 rows=19000 wid
th=16)
   ->  Merge Full Join  (cost=2718.69..2963.69 rows=9500 width=16)
         Merge Cond: test005.id = test006.id2
         ->  Sort  (cost=776.39..801.39 rows=5000 width=8)
               Sort Key: test005.id
               ->  Seq Scan on test005  (cost=0.00..112.00 rows=5000 width=8)
         ->  Sort  (cost=1942.30..1989.80 rows=9500 width=8)
               Sort Key: test006.id2
               ->  Redistribute Motion 2:2  (slice1; segments: 2)  (cost=0.00..5
92.00 rows=9500 width=8)
                     Hash Key: test006.id2
                     ->  Seq Scan on test006  (cost=0.00..212.00 rows=9500 width
=8)
(11 rows)
```

情况 3：select * from A full outer join B on A.id2 = B.id2

同上，为了实现全连接，不能将表广播，只能重分布。

```
testdb=# explain select * from test005 full outer join test006 on test005.id2 =
test006.id2;
                                          QUERY PLAN
------------------------------------------------------------------------------
------------------------
 Gather Motion 2:1  (slice3; segments: 2)  (cost=2918.69..3163.69 rows=19000 wid
th=16)
   ->  Merge Full Join  (cost=2918.69..3163.69 rows=9500 width=16)
         Merge Cond: test005.id2 = test006.id2
         ->  Sort  (cost=976.39..1001.39 rows=5000 width=8)
               Sort Key: test005.id2
               ->  Redistribute Motion 2:2  (slice1; segments: 2)  (cost=0.00..3
12.00 rows=5000 width=8)
                     Hash Key: test005.id2
                     ->  Seq Scan on test005  (cost=0.00..112.00 rows=5000 width
=8)
         ->  Sort  (cost=1942.30..1989.80 rows=9500 width=8)
               Sort Key: test006.id2
               ->  Redistribute Motion 2:2  (slice2; segments: 2)  (cost=0.00..5
92.00 rows=9500 width=8)
                     Hash Key: test006.id2
                     ->  Seq Scan on test006  (cost=0.00..212.00 rows=9500 width
=8)
```

因此，在 greenplum 中，对于选择广播还是重分布，是依据两种策略的代价来进行决策的。Greenplum 会对广播和重分布的代价分别进行估计，然后选择代价较小的策略。当然，由于左连接和全连接的限制，在某些情况下广播策略不能被采用。

6.4.2 查询执行实验要求

1、 理解查询执行中的数据广播和数据重分布，分析在何种情况下选择哪种策略，以具体实验验证；

2、 借助 explain 指令，分析 select、join、group 等常用 sql 语句的执行流程；

3、 查看相关文档理解 SQL 中窗口函数的语义，分析窗口函数的执行语义，以实验验证；

4、 查看 GP 代码，以简单 SELECT 语句为例，分析 SQL 语句执行逻辑，追踪函数的调用关系，只需分析代码目录下 excutor 中的代

码。

6.5 查考文献

[1]Selinger P G, Astrahan M M, Chamberlin D D, et al. Access path selection in a relational database management system[C]//Proceedings of the 1979 ACM SIGMOD international conference on Management of data. ACM, 1979: 23-34.

[2]Pirahesh H, Hellerstein J M, Hasan W. Extensible/rule based query rewrite optimization in Starburst[C]//ACM Sigmod Record. ACM, 1992, 21(2): 39-48.

[3] Ailamaki A, DeWitt D J, Hill M D, et al. DBMSs on a modern processor: Where does time go?[C]//VLDB" 99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK. 1999 (DIAS-CONF-1999-001): 266-277.

[4]Bajda-Pawlikowski K, Abadi D J, Silberschatz A, et al. Efficient processing of data warehousing queries in a split execution environment[C]//Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011: 1165-1176.

[5]Shankar S, Nehme R, Aguilar-Saborit J, et al. Query optimization in microsoft SQL server PDW[C]//Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012: 767-776.

[6] Vitorovic A, Elseidy M, Koch C. Load balancing and skew resilience for parallel joins[C]. ICDE, 2016.

[7] Trummer I, Koch C. Parallelizing query optimization on shared-nothing architectures[J]. Proceedings of the VLDB Endowment, 2016, 9(9): 660-671.

[8]Dey A, Bhaumik S, Harish D ,, et al. Efficiently approximating query optimizer plan diagrams[J]. Proceedings of the Vldb Endowment, 2008, 1(2):1325-1336.

[9]Chatziantoniou D, Ross K A. Groupwise Processing of Relational Queries[C]// International Conference on Very Large Data Bases. Morgan Kaufmann Publishers Inc. 1997:476--485.