Greenplum：基于 PostgreSQL 的分布式数据库内核揭秘

Greenplum 是最成熟的开源分布式分析型数据库（今年 6 月份预计发布的 Greenplum 6 之 OLTP 性能大幅提升，将成为一款真正的 HTAP 数据库，评测数据将于近期发布），Gartner 2019 最新评测显示 Greenplum 在经典数据分析领域位列全球第三，在实时数据分析领域位列并列第四。两个领域中前十名中唯一一款开源数据库产品。这意味着如果选择一款基于开源的产品，前十名中别无选择，唯此一款。Gartner 报告原文。

那么 Greenplum **分布式**数据库是如何炼成？众所周知 Greenplum 基于 PostgreSQL。PostgreSQL 是最先进的单节点数据库，其相关内核文档、论文资源很多。而有关如何将单节点 PostgreSQL 改造成分布式数据库的资料相对较少。本文从 6 个方面介绍将单节点 PostgreSQL 数据库发展成分布式 MPP 数据库所涉及的主要工作。当然这些仅仅是极简概述，做到企业级产品化耗资数亿美元，百人规模的数据库尖端人才团队十几年的研发投入结晶而成。

虽然不是必需，然而了解 PostgreSQL 基本内核知识对理解本文中的一些细节有帮助。Bruce Momjian 的 PPT 是极佳入门资料。

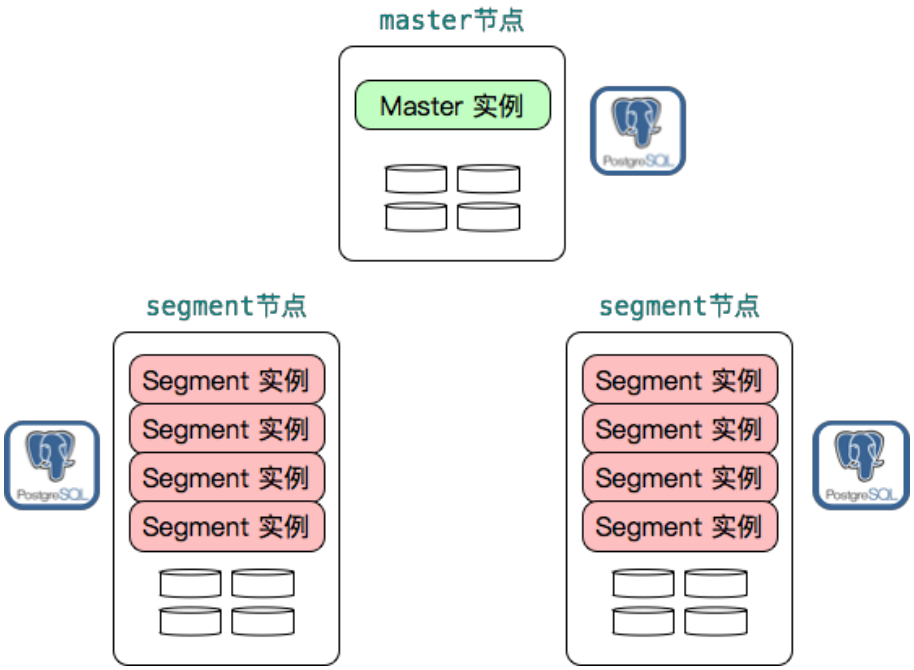
1. Greenplum 集群化概述

PostgreSQL 是世界上最先进的单机开源数据库。Greenplum 基于 PostgreSQL，是世界上最先进的开源 MPP 数据库 （有关 Greenplum 更多资讯请访问 Greenplum 中文社区)。从用户角度来看，Greenplum 是一个完备的关系数据库管理系统（RDBMS)。从物理层面，它内含多个 PostgreSQL 实例，这些实例可以单独访问。为了实现多个独立的 PostgreSQL 实例的分工和合作，呈现给用户一个逻辑的数据库，Greenplum 在不同层面对数据存储、计算、通信和管理进行了分布式集群化处理。Greenplum 虽然是一个集群，然而对用户而言，它封装了所有分布式的细节，为用户提供了单个逻辑数据库。这种封装极大的解放了开发人员和运维人员。

把单节点 PostgreSQL 转化成集群涉及多个方面的工作，本文主要介绍数据分布、查询计划并行化、执行并行化、分布式事务、数据洗牌（shuffle）和管理并行化等 6 个方面。

Greenplum 在 PostgreSQL 之上还添加了大量其他功能，例如 Append-Optimized 表、列存表、外部表、多级分区表、细粒度资源管理器、ORCA 查询优化器、备份恢复、高可用、故障检测和故障恢复、集群数据迁移、扩容、MADlib 机器学习算法库、容器化执行 UDF、PostGIS 扩展、GPText 套件、监控管理、集成 Kubernetes 等。

下图展示了一个 Greenplum 集群的俯瞰图，其中一个 master 节点，两个 segment 节点，每个 segment 节点上部署了 4 个 segment 实例以提高资源利用率。每个实例，不管是 master 实例还是 segment 实例都是一个物理上独立的 PostgreSQL 数据库。

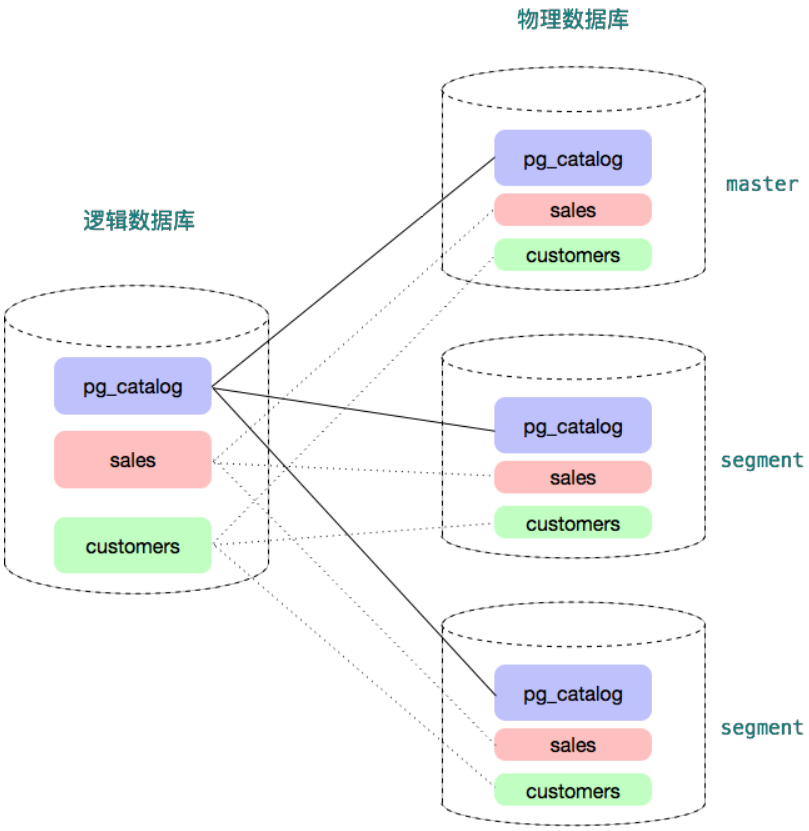


2. 分布式数据存储

数据存储分布化是分布式数据库要解决的第一个问题。分布式数据存储基本原理相对简单，实现比较容易，很多数据库中间件也可以做到基本的分布式数据存储。Greenplum 在这方面不单单做到了基本的分布式数据存储，还提供了很多更高级灵活的特性，譬如多级分区、多态存储。

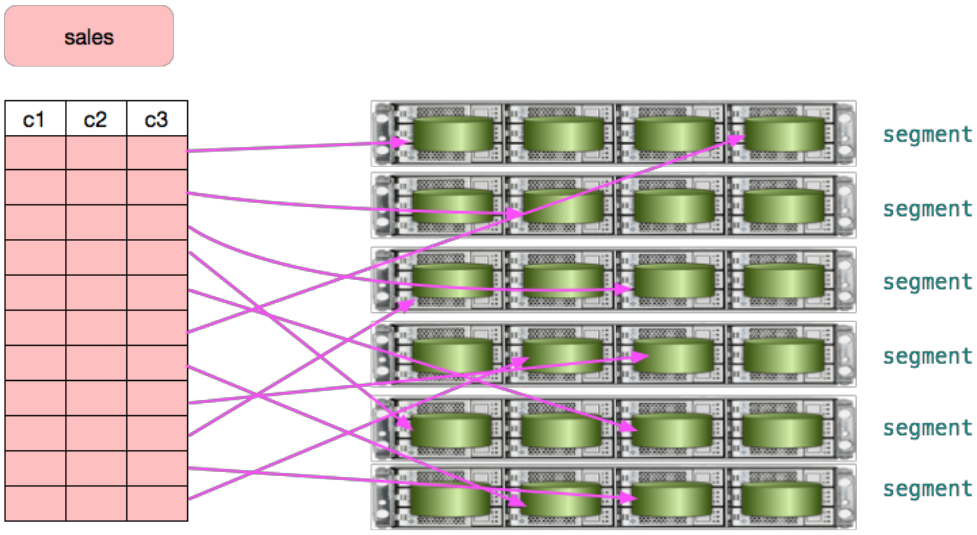
Greenplum 6 进一步增强了这一领域，实现了一致性哈希和复制表，并允许用户根据应用干预数据分布方法。

如下图所示，用户看到的是一个逻辑数据库，每个数据库有系统表（例如 pg_catalog 下面的 pg_class, pg_proc 等）和用户表（下例中为 sales 表和 customers 表）。在物理层面，它有很多个独立的数据库组成。每个数据库都有它自己的一份系统表和用户表。master 数据库仅仅包含元数据而不保存用户数据。master 上仍然有用户数据表，这些用户数据表都是空表，没有数据。优化器需要使用这些空表进行查询优化和计划生成。segment 数据库上绝大多数系统表（除了少数表，例如统计信息相关表）和 master 上的系统表内容一样，每个 segment 都保存用户数据表的一部分。



在 Greenplum 中，用户数据按照某种策略分散到不同节点的不同 segment 实例中。每个实例都有自己独立的数据目录，以磁盘文件的方式保存用户数据。使用标准的 INSERT SQL 语句可以将数据自动按照用户定义的策略分布到合适的节点，然而 INSERT 性能较低，仅适合插入少量数据。Greenplum 提供了专门的并行化数据加载工具以实现高效数据导入，详情可以参考 gpfdist

和 gpload 的官方文档。此外 Greenplum 还支持并行 COPY，如果数据已经保存在每个 segment 上，这是最快的数据加载方法。下图形象的展示了用户的 sales 表数据被分布到不同的 segment 实例上。



除了支持数据在不同的节点间水平分布，在单个节点上 Greenplum 还支持按照不同的标准分区，且支持多级分区。Greenplum 支持的分区方法有：

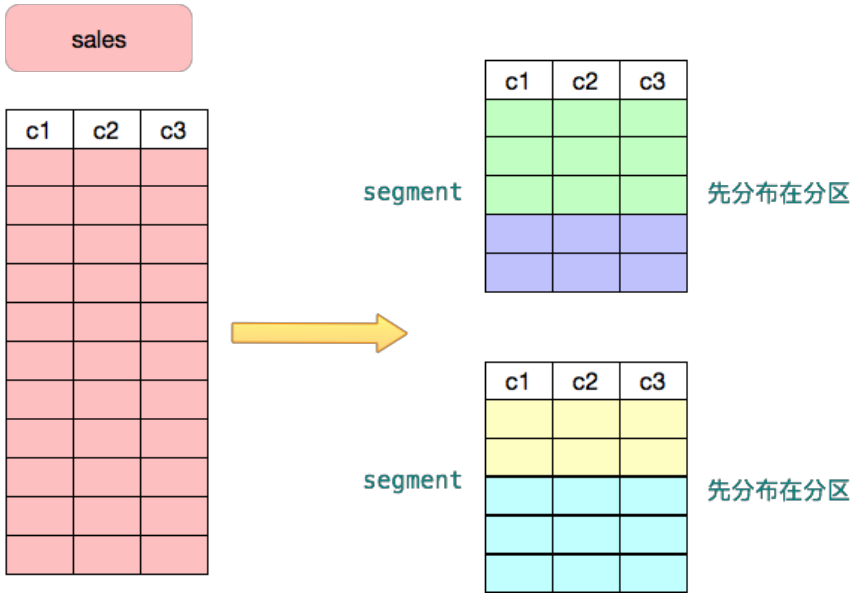
- 范围分区：根据某个列的时间范围或者数值范围对数据分区。譬如以下 SQL 将创建一个分区表，该表按天分区，从 2016-01-01 到 2017-01-01 把全部一年的数据按天分成了 366 个分区：

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( START (date '2016-01-01') INCLUSIVE
  END (date '2017-01-01') EXCLUSIVE
  EVERY (INTERVAL '1 day') );
```

- 列表分区：按照某个列的数据值列表，将数据分不到不同的分区。譬如以下 SQL 根据性别创建一个分区表，该表有三个分区：一个分区存储女士数据，一个分区存储男士数据，对于其他值譬如 NULL，则存储在单独 other 分区。

```
CREATE TABLE rank (id int, rank int, year int, gender char(1), count int )
DISTRIBUTED BY (id)
PARTITION BY LIST (gender)
( PARTITION girls VALUES ('F'),
  PARTITION boys VALUES ('M'),
  DEFAULT PARTITION other );
```

下图展示了用户的 sales 表首先被分布到两个节点，然后每个节点又按照某个标准进行了分区。分区的主要目的是实现分区裁剪以通过降低数据访问量来提高性能。分区裁剪指根据查询条件，优化器自动把不需要访问的分区过滤掉，以降低查询执行时的数据扫描量。PostgreSQL 支持静态条件分区裁剪，Greenplum 通过 ORCA 优化器实现了动态分区裁剪。动态分区裁剪可以提升十几倍至数百倍性能。



Greenplum 支持多态存储，即单张用户表，可以根据访问模式的不同使用不同的存储方式存储不同的分区。通常不同年龄的数据具有不同的访问模式，不同的访问模式有不同的优化方案。多态存储以用户透明的方式为不同数据选择最佳存储方式，提供最佳性能。Greenplum 提供以下存储方式:

- 堆表（Heap Table）：堆表是 Greenplum 的默认存储方式，也是 PostgreSQL 的存储方式。支持高效的更新和删除操作，访问多列时速度快，通常用于 OLTP 型查询。
- Append-Optimized 表：为追加而专门优化的表存储模式，通常用于存储数据仓库中的事实表。不适合频繁的更新操作。
- AOCO (Append-Optimized, Column Oriented) 表：AOCO 表为列表，具有较好的压缩比，支持不同的压缩算法，适合访问较少的列的查询场景。
- 外部表：外部表的数据存储在外部（数据不被 Greenplum 管理），Greenplum 中只有外部表的元数据信息。Greenplum 支持很多外部数据源譬如 S3、HDFS、文件、Gemfire、各种关系数据库等和多种数据格式譬如 Text、CSV、Avro、Parquet 等。

如下图所示，假设前面提到的 sales 表按照月份分区，那么可以采用不同的存储策略保存不同时间的数据，例如最近三个月的数据使用堆表（Heap）存储，更老的数据使用列存储，一年以前的数据使用外部表的方式存储在 S3 或者 HDFS 中。

数据分布是任何 MPP 数据库的基础，也是 MPP 数据库是否高效的关键之一。通过把海量数据分散到多个节点上，一方面大大降低了单个节点处理的数据量，另一方面也为处理并行化奠定了基础，两者结合起来可以极大的提高整个系统的性能。譬如在一百个节点的集群上，每个节点仅保存总数据量的百分之一，一百个节点同时并行处理，性能会是单个配置更强节点的几十倍。如果数据分布不均匀出现数据倾斜，受短板效应制约，整个系统的性能将会和最慢的节点相同。因而数据分布是否合理对 Greenplum 整体性能影响很大。

Greenplum 6 提供了以下数据分布策略。

- 哈希分布
- 随机分布
- 复制表（Replicated Table）

Hash 分布

哈希分布是 Greenlum 最常用的数据分布方式。根据预定义的分布键计算用户数据的哈希值，然后把哈希值映射到某个 segment 上。 分布键可以包含多个字段。分布键选择是否恰当是 Greenplum 能否发挥性能的主要因素。好的分布键将数据均匀分布到各个 segment 上，避免数据倾斜。

Greenplum 计算分布键哈希值的代码在 cdbhash.c 中。结构体 CdbHash 是处理分布键哈希的主要数据结构。 计算分布键哈希值的逻辑为：

- 使用 makeCdbHash(int segnum) 创建一个 CdbHash 结构体
- 然后对每个 tuple 执行下面操作，计算该 tuple 对应的哈希值，并确定该 tuple 应该分布到哪个 segment 上：
    - cdbhashinit()：执行初始化操作
    - cdbhash()，这个函数会调用 hashDatum() 针对不同类型做不同的预处理，最后 addToCdbHash() 将处理后的列值添加到哈希计算中
    - cdbhashreduce() 映射哈希值到某个 segment

CdbHash 结构体：

```
typedef struct CdbHash
{
   uint32    hash;                /* 哈希结果值 */
   int       numsegs;             /* segment 的个数 */
   CdbHashReduce reducealg;       /* 用于减少桶的算法 */
   uint32    rrindex;             /* 循环索引 */
} CdbHash;
```

主要的函数

- makeCdbHash(int numsegs): 创建一个 CdbHash 结构体，它维护了以下信息：
    - Segment 的个数
    - Reduction 方法
        - 如果 segment 个数是 2 的幂，则使用 REDUCE_BITMASK，否则使用 REDUCE_LAZYMOD.
    - 结构体内的 hash 值将会为每个 tuple 初始化，这个操作发生在 cdbhashinit() 中。
- void cdbhashinit(CdbHash *h)
  h->hash = FNV1_32_INIT; 重置 hash 值为初始偏移基础量
- void cdbhash(CdbHash *h, Datum datum, Oid type): 添加一个属性到 CdbHash 计算中，也就是添加计算 hash 时考虑的一个属性。 这个函数会传入函数指针：addToCdbHash。
- void addToCdbHash(void *cdbHash, void *buf, size_t len); 实现了 datumHashFunction

h->hash = fnv1_32_buf(buf, len, h->hash); // 在缓冲区执行 32 位 FNV 1 哈希

通常调用路径是： evalHashKey -> cdbhash -> hashDatum -> addToCdbHash

- unsigned int cdbhashreduce(CdbHash *h): 映射哈希值到某个 segment，主要逻辑是取模，如下所示：

```
switch (h->reducealg)
{
  case REDUCE_BITMASK:
    result = FASTMOD(h->hash, (uint32) h->numsegs);      /* fast mod (bitmask) */
    break;

  case REDUCE_LAZYMOD:
    result = (h->hash) % (h->numsegs); /* simple mod */
    break;
}
```

对于每一个 tuple 要执行下面的 flow：

- void cdbhashinit(CdbHash *h)
- void cdbhash(CdbHash *h, Datum datum, Oid type)
- void addToCdbHash(void *cdbHash, void *buf, size_t len)
- unsigned int cdbhashreduce(CdbHash *h)

随机分布

如果不能确定一张表的哈希分布键或者不存在合理的避免数据倾斜的分布键，则可以使用随机分布。随机分布会采用循环的方式将一次插入的数据存储到不同的节点上。随机性只在单个 SQL 中有效，不考虑跨 SQL 的情况。譬如如果每次插入一行数据到随机分布表中，最终的数据会全部保存在第一个节点上。

```
test=# create table t1 (id int) DISTRIBUTED RANDOMLY;
CREATE TABLE
test=# INSERT INTO t1 VALUES (1);
INSERT 0 1
test=# INSERT INTO t1 VALUES (2);
INSERT 0 1
test=# INSERT INTO t1 VALUES (3);
INSERT 0 1
test=# SELECT gp_segment_id, * from t1;
 gp_segment_id | id
---------------+----
             1 |  1
```

```
             1 |  2
             1 |  3
```

有些工具使用随机分布实现数据管理，譬如扩容工具 gpexpand 在增加节点后需要对数据进行重分布。在初始化的时候，gpexpand 会把所有表都标记为随机分布，然后执行重新分布操作，这样重分布操作不影响业务的正常运行。（Greenplum 6 重新设计了 gpexpand，不再需要修改分布策略为随机分布）

复制表（Replicated Table）

Greenplum 6 支持一种新的分布策略：复制表，即整张表在每个节点上都有一个完整的拷贝。

```
test=# CREATE TABLE t2 (id int) DISTRIBUTED REPLICATED;
CREATE TABLE
test=# INSERT INTO t2 VALUES (1), (2), (3);
INSERT 0 3
test=# SELECT * FROM t2;
 id
----
  1
  2
  3
(3 rows)

test=# SELECT gp_segment_id, * from t2;
 gp_segment_id | id
---------------+----
             0 |  1
             0 |  2
             0 |  3
```

复制表解决了两个问题：

- UDF 在 segment 上不能访问任何表。由于 MPP 的特性，任何 segment 仅仅包含部分数据，因而在 segment 执行的 UDF 不能访问任何表，否则数据计算错误。

```
yydzero=# CREATE FUNCTION c() RETURNS bigint AS $$
yydzero$#  SELECT count(*) from t1 AS result;
yydzero$# $$ LANGUAGE SQL;
CREATE FUNCTION
yydzero=# SELECT c();
 c
---
 6
(1 row)
```

```
yydzero=# select c() from t2;
ERROR:  function cannot execute on a QE slice because it accesses relation "public.t1"
(seg0 slice1 192.168.1.107:25435 pid=76589)
```

如果把上面的 t1 改成复制表，则不存在这个问题。

复制表有很多应用场景，譬如 PostGIS 的 spatial_ref_sys（PostGIS 有大量的 UDF 需要访问这张表）和 PLR 中的 plr_modules 都可以采用复制表方式。在支持这个特性之前，Greenplum 只能通过一些小技巧来支持诸如 spatial_ref_sys 之类的表。

- 避免分布式查询计划：如果一张表的数据在各个 segment 上都有拷贝，那么就可以生成本地连接计划，而避免数据在集群的不同节点间移动。如果用复制表存储数据量比较小的表（譬如数千行），那么性能有明显的提升。 数据量大的表不适合使用复制表模式。

3. 查询计划并行化

PostgreSQL 生成的查询计划只能在单节点上执行，Greenplum 需要将查询计划并行化，以充分发挥集群的优势。

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。（本小节需要理解 PostgreSQL 优化器基础知识，可参阅 src/backend/optimizer/README）

优化实例

在介绍技术细节之前，先看几个例子。

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);
```

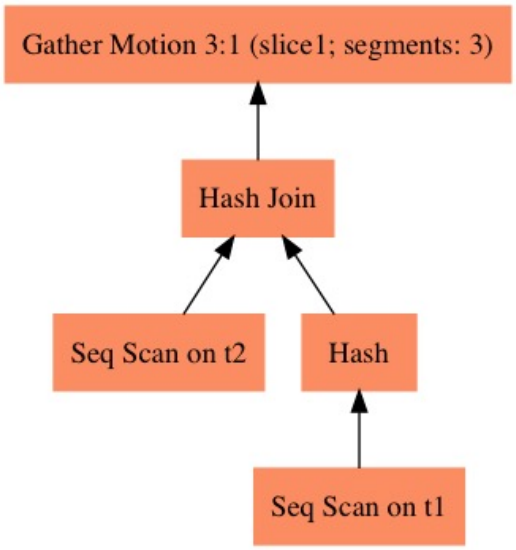
SQL1:

```
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
```

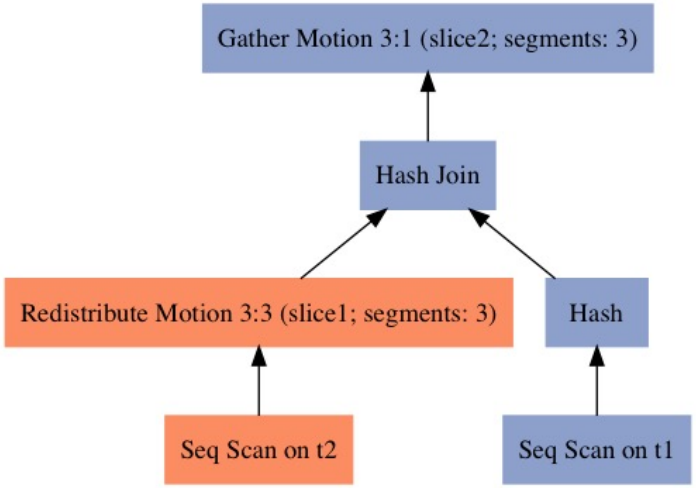


SQL2:

```
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
```



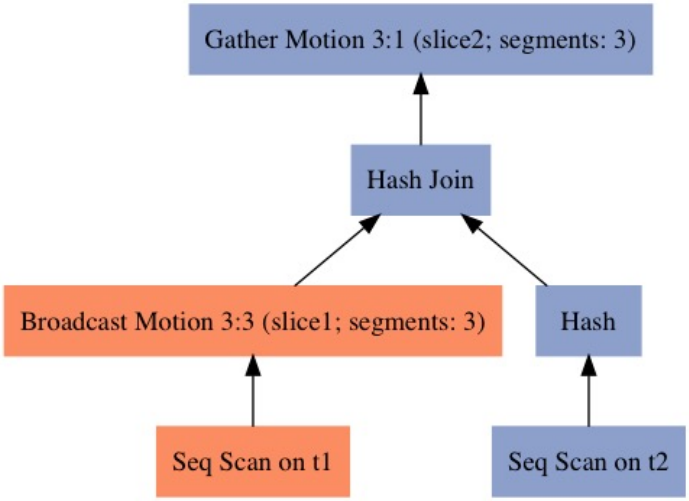
SQL3:

```
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。最新的 master 代码对这个查询生成的计划会对两个表选择重分布，为何这么做可以作为一个思考题：）。

```
                                        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
```



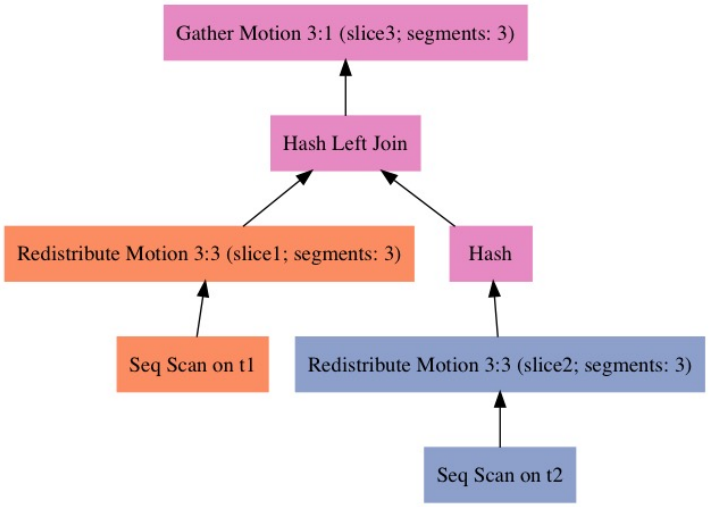
SQL4:

```
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
```



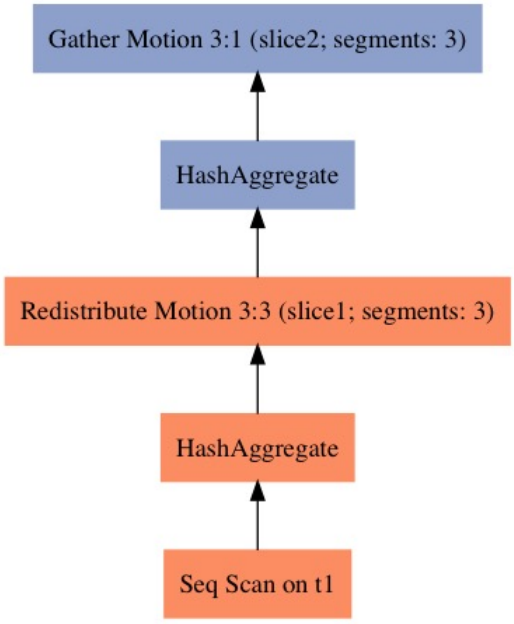
SQL5:

```
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 显示不同类型的 JOIN 对数据移动类型(Motion 类型)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 为查询优化引入的新数据结构和概念

前面几个直观的例子展示了 Greenplum 对不同 SQL 生成的不同分布式查询计划。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：

- 新增一种节点（Node）类型：Flow
- 新增一种路径（Path）类型：CdbMotionPath
- 新增一个新的查询计划（Plan）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；

新 Node 类型：Flow

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

新 Path 类型：CdbMotionPath

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。

新 Plan 算子：Motion

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 motionNode 等。

生成分布式查询计划

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（也称为 SPJ：Select/Projection/Join）的优化，由 query_planner() 实现
- 高级查询特性(Non-SPJ)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。

standard_planner

subquery_planner: 优化的主入口，若有子查询，则会递归调用

对 Query 对象执行某些一次性的处理

grouping_planner

query_planner

最基本的优化，不会处理top-level 操作。

cdb_grouping_planner

优化高级查询特性，譬如聚集，并进行MPP优化

很多其他高级查询特性的处理

修复共享 id 信息, 避免DAG环。并行化前必须解决这个问题。

set_plan_references

PostgreSQL 优化器的最后阶段，此时已经获得完整的计划树，仅做些方便执行器执行的调整，譬如扁平化子查询并调整 Vars。

cdbparallelize: 执行计划最终并行化，加flow并设置dispatch字段

prescan:扫描plan，并设置其 flow 节点

prescan 遍历查询树，合理标记以决定后续是否执行 apply_motion。
还会对 SubPlan 表达式进行专门处理。分别根据其查询树的特点采取不同的方法进行 plan 树的标记或者转换：InitPlan, 无关多行子查询、关联子查询

apply_motion:

为plantree的根节点添加motion
Prescan 处理 SubPlan 的时候会做合适的标记或者变形

```
result = makeNode(PlannedStmt);
result-> planTree = top_plan;
result->subplans = glob->subplans
return result; // 返回最终的 PlannedStmt
```

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。

如何确定 locus?

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

多个子查询间的并行化

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。

4. 分布式执行器

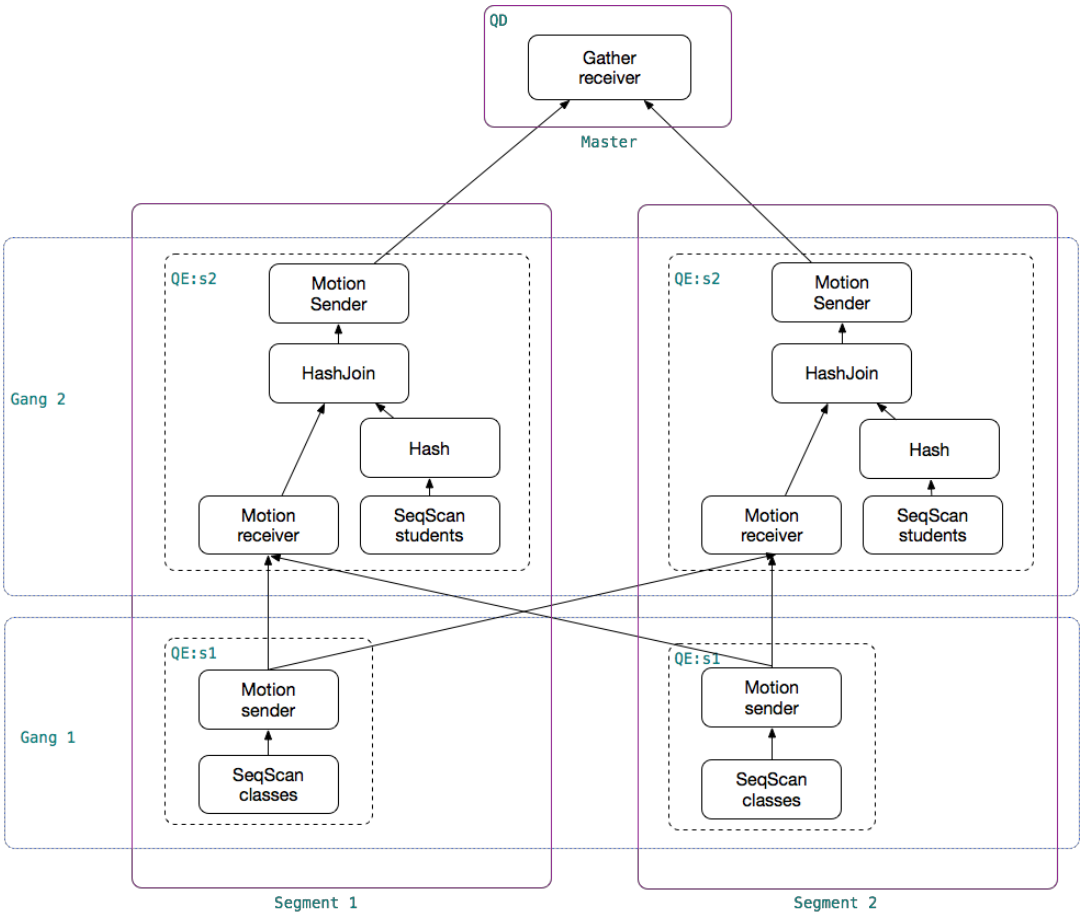
现在有了分布式数据存储机制，也生成了分布式查询计划，下一步是如何在集群里执行分布式计划，最终返回结果给用户。

Greenplum 执行器相关概念

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 width=10)
         ->  Hash  (cost=2.03..2.03 rows=2 width=12)
               ->  Seq Scan on students s  (cost=0.00..2.03 rows=2 width=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

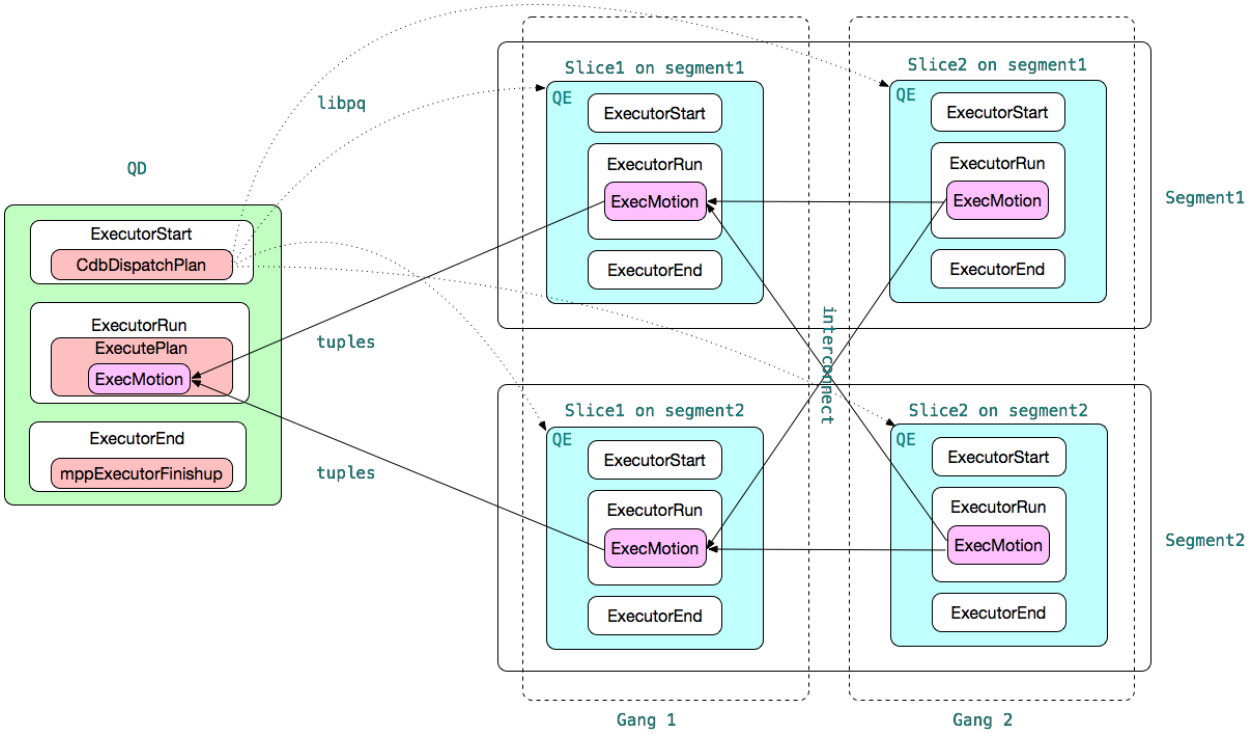
行其查询请求的 PostgreSQL Backend 进程。通常 QE 执行整个查询的一部分（称为 Slice）。QE 的入口是 exec_mpp_query()。

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。

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 CdbDispatchPlan 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。Greenplum 从 6.0 开始去掉了异步方式。
- 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。

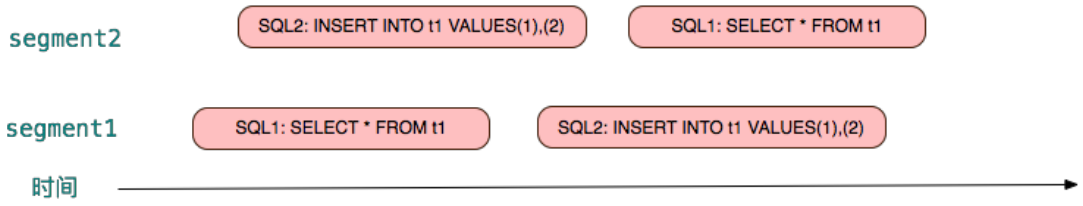
5. 分布式事务

Greenplum 使用两阶段提交（2PC）协议实现分布式事务。2PC 是数据库经典算法，此处不再赘述。本节概要介绍两个 Greenplum 分布式事务的实现细节：

- 分布式事务快照：实现 master 和不同 segment 间一致性
- 共享本地快照：实现 segment 内不同 QEs 间的一致性

分布式快照

在分布式环境下，SQL 在不同节点上的执行顺序可能不同。譬如下面例子中 segment1 首先执行 SQL1，然后执行 SQL2，所以新插入的数据对 SQL1 不可见；而 segment2 上先执行 SQL2 后执行 SQL1，因而 SQL1 可以看到新插入的数据。这就造成了数据的不一致。



Greenplum 使用分布式快照和本地映射实现跨节点的数据一致性。Greenplum QD 进程承担分布式事务管理器的角色，在 QD 开始一个新的事务（StartTransaction）时，它会创建一个新的分布式事务 id、设置时间戳及相应的状态信息；在获取快照（GetSnapshotData）时，QD 创建分布式快照并保存在当前快照中。和单节点的快照类似，分布式快照记录了 xmin/xmax/xip 等信息，结构体如下所示：

```
typedef struct DistributedSnapshot
{
       DistributedTransactionTimeStamp      distribTransactionTimeStamp;
       DistributedTransactionId xminAllDistributedSnapshots;
       DistributedSnapshotId distribSnapshotId;

       DistributedTransactionId xmin;       /* XID < xmin 则可见 */
       DistributedTransactionId xmax;       /* XID >= xmax 则不可见 */
       int32          count;         /*  inProgressXidArray 数组中分布式事务的个数 */
       int32          maxCount;

       /* 正在执行的分布式事务数组 */
       DistributedTransactionId        *inProgressXidArray;
} DistributedSnapshot;
```

执行查询时，QD 将分布式事务和快照等信息序列化，通过 libpq 协议发送给 QE。QE 反序列化后，获得 QD 的分布式事务和快照信息。这些信息被用于确定元组的可见性（HeapTupleSatisfiesMVCC）。所有参与查询的 QEs 都使用 QD 发送的同一份分布式事务和快照信息判断元组的可见性，因而保证了整个集群数据的一致性，避免前面例子中出现的不一致现象。

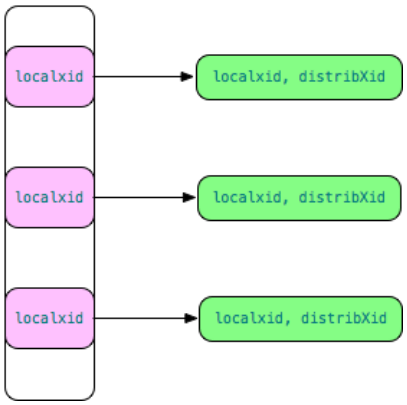
在 QE 上判断一个元组对某个快照的可见性流程如下：

- 如果创建元组的事务：xid（即元组头中的 xmin 字段）还没有提交，则不需要使用分布式事务和快照信息；
- 否则判断创建元组的事务 xid 对快照是否可见

- 首先根据分布式快照信息判断。根据创建元组的 xid 从分布式事务提交日志中找到其对应的分布式事务：distribXid，然后判断 distribXid 对分布式快照是否可见：
  - 如果 distribXid < distribSnapshot->xmin，则元组可见
  - 如果 distribXid > distribSnapshot->xmax，则元组不可见
  - 如果 distribSnapshot->inProgressXidArray 包含 distribXid，则元组不可见
  - 否则元组可见
- 如果不能根据分布式快照判断可见性，或者不需要根据分布式快照判断可见性，则使用本地快照信息判断，这个逻辑和 PostgreSQL 的判断可见性逻辑一样。

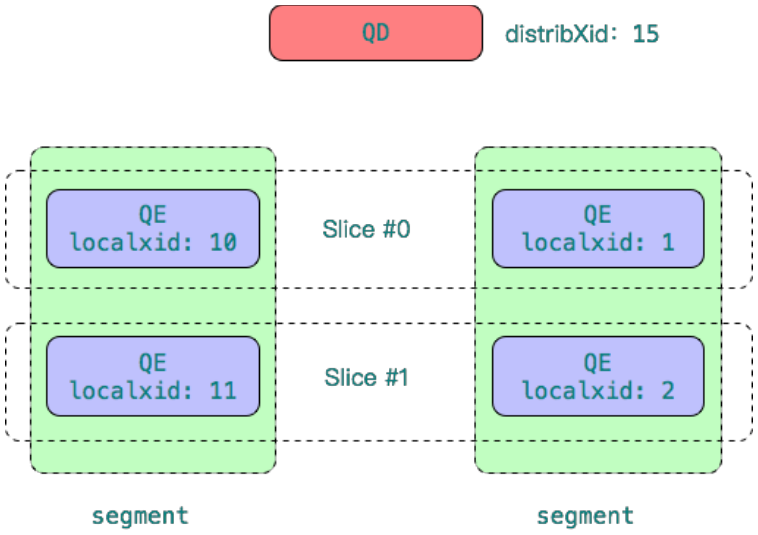
和 PostgreSQL 的提交日志 clog 类似，Greenplum 需要保存全局事务的提交日志，以判断某个事务是否已经提交。这些信息保存在共享内存中并持久化存储在 distributedlog 目录下。

为了提高判断本地 xid 可见性的效率，避免每次访问全局事务提交日志，Greenplum 引入了本地事务-分布式事务提交缓存，如下图所示。每个 QE 都维护了这样一个缓存，通过该缓存，可以快速查到本地 xid 对应的全局事务 distribXid 信息，进而根据全局快照判断可见性，避免频繁访问共享内存或磁盘。



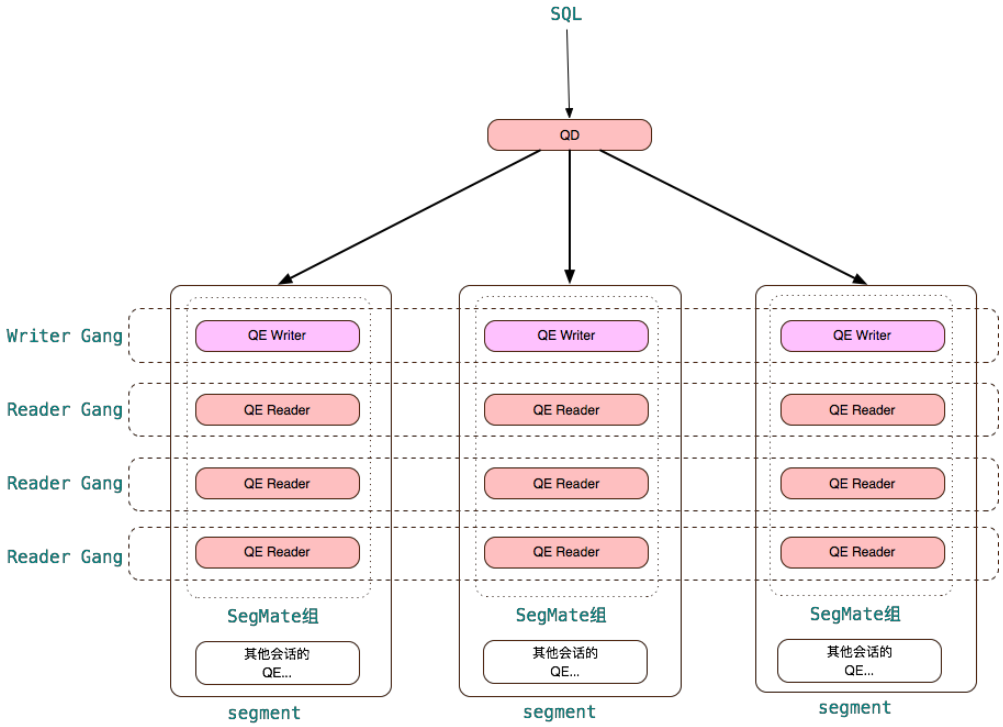
共享本地快照（Shared Local Snapshot）

Greenplum 中一个 SQL 查询计划可能含有多个 slices，每个 Slice 对应一个 QE 进程。任一 segment 上，同一会话（处理同一个用户 SQL）的不同 QE 必须有相同的可见性。然而每个 QE 进程都是独立的 PostgreSQL backend 进程，它们之间互相不知道对方的存在，因而其事务和快照信息都是不一样的。如下图所示。

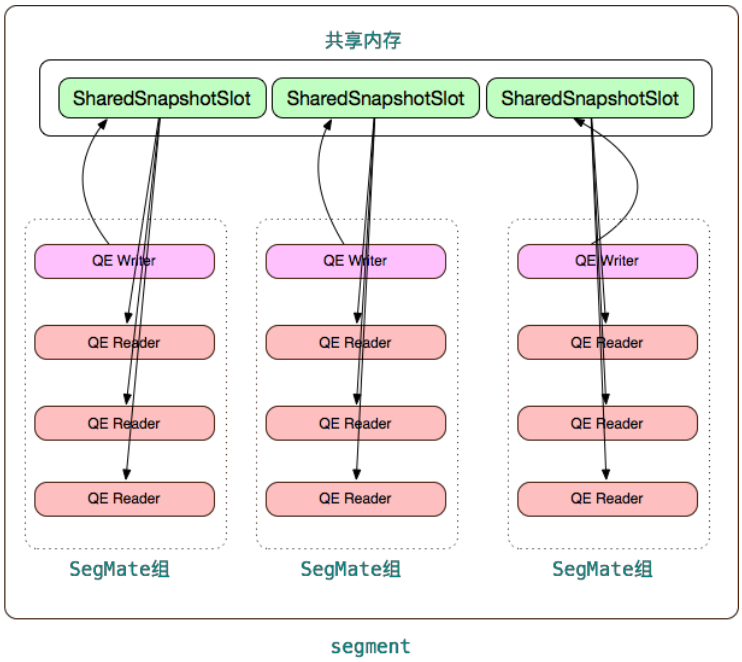


为了保证跨 slice 可见性的一致性，Greenplum 引入了 “共享本地快照(Shared Local Snapshot)” 的概念。每个 segment 上的执行同一个 SQL 的不同 QEs 通过共享内存数据结构 SharedSnapshotSlot 共享会话和事务信息。这些进程称为 SegMate 进程组。

Greenplum 把 SegMate 进程组中的 QE 分为 QE writer 和 QE reader。QE writer 有且只有一个，QE reader 可以没有或者多个。QE writer 可以修改数据库状态；QE reader 不能修改数据库的状态，且需要使用和 QE writer 一样的快照信息以保持与 QE writer 一致的可见性。如下图所示。



“共享”意味着该快照在 QE writer 和 readers 间共享，“本地” 意味着这个快照是 segment 的本地快照，同一用户会话在不同的 segment 上可以有不同的快照。segment 的共享内存中有一个区域存储共享快照，该区域被分成很多槽（slots）。一个 SegMate 进程组对应一个槽，通过唯一的会话 id 标志。一个 segment 可能有多个 SegMate 进程组，每个进程组对应一个用户的会话，如下图所示。



QE Writer 创建本地事务后，在共享内存中获得一个 SharedLocalSnapshot 槽，并它自己的本地事务和快照信息拷贝到共享内存槽中，SegMate 进程组中的其他 QE Reader 从该共享内存中获得事务和快照信息。Reader QEs 会等待 Writer QE 直到 Writer 设置好共享本地快照信息。

只有 QE writer 参与全局事务，也只有该 QE 需要处理 commit/abort 等事务命令。

6. 数据洗牌（Shuffle）

相邻 Gang 之间的数据传输称为数据洗牌（Data Shuffling）。数据洗牌和 Slice 的层次相吻合，从下到上一层一层通过网络进行数据传输，不能跨层传输数据。根据 Motion 类型的不同有不同的实现方式，譬如广播和重分布。

Greenplum 实现数据洗牌的技术称为 interconnect，它为 QEs 提供高速并行的数据传输服务，不需要磁盘 IO 操作，是 Greenplum 实现高性能查询执行的重要技术之一。interconnect 只用来传输数据（表单的元组），调度、控制和错误处理等信息通过 QD 和 QE 之间的 libpq 连接传输。

Interconnect 有 TCP 和 UDP 两种实现方式，TCP interconnect 在大规模集群中会占用大量端口资源，因而扩展性较低。Greenplum 默认使用 UDP 方式。UDP interconnect 支持流量控制、网络包重发和确认等特性。

7. 分布式集群管理

分布式集群包含多个物理节点，少则四五台，多则数百台。管理如此多机器的复杂度远远大于单个 PostgreSQL 数据库。为了简化数据库集群的管理， Greenplum 提供了大量的工具。下面列出一些常用的工具，关于更多工具的信息可以参考 Greenplum 数据库管理员官方文档。

- gpactivatestandby： 激活 standby master，使之成为 Greenplum 数据库集群的主 master。
- gpaddmirrors：为 Greenplum 集群添加镜像节点，以提高高可用性
- gpcheckcat：检查 Greenplum 数据库的系统表，用以辅助故障分析。

- gpcheckperf：检查 Greenplum 集群的系统性能，包括磁盘、网络和内存的性能。
- gpconfig：为 Greenplum 集群中的所有节点进行参数配置。
- gpdeletesystem：删除整个 Greenplum 集群
- gpexpand：添加新机器到 Greenplum 集群中，用以扩容。
- gpfdist：Greenplum 的文件分发服务器，是 Greenplum 数据加载和卸载的最主要工具。gpfdist 充分利用并行处理，性能非常高。
- gpload：封装 gpfdist 和外部表等信息，通过配置 YAML 文件，可以方便的加载数据到 Greenplum 数据库中。支持 INSERT、UPDATE 和 MERGE 三种模式。
- gpinitstandby：为 Greenplum 集群初始化 standby master
- gpinitsystem：初始化 Greenplum 集群
- gppkg：Greenplum 提供的软件包管理工具，可以方便的在所有节点上安装 Greenplum 软件包，譬如 PostGIS、PLR 等。
- gprecoverseg：恢复出现故障的主 segment 节点或者镜像 segment 节点
- gpssh/gpscp：标准的 ssh/scp 只能针对一个目标机器进行远程命令执行和文件拷贝操作。gpssh 可以同时在一组机器上执行同一个命令；gpscp 同时拷贝一个文件或者目录到多个目标机器上。 很多 Greenplum 命令行工具都使用这两个工具实现集群并行命令执行。
- gpstart：启动一个 Greenplum 集群。
- gpstop：停止一个 Greenplum 集群
- gpstate：显示 Greenplum 集群的状态
- gpcopy：将一个 Greenplum 数据库的数据迁移到另一个 Greenplum 数据库中。
- gp_dump/gp_restore：Greenplum 数据备份恢复工具。从 Greenplum 5.x 开始，推荐使用新的备份恢复工具： gpbackup/gprestore。
- packcore：packcore 可以将一个 core dump 文件及其所有的依赖打成一个包，可以在其他机器上进行调试。非常有用的一个调试用具。
- explain.pl：把 EXPLAIN 的文本结果转换成图片。本节中用的计划树图片都是使用这个工具生成的。

8. 动手实践

上面概要介绍了把单个 PostgreSQL 数据库变成分布式数据库涉及的 6 个方面的工作。若对更多细节感兴趣，最有效的方式是动手改改代码实现某些新特性。下面几个项目可以作为参考：

- 数据存储：实现 partial table，使得一张表或者一个数据库仅仅使用集群的一个子集。譬如集群有 200 个节点，可以创建只是用 10 个节点的表或者数据库。
- 资源管理：目前的 Gang 只能在一个会话内部共享，实现 Gang 的跨会话共享，或者 Gang 共享池。
- 调度：目前 dispatcher 将整个 plan 发送给每个 QE，可以发送单个 slice 给负责执行该 slice 的 QE
- 性能优化：分析 Greenplum 分布式执行的性能瓶颈，并进行进一步的优化，特别是 OLTP 型查询的性能优化，以实现更高 tps。
- 执行器优化：目前 Greenplum 使用 zstd 压缩 AO 数据和临时数据，zstd 造成的一个问题是内存消耗较大，如何优化操作大量压缩文件时的内存消耗是一个很有挑战的课题。有关更多细节可以参考这个讨论（最后部分有简单的问题重现方法）。

对这些项目有兴趣者可以联系 yyao AT pivotal DOT io 提供更多咨询或帮助。实现以上任何一个功能者，可以走快速通道加入 Greenplum 内核开发团队，共应挑战共享喜悦 :)

9. 后记

想了一会，本想写一点打鸡血的话吸引更多人加入数据库内核开发行列，然觉不合自性，作罢。

Greenplum 酒文化比较浓厚，还是分享一个与酒有关的小故事收尾吧。

13/14 年左右有幸和数据库老前辈 Dan Holle（Teradata CTO，第七号员工）有诸多交集。老爷子谈吐优雅而不失幽默，从事 MPP 数据库已有 30 余年。每次喝酒至少放两瓶酒于面前，且同时喝两瓶酒，笑谈此为并行处理；若某一瓶喝的多了，必拿起另一瓶再喝一点，以确保两瓶余量保

持一致，笑谈此为避免倾斜。经常左瓶喝的多了点，拿起右瓶补一口，补多了，再拿起左瓶补一口。如此左右互补，无需山东人劝酒，自己很快进入状态。

老先生对 MPP 数据库之爱已融入生活中，令人敬佩。而正是许多这样数十年如一日的匠人成就了当今数据库领域的辉煌。期待更多人加入，幕天席地把酒言欢！

关于作者

姚延栋，山东大学本科，中科院软件所研究生，Greenplum 研发总监。PostgreSQL 中文社区委员，Greenplum 中文社区发起人。致力于 Greenplum/PostgreSQL 开源数据库产品、社区和生态的发展。