Greenplum Database管理员指南

版本 V6.0

2020年07月19日

如果您觉得有帮助，感谢打赏

序言

术语约定

GP	:	Greenplum数据库
Master	:	GP的控制节点/实例
Standby	:	GP的备用控制节点/实例
Host(主机)	:	GP的一台独立的机器设备
Instance	:	GP的计算实例，很多时候也叫Segment
Primary	:	GP的主计算实例
Mirror	:	GP的镜像计算实例
MPP	:	大规模并行处理
算子	:	执行计划中的运算操作

背景简介

多年前，编者翻译了GP4.2.2的AdminGuide，如今，GP已经历经了无数个版本更新和迭代，编者也有了更多的感悟，放眼GP的中文资料，为之动容，就想着再为GP的发展壮大多做那么一点点贡献，挤出一点时间，重新梳理和打磨这个文档，并完全根据最新的版本特性进行重新整理，希望能对中文爱好者提供一些帮助，在编写过程中，仍会参考官方文档，但绝不是简单的翻译，甚至有些内容会与官方文档不一致。

编者提醒，升级版本极其重要，4版本早该淘汰了，5版本和6版本都带来了极大的性能和稳定性的提升。

声明：

本书中的绝大部分内容都经过编者重新考量和实测验证，有些观点与官方手册有出入，仅代表编者本人观点，与官方手册无关。本书中可能会提及一些非官方的命令和工具等，仅用于讲解相关知识，如有缺失相关细节的情况，请谅解。

致读者

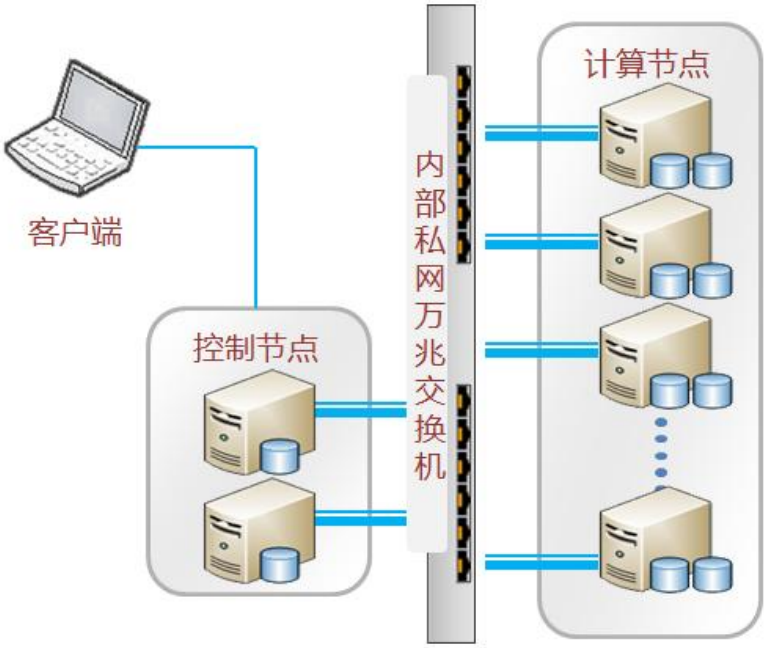
如果您在阅读和参考本书的过程中发现有任何不妥之处，或者有任何的建议和意见，欢迎联系编者，本书主要针对GP数据库的爱好者进行编写，包括产品的安装和使用说明，以及最佳实践等内容。本书的发布更新情况与编者的时间有关，不做承若。

编写： 陈淼
电邮： miaochen@mail.ustc.edu.cn

目录

第一章：GP 数据库架构

目前 GP 数据库已经开源多年，多年来一直由 Pivotal 公司商业运营，GP 是一个 MPP 数据库产品，采用 Share-Nothing 架构，可管理和处理分布在多个不同主机上的大规模数据集。对于 GP 数据库来说，一个数据库实体是由多个独立的 PostgreSQL 实例构成的，它们分布在不同的主机上，实例之间协同工作，用户可以像对待一个普通的单机数据库那样，进行访问和执行 SQL 操作。其中 Master 是整个系统的访问入口，负责处理客户端的连接和 SQL 命令、协调系统中的其他实例协同工作，计算实例负责管理和处理具体的用户数据。



这一章节讲述组成 GP 数据库系统的组件及如何协同工作：

- 管理节点：Master
- 计算实例：Instance
- 内联网络：Interconnect
- 冗余与故障切换
- 并行数据装载
- 管理与监控

管理节点：Master

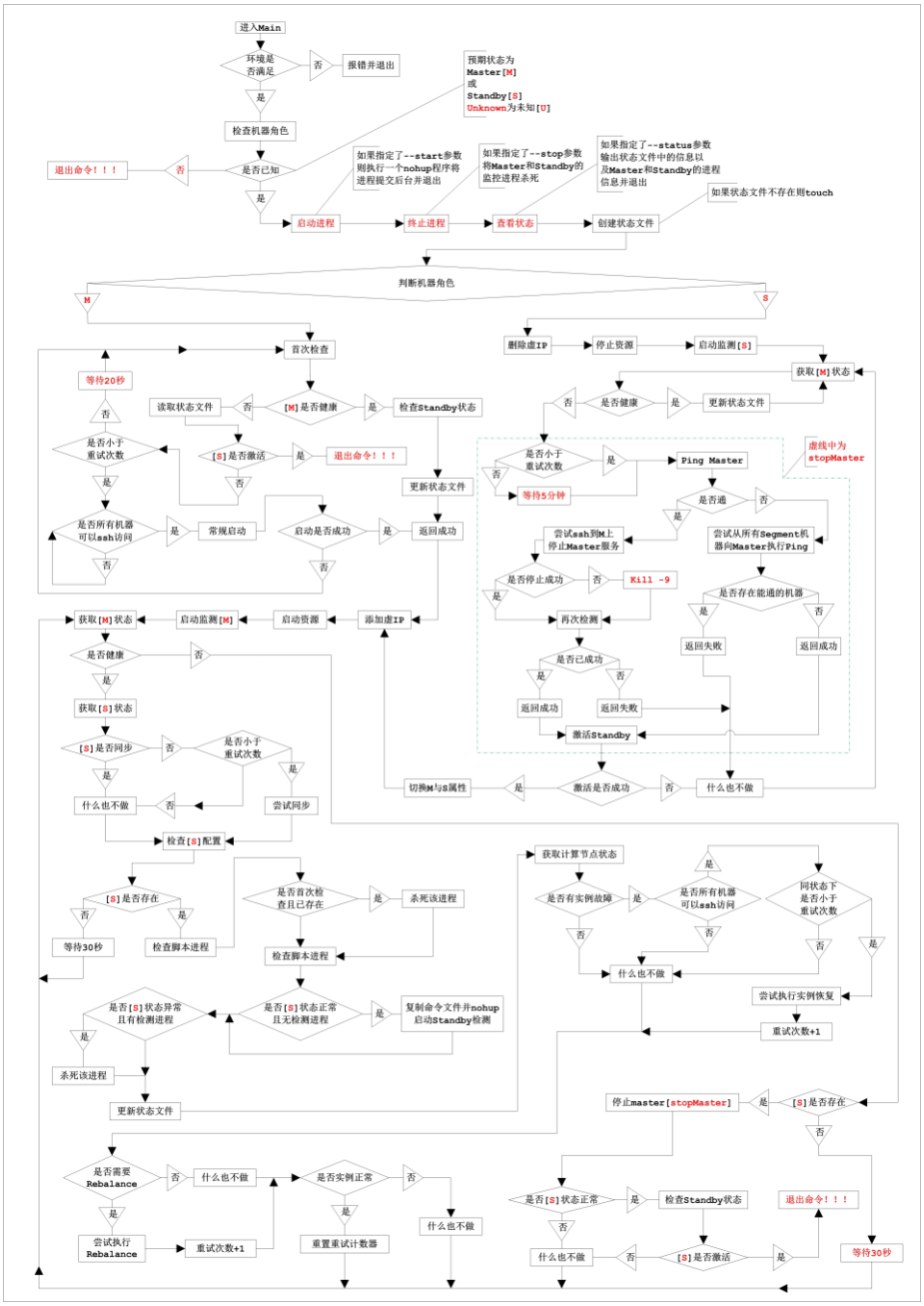
Master 作为 GP 的访问入口，主要负责处理客户端连接的访问以及用户提交的 SQL 语句的解析、生成执行计划、优化执行计划等。

GP 是基于 PostgreSQL 发展而来，用户端可以如同访问 PostgreSQL 那样与 GP

进行交互。可以通过 PostgreSQL 客户端程序(如 psql、pgAdminIII)和应用程序接口(APIs(如 JDBC、ODBC))连接 GP。

Master 上存储着全局系统表(Global System Catalog)(包含数据库系统自身元数据的系统表)，但不存储任何用户数据，用户数据只存储在 Instance 上。Master 负责客户端认证、SQL 命令接收并生成并行执行计划、在 Instance 之间分配工作负载、整合 Instance 处理结果、将 Instance 计算的最终结果汇总并反馈给客户端程序。

目前，GP 还不支持多 Active Master，不过，已经有很多人用工具或者脚本的形式实现了 Master 和 Standby 的自动 FailOver，编者也实现过自动切换命令，当 Master 出现无法正常工作的故障时，自动激活 Standby 来接管 Master 的任务。



如上图所示，这是编者实现的脚本命令所遵循的逻辑流程图。

Master 的连接数是有限的，缺省值为 250 个，如果要大规模提升连接的可用数量，可以配置使用 GP 自带的 pgbouncer 连接池，这对于一些应用场景会很有帮助，例如 SAS 等软件连接 GP 时，由于这些软件自身无法严格限制连接数，pgbouncer 会是一个有效的缓解连接数过大的方案，例如按照如下方式进行配置：

```
$ cat pgbouncer.ini
[databases]
dwdb = host=127.0.0.1 port=5432 dbname=dwdb auth_user=gpadmin
[pgbouncer]
;;pool_mode = session
pool_mode = transaction
max_client_conn = 2000
default_pool_size = 20
min_pool_size = 5
listen_port = 6432
listen_addr = *
auth_type = hba
auth_file = /home/gpadmin/pgbouncer_users.list
auth_hba_file = /home/gpadmin/pgbouncer_hba.conf
logfile = /data/pgbouncer.log
pidfile = /tmp/.pgbouncer.pid
admin_users = gpadmin
stats_users = gpadmin
ignore_startup_parameters = extra_float_digits,gp_session_role
$ cat restart_pgbouncer.sh
ps ax|grep -w pgbouncer|grep -vw grep|awk '{system("kill -9 "$1)}'
pgbouncer -v -R -d pgbouncer.ini
```

计算实例：Instance

在 GP 系统中，Instance 才是承担数据存储和查询处理的角色。用户 Table 和相应的 Index 都分布在 GP 系统中各 Instance 上，每个 Instance 存储着一部分数据(对于复制表来说，每个 Instance 存储一份完整的数据)。Instance 才是真正进行数据处理的地方。缺省情况下，用户不能跳过 Master 直接访问 Instance，而只能通过 Master 来访问整个数据库系统；不过，对于管理员和运维人员来说，有时需要使用 Utility 模式来访问 Instance，访问方法是：

```
[gpamdin@mdw001 ~]$ PGOPTIONS='-c gp_session_role=utility' psql
```

在GP推荐的硬件配置环境下，每个Instance需要对应数个CPU Core，具体的比例需要根据数据库的适用场景进行综合评估。例如在生产环境，每个Instance所在的主机配置了2个16 Core的CPU，可根据不同的场景，配置4 ~ 12个不等的Primary，这个数字的选择需要由富有经验的工程师进行评估，每个Instance所在主机配置的Primary越多，响应并发的能力越弱，但单个任务的处理能力越强。实际上，每个主机的实例个数，还与其他资源有关，如，磁盘性能，网络性能，内存容量。

内联网络：Interconnect

网络层是GP系统的重要组件，在用户执行查询时，每个Instance都需要执行相应的处理，网络层涉及到Instance之间的通信和数据传输，网络层可以使用标准的以太网协议。

在缺省情况下，网络层使用UDPIFC协议。这是经过改善的UDP协议，在UDP协议的基础上增强了数据包校验，其可靠性与TCP协议相似，但其性能和扩展性远好于TCP协议。当集群规模较小，网络的稳定性较差的时候，如果UDPIFC协议会遭遇一些故障，可以考虑使用TCP协议，例如只有几十台主机时。通常，还是强烈建议配备稳定的网络环境，使用UDPIFC协议。

冗余与故障切换

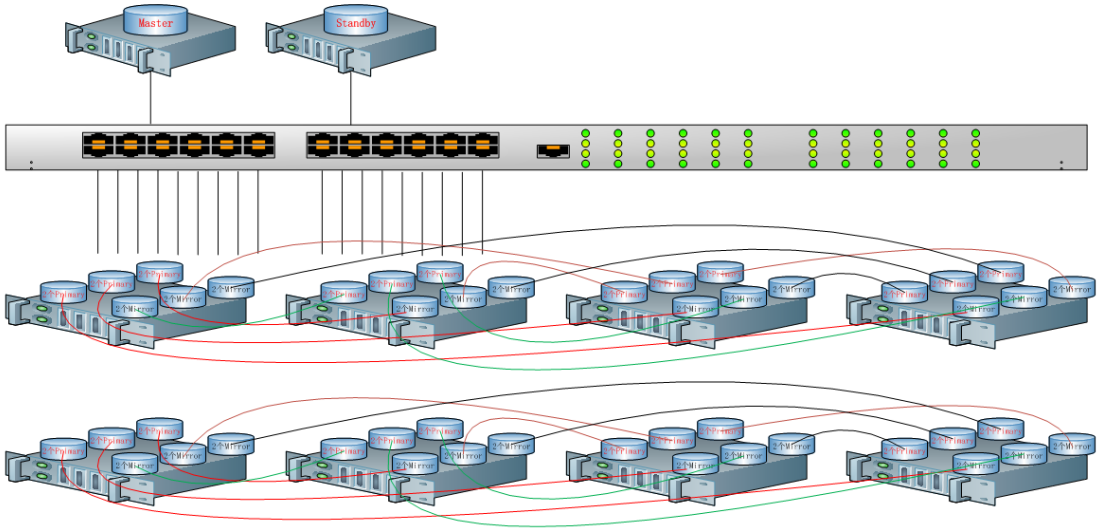
GP提供了避免单点故障的部署选项。本节讲述GP的冗余组件。

- Instance镜像
- Master镜像
- 网络层冗余

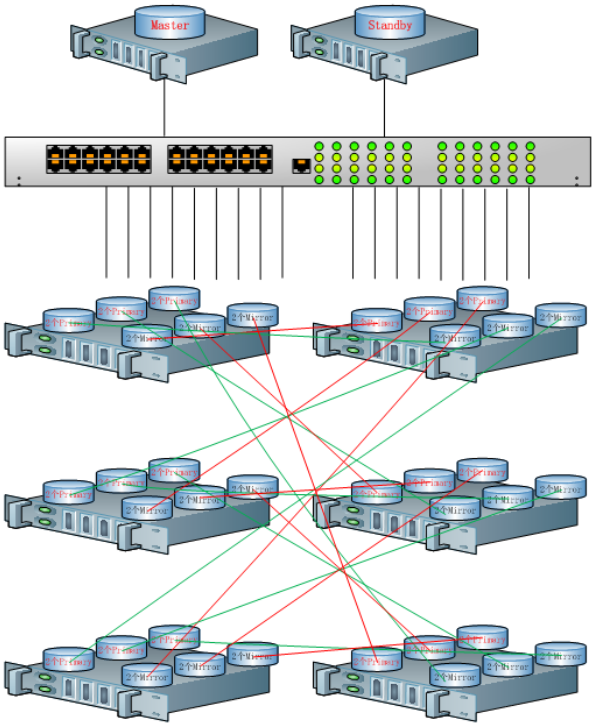
Instance镜像

在部署GP系统时，可以选择配置Mirror，如果开始时没有配置Mirror，后期也可以规划添加Mirror，当然，如要要删除已有的Mirror也是可以的，不过需要手动操作，因为GP并未提供删除Mirror的标准命令。

Mirror使得数据库查询在Primary不可用时自动切换到Mirror上。为了配置Mirror，GP系统需要有足够多的主机，从而可以保证冗余的Instance总是在与Primary不同的Host主机上。Mirror总是位于不同于Primary的Host主机上。



如上图所示，这是一种循环混合镜像模式，每 4 台机器组成一个镜像组，每台机器上有 6 个 Primary，每台机器的 6 个 Primary 对应的 Mirror 均匀分布在另外三台机器上。编者还实现了多种镜像模式，例如，循环镜像，所有的 Instance 主机组成一个环，每台机器的镜像都在下一台机器上；结对镜像，如下图所示，将一群数量为偶数的机器，分为两组，每台机器的镜像分散在对面组的机器上。关于如何选择镜像模式，以及如何分散镜像关系，可以根据用户的实际需求进行评估和实施。



目前，编者的一键式集群配置安装初始化命令已经内置了两种镜像模式，分别为 RING 和 PAIR。RING 是一种带有环状关系的镜像模式，典型的特征是，一组机器形成对等的环，环上的每台机器，其对应的 Mirror 会散落在后面的一台或者多台机器上，

这种模式包含了 gpinitsystem 命令缺省支持的两种镜像模式：GROUP 和 SPREAD。PAIR 模式是一种两组配对互为镜像的模式，是一种更能兼顾性能和安全性的方案。

Instance 故障切换与恢复

在 GP 系统启用 Mirror 的情况下，当 Primary 不可访问时，Master 会自动将任务切换到对应的 Mirror 上，此时，Mirror 取代 Primary 的作用。只要剩余的可用 Instance 能够保证数据的完整性，在 Instance 或者 Host 主机宕机时，GP 系统仍可继续保持服务可用的状态。

每当 Master 无法连接到 Primary 时，该 Primary 在 GP 的系统表中将被标记为失败状态，Master 会激活/唤醒对应的 Mirror 取代原有的 Primary。在采取相应的措施将该失败的 Primary 恢复到健康状态之前，该 Primary 一直保持失败状态。失败的 Primary 可以在系统处于运行状态下被恢复回来。恢复进程仅仅复制失败期间发生变化的增量差异，当然，如果失败时间太久或者因失败的 Instance 文件有损毁，将需要全量复制或者需要选择全量复制。在 6 之前的版本，GP 的 Primary 和 Mirror 之间采用的是 filerep 的方式进行 block 级别的变化复制的方式，从 6 版本开始，使用 WAL 复制，这将可以从根本上解决以往的 block 损毁被复制到 Mirror 上的问题，也不再需要 persistent 系统表了。

在未启用 Mirror 的情况下，任何的 Instance 失败都会导致系统自动停止服务。在继续使用系统之前，必须恢复所有失败的 Instance。

Master 镜像

如同 Instance 需要 Mirror 一样，可以在另一台主机上为 Master 部署一个备份/镜像，按照惯例将其称为 Standby。在 Master 不可用时，Standby 将可以被激活以接替 Master 的角色。Standby 与 Master 之间保持 WAL 同步，保证与 Master 之间的一致性。

在 Master 失效时，WAL 同步的复制进程会自动停止，同时，Standby 可以被激活。在 Standby 上，冗余的日志会被用来将状态恢复到最后成功提交(commit)的状态。激活的 Standby 实际上会成为 GP 的新 Master，通过 Master Port(该端口需要设置和 Master 的相同)接受客户端的链接访问。一旦 Standby 被激活，旧的 -- 那个失败了的 Master 将脱离集群，不再属于这个集群，要想将其重新加入集群中，需要使用 gpinitstandby 命令将其添加为 Standby 的角色。如果因为误操作，导致一个不应该被激活(例如其已经与 Master 失去同步很久了)的 Standby 被激活了，这个时候，应该尽快停止数据库，将旧的 Master 以 Master 的角色恢复回来，这是一个复杂的问题，需要在确保风险可控的前提下进行操作。

由于 Master 不存储用户数据，在 Master 和 Standby 之间仅仅是系统表的数据需要被同步。这些表的数据量与用户的业务表相比，很小，而且较少发生变化，一旦发生变化，就会自动同步到 Standby 从而保证与 Master 的一致性。

会有很多用户问，Master 和 Standby 在绝大多数时间内，资源非常空闲，跟 Instance 主机相比，相当于完全空闲，那么是否可以将 Master 和 Standy 设置到 Instance 主机上呢？从理论的角度来说，答案是肯定的，因为 GP 数据库的集群概念是虚拟的，并没有严格限制不同角色必须分离，但，对于生产环境来说，除非可以 100%确保计算节点机器的资源不会被耗尽，否则，都应该尽最大可能避免 Master 和 Standby 设置到 Instance 主机上，因为，这种情况，一旦系统在处理负载很高的任务，Master 将很难获得足够的资源，其响应会变慢，稳定性会下降。换句话说，如果可以确保集群是非常良性的运转，不会有任务造成 Master 很大的压力，可以适当配置计算能力稍差的机器。

网络层冗余

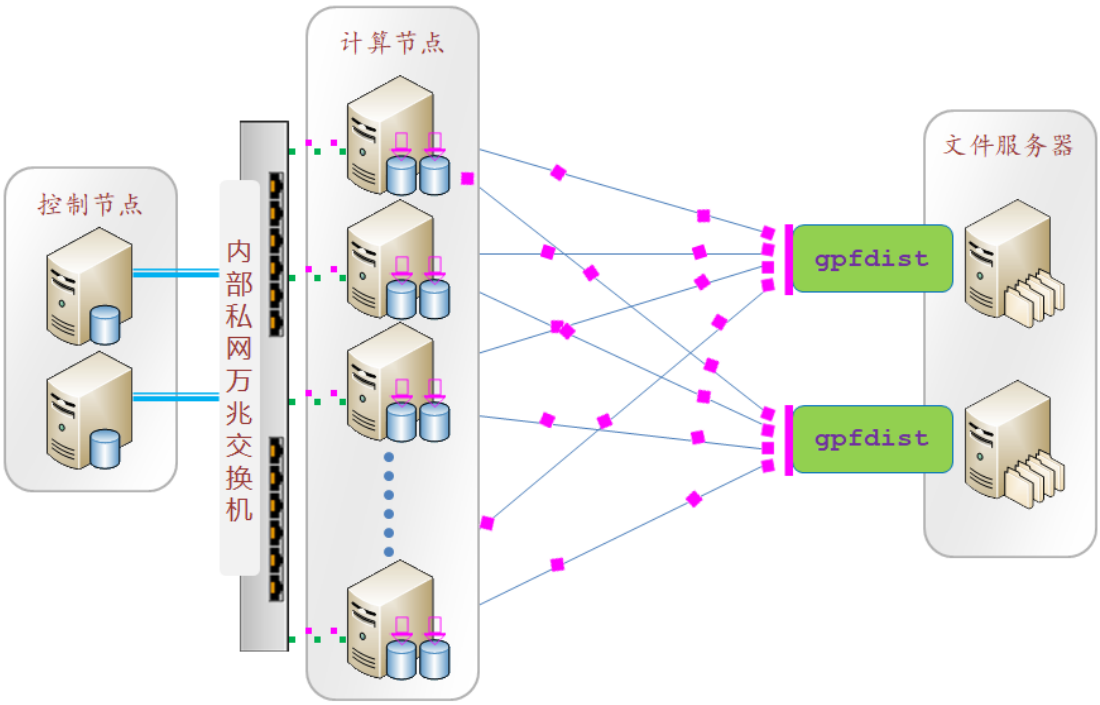
网络层关系到 Instance 之间的通信，其依靠基础网络设施，高可用网络层可以通过部署双重网络实现。虽然，在配置 Mirror 的情况下，通过不同网段间的 Primary 与 Mirror 之间的对应关系也可以达到网络保障的效果，但依然强烈建议采用网卡绑定的方式实现网络的高可用。建议采用支持 802.3ad 协议的交换机以实现多网口的链路聚合，这样，在操作系统层面，多个物理网口将聚合并表现为一个 IP 地址，当任何的网络或者交换机出现故障时，在操作系统级别将不会有任何的连接性异常的感知，只是网络带宽出现下降，整个数据库集群的 Instance 状态将不会受到任何影响。如果选择将 Primary 和 Mirror 分布在不同的网段，出现任何的网络故障时，总会有 Instance 的状态发生变化，这对上层应用可能是有感知的。

并行数据装载

海量数据仓库的一个重大挑战是，要在一个受限的时间窗口内完成大量数据的装载。

GP 通过外部表(External Table)支持高速并行数据装载。外部表可以使用[单条记录出错隔离]模式，以允许在装载数据过程中将出错的数据记录下来。可以设置错误容忍的阈值，以实现对数据装载质量的控制。也可以对错误信息进行分析，以帮助改善数据装载的质量。

结合使用外部表和 GP 的并行文件分发服务(gpfdist)，管理员可以实现最大化的利用网络带宽资源以实现高速并行装载。



上图展示了 GP 外部表和 gpfdist 是如何配合，以实现高速数据装载的，该模式的性能是完全线性扩展的，数据直接在 gpfdist 和 Instance 之间并行传输，数据的重分布直接在 Instance 之间完成，整个架构没有瓶颈点。

管理与监控

对 GP 系统的管理，可以通过一系列的命令行来实现，它们都存放在$GPHOME/bin 目录下。GP 提供的命令可以实现如下的管理任务：

- 在主机上批量执行命令(gpssh)
- 初始化 GP 系统(gpinitsystem)
- 启动(gpstart)或关闭 GP 系统(gpstop)
- 添加或移除 Host 主机(gpstop --host)
- 扩展 Instance 以及在新节点间重新分布 Table(gpexpand)
- 监控和恢复失败的 Instance(gpstate & gprecoverseg)
- 监控和恢复失败的 Master(gpstate & gpinitstandby)

- 备份和恢复数据库(并行)(gpbackup & gprestore)
- 并行装载数据(gpload)
- 集群之间并行数据传输(gpcopy)
- 系统状态报告(gpstate)

编者在这么多年的GP使用和支持过程中，也积累了一些不错的命令，例如：

- 一键式集群安装部署初始化命令
- Master和Standby自动切换命令
- 更灵活的并行数据库备份恢复命令
- 高速DDL备份命令
- 并行DDL恢复命令
- 更先进的跨集群数据同步命令
- 集群间的表结构差异增量比对命令
- 良好兼容的pgAdminIII客户端

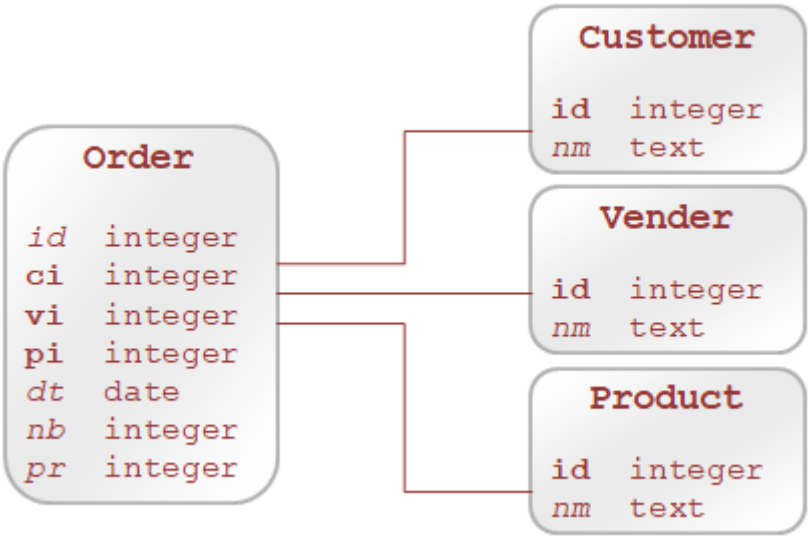
第二章：分布式数据库概念

GP 是一个分布式数据库系统。这就意味着在物理上，数据是存储在多个数据库上的(称为 Instance)。这些独立的数据库通过网络进行通信(称为内联网络)。分布式数据库的一个基本特征是，用户和客户端程序在访问时如同访问一个单机数据库(GP 访问 Master)，数据库内部的分布式实现不需要用户过多的关心，对于客户端应用来说，访问 GP 数据库与单机数据库没有什么区别。不过，对于开发人员和 DBA 来说，要更好的用好 GP 数据库，还是需要了解和掌握分布式数据库的概念，了解 GP 的架构和工作原理，这样才能更好的发挥 GP 的分布式优势，也就是说，学好这些知识是极其重要的。

数据是如何存储的

要理解 GP 是如何在不同的 Instance 之间存储数据，可以参考下图所示的简单逻辑关系，主键(Primary Key)被使用黑体标记，外键(Foreign Key)关系通过连线标明。

用数据仓库的术语来说，这种数据模型称为星型模型。在这种数据库模型下，Order 表通常被称为事实表(Fact Table)，其他表(Customer、Vendor、Product)被称为维表(Dimension Table)。不管是哪张表，虽然对于用户来说，看起来就是一张表，实际上，在每个 Instance 上都有这样 4 张表，分别叫 Order、Customer、Vender 和 Product，针对某一张表来说，每个 Instance 上都存储有一部分数据，所有 Instance 上的这张表的数据的集合组成了这张表的全部数据，这类似分库分表的 Sharding 概念，这样理解起来可能会容易一些。



GP 系统中所有的用户表都是分布的(复制表除外)，这意味着数据被拆分成无重叠

的记录集合。每部分存储在一个 Instance 中。数据通过复杂的 HASH 算法分布到所有 Instance。HASH KEY(一个或者多个)由管理员在定义 Table 时指定。

GP 从底层上来说，通过一系列相关的独立 Database 实现，由一个 Master 和数个 Instance 组成。Master 不存储用户数据。Instance 存储每张表无重叠的一部分数据子集(对于复制表，每个 Instance 都存储一份完整的数据)。

解读 GP 分布策略

在 GP 中创建(Create)或者修改(Alter)表时，有一个额外的 DISTRIBUTED 子句用以定义表的分布策略(Distribution Policy)。分布策略决定了表中的数据记录如何被分散到不同的 Instance 上。GP 提供了 3 种分布策略：

- **HASH 分布**

使用 HASH 分布时，一个或数个(强烈建议避免选多个)Table Column 可以被用作 Distribution Key(简称 DK)。通过 DK 计算出一个 HASH 值用来决定每条记录分散到哪个 Instance 上。相同 Key 值的记录会 HASH 到相同的 Instance。选择一个唯一性较高的字段作为 DK，可以确保尽可能平坦的分布数据。虽然，主键往往是唯一性最好的字段，但是，不建议为了选择一个分布键而去增加一个主键，这是一种逻辑颠倒的做法，通常，应该选择一个常用于大表之间关联的某个唯一性较高的字段作为分布键，一般这个字段可能在其他某个表中具有主键特征，例如，客户 ID，例如会员卡号，例如手机号码，例如身份证号码，等等，在选择分布键时，仅需要考虑大表与大表之间的关联，任何涉及到小表关联的场景均不应作为选择分布键的考虑因素。

如果可以，尽可能只选择一个字段作为分布键，因为，只有当关联字段包含全部的分布键时，分布键才对关联有帮助，除了空集(没有分布键的存储策略就是 Randomly 随机分布)，只有仅包含一个元素的集合才最容易成为其他集合的子集。如果可以确保组合分布键常常会被关联查询的字段全部包含，且没有一个合适的字段单独作为分布键，选择组合分布键也是可以的，但这只应该作为特例来考虑。

- **随机(Random)分布**

使用随机分布，数据记录被循环的分布到所有 Instance 上。相同值的记录可能会分布在不同的 Instance。随机分布可以确保数据分布的平坦性，但为了确保数据分布对查询有帮助，应该尽可能的使用 HASH 分布。

对于一些尺寸很小的表(叫维表或者参考表)来说，无所谓如何分布，所以，这样的表完全可以按照主键分布或者使用随机分布，对整体的分析查询性能不会有明显的影响。

- **复制(Replicated)分布**

复制分布，会在每个Instance上都存储一份完整的数据，复制表是在6版本新引入的数据分布策略，这里需要特别指出，复制表，因为需要在每个Instance上存储一份完整的数据，数据量大的事实表不适合选择复制这种分布策略，如果这么做，将会极大的浪费存储空间，同时，未必会带来性能的改善，对于复制表的理解，应该仅限于：复制表的存在，等于提前把广播做好了，减少了执行计划的复杂度，对于一些非常小的表，涉及的业务场景追求极致的性能时才考虑，对于通常的分析型场景，无需考虑复制表。对分布策略要理解透彻，不能过度迷信某一种分布策略，时常在社区听到有人说，复制表的性能更好，这是一种片面的理解，只能说，在某些特定的情况下，会表现出更好的性能。记住一句话即可，复制表的作用仅仅是提前把广播(Broadcast)做好了，仅仅如此，而已。

第三章：角色权限管理

GP通过角色(Role)的概念来管理数据库的访问权限。Role的概念包含两个子概念用户(User)和组(Group)。一个Role可以是一个DB User或者一个Group或者两者兼备。Role可以是DB对象(例如Table)的Owner(只是一种权限的体现，不是专有对象)并可以分配该对象的权限给其他Role从而实现对该对象的权限管理。Role还可以成为其他Role的成员，因此Role可以继承其父级Role的对象权限。

每个GP系统都包含一系列的Role(User或Group)。这些Role与运行在OS上的Role是没有关系的。如果是出于便利考虑，可以选择使用与OS Role相关联的GP Role，这样对于一些缺省使用OS User名称作为DB User的应用来说会比较方便，不过，往往不太需要这样的设计，因为极少有需要直接在Master主机上来访问GP的情况存在。

在GP中User通过Master登录和认证。而对于Instance的访问是Master通过内部实现完成的，与当前登录的User信息无关。

Role是定义在GP系统级别的，这就意味着，在其所在集群的全部DB实例中都是有效的，也就是说，Role是独立于DB存在的，特定的Role可以是某个DB的Owner，但这不等于说这个DB是该Role专有的，Owner只是一种权限的体现，没有绝对的隶属关系，例如SUPERUSER可以不受任何限制的访问任何DB和任何对象。

为了能够启动GP系统，在初始化系统时会自动包含一个SUPERUSER，该Role与执行该初始化操作的OS User相关。该Role与该OS User具有相同的Name。按照管理，这个Role使用gpadmin。为了能够创建其他Role，一开始需要使用该初始化Role来访问数据库系统。

角色与权限安全的最佳实践

- **保护系统User gpadmin**。GP需要使用一个Linux的User来安装和初始化GP系统。按照惯例，该系统User使用gpadmin。gpadmin用户作为GP系统的默认SUPERUSER，同时是GP安装目录及相关数据文件的Owner。默认的管理员账户是GP系统的基本要素，如果没有该账户，整个数据库系统将无法运行，**GP集群不可以使用root用户进行初始化**，另外，没有办法限制gpadmin用户的访问权限，因为这是第一个SUPERUSER。gpadmin用户可以绕过GP的所有权限限制。任何人通过gpadmin登录到GP主机后，都可以Read、Alter、Delete任何数据，包括系统表的访问和任何数据库操作。因此，保护好gpadmin用户账号是很重要的。超级用户(gpadmin)只应该用于执行特定的系统管理任务(例如备份恢复、故障处理、升级、扩容等)。一般的数据库访问不应使用gpadmin账号，ETL等生产系统也不应该使用gpadmin账号。**不要闲的无聊将gpadmin修改为NOSUPERUSER，弄不好如果系统中一个SUPERUSER都没了，可能就悲剧了。**

- **为每个登录的 User 分配不同的 Role。**出于登录和审计的需要，每个被允许登录到 GP 的使用者都应该分配一个属于自己的 Role。对于应用程序(APP)或者 Web 应用来说，应该考虑为每个 APP 或者 Server 创建独立的 Role。

- **使用 Group 来管理访问权限。**当登录的用户数量较多，且经常需要为类似的用户授予某一类的权限时，可以通过 Group 来进行权限的管理，将库中对象的权限赋予 Group，当某个 User 需要某类权限时，将该 User 加入到相应的 Group 中，此时该 User 就拥有了该 Group 所拥有的对象权限，Role 属性相关的权限无法被自动继承。

- **控制具有 SUPERUSER 属性的 User 数量。**具有 SUPERUSER 属性的 Role 将可以像 gpadmin 那样绕过 GP 的所有权限限制，包括资源队列(Resource Queue/RQ)或者资源组(Resource Group/RG)的限制。所以，应该仅为系统管理员分配 SUPERUSER 权限，**拥有 SUPERUSER 权限，几乎可以做任何事情，例如，删除任何的表，删除任何的 Role，删除任何的 DB，甚至删除操作系统上 gpadmin 用户有操作权限的任何文件和目录。**

创建用户 User Role

User Role 意味着其可以登录 DB 并发起 DB 会话。因此，在使用 CREATE ROLE 来创建一个 User 时，需要指定 LOGIN 权限。例如：

```
=# CREATE ROLE jsmith WITH LOGIN;
```

Role 还需要设置一系列的属性来决定其可以执行哪些数据库操作。可以在 CREATE ROLE 的时候指定这些属性，也可以在 CREATE 之后使用 ALTER ROLE 命令来完成。

修改 ROLE 属性

属性	描述
SUPERUSER | NOSUPERUSER	SUPERUSER 可以绕过所有所有权限限制，SUPERUSER 使用有风险，仅在需要的时候使用。CREATE ROLE 时缺省属性为 NOSUPERUSER。
CREATEDB | NOCREATEDB	是否有 CREATE DATABASE 权限。缺省为 NOCREATEDB。
CREATEROLE | NOCREATEROLE	是否有 CREATE 和管理其他 ROLE 的权限，缺省为 NOCREATEROLE。
CREATEEXTTABLE | NOCREATEEXTTABLE [(attribute='value'[, ...])] where attributes and values are:	是否有创建外部表的权限，缺省为 NOCREATEEXTTABLE。在设置外部表权限时还需要指定外部表的权限类型，包括[可读|可写]以及[gpfdist 协议|http 协议]等。

type='readable'|'writable' protocol='gpfdist'|'http'	
INHERIT | NOINHERIT	决定该 Role 是否继承其所属 Group 的权限。缺省属性为 INHERIT。INHERIT 表明，该 Role 无需单独指定对象的访问权限，该 Role 所属的 Group 具有的权限将会自动被该 Role 继承。
LOGIN | NOLOGIN	决定 ROLE 是否可以登录 DB。具有 LOGIN 属性的 ROLE 就是 USER。不具有 LOGIN 属性的 ROLE 往往被用来做权限管理(GROUP)。缺省为 NOLOGIN。
CONNECTION LIMIT connlimit	对于可以 LOGIN 的 Role 来说，决定其同时最多可以有多少个连接。缺省值为-1(无限制)。
PASSWORD 'password'	设置 Role 的 PASSWORD。如果暂时不打算让该 Role 登陆数据库，可忽略该属性，如果不指定密码，PASSWORD 会被设置为 NULL 并且始终无法登录。空密码也可以明确定义为 PASSWORD NULL。
ENCRYPTED | UNENCRYPTED	指定密码是否加密，缺省行为受 password_encryption 参数决定(缺省为 ON)。如果目前的密码已经使用了加密存储，可忽略该属性。
VALID UNTIL 'timestamp'	设置在指定的日期后该 Role 的密码失效。不设置的情况下为永远有效。
RESOURCE QUEUE queue_name	将 Role 分配到指定的资源队列(RQ)。所有的语句都受到该 RQ 的约束。需要注意的是该属性不会被继承，必须为每个 USER 指定该属性。缺省的是 pg_default。
RESOURCE GROUP group_name	将 Role 分配到指定的资源组(RG)。所有的语句都受到该 RG 的约束。需要注意的是该属性不会被继承，必须为每个 USER 指定该属性。缺省的是 default_group。
DENY {deny_interval | deny_point}	定义限制 Role 登录的时间段，在指定的时间段内不允许登录。可以指定日期或者日期加时间的格式。这些信息存储在 pg_catalog.pg_auth_time_constraint 系统表中。该功能鲜有使用，该系统表的维护一直存在明显的问题，该表没有约束限制，完全相同的限制信息可以被重复的存储在该系统表中。

例如下面的例子：

```
=# ALTER ROLE reuser WITH PASSWORD 'passwd123';
=# ALTER ROLE reuser VALID UNTIL 'infinity';
=# ALTER ROLE reuser LOGIN;
=# ALTER ROLE reuser RESOURCE QUEUE adhoc;
=# ALTER ROLE reuser DENY DAY 'Sunday';
```

除了上述的一些自有的属性外，ROLE 还可以配置一些参数的属性值，例如设置缺省的搜索路径：

```
=# ALTER ROLE admin SET search_path TO myschema, public;
```

需要注意的是，在 Role 上设置参数要进行适当的验证检查，例如，要在 Role 上设置内存参数 max_statement_mem 和 statement_mem 时需要注意，在系统级别，max_statement_mem 这个参数的值是多少，缺省为 2GB，如果在 Role 级别要设置的 statement_mem 超过系统级别的 max_statement_mem 值，虽然 Role 上的 max_statement_mem 参数值足够大，但仍可能会报错，因为这取决于哪个参数设置先生效。

创建用户组 Group Role

对于管理一组 User 来说，将它们绑定到一个 Group 是很方便的。通过这种方式，一组 User 可以通过一个 Group 来统一授予权限和回收权限。在 GP 中，通过 CREATE ROLE 的方式来创建 GROUP，并通过以角色作为权限实体的方式通过 GRANT 命令来为 USER 分组。

通过 CREATE ROLE 命令创建新的 GROUP ROLE：

```
=# CREATE ROLE admin CREATEROLE CREATEDB;
```

一旦 GROUP 创建好之后，就可以通过 GRANT 和 REVOKE 添加或者删除 Member(USER ROLE)了：

```
=# GRANT admin TO john, sally;
=# REVOKE admin FROM bob;
```

为了合理的管理对象权限，需要将合适的权限赋予 GROUP ROLE。因为其所有的 USER ROLE 成员都会继承该 GROUP ROLE 的对象权限。例如：

```
=# GRANT ALL ON TABLE mytable TO admin;
=# GRANT ALL ON SCHEMA myschema TO admin;
=# GRANT ALL ON DATABASE mydb TO admin;
```

不过，对于 ROLE 的 LOGIN、SUPERUSER、CREATEDB 和 CREATEROLE 等属性是不会像普通的权限一样被继承的。GROUP 的 USER 成员需要明确的使用 SET ROLE 命令进行获取。例如，admin ROLE 具备 CREATEDB 和 CREATEROLE 属性，sally 是 admin 的成员，其可以通过下面的语句来获取 admin ROLE 的属性权限：

```
=# SET ROLE admin;
=# SELECT current_role;
admin
```

这也提醒了我们，在使用 Group 做权限管理时要小心控制这种权限，同时，需要说明的是，对于普通 Role 来说，不要因为 SET ROLE 可以获得更高的权限而动歪心思，因为数据库日志中记录的操作信息仍然是你的大名。

管理对象权限

当一个对象(Table、View、Sequence、Database、Function、Language、Schema、Tablespace)被创建时，其会被分配一个所有者(Owner)。通常 Owner 是执行了创建语句的 User。对于大多数的对象(Object)来说，缺省只有其 Owner(或者 SUPERUSER)可以对其做不受限制的操作。要允许其他 User 使用该对象，需要使用 Grant 进行授权，有趣的是，Owner 还可以把自己的权限 Revoke，Revoke 之后自己就没有了相关的权限(还可以再 Grant 回来)，这真是一个神奇的设计。GP 对于每种对象支持的权限为：

对象类型	权限
Tables、Views、Sequences	SELECT INSERT UPDATE DELETE RULE ALL
External Tables	SELECT RULE ALL
Databases	CONNECT CREATE TEMPORARY | TEMP ALL
Functions	EXECUTE
Procedural Languages	USAGE
Schemas	CREATE USAGE ALL

注意：每个对象的权限必须被独立的授权。例如：被授予了一个 DB 的 ALL 授权不等于把该 DB 的所有对象都授权了。这仅仅是授权了 DB 自身的全部权限(CONNECT、CREATE、TEMPORARY)。

使用 GRANT 命令给指定的 Role 授予一个对象权限。例如：

```
=# GRANT INSERT ON mytable TO jsmith;
```

如果要控制Column级别的权限，可以在Grant的时候列出Column的名称，缺省在不列出Column名称的情况下，包含全部字段的权限。例如：

```
=# GRANT SELECT(col1) on TABLE mytable TO jsmith;
```

还可以通过DROP OWNED和REASSIGN OWNED命令来取消Role的Owner权限(只有该对象的Owner或者SUPERUSER可以执行这样的操作)。例如：

```
=# REASSIGN OWNED BY sally TO bob;
=# DROP OWNED BY visitor;
```

这里需要注意，DROP OWNED是删除所有Owner为指定Role的对象，这可能是一个风险很高的操作，而REASSIGN OWNED是找到所有Owner为指定Role的对象，将其Owner改为新的Role。当我们需要删除某个Role而并不希望删除任何依赖的对象时，应该使用REASSIGN OWNED而不是DROP OWNED。

在6版本开始，可以通过GRANT ALL IN SCHEMA命令来完成指定Schema内全部某类对象的授权。例如：

```
=# GRANT ALL ON ALL TABLES IN SCHEMA public TO bob;
=# REVOKE ALL ON ALL TABLES IN SCHEMA public FROM bob;
=# REVOKE ALL ON ALL FUNCTIONS IN SCHEMA public FROM bob;
=# REVOKE ALL ON ALL SEQUENCES IN SCHEMA public FROM bob;
```

需要注意的是，GRANT ALL IN SCHEMA语法只是将当前状态下Schema内现有的对象进行授权，之后创建的对象不包含在本次授权中，从原理上来说，GRANT ALL IN SCHEMA语法是一种内置的循环授权的方式，并不是在Schema上保存权限信息。

模拟Row级别的权限控制

GP目前还不支持Row级别的权限控制。Row级别的权限控制可以通过View的方式来模拟。可以通过添加一个Column的形式来存储权限信息，使用基于该Column的View来控制访问的Row，再将该View授权给相应的User。

密码加密

缺省情况下， GP使用MD5算法对库内存储的ROLE的密码进行加密存储，所有具有权限查看pg_authid系统表的用户都可以看到加密后的密码，不过这些密码都是经过MD5加密后的字符串，由于MD5加密算法的不可逆性，查看者无法看到真实的原始明文密码。当进行DDL的备份和恢复时，操作的是加密后的字符串，无法获取真实

的明文密码串。在设置密码的时候，密码就被加密了：

```
=# CREATE USER name WITH ENCRYPTED PASSWORD 'password';
=# CREATE ROLE name WITH LOGIN ENCRYPTED PASSWORD 'password';
=# ALTER USER name WITH ENCRYPTED PASSWORD 'password';
=# ALTER ROLE name WITH ENCRYPTED PASSWORD 'password';
```

当 password_encryption 参数设置为 on 时，ENCRYPTED 关键字是可以省略的，password_encryption 的缺省值为 on，password_encryption 的值决定了，当不指定 ENCRYPTED 或者 UNENCRYPTED 关键字时的缺省动作，当设置为 on 时，缺省等效于指定了 ENCRYPTED，即，对密码进行加密存储，如果指定了 UNENCRYPTED，在 pg_authid 系统表中的密码将会以明文的方式进行存储。例如：

```
=# CREATE ROLE name WITH UNENCRYPTED PASSWORD 'password';
=# SELECT rolpassword FROM pg_authid WHERE rolname='name';
 rolpassword
```

密码除了使用 MD5 进行加密，还可以使用 SHA-256 算法进行加密，该算法生成一个 64 字节的十六进制字符串，前缀为 sha256 字符。MD5 算法生成的加密密码前缀为 md5 字符。pg_authid 系统表中存储的加密密码，是通过对密码拼接用户名之后的字符串执行相应的加密算法得到的，同时以加密时的加密算法名作为前缀。例如：

```
=# CREATE ROLE name1 WITH PASSWORD 'password';
=# SET password_hash_algorithm TO 'sha-256';
=# CREATE ROLE name2 WITH PASSWORD 'password';
=# SELECT rolpassword FROM pg_authid WHERE rolname = 'name1';
md5c3ac1a95fe4ba7cac352448ff1e5d6ec
=# SELECT rolpassword FROM pg_authid WHERE rolname = 'name2';
sha2569fca82e96e5092248ec38609bfc503dc3e138b4a83e719c69e325128e289ac40
$ echo -n passwordname1|md5sum
c3ac1a95fe4ba7cac352448ff1e5d6ec  -
$ echo -n passwordname2|sha256sum
9fca82e96e5092248ec38609bfc503dc3e138b4a83e719c69e325128e289ac40  -
```

基于时间的登录认证

GP 允许管理员限制 Role 的登录时间。可以使用 CREATE ROLE 或者 ALTER ROLE 命令来管理基于时间的限制，使用 ALTER ROLE 可以随时修改时间限制。

访问限制可以控制到具体时间点，且约束的改变不需要删除或者重建 ROLE。为权限的管理提高了灵活性。

时间约束仅仅对于指定的Role有效。如果一个Role被另一个受时间约束的Role包含，该时间约束不会被继承。

时间约束仅仅是在LOGIN的时候有效。SET ROLE和SET SESSION AUTHORIZATION命令对于时间约束不受任何影响，也就是说，即便执行了这些语句也无法继承时间约束的设置。

需要的权限

要设置时间约束，SUPERUSER或者CREATEROLE权限是必须的。另外没有任何User可以给SUPERUSER设置时间约束，否则会得到如下报错：

```
ERROR:  cannot alter superuser with DENY rules
```

如何添加时间约束

有两种办法添加时间约束。在CREATE ROLE或者ALTER ROLE的时候使用DENY关键字并跟随如下的选项来实现：

- 某天或者某个时间的访问限制(不需要BETWEEN关键字)，例如：周二不允许登录。
- 一个有开始时间和结束时间的访问限制(需要BETWEEN AND)，例如：周二下午10点到周三上午8点不允许登录。

还可以指定多个限制，例如：周二的任何时间不允许登录并且周五的下午3点到5点不允许登录。

指明日期和时间

有两种方法指明哪一天。使用DAY关键字并紧跟英文的星期几或者0~6的数字，如下表所示：

英文表述	数字表述
DAY 'Sunday'	DAY 0
DAY 'Monday'	DAY 1
DAY 'Tuesday'	DAY 2
DAY 'Wednesday'	DAY 3
DAY 'Thursday'	DAY 4

DAY 'Friday'	DAY 5
DAY 'Saturday'	DAY 6

每日中的时间可以使用 12 小时或者 24 小时格式。在 TIME 关键字之后跟随单引号引起来的时间格式。仅仅含有小时和分钟的时间即可(也可以使用秒单位)，且使用冒号(:)作为分隔符。如果使用 12 小时格式，需要指定 AM 或者 PM 结尾以确定上下午。下面的例子表明了几种时间格式：

```
TIME '14:00' (24 小时格式的时间)
TIME '02:00 PM' (12 小时格式的时间)
TIME '02:00' (24 小时格式的时间) 其等价于 TIME '02:00 AM'
```

注意：时间约束是强制以服务器时间为准的。时区信息会被忽略。

指定时间段

要指定限制访问的时间段，需要两个[日期/时间]来确定，且通过 BETWEEN 和 AND 关键字连接。DAY 是必须的。

```
BETWEEN DAY 'Monday' AND DAY 'Tuesday'
BETWEEN DAY 'Monday' TIME '00:00' AND DAY 'Monday' TIME '01:00'
BETWEEN DAY 'Monday' TIME '12:00 AM' AND DAY 'Tuesday' TIME '02:00 AM'
BETWEEN DAY 'Monday' TIME '00:00' AND DAY 'Tuesday' TIME '02:00'
BETWEEN DAY 1 TIME '00:00' AND DAY 2 TIME '02:00'
```

最后 3 句是等价的。

注意：日期间隔不能跨越 Saturday(周六)。

```
Incorrect: DENY BETWEEN DAY 'Saturday' AND DAY 'Sunday'
```

正确语法为：

```
DENY DAY 'Saturday'
DENY DAY 'Sunday'
```

例子：

下面的例子说明在 CREATE ROLE 和 ALTER ROLE 的时候使用时间约束。这里只是展示了时间约束部分的命令。关于 CREATE ROLE 和 ALTER ROLE 的细节可参考相关章节。

例 1-创建包含时间约束的 ROLE，限制周末访问。

```
=# CREATE ROLE generaluser DENY DAY 'Saturday' DENY DAY 'Sunday';
```

例 2-修改 ROLE 添加时间约束，每天晚上 2:00 到 4:00 限制访问。

```
=# ALTER ROLE generaluser
DENY BETWEEN DAY 'Monday' TIME '02:00' AND DAY 'Monday' TIME '04:00'
DENY BETWEEN DAY 'Tuesday' TIME '02:00' AND DAY 'Tuesday' TIME '04:00'
DENY BETWEEN DAY 'Wednesday' TIME '02:00' AND DAY 'Wednesday' TIME '04:00'
DENY BETWEEN DAY 'Thursday' TIME '02:00' AND DAY 'Thursday' TIME '04:00'
DENY BETWEEN DAY 'Friday' TIME '02:00' AND DAY 'Friday' TIME '04:00'
DENY BETWEEN DAY 'Saturday' TIME '02:00' AND DAY 'Saturday' TIME '04:00'
DENY BETWEEN DAY 'Sunday' TIME '02:00' AND DAY 'Sunday' TIME '04:00';
```

例 3-修改 ROLE 添加时间约束，周三或者周五下午 3:00 到 5:00 限制访问。

```
=# ALTER ROLE generaluser
DENY BETWEEN DAY 'Wednesday'
DENY BETWEEN DAY 'Friday' TIME '15:00' AND DAY 'Friday' TIME '17:00;
```

删除时间约束

要删除时间约束，可使用 ALTER ROLE 命令。跟着 DROP DENY FOR，并跟着日期/时间。

```
DROP DENY FOR DAY 'Sunday'
```

任何与该条件有交集(互相有重叠关系)的约束都会被移除。例如存在的约束为 BETWEEN DAY 'Monday' AND DAY 'Tuesday'。那么删除'Monday'限制时会移除整个限制。原则是有交集即移除。例如：

```
=# ALTER ROLE generaluser DROP DENY FOR DAY 'Monday'…
```

这些限制登录的信息存储在 pg_catalog.pg_auth_time_constraint 系统表中。

第四章：配置客户端认证

在GP系统初始化成功之后，系统包含一个预定义的SUPERUSER ROLE。该USER的USER NAME与初始化GP系统的OS USER同名。该ROLE按照惯例使用gpadmin。缺省情况下，系统会被设置为只允许gpadmin从本地连接。为了让其他ROLE可以连接数据库，或者允许从远程主机连接数据库，必须配置相应的许可以允许这些连接。本章介绍如何配置客户端连接和认证。

- 允许连接到Master
- 限制并发连接

允许连接到Master

客户端的访问许可是通过一个叫做pg_hba.conf(也是标准的PostgreSQL的认证文件)的配置文件来控制的。关于该文件的细节可以参考PostgreSQL的文档。

在GP中，Master的pg_hba.conf文件控制着客户端连接到GP系统的认证。在Instance上也存在pg_hba.conf文件，通常此文件已经被正确配置为允许从Master访问。不过根据以往的经验来看，也出现过配置错误的情况，该情况会导致gpexpand之类的操作报错失败。通常来说，Instance是不需要接受外部客户端连接的(如果需要，必须通过Utility模式连接)，不太有必要去修改pg_hba.conf文件。编者编写的一些增值服务的命令中有些会涉及需要修改Instance的pg_hba.conf文件，不过，缺省情况下这些命令会自动完成这些必要的修改操作。

pg_hba.conf是一个平面文件，按照行来区分每条记录。空行会被忽略，任何在井号(#)后的字符串都会被忽略。每行记录由一系列Space和Tab混合分割的字段组成。如果需要在字段中出现空白字符，需要将字段用引号引起来。记录不可跨行。每条远程客户端的访问许可，都像这种格式：

```
host database role CIDR-address authentication-method
```

而每个UNIX本地连接的访问许可，都像这种格式：

```
local database role authentication-method
```

这些字段的含义如下：

字段	描述
local	匹配UNIX嵌套连接。如果没有这种记录，UNIX嵌套连接是不被允许的。
host	匹配TCP/IP方式的连接。除非该Server属于一个合适的IP

	段，否则其访问是不被允许的。
hostssl	匹配 TCP/IP 方式的 SSL 加密连接。这个配置需要配合 SSL 参数的设置，该参数在 GP 启动时生效。
hostnossl	匹配 TCP/IP 方式的非 SSL 加密连接。
database	设置该记录匹配的 DB Name。all 可以匹配全部 DB。多个 DB Name 可以使用逗号(,)分割。或者使用@符号跟随文件名的方式指定，该文件包含需要匹配的 DB Name。
role	匹配哪个 ROLE。all 可以匹配全部的 ROLE。如果想把一个 GROUP 的所有成员匹配上，可以在 ROLE Name 前使用加号(+)表示。多个 ROLE Name 可以使用逗号(,)分割。或者使用@符号跟随文件名的方式指定，该文件包含需要匹配的 ROLE Name。
address	指定该记录匹配的客户端 IP 地址或者范围，也可以是一个 hostname。如果是 IP，其包含一个标准的小数点分割的 IP 地址和一个掩码长度值。IP 地址只能使用数字形式。掩码长度表示 IP 地址高位与客户端 IP 匹配的长度。指定的掩码长度右边的二进制 IP 地址必须是 0。IP 地址与分隔符(/)和掩码长度之间不可以有任何的空字符。例如 172.20.143.89/32。其只能匹配 172.20.143.89IP 地址。172.20.143.0/24 可以匹配 172.20.143 开始的任何 IP 地址。要匹配单个 IP 地址 IPv4 使用 32 作为掩码长度，IPv6 使用 128 作为掩码长度。如果使用 hostname 来配置，hostname 的解析依赖/etc/hosts 文件的配置或者 DNS 的解析，如果 hostname 解析出的 IP 地址与访问时的 IP 地址不能匹配，则访问会被拒绝。通常可能没有必要使用 hostname 来进行配置，这个特性主要是为了 gp4k 而新增的功能。
IP-address IP-mask	通过标准子网掩码的格式作为掩码长度的可选方案。其被作为一个单独的字段。255.0.0.0 等效于 IPv4 的 8 位掩码长度。255.255.255.255 等效于 IPv4 的 32 位掩码长度。例如：192.168.0.0 255.255.0.0 与 192.168.0.0/16 等价
authentication -method	指定连接时使用的认证方法。例如 trust 为不需要密码，md5 为使用 md5 加密认证。更多细节可以查看 PostgreSQL8.4 的文档中认证方法的部分。

编辑 pg_hba.conf 文件

下面的例子展示如何编辑 Master 上的 pg_hba.conf 文件从而允许远程的客户端通过加密认证的方式访问数据库。

编辑 pg_hba.conf 文件

1. 文本编辑器(例如 VI)打开$MASTER_DATA_DIRECTORY/pg_hba.conf 文件。

2. 为每类需要允许的连接添加一行记录。记录是被顺序读取的，所有记录应该备有序的安排。通常前面的记录匹配更少的连接但要求较弱的认证，后面的记录匹配更多的连接但要求更严格的认证。例如：

```
# allow the gpadmin user local access to all databases
# using ident authentication
local all gpadmin ident sameuser
host all gpadmin 127.0.0.1/32 ident
host all gpadmin ::1/128 ident
# allow the 'dba' role access to any database from any
# host with IP address 192.168.x.x and use md5 encrypted
# passwords to authenticate the user
# Note that to use SHA-256 encryption, replace md5 with
# password in the line below
host all dba 192.168.0.0/32 md5
# allow all roles access to any database from any
# host and use ldap to authenticate the user. Greenplum role
# names must match the LDAP common name.
host all all 192.168.0.0/32 ldap ldapserver=usldap1 ldapport=1389
ldapprefix="cn=" ldapsuffix=",ou=People,dc=company,dc=com"
```

3. 保存并关闭文件

4. 重新加载 pg_hba.conf 文件从而使得刚刚的修改生效：

```
$ gpstop -u
```

注意：pg_hba.conf 文件中的记录是顺序匹配的，当某个登录被前面的记录匹配了，将不会继续匹配后面的记录。所以，一定要避免记录之间有互相包含的关系出现，否则不容易发现登录失败的原因。编辑这个文件时一定要注意保证输入的正确性，避免 Windows 隐藏符号等特殊不可见字符的出现，也不能随意修改初始化时自动生成的记录，错误的修改可能会导致数据库无法访问，甚至出现无法执行 gpstop 等尴尬情况，那时只能使用 kill -9 来强行杀死数据库的服务进程，这是不应该发生的。

限制并发连接

为了限制对 GP 系统的并发访问，可以通过配置 Server 参数 max_connections 来实现。这是一个本地化参数，就是说，需要把 Master，Standby 以及所有的 Instance 都修改。通常建议 Instance 的值是 Master 的 5-10 倍，不过这个规律并非总是如此，在 max_connections 比较大的时候通常没有这么高的倍数，2-3 倍也是允许的，但无论如何 Instance 的值不能小于 Master。在设置

max_connections 时，其依赖的参数 max_prepared_transactions 参数也需要修改，该参数的值至少要和 Master 上的 max_connections 值一样大，另外 Instance 上的值要与 Master 相同。

例如：

在$MASTER_DATA_DIRECTORY/postgresql.conf(包括 Standby)中：

```
max_connections=100
max_prepared_transactions=100
```

在所有的 Instance 上 SEGMENT_DATA_DIRECTORY/postgresql.conf 中：

```
max_connections=500
max_prepared_transactions=100
```

修改最大连接数的步骤

1. 通过 gpstate 命令确认数据库状态无异常，如：

```
$ gpstate -e
$ gpstate
$ gpstate -f
```

2. 使用 gpconfig 命令修改参数值：

```
$ gpconfig -c max_connections -v 1500 -m 500
$ gpconfig -c max_prepared_transactions -v 500
```

3. 执行 CHECKPOINT 操作：

```
$ psql postgres -c "CHECKPOINT"
```

4. 停止数据库：

```
$ gpstop -f
```

5. 重启数据库：

```
$ gpstart -a
```

注意：增加该参数的值可能需要更多的共享内存，需要注意操作系统参数方面内存相关的参数配置。

客户端/服务端间的加密连接

GP原生支持客户端与Master服务端之间的SSL连接。SSL连接可以有效的防止第三方对包的窥探，防止中间层的攻击。在非安全网络环境中有必要使用SSL，且在使用权限认证时更为必要。使用SSL需要在客户端和Master端都安装有OpenSSL。在设置参数ssl=on(在Master的postgresql.conf文件)后启动就开启了SSL。在使用SSL模式启动时，数据库会查找Master目录下的server.key(服务器密钥)文件和server.crt(服务器证书)文件。这些文件必须被正确的安装，否则数据库系统将无法启动。

重要提示：不要为server.key设置访问口令。数据库不会为密钥提示输入口令，这样会导致出错并无法启动数据库系统。

关于如何安装OpenSSL，可参考PostgreSQL的相关文档。

第五章：访问数据库

本章介绍可以使用哪些客户端工具连接GP，以及如何建立数据库会话：

- 建立数据库会话
- 支持的客户端应用
- 连接故障排除

建立数据库会话

用户可以使用任何PostgreSQL兼容的客户端程序连接到GP，例如psql。用户或者管理员总是通过访问Master来连接到GP数据库，通常，Instance不能直接使用客户端连接，如确有必要，需要使用Utility模式来连接。

要连接到GP的Master,需要知道下面这些连接参数并在客户端程序进行正确的配置。

连接参数	描述	环境变量
Application name	连接到数据库的应用名称，该参数为可选项。	$PGAPPNAME
Database name	需要连接的数据库名称。对于新初始化的系统来说，首次可以使用postgres。	$PGDATABASE
Host name	要连接的GP Master的主机名称。缺省为localhost。远程连接可能需要IP地址。	$PGHOST
Port	GP Master上Instance的端口号。缺省为5432.	$PGPORT
User name	要连接的用户名。其没必要与OS的User Name相匹配。在不知道User Name的情况下最好联系询问数据库管理员。每个GP系统都有一个初始化时产生的SUPERUSER。该User Name与初始化的OS User Name相同(通常为gpadmin)。	$PGUSER

支持的客户端应用

可以使用这些客户端应用连接GP：

- 在GP安装时提供的客户端应用。psql提供了交互式的命令行方式访问GP。
- 针对GP的pgAdminIII作为一种强化版本支持GP。从1.10.0版本开始，

PostgreSQL 客户端工具 pgAdminIII 开始支持 GP 特性。安装包可以从 pgAdmin 网站下载。因 pgAdminIII 很久没有更新了，对于 5 版本和 6 版本中的很多特性没有支持，编者为此做了定制开发，针对 5 版本和 6 版本的新功能和系统表的修改进行了必要的修改，可以很好的兼容开源 5 版本和 6 版本。

- 使用标准数据库应用接口，例如 ODBC、JDBC，用户可以开发出自己的客户端程序。由于 GP 基于 PostgreSQL 而来，可以直接使用 PostgreSQL 驱动访问 GP。
- 使用标准数据库应用接口的客户端程序，例如使用 ODBC、JDBC 的客户端程序，可以通过配置的方式连接到 GP。

GP 的客户端应用程序

在 GP 安装时在 Master 主机的$GPHOME/bin 下有一系列的客户端应用程序。下面是一些常用的客户端应用程序：

名称	用途
createdb	创建新的数据库
createlang	创建新的程序语言
createuser	创建新的数据库 ROLE
dropdb	删除数据库
droplang	删除程序语言
psql	PostgreSQL 交互式命令
reindexdb	将数据库重建索引
vacuumdb	回收数据库的磁盘空间并分析数据库

在使用这些客户端应用程序时，必须通过 GP Master 实例来访问数据库，必须知道目标 DB Name、Host Name、Port 以及连接使用的 User Name。这些参数在命令行可以分别使用-d、-h、-p 和-U 提供。没有指定任何选项的参数会优先被解读为 DB Name。

那些有缺省值的参数可以不指定。缺省的 Host Name 是 localhost。缺省的 Port 是 5432。缺省的 User Name 是当前的 OS User Name，在不指定数据库名称参数时，当前用户的 User Name 会被当作 DB Name 来使用。GP 的 User Name 与 OS User Name 未必相同。

如果缺省参数值是错误的，可以选择将正确的值保存在环境变量 PGDATABASE、PGHOST、PGPORT、PGUSER 中。在~/.pgpass 中设置合适的值可以避免反复输入密码的麻烦。

使用 psql 连接

根据缺省值或者环境变量，下面的例子说明如何通过 psql 连接：

```
$ psql -d gpdatabase -h master_host -p 5432 -U gpadmin
$ psql gpdatabase
$ psql
```

如果还没有用户数据库存在，可以连接系统数据库 postgres。例如：

```
$ psql postgres
```

在成功连接到数据库之后，psql 会出现一个提示符，包含连接的 DB Name 和一串字符(=>)(或者=#，仅当该用户是 SUPERUSER 时)。例如：

```
postgres=>
```

在提示符处，就可以直接输入 SQL 命令执行了。SQL 命令必须以分号(;)结尾才能将命令发给 Master 并被数据库执行。例如：

```
=> SELECT * FROM mytable;
```

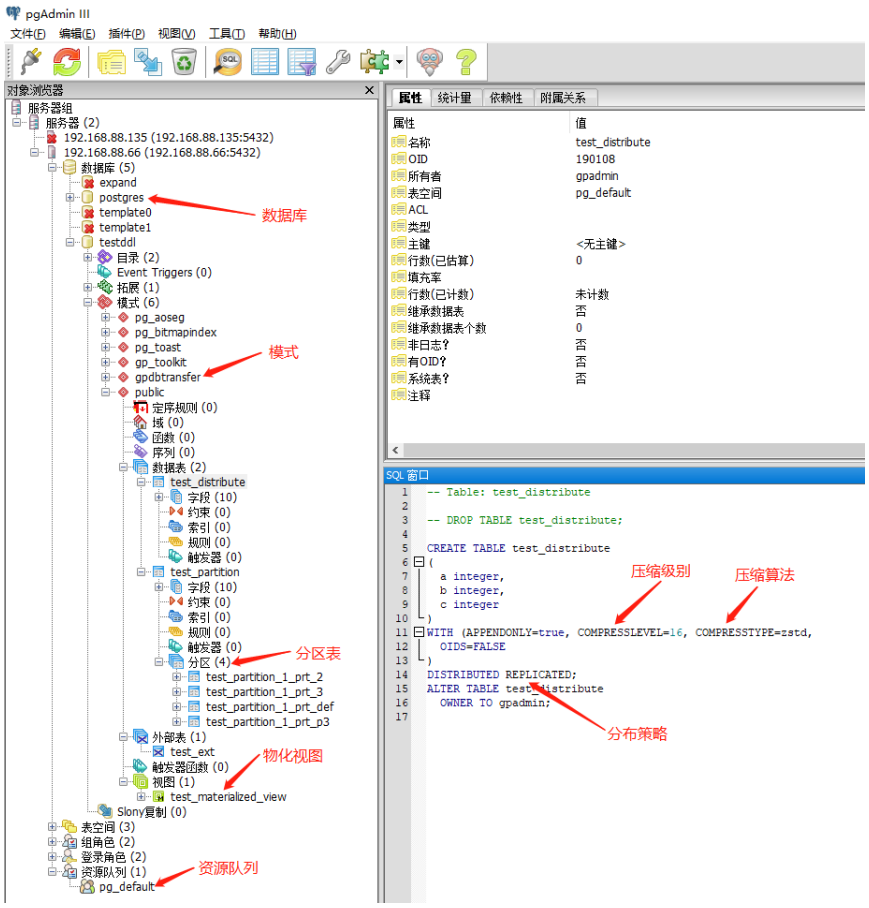
要得到关于 psql 客户端应用程序的更多信息，可以查看 PostgreSQL 的相关文档。

针对 GP 的 pgAdminIII

如果更喜欢图形化界面(有谁不喜欢呢)，可以使用针对 GP 的 pgAdminIII。该 GUI 客户端除了支持标准 PostgreSQL 外，还支持一些 GP 的专有特性。

针对 GP 的 pgAdminIII 支持下列的 GP 专有特性：

- 外部表(External tables)
- 追加优化表(Append-Optimize table)，压缩表(Compressed Append-Optimize table)
- 图形化的解释器(EPLAIN ANALYZE)
- Server 的参数配置



安装针对 GP 的 pgAdminIII

支持 5 版本和 6 版本的安装包可以从编者处获取 (目前仅提供给编者服务的客户使用，本文档中提到的其他编者自己的工具命令也是仅提供给编者服务的客户使用，至少到目前为止，没有公开传播)，获取的将是一个 rar 的压缩包文件，直接解压成目录后即可运行使用。

使用 pgAdminIII 执行管理操作

该节介绍诸多 GP 管理操作中的两个重要部分:编辑服务器配置,图形化查看执行计划。

编辑服务器配置

pgAdminIII提供了修改Server配置文件postgresql.conf的方式：通过[工具]>[服务器配置]> postgresql.conf。不过，强烈不推荐使用这种方式修改服务器参数配置，请使用gpconfig命令完成参数的修改配置。

远程编辑服务器配置

1. 连接到需要修改的数据库。如果连接了多个数据库，要确保已经选中需要修改的数据库。

2. 选择[工具]>[服务器配置]>**postgresql.conf**菜单。配置信息将会以列表的形式打开。
3. 双击需要修改的参数打开一个参数设置对话框。
4. 输入参数的新值。修改好之后点击[确定]按钮保存修改，或者点击[取消]按钮放弃修改。
5. 如果修改的参数可以通过重新加载配置的方式生效，点击左上角的绿色箭头来完成。有些参数的修改是需要重启数据库(不是gpstop -u)才能生效的。

查看执行计划

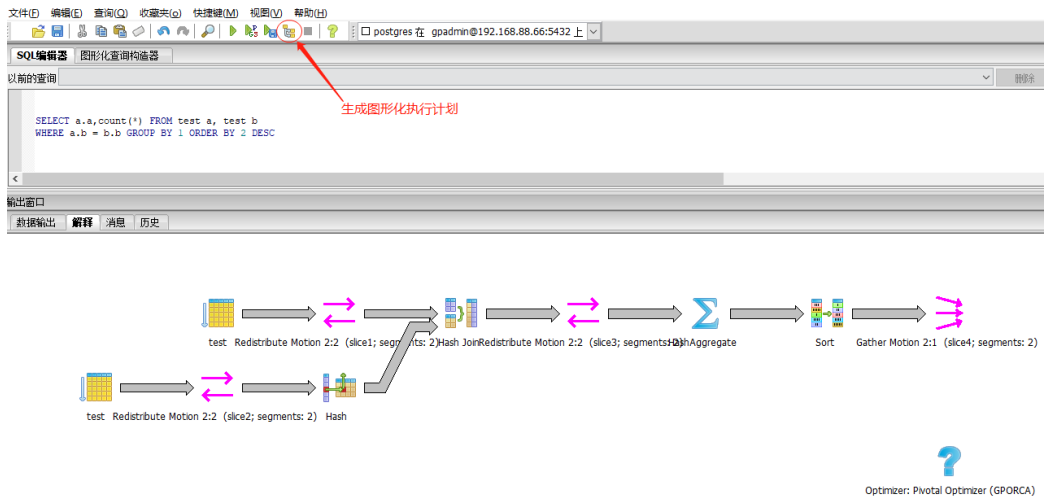
使用pgAdminIII工具，可以通过执行EXPLAIN命令查看执行计划。输出内容包括GP的分布式查询处理步骤，如Hash，Sort，Merge，Join，Filter等，以及Instance之间数据移动Motion。还可以查看图形化的执行计划，这将非常有助于对执行计划进行直观的分析。

查看图形化的执行计划

1. 在正确的数据库连接下，选择[工具]>[查询工具]。
2. 使用SQL编辑器输入查询语句，还可以通过图形化对象编辑器或者打开一个SQL文件的方式。
3. 选择[查询]>[解释选项]确认下面的选项：

   详细模式--如果想查看图形化的执行计划，需要取消该选项
4. 点击查询面板上端的执行计划按钮，或者使用快捷键[F7]，或者选择[查询]>[解释]。

执行计划在屏幕的底部展现。例如：



DB应用程序接口

若需要开发针对GP的应用程序，PostgreSQL提供的一些通用的API同样可以应用在GP上。这些驱动包并没有与GP一起发布，而是一些独立的项目，需要单独下载和安装配置从而连接GP。有下面这些驱动可以获取：

API	PostgreSQL Driver	下载连接
ODBC	pgodbc	可以从 GP 或者 PG 的官网获得。
JDBC	pgjdbc	可以从 GP 或者 PG 的官网获得。
Perl DBI	pgperl	http://gborg.postgresql.org/project/pgperl
Python DBI	pygresql	http://www.pygresql.org

使用通用API来访问GP的说明：

1. 下载相应的语言平台和对应的API。例如下载JDK和JDBC。
2. 编写相应的程序连接GP。需要注意SQL的语法支持问题。

下载合适的PostgreSQL驱动并配置到Master Instance的连接。

第三方客户端工具

很多第三方的ETL和BI工具使用标准的API如ODBC、JDBC，都可以通过配置连接到GP。下面这些工具经过用户证实可以很好的协同GP一同工作：

- Business Objects
- Microstrategy
- Informatica Power Center
- Microsoft SQL Server Integration Services (SSIS) and Reporting Services (SSRS)
- Ascential Datastage
- SAS
- Cognos

GP专业服务可以协助用户配置他们选定的第三方工具协同GP工作。

连接故障排除

有很多导致客户端程序无法成功连接GP的原因。本节介绍一些常见的问题并说明如何解决这些问题。

问题	解决办法
No pg_hba.conf entry for host or user	要允许远程客户端连接到GP，必须正确的配置Master的pg_hba.conf文件。
Greenplum Database is not running	在Master失败的情况下，用户是无法连接的。可以通过在Master 上使用gpstate工具检查GP系统是否已经启动。
Network problems Interconnect timeouts	从远程连接 Master时，网络问题可能会妨碍连接，例如DNS解析错误。要排除这种问题可先ping Master主机，先确保需要连接的主机可以ping通Master。另外需要分清localhost与实际的Host Name。当然有时候还存在网络防火墙的问题，这种情况下，可以ping通Master，在Master主机通过psql可以连接数据库，但从客户端连接Master无法连接，此时应该寻求相关的网络管理人员的帮助。
Too many clients already	缺省情况下，Master最大连接数是250，Instance为750。超过这个限制的请求会被拒绝。该限制是由postgresql.conf文件中的max_connections参数配置的。若修改Master上的该参数，还必须修改Instance上该参数为合适的值。

第六章：资源管理

本章介绍GP的资源管理的概念，GP提供了一些功能来帮助用户管理资源，根据业务的情况来控制资源的使用，防止出现资源的恶性竞争。可以通过资源管理来限制并发执行的查询数量，内存的消耗量，以及CPU的使用量。GP提供了两种资源管理的方案，资源队列和资源组。

注意：在RedHat6或者CentOS6中使用资源组是有问题的，这是因为早期的cgroup有缺陷，最好将Kernel升级到2.6.32-696或者更高的版本以修复已知问题，从而可以更好的使用资源组功能。这些问题在Redhat7或者CentOS7中已经修复。

资源队列或者资源组这两种资源管理方案同时只能选择使用一种，无法在一个集群中同时使用两种管理方案。

在初始化数据库时，缺省启用的是资源队列方案。在使用资源队列的时候，可以创建和管理分配资源组(不是完全没有限制的，例如不能设置CPUSET)，但要真正启用资源组方案，必须明确的启用资源组，且需要重启数据库以使其生效。

下表列举了资源队列和资源组之间的差异：

功能点	资源队列	资源组
并发	查询级别的控制	事务级别的控制
CPU	指定查询的优先级	指定 CPU 资源的百分比或 CPU Core
内存	在资源队列中控制，允许超过限制	在事务级别控制，更精准，不允许超过限制
内存隔离	无	在资源组之间隔离，在同一资源组内的不同事务之间隔离
用户	资源配额仅对非管理员用户有效	资源配额对管理员用户和非管理员用户同样有效
排队	仅当没有槽位可用时开始排队	当没有槽位可用时或者没有足够的可用内存时开始排队
查询失败	当没有足够的可用内存时可能会立即失败	当事务达到资源组限制的内存，且没有更多的共享资源组内存时，此时事务仍需要更多的内存，会导致查询失败
跳过限制	对超级用户和某些操作和函数不限制	对 SET、RESET 和 SHOW 命令不限制
外部组件	无	可以管理PL/Container的CPU和内存资源

使用资源组

在GP数据库中，可以使用资源组(RESOURCE GROUP)来压制CPU、限制内存和控制最大并发事务数量。在创建了资源组之后，可以将该资源组分配给多个ROLE，还可以分配给PL/Container这种外部组件，以控制这些ROLE和外部组件的资源。

当把一个资源组分配给一个ROLE(基于角色的资源组)时，资源的限制将影响到该资源组上的所有ROLE。例如，一个资源组上的内存配额，将限制该资源组上所有ROLE正在执行的事务的总的内存使用量上限。

同样，当把一个资源组分配给一个外部组件时，资源配额将影响到该外部组件上的所有正在执行的实例。例如，为PL/Containe创建了一个资源组，该资源组的内存配额，将限制该外部组件上所有正在运行的实例的总内存使用上限。

- **资源组基于角色或基于外部组件**
- **资源组的属性**
  - **内存管理模式**
  - **并发事务数限制**
  - **CPU配额**
  - **内存配额**
- **配置与使用资源组**
  - **启用资源组**
  - **创建资源组**
  - **配置基于内存限制的查询终止**
  - **分配资源组给ROLE**
- **监控资源组状态**
- **转移查询的资源组**

资源组基于角色或基于外部组件

GP有两类资源组，分别是为ROLE管理资源的资源组和为外部组件(如PL/Container)管理资源的资源组。资源组最普遍的用途是用于限制GP数据库中活跃事务的数量，当然，也可以用于管理CPU和内存资源的使用量。

基于角色的资源组通过Linux Kernel的cgroup来管理CPU资源，GP数据库使用名为vmtracker的内存管理模式为基于角色的资源组进行内存的管理。

当执行一个查询时，数据库会根据ROLE所属的资源组的资源配额进行评估，当所属资源组的资源没有达到配额限制，且资源组的并发事务限制也没有达到，查询将会立

即被执行。假如，资源组中的并发事务数限制已经达到，后续的查询将要进行排队等待，直到前面有查询结束。当增加资源组的并发事务数量限制和内存配额，正在排队的查询也可能会马上得到执行，也就是说这种限制会随着资源组属性的修改而重新评估。在基于角色的资源组中，查询的排队是先进先出的，先排队的先得到执行，后排队的只有等到前面排队的事务都得到执行并且空出资源时才会被执行，数据库会定期评估负载情况以决定资源的分配和执行排队的语句。

使用基于外部组件的资源组来管理外部组件的 CPU 和内存资源。这种资源组，使用 cgroup 来管理外部组件的 CPU 和内存的使用总量。

注意：GP 的容器化部署，例如 Greenplum for Kubernetes(GP4K)，可能会创建一组嵌套的 cgroup 配置来管理系统资源，这可能会影响 GP 的资源组管理 CPU 的使用率、Core 数量和内存使用量，资源组的资源限制将受限于上层的资源配额。

例如，GP 数据库运行在 cgroup 的限制中，那么 GP 的 cgroup 则会嵌套在上层的 cgroup 中，上层的系统为 GP 配置了 60%的系统 CPU 配额，GP 的资源组配置了 90%的 CPU 配额，那么 GP 可以利用的系统 CPU 为 60% x 90% = 54%。

嵌套的 cgroup 不影响基于外部组件(如 PL/Container)的资源组对内存的配额，只有 GP 的资源组的 cgroup 被配置为顶级 cgroup 时才能管理外部组件的内存配额。

资源组的属性

在创建资源组时，需要确定内存管理模式以确定资源组的类型，以及，配置该资源组可用的 CPU 和内存的资源配额。

资源组的属性有：

属性(限制类型)	描述
MEMORY_AUDITOR	资源组的内存管理模式。基于角色的资源组需要配置为 vmtracker(缺省值)，基于外部组件的资源需要配置为 cgroup。
CONCURRENCY	最大活跃并发事务数量，idle 的事务也包含在内。
CPU_RATE_LIMIT	资源组的 CPU 资源的百分比配额。
CPUSET	资源组保留的 CPU core 的数量。
MEMORY_LIMIT	资源组的内存资源的百分比配额。
MEMORY_SHARED_QUOTA	资源组中，事务之间可共享的内存百分比。
MEMORY_SPILL_RATIO	资源组中，内存密集型事务使用的内存百分比上限，超过该限制后需要溢出到文件。

注意：资源组对 SET、RESET 和 SHOW 命令不做资源限制，因为这种操作本来就不需要消耗很多资源。

内存管理模式

通过 MEMORY_AUDITOR 属性来确定资源组的类型，vmtracker 表明这是一个基于 ROLE 的资源组，cgroup 表明这是一个基于外部组件的资源组。MEMORY_AUDITOR 属性的缺省值是 vmtracker，即，基于 ROLE 的资源组。为资源组指定内存管理模式来确定 GP 是否以及如何使用资源配额属性来管理 CPU 和内存资源：

属性(限制类型)	基于 ROLE	基于外部组件
CONCURRENCY	Yes	No；必须是 0，就是不管
CPU_RATE_LIMIT	Yes	Yes
CPUSET	Yes	Yes
MEMORY_LIMIT	Yes	Yes
MEMORY_SHARED_QUOTA	Yes	组件相关
MEMORY_SPILL_RATIO	Yes	组件相关

注意：对于内存管理模式配置为 vmtracker 的资源组，GP 支持基于内存使用量来自动终止查询。与参数 runaway_detector_activation_percent 有关。

并发事务数限制

CONCURRENCY 属性配置基于 ROLE 的资源组中最大并发事务的数量。

注意：CONCURRENCY 限制不适用于基于外部组件的资源组，这种资源组必须设置该属性为 0。

基于角色的资源组会从逻辑上根据 CONCURRENCY 属性划分等量的槽位，同时会为这些槽位配额相同百分比的内存资源。

基于角色的资源组 CONCURRENCY 属性的缺省值为 20。

在资源组达到并发事务数限制(CONCURRENCY)后，任何提交的事务都将排队，当正在执行的事务完成之后，在有充足内存资源的情况下，数据库将开始执行最早排队的事务。

可以通过设置参数gp_resource_group_bypass为TRUE绕过资源组的并发事务限制，不过，通过这种方式提交的事务，其资源是受限的，而且仍有可能会因为资源不足而失败。

可以设置 gp_resource_group_queuing_timeout 参数来指定事务排队的时间长度，超时之后，数据库将 cancel 该事务，该参数缺省值为0，意思是排队时间长度没有限制。

CPU 配额

通过 CPU_RATE_LIMIT 来配置资源组可用的 CPU 的百分比，使用 CPUSET 来配置资源组专用的 CPU 的 Core 数量，二者是不同的 CPU 资源的配额模式。配置资源组时，必须选择其中一种。

GP 允许在不同的资源组中同时使用这两种 CPU 配额模式，也可以在使用过程中随时修改 CPU 的配额模式。

参数 gp_resource_group_cpu_limit 用于配置每个主机上最大分配给 GP Instance 的 CPU 资源的百分比。不管使用哪种 CPU 配额模式，该参数都控制着所有资源组的最大 CPU 使用率。其余的资源需要留给操作系统和数据库的主进程使用。参数 gp_resource_group_cpu_limit 的缺省值为0.9(90%)。

注意：如果在 GP 集群的主机上还有其他程序，gp_resource_group_cpu_limit 的缺省值是没有为其他程序预留 CPU 资源的，如果要为其他程序保留 CPU 资源，可能需要调整该参数。

注意：应该尽量避免将 gp_resource_group_cpu_limit 设置为大于0.9，这样可能会导致 GP 工作负载抢占所有的 CPU 资源，从而影响数据库的主进程获取不到足够的 CPU 资源。

按照 Core 来配额 CPU

使用 CPUSET 属性来指定哪些 CPU 的 Core 为资源组专用，被指定的 Core 必须是在系统中存在的，且不能已经分配给其他资源组，虽然 GP 将这些 Core 指定为该资源组专用，但操作系统中的其他非 GP 数据库的进程仍然可以使用这些 CPU Core。

使用 CPUSET 时，需要使用单引号引起来一串以逗号分隔的 Core 序号或者序号范围。例如：'1,3-5'。

将 CPU 的 Core 分配给 CPUSET 型的资源组时，需要考虑以下因素：

- 使用CPUSET型的资源组时，指定的Core是被资源组独占的，即便该资源组中没有正在执行的事务，这些CPU Core仍处于空闲状态，不会被其他资源组使用，所以，应该尽可能的避免设置过多的Core数量，避免系统的CPU资源浪费。

- 不要分配0号Core，0号Core需要保留以备不时之需：
  - 缺省资源组admin_group和default_group至少需要一个Core，当把所有的Core都指定为其他资源组专用时，缺省的资源组就只能使用0号Core，此时，admin_group和default_group将与占用了0号Core的资源组共用0号Core。
  - 当使用一个新的机器来替换现有集群中的一个机器，而新的机器的CPU Core的数量减少了，就没有足够的Core来满足CPUSET的配置，数据库将会把0号Core分配给这些资源组以避免启动失败。

- 将CPU Core分配给资源组时，应该尽量使用数字小的序号。否则，当更换一个CPU Core的数量减少的机器时，或者将数据库备份并恢复到一个CPU Core的数量变少的集群时，资源组的创建可能会失败，因为在新的机器上数字大的Core序号可能不存在。

通过 CPUSET 配置的资源组，在 CPU 资源的使用上有更高的优先级，其最高的 CPU 使用率是分配的 Core 数量在节点全部 Core 数量的百分比。

在使用 CPUSET 配置资源组时，将不能设置 CPU_RATE_LIMIT 属性，会被自动设置为-1，这两个属性不能同时配置，只能二选一。

按照百分比来配额 CPU

对于 GP 集群中的每个机器来说，CPU 资源是可以按照百分比进行均分的，通过 CPU_RATE_LIMIT 来配置的资源组，系统将会为其保留指定百分比的 CPU 资源。在设置 CPU_RATE_LIMIT 参数时，可设置的最小值是 1，最大值是 100，同时，系统中所有资源组的 CPU 百分比的总和不能超过 100。

通过 CPU_RATE_LIMIT 参数设置的所有资源组的 CPU 资源配额的总和还受限于：

被 CPUSET 独占剩余的 CPU Core 的数量 ÷ 机器上 CPU Core 的总数 × 100 × gp_resource_group_cpu_limit 参数的值。

通过 CPU_RATE_LIMIT 配置的资源组，其可以使用的 CPU 资源是弹性的，不是固定的，数据库可能会将空闲的资源组的 CPU 资源分配给其他繁忙的资源组使用，不过，一旦被划走 CPU 资源的资源组开始有事务执行，CPU 资源将会重新分配回去。对于有多个资源组在繁忙的情况，他们将按照 CPU_RATE_LIMIT 配置的值，按照比例获取空闲资源组的 CPU 资源，例如，CPU_RATE_LIMIT 设置为 40 的资源组获得的 CPU 资源将是 CPU_RATE_LIMIT 设置为 20 的资源组的 2 倍。

在使用 CPUSET 配置资源组时，将不能设置 CPU_RATE_LIMIT 属性，会被自动设置为-1，这两个属性不能同时配置，只能二选一。

内存配额

启用资源组之后，内存的使用可以在 GP 数据库的 Host、Instance 和资源组层面来管理和控制，还可以通过资源组在事务层面来控制。

参数 gp_resource_group_memory_limit 设置了每个 GP Host 主机上可以分配给资源组的系统内存的最大百分比。gp_resource_group_memory_limit 的缺省值为 0.7(70%)。

GP 数据库在 Host 主机上的可用内存在 Primary 之间平均分配，当启用资源组来管理资源时，分配给每个 Instance 的内存是，Host 的总可用内存 × gp_resource_group_memory_limit ÷ Host 主机上的 Primary 总数，可通过下面的公式计算得出：

```
rg_perseg_mem = ((RAM × (vm.overcommit_ratio ÷ 100) + SWAP) × gp_resource_group_memory_limit) ÷ num_active_primary_segments
```

其中[RAM × (vm.overcommit_ratio ÷ 100) + SWAP]为 Linux 可用内存的计算方式。

每个资源组都可以配置一定百分比的专享内存，在创建资源组时通过 MEMORY_LIMIT 属性来配置，MEMORY_LIMIT 属性的最小取值为 0，最大取值为 100。当设置 MEMORY_LIMIT 为 0 时，GP 将不会为该资源组配置专享内存，而是使用全局共享内存来满足该资源组中的内存需求。可以参见”全局共享内存”章节。

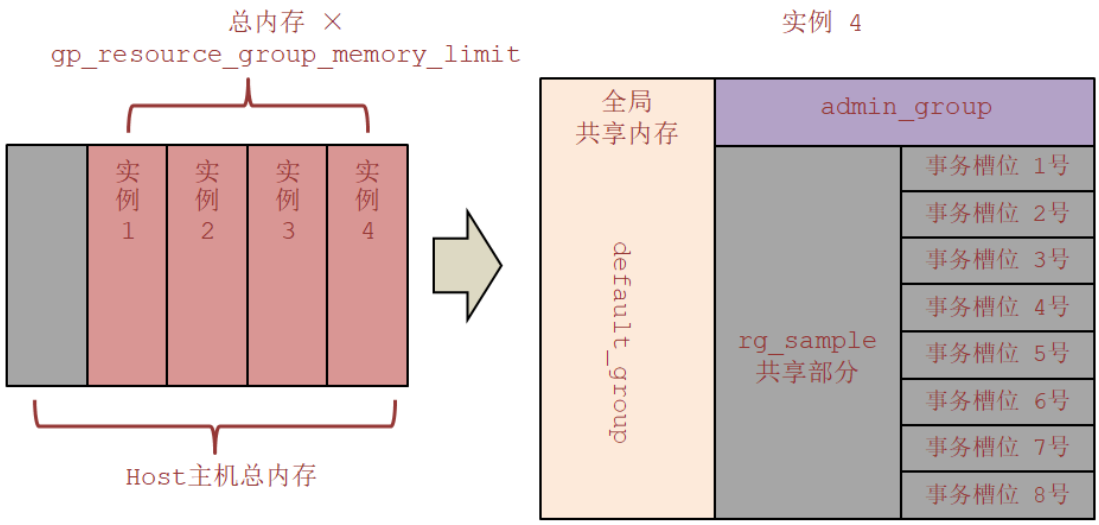
注意： GP 数据库中所有资源组的 MEMORY_LIMIT 总和，不能超过 100.

基于 ROLE 的内存配额的更多配置

对于基于 ROLE 的内存配额来说，为资源组配置的专享内存(MEMORY_LIMIT 不为 0)将会继续分为定额部分和共享部分。创建资源组时，MEMORY_SHARED_QUOTA 属性用于指定，为资源组配额的专享内存中，多少百分比作为共享内存，这部分的共享内存，该资源组中的并发事务都可以使用，并且采用先到先得的原则来分配，正在运行的事务可能完全不用，也可能用一些，也可能使用到全部。

MEMORY_SHARED_QUOTA 属性的取值范围是 0 到 100，缺省值是 80。

正如之前说的，CONCURRENCY 属性控制着资源组中最大的并发事务数量，如果为资源组配置了专享内存(MEMORY_LIMIT 不为 0)，定额部分的内存将按照最大并发事务数量来平均分配，每个槽位将获得完全相同的份额，也就是说，即便没有并发事务在执行，这部分定额内存也是保留的。




```
CREATE RESOURCE GROUP rg_sample WITH(CONCURRENCY=8,
    CPU_RATE_LIMIT=30, MEMORY_LIMIT=60, MEMORY_SHARED_QUOTA=50);
```

此图展示的是内存配额的情况，该图与官方文档中有不同，因为 default_group 资源组的 memory_limit 是 0，应该是只能使用全局共享内存的资源。

当一个查询的内存消耗超过了资源组中定额部分的限制，将可以从该资源组的共享部分获取，因此，对于一个事务槽位来说，可以获得的最大内存使用量，是定额部分+共享部分。不过，当有全局共享内存时，资源组的共享部分用完之后，还可以从全局共享内存获得内存资源。

全局共享内存

为所有资源组(包括缺省的 admin_group 和 default_group)配置的MEMORY_LIMIT属性的总和，是数据库为这些资源组预留的专享配额。如果总和小于100，剩余部分就会作为全局共享内存来管理。全局共享内存，只有使用了vmtracker内存管理模式(基于ROLE的资源组)的资源组可以使用。

如果可以的话，数据库会在为查询分配了资源组的定额部分的内存和共享部分(如果有的话)的内存之后，为事务分配全局共享内存。这部分内存的分配也是按照先到先得的原则。

注意：数据库会统计(但不主动监测)资源组中事务的内存使用情况，如果资源组中的事务使用内存的情况达到了以下所有条件，该事务将会失败：

- 资源组中共享部分的内存耗尽。
- 全局共享内存耗尽。
- 事务请求更多的内存。

当预留了一些全局共享内存(例如10%到20%)时，数据库通过资源组来管理内存使用将会更有效。全局共享内存会有助于降低大量内存消耗型查询出现异常的概率。

算子内存配额

大多数的算子(我们将执行计划中的Hash，Sort，Join，Agg等运算操作统一称为算子)都不是内存密集型的算子，也就是说，在执行过程中，数据库分配的内存足够其使用。有些内存密集型的算子，例如，Join和Sort，可能内存中放不下大量的数据，数据就需要溢出到磁盘上。

参数gp_resgroup_memory_policy用于控制一个查询中所有算子的内存分配策略，对于资源组，目前支持eager_free(缺省值)和auto两种内存策略。当设置为auto策略时，数据库的资源组内存管理，将会为非内存密集型算子分配固定尺寸的内存，剩余的内存分给内存密集型算子。当设置为eager_free策略时，内存资源的分配会更优化，数据库会将已经完成的算子释放的内存重新分配给后续的算子。

MEMORY_SPILL_RATIO属性用于设置资源组中内存密集型事务的内存使用上限。当达到该上限时，事务将会产生溢出文件到磁盘上，编者也没有找到关于这个比例的准确解释，编者认为应该是资源组可以利用的内存总量的百分比，设置一个较低的值将可以允许更多的并发事务，设置过高的值会争抢其他事务的内存资源。数据库会根据MEMORY_SPILL_RATIO的设置来确定初始分配给一个事务的内存尺寸。

MEMORY_SPILL_RATIO属性的取值范围是0到100，缺省值是0。当

MEMORY_SPILL_RATIO 属性的值为 0 时，数据库将根据 statement_mem 参数的值来确定初始分配给一个事务的内存尺寸。

注意：当设置 MEMORY_LIMIT 属性为 0 时，MEMORY_SPILL_RATIO 属性也必须设置为 0。

在 SESSION 中，还可以通过设置 memory_spill_ratio 参数的值来设置当前事务的 MEMORY_SPILL_RATIO 属性。

官方文档上说，对于低内存消耗型的查询来说，设置如下的参数可以提升查询的性能，编者觉得，有待验证，至少，这种操作可能没有显著的性能提升。

```
=# SET memory_spill_ratio=0;
=# SET statement_mem='10 MB';
```

使用专享内存还是使用全局共享内存

如果没有为资源组配额专享内存 (MEMORY_LIMIT 和 MEMORY_SPILL_RATIO 属性被设置为 0)，将会产生如下影响：

- 全局共享内存的尺寸会增加。
- 资源组的功能将会与资源队列相似，通过statement_mem参数的值来确定初始分配给一个事务的内存尺寸。
- 该资源组中的事务将与其他资源组中的事务竞争全局共享内存，并且采用先到先得的原则来分配。
- 数据库无法保证一定能为该资源组中的事务分配到内存。
- 当同时有多个事务需要从全局共享内存获取配额时，该资源组中的查询出现内存不足的风险增加。

要降低重要资源组中查询出现内存不足的风险，可以考虑为该资源组配置一些专享内存，这样做会降低全局共享内存的尺寸，但为了降低该资源组出现内存不足的风险，这是一种权衡的选择。

其他内存事项

基于ROLE的资源组会管理所有通过palloc()函数分配的内存，通过linux的malloc()函数分配的内存将不会被资源组管理。因此，为了确保基于ROLE的资源组能够准确的管理内存的使用情况，应该避免在自定义函数中使用 malloc()函数来分配大量内存。

配置与使用资源组

注意：在RedHat6或者CentOS6中使用资源组是有问题的，这是因为早期的cgroup有缺陷，最好将Kernel升级到2.6.32-696或者更高的版本以修复已知的问题，从而可以更好的使用资源组功能。这些问题在Redhat7或者CentOS7中已经修复。

环境要求

GP数据库，使用Linux的cgroup来管理资源组的CPU资源，同时，使用cgroup来管理基于外部组件的资源组的内存资源。通过 cgroup，GP 数据库将数据库进程的CPU 资源和外部组件的内存资源与主机上其他进程进行隔离。这就可以将每个资源组的CPU资源和外部组件的内存资源进行限制。

关于cgroup的更多信息，可以参考对应Linux发行版的Control Groups文档。

在GP数据库的每个Host主机上完成如下cgroup的配置(使用编者的一键部署命令时，除了安装必要的操作系统组件，其他配置工作都会自动完成)：

1、如果是在 SuSE11+操作系统运行 GP 数据库集群，需要将所有 Host 主机启用swapaccount内核参数并重启机器，之后才能继续进行cgroup的配置。

2、使用ROOT用户或者通过sudo权限创建GP数据库的cgroup配置文件：

```
$ sudo vi /etc/cgconfig.d/gpdb.conf
```

3、在/etc/cgconfig.d/gpdb.conf配置文件中加入如下配置信息：

```
group gpdb {
    perm {
        task {
            uid = gpadmin;
            gid = gpadmin;
```

```
        }
        admin {
            uid = gpadmin;
            gid = gpadmin;
        }
    }
    cpu {
    }
    cpuacct {
    }
    memory {
    }
    cpuset {
    }
}
```

这些配置，用于设置由 gpadmin 用户管理 CPU 和 CPU Core 以及内存的控制，GP 数据库只针对基于外部组件的资源组使用 cgroup 来管理内存资源。

4、 如果操作系统中没有安装并运行 cgroup 的相关组件，需要在 GP 集群的所有 Host 主机上进行安装和启用。根据不同的操作系统，这些命令会略有不同，使用 ROOT 用户或者通过 sudo 权限来执行这些命令：

- Redhat/CentOS 7.x系统

```
$ sudo yum install libcgroup-tools
$ sudo cgconfigparser -l /etc/cgconfig.d/gpdb.conf
$ sudo systemctl enable cgconfig.service
```

- Redhat/CentOS 6.x系统

```
$ sudo yum install libcgroup
$ sudo service cgconfig start
$ sudo chkconfig cgconfig on
```

- SuSE 11+系统

```
$ sudo zypper install libcgroup-tools
$ sudo cgconfigparser -l /etc/cgconfig.d/gpdb.conf
```

5、 运行以下命令验证是否正确设置了 GP 数据库 cgroup 配置：

```
$ CGROUP_MOUNT_POINT=`df -h|grep cgroup|awk '{print $NF}'`
$ if [ "$CGROUP_MOUNT_POINT" != "" ];then
```

```
> ls -l $CGROUP_MOUNT_POINT/{cpu,cpuacct,cpuset,memory}/|grep gpdb
> fi
```

如果输出 4 行 owner 为 gpadmin:gpadmin 的目录，则表明设置成功了。

启用资源组

在安装 GP 时缺省使用资源队列来管理资源。要使用资源组取代资源队列来管理资源，必须修改参数 gp_resource_manager 为 group。

1、修改参数 gp_resource_manager 的值为 group：

```
$ gpconfig -s gp_resource_manager
$ gpconfig -c gp_resource_manager -v "group"
```

2、重启 GP 数据库：

```
$ psql postgres -c "CHECKPOINT"
$ gpstop -af
$ gpstart -a
```

启用资源组之后，任何 ROLE 提交的事务都会经过该 ROLE 所属的资源组来执行，并受到该资源组的限制(并发事务数量、CPU 百分比和内存百分比)。类似的，外部组件的 CPU 和内存资源也受到分配到该外部组件的资源组的限制。

GP 数据库缺省创建了两个资源组，分别为 admin_group 和 default_group。一旦启用了资源组，未明确分配资源组的 ROLE，会根据 ROLE 的类型分配一个缺省资源组，SUPERUSER 会被分配 admin_group，普通 ROLE 会被分配 default_group。

两个缺省资源组的缺省属性如下：

属性(限制类型)	admin_group	default_group
CONCURRENCY	10	20
CPU_RATE_LIMIT	10	30
CPUSET	-1	-1
MEMORY_LIMIT	10	0
MEMORY_SHARED_QUOTA	80	80
MEMORY_SPILL_RATIO	0	0
MEMORY_AUDITOR	vmtracker	vmtracker

需要注意的是，缺省的两个资源组的 CPU_RATE_LIMIT 属性和 MEMORY_LIMIT 属性也计入 Host 主机的总百分比，当创建创建新的资源组时，可能需要调整这两个缺省资源组的属性配置。

创建资源组

要创建一个资源组，需要提供资源组的名称以及 CPU 配额模式，还可以配置一些可选项：最大并发事务数量，内存配额，内存的共享部分占比，内存溢出到文件的阈值。使用 CREATE RESOURCE GROUP 命令来创建一个新的资源组。

创建资源组时必须指定 CPU_RATE_LIMIT 或者 CPUSET 的值，这用于限制该资源组可用的 CPU 的百分比。可以指定 MEMORY_LIMIT 为资源组配置专享内存配额的百分比，如果 MEMORY_LIMIT 设置为 0，GP 将不会为该资源组配置专享内存，而是使用全局共享内存来满足该资源组中的内存需求。

例如，创建一个名称为 rgroup1 的资源组，CPU 配额为 20，内存配额为 25，内存溢出到文件的阈值为 20：

```
=# CREATE RESOURCE GROUP rgroup1 WITH
    (CPU_RATE_LIMIT=20, MEMORY_LIMIT=25,MEMORY_SPILL_RATIO=20);
```

分配了 rgroup1 资源组的所有 ROLE 将共享 20%的 CPU 配额，类似的，这些 ROLE 也将共享 20%的内存配额。使用缺省的内存管理模式 vmtracker 以及缺省的 CONCURRENCY 为 20。

如果要创建基于外部组件的资源组，除了必须指定 CPU_RATE_LIMIT 或者 CPUSET 的值，还必须设置 MEMORY_LIMIT 的配额，同时还必须设置 MEMORY_AUDITOR 为 cgroup，并且明确的设置 CONCURRENCY 为 0。例如要创建一个名称为 rgroup_extcomp 的资源组，CPU 配额为 1 Core，内存配额为 15：

```
=# CREATE RESOURCE GROUP rgroup_extcomp WITH
    (MEMORY_AUDITOR=cgroup, CONCURRENCY=0, CPUSET='1', MEMORY_LIMIT=15);
```

使用 ALTER RESOURCE GROUP 来调整资源组的配额。要修改资源组的配额，需要为资源组的属性指定一个新的值。例如：

```
=# ALTER RESOURCE GROUP rg_role_light SET CONCURRENCY 7;
=# ALTER RESOURCE GROUP exec SET MEMORY_SPILL_RATIO 25;
=# ALTER RESOURCE GROUP rgroup1 SET CPUSET '2,4';
```

注意：不能设置admin_group资源组的CONCURRENCY属性为0。

使用DROP RESOURCE GROUP命令来删除资源组，要删除一个资源组，该资源组不能被分配给任何ROLE，同时，该资源组上不能有任何活动的事务和等待的事务。如果删除一个基于外部组件的资源组，该资源组上正在运行的实例将会被杀死。例如：

```
=# DROP RESOURCE GROUP exec;
```

配置基于内存限制的查询终止

当有全局共享内存时，runaway_detector_activation_percent参数设置了全局共享内存的阈值，当全局共享内存达到参数指定的利用率时，将触发资源组中的查询被终止，这只针对内存管理模式为vmtracker 的资源组。编者的理解是，当阈值达到时，那些继续向全局共享内存申请配额的事务将会被终止，其他不需要向全局共享内存申请配额的事务将不受影响。

什么时候有全局共享内存，当系统中所有资源组MEMORY_LIMIT属性的值的和小于100时。例如，系统中配置了3个资源组，MEMORY_LIMIT属性的值分别为10、20、30，那么全局共享内存为100 - (10 + 20 + 30) = 40。

分配资源组给ROLE

在创建了一个MEMORY_AUDITOR属性为缺省值vmtracker的资源组后，该资源组就可以分配给ROLE了(可以是一个或者多个)。通过CREATE ROLE或ALTER ROLE命令的RESOURCE GROUP子句来分配资源组给ROLE。如果在CREATE ROLE时没有设置资源组，会根据ROLE的类型分配一个缺省资源组，SUPERUSER会被分配admin_group，普通ROLE会被分配default_group。

使用ALTER ROLE或CREATE ROLE命令来分配一个资源组给ROLE。例如：

```
=# ALTER ROLE bill RESOURCE GROUP rg_light;
=# CREATE ROLE mary RESOURCE GROUP exec;
```

可以将一个资源组分配给一个或者多个ROLE。如果ROLE有层级关系，将一个资源组分配给一个上层的ROLE，这个设置并不会传递到该组的其他ROLE，也就是说，

ROLE 的资源组属性不可继承。

注意：不能将创建的基于外部组件的资源组分配给一个 ROLE。

如果想要将一个资源组从一个 ROLE 移除，并按照缺省的规则分配一个缺省资源组，可以修改 ROLE 并分配一个名为 NONE 的资源组。例如：

```
=# ALTER ROLE mary RESOURCE GROUP NONE;
```

监控资源组状态

本章节介绍查看资源组状态信息的方法。

查看资源组配额

通过 gp_toolkit.gp_resgroup_config 视图可以查看资源组的配额设置。不过，编者建议可以重建该视图，原有的视图定义实在不够优雅：

```
=# CREATE OR REPLACE VIEW gp_toolkit.gp_resgroup_config AS
SELECT groupid,groupname,
    vs[1] AS concurrency,
    vs[2] AS cpu_rate_limit,
    vs[3] AS memory_limit,
    vs[4] AS memory_shared_quota,
    vs[5] AS memory_spill_ratio,
    DECODE(vs[6],null,'vmtracker','0','vmtracker',
        '1','cgroup','unknown') AS memory_auditor,
    vs[7] AS cpuset
FROM (
    SELECT g.oid AS groupid,
        g.rsgname AS groupname,
        array_agg(c.value order by reslimittype) vs
    FROM pg_resgroup g, pg_resgroupcapability c
    WHERE g.oid = c.resgroupid
    GROUP BY 1,2
) t;
ALTER TABLE gp_toolkit.gp_resgroup_config OWNER TO gpadmin;
```

```
GRANT ALL ON TABLE gp_toolkit.gp_resgroup_config TO gpadmin;
GRANT SELECT ON TABLE gp_toolkit.gp_resgroup_config TO public;
```

查看资源组的配额设置：

```
=# SELECT * FROM gp_toolkit.gp_resgroup_config;
```

	groupid oid	groupname name	concurrency text	cpu_rate_limit text	memory_limit text	memory_shared_quota text	memory_spill_ratio text	memory_auditor text	cpuset text
1	6437	default gr	20	30	0	80	0	vmtracker	-1
2	6438	admin grou	10	10	10	80	0	vmtracker	-1

查看资源组的查询状态和 CPU 内存使用量

通过 gp_toolkit.gp_resgroup_status 视图来查看资源组的事务活跃情况，排队情况以及实时的 CPU 和内存的使用量：

```
=# SELECT * FROM gp_toolkit.gp_resgroup_status;
```

不过，编者觉得这个视图没法看，CPU 和内存使用量的字段是个很大的 Json，难以阅读。像下面这样可能会容易阅读一些：

```
=# SELECT rsgname,groupid,num_running,num_queueing,
    num_queued,num_executed,total_queue_duration,
    json_each(cpu_usage)::text cpu_usage,
    json_each(memory_usage)::text memory_usage
   FROM gp_toolkit.gp_resgroup_status
ORDER BY rsgname,cpu_usage;
```

查看资源组在每个 Host 主机的 CPU 和内存使用量

通过 gp_toolkit.gp_resgroup_status_per_host 视图来查看每个资源组在每个 Host 主机上的 CPU 和内存的实时使用量：

```
=# SELECT * FROM gp_toolkit.gp_resgroup_status_per_host;
```

查看资源组在每个 Instance 的 CPU 和内存使用量

通过 gp_toolkit.gp_resgroup_status_per_segment 视图来查看每个资源组在每个 Instance 上的 CPU 和内存的实时使用量：

```
=# SELECT * FROM gp_toolkit.gp_resgroup_status_per_segment;
```

查看资源组分配到 ROLE 的情况

通过关联 pg_roles 和 pg_resgroup 两张系统表来查看资源组分配到 ROLE 的情况：

```
=# SELECT rolname, rsgname FROM pg_roles, pg_resgroup
   WHERE pg_roles.rolresgroup=pg_resgroup.oid;
```

查看资源组中正在执行和排队的查询

通过 pg_stat_activity 系统视图来查看资源组中有哪些查询正在执行，或者正在排队，以及排队的时间等：

```
=# SELECT query, waiting, rsgname, rsgqueueduration FROM pg_stat_activity;
```

pg_stat_activity 视图，可以查看执行语句的状态信息，这与 PostgreSQL 相似。如果一个查询使用了外部组件(如 PL/Container)，该查询将会有两个部分，一部分是查询本身在 GP 数据库中运行，而另一部分 UDF 则在 PL/Container 容器中运行，在 GP 数据库中运行的查询本身由 ROLE 的资源组来管理，在 PL/Container 运行的 UDF 由 PL/Container 的资源组管理，后者在 pg_stat_activity 视图中无法体现，数据库无法获取 PL/Container 外部组件中的运行情况。

终止资源组中正在运行或者排队的事务

有时，可能需要终止资源组中正在执行的事务或者还在排队的事务。例如想把某个还在排队的查询移除，或者想把某个已经执行很久的事务中断，亦或是想把某个占用并发数槽位的IDLE事务清除以让给其他ROLE使用。

缺省情况下，事务可以无限期的在资源组中排队，如果想要为排队设置超时时间，可以设置gp_resource_group_queuing_timeout参数，该参数指定一个毫秒数，当事务排队时间超过这个设置时，将会被数据库中断。

要手动终止一个事务，首先要确定该查询相关的进程号(pid)，得到了该pid之后，通过调用pg_cancel_backend()函数来终止该查询。

例如，通过如下语句查看所有资源组中正在执行和排队的语句，如果查询没有结果，说明资源组中没有正在执行的事务或者排队的事务。

```
=# SELECT rolname, g.rsgname, pid, waiting, state, query, datname
   FROM pg_roles, gp_toolkit.gp_resgroup_status g, pg_stat_activity
   WHERE pg_roles.rolresgroup=g.groupid
   AND pg_stat_activity.usename=pg_roles.rolname;
```

例如，要终止pid为2395的查询：

```
=# SELECT pg_cancel_backend(2395);
```

还可以为pg_cancel_backend()函数提供一个可选的消息参数，用于通知该查询的ROLE，告知为何终止了其执行的事务。例如：

```
=# SELECT pg_cancel_backend(2395,'因系统维护暂停使用');
```

该事务的ROLE会收到如下信息：

```
ERROR:  canceling statement due to user request: "因系统维护暂停使用"
```

注意：尽可能避免使用操作系统的KILL命令来杀死GP数据库的任何进程，当然，有时如果万不得已，在尽量确保安全的情况下，也不是绝对不能使用，但服务支持可能会不支持这种擅自操作导致的严重后果，所以，如有必要，请先获得服务支持的许可。

转移查询的资源组

数据库的SUPERUSER可以执行gp_toolkit.pg_resgroup_move_query()函数来将一个正在执行的事务转移到另一个资源组，这样，在不中断该查询的情况下，可以将其转移到一个资源配额更多的资源组。

注意：只能转移一个正在执行的事务，不能转移因为并发事务限制或者内存限制导致排队和等待的事务。

调用pg_resgroup_move_query()函数需要提供两个参数，PID和新的资源组名称。例如：

```
=# SELECT gp_toolkit.pg_resgroup_move_query(2514,'default_group');
```

在调用pg_resgroup_move_query()函数时，该查询将受到新的资源组的配额限制：

- 如果目标资源组的最大并发事务数量已经超了，该查询会进行排队等待状态。
- 如果目标资源组的内存资源不足，调用该函数时会收到如下报错信息：

```
group <group_name> doesn't have enough memory . . .
```

在这种情况下，可以增加目标资源组的内存配额，或者等有资源空缺时再转移。

转移查询的资源组之后，该查询将与目标资源组中的其他事务竞争资源，因此，无法保证目标资源组中的事务不受影响，目标资源组中的事务可能会受此影响而导致查询失败，保留足够的全局共享内存，可能会有效降低这种风险。

pg_resgroup_move_query()函数只是把指定的事务转移到目标资源组中，其所在Session中之后的事务仍然通过原来的资源组执行。

注意：转移查询的资源组，这个功能是6.8版本才引入的功能。

如果是从6.8之前的版本升级而来，需要手动创建函数来使用该用能：

```
CREATE FUNCTION gp_toolkit.pg_resgroup_check_move_query(IN session_id
int, IN groupid oid, OUT session_mem int, OUT available_mem int)
RETURNS SETOF record
AS 'gp_resource_group', 'pg_resgroup_check_move_query'
VOLATILE LANGUAGE C;
GRANT EXECUTE ON FUNCTION
gp_toolkit.pg_resgroup_check_move_query(int, oid, OUT int, OUT int)
```

```
TO public;
CREATE FUNCTION gp_toolkit.pg_resgroup_move_query(session_id int4,
groupid text)
RETURNS bool
AS 'gp_resource_group', 'pg_resgroup_move_query'
VOLATILE LANGUAGE C;
GRANT EXECUTE ON FUNCTION gp_toolkit.pg_resgroup_move_query(int4,
text) TO public;
```

如果要将版本降到6.7或者更早的6版本，可以手动删除这些函数：

```
DROP FUNCTION gp_toolkit.pg_resgroup_check_move_query(IN int, IN oid,
OUT int, OUT int);
DROP FUNCTION gp_toolkit.pg_resgroup_move_query(int4, text);
```

使用资源队列

在出现资源组之前，GP一直是使用资源队列来管理资源，资源队列最常见的使用场景就是限制并发查询的数量。资源队列是基于查询语句来做并发控制的，而资源组是基于事务来做并发控制。所以在资源队列中可能会出现多语句的事务之间的死锁现象，一边在等资源队列的锁，另一边在等对象的锁，这种情况在资源组中不会出现，因为资源组是以事务为单位进行排队的。因此，资源队列存在死锁风险，虽然概率很低。

资源队列如何工作

在安装GP时缺省使用资源队列来管理资源。所有的ROLE都必须分配到资源队列。如果管理员创建ROLE时没有指定资源队列，该ROLE将会被分配到缺省的资源队列pg_default。

建议管理员为不同类型工作负载创建结构性独立的资源队列。例如，可以为高级用户、WEB用户、报表管理等创建不同的资源队列。可以根据相关工作的负载压力设置合适的资源队列限制。目前资源队列的限制包括：

- 活动语句数量。同时正在执行的最大语句数量。这往往是资源队列的唯一用处。
- 活动语句内存使用量。所有正在执行的语句使用的总内存不能超过该限制。
- 活动语句优先级。该值设定了该资源队列相对于其他资源队列在CPU资源使用上的

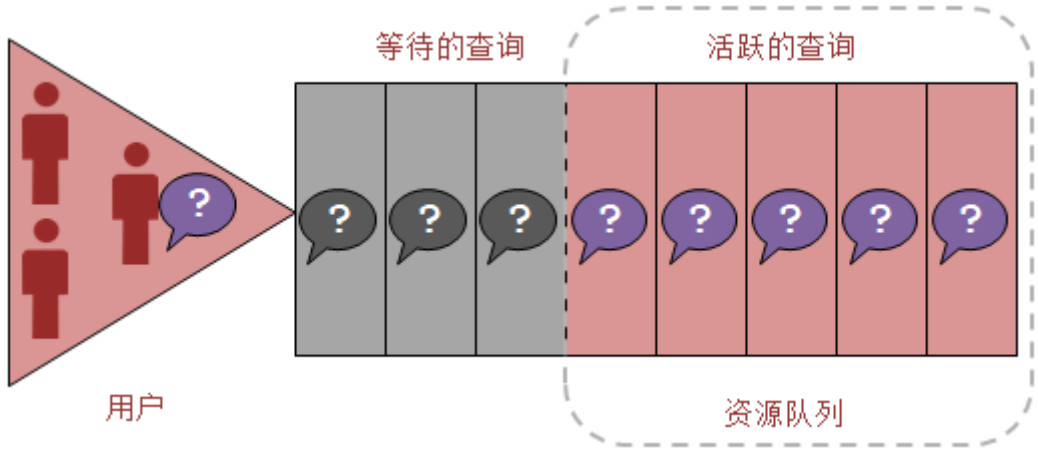
优先级，这里说的优先是相对的。这种优先级，几乎是没用的，MAX和MIN之间可能也测不出差异，无法达到资源组的CPU压制效果，如果能达到，也就不需要资源组了。

- 活动语句的成本限制。该值是由执行计划做的成本评估，该值以涉及的磁盘页(disk page)作为计量单位。

资源队列创建好之后，ROLE(User)可以被分配到合适的资源队列。一个资源队列可以分配多个ROLE，但每个ROLE只能被分配到一个资源队列。

资源队列如何工作

在数据库运行时，用户提交一个查询，该查询会被数据库根据其所在的资源队列的资源进行评估。如果评估认为该查询不会超过资源限制，该查询将被立即执行。如果评估认为该查询超过了资源限制(例如最大活动语句数的查询正在执行)，该查询需要等到有足够的资源时才能被执行。查询按照先进先出的方式排队。在查询优先级启用的情况下，系统会定期的重新分配计算资源。编者想说，排队也是要消耗内存的，资源队列的排队相当于是在等待锁，而且查询的执行计划已经生成并开始执行，只是未获取必要的锁，实际上内存已经分配，如果有大量消耗大量内存的语句在排队，可能会出现内存不足的报错。



超级用户是不受资源队列限制的。超级用户的查询语句总是被立即执行，不管其所在的资源队列如何限制。

内存限制如何工作

在资源队列上设置的内存限制使得每个Instance上该资源队列能够使用的内存总和不能超过设定的最大值。每个查询语句分配的内存大小是资源队列的内存限制除以最大活动语句数量(建议与活动语句数限制结合使用，而不是与cost限制结合使用，如果是与cost限制结合使用，将按照cost的权重进行分配)。例如，资源队列的内存限制为2000MB，活动语句数限制为10，那么每条执行语句可以得到200MB的内存。缺省的内存分配可以针对每条语句通过设置statement_mem参数来覆盖(最大可以达到资

源队列限制的值）。一旦一条语句开始执行，其分配的内存一直到执行结束才会释放（即便其实际使用的内存小于分配的内存）。

执行优先级如何工作

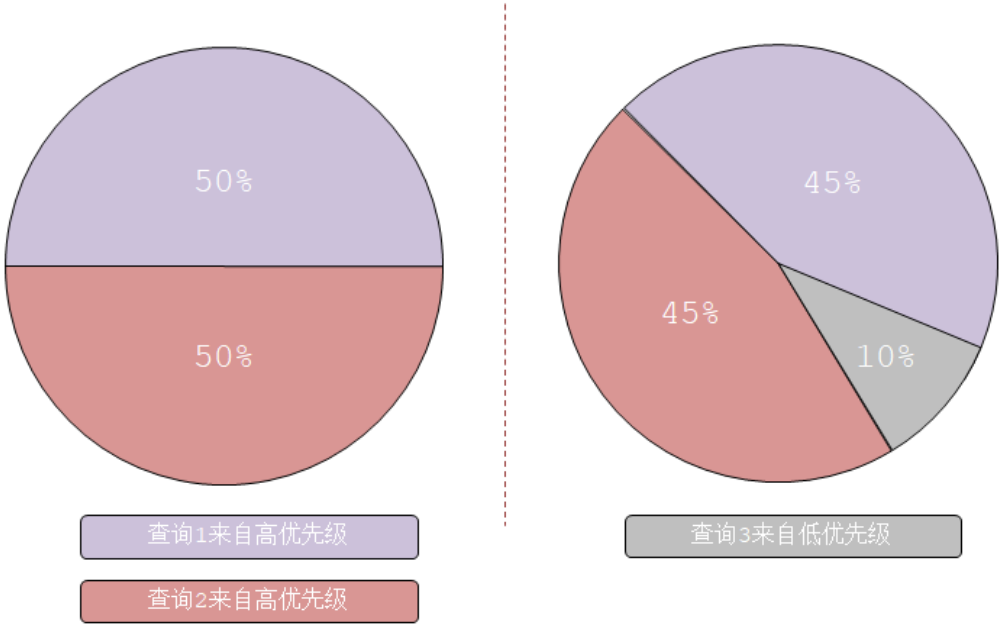
资源限制是针对活动语句来说的，内存和成本的限制属于是否许可类型，其决定查询语句是允许进入查询状态还是保持排队状态。在语句处于活动状态时，其需要分享CPU资源，这部分的资源由资源队列的优先级控制。当一个更高优先级的语句进入运行状态时，其要求获得更多的CPU资源，相应的需要减少其他运行中语句的CPU资源。

语句的规模和复杂程度不会影响到CPU资源的分配。例如一个简单低成本的查询与一个庞大复杂的查询同时执行却有着相同的优先级，他们在同一时间段内将分配到相同的CPU资源。当一个新的语句开始执行，CPU资源分配的比重需要被重新计算，不过，相同优先级的语句之间获得的CPU资源仍然是相同的。

例如，管理员想要创建3个资源队列：adhoc用于做持续查询的业务分析，reporting用于做定期的报表工作，executive用于高级用户查询。管理员希望确保定期报表工作不受到adhoc分析查询不可预测的资源消耗影响，希望高级用户提交的查询能够获得更多计算资源。因此，资源队列优先级可以设置为这样：

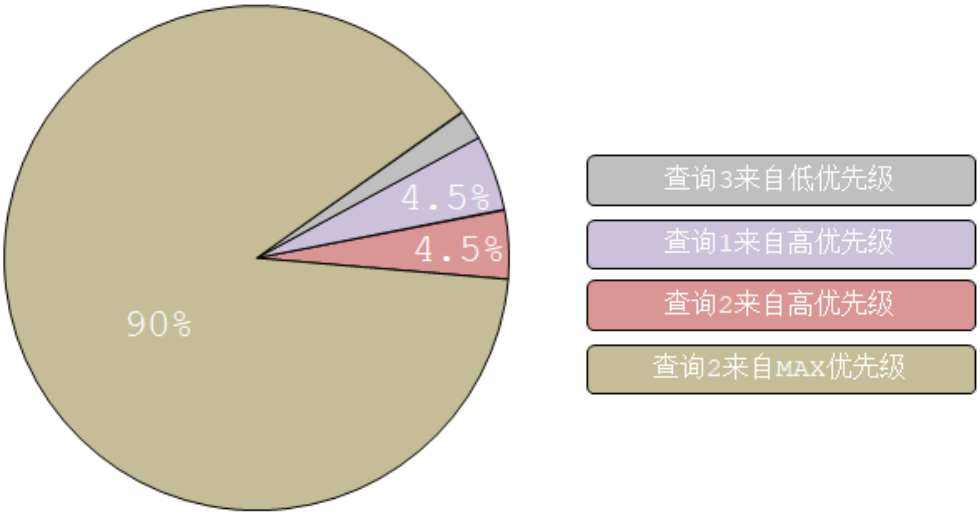
- adhoc -- Low priority
- reporting -- High priority
- executive -- Maximum priority

运行时，CPU资源由正在运行语句的优先级决定如何分配。如果语句1与语句2来自reporting队列且同时运行，他们将获得相同CPU资源。当一个来自adhoc队列语句开始运行，其需要较少的CPU资源。CPU资源的分配将做调整，他们之间的比重受他们的优先级决定：



注意：该图显示的是粗略的百分比。在CPU的使用上，并不总是按照精确计算来分配资源。

当一个来自executive的语句开始执行，CPU资源将根据他们的优先级重新调整。该语句相对于adhoc和reporting来说可能是很简单的，但在其执行完之前，其仍获取最大份额的CPU资源。



资源队列评估的语句类型

并不是所有的SQL语句都被资源队列评估和限制。缺省状态下只有SELECT、SELECT INTO、CREATE TABLE AS SELECT和DECLARE CURSOR语句被评估和限制。若将Server端参数resource_select_only设置为off，INSERT、UPDATE和DELETE语句也将被评估和限制。

使用资源队列做资源管理的步骤

在GP中使用资源队列做资源管理涉及到如下几个任务：

1. 创建资源队列并设置合适的限制。
2. 为User Role指定资源队列。
3. 使用资源队列相关的系统视图监控和管理资源队列。

配置资源队列管理资源

在安装GP时资源管理缺省使用资源队列。缺省的资源队列是pg_default，活动语句数量限制为20，成本(Cost)无限制，内存(Memory)无限制，中(Medium)优先级。建议创建不同类型的资源队列。

配置资源队列

1. 以下为一般的资源队列配置参数

- max_resource_queues -- 设置最多可以有多少个资源队列
- max_resource_portals_per_transaction -- 设置每个事务最多可以打开几个游标(Cursor)。值得注意的是每个游标需要占用资源队列的一个活动查询。
- resource_select_only -- 若设置为on，SELECT、SELECT INTO、CREATE TABLE AS SELECT和DECLARE CURSOR语句被评估限制。若设置为off，INSERT、UPDATE和DELETE语句也将被评估和限制。
- resource_cleanup_gangs_on_wait -- 在开始一个新的查询之前先清空所在资源队列中其他空闲的工作进程。
- stats_queue_level -- 打开资源队列使用信息收集，这样就可以通过查询系统视图pg_stat_resqueues查看资源队列的使用情况。

2. 以下参数与内存使用有关：

- gp_resqueue_memory_policy -- 设置GP内存管理模式。设置为none的情况下内存管理与4.1版本之前相同。设置为auto的情况下，内存受statement_mem和资源队列内存限制的控制。缺省为eager_free，这种模式下内存的管理和分配会更高效，并不是说这种模式下内存就不受statement_mem和资源队列的限制，执行计划会分为很多个算子，当设置为eager_free时，在每个算子执行结束之后，会尽快回收那些分配给已经结束的算子的内存并分配给之后的算子。
- statement_mem与max_statement_mem -- 用于为每个活动语句分配内存(可以复写资源队列的缺省值)。max_statement_mem由SUPERUSER设置，应考虑避免普通用户的超负荷使用。在任何时候statement_mem都必须小于max_statement_mem。
- gp_vmem_protect_limit -- 限制每个Instance上所有语句可以使用的内存总量的上限值。导致内存使用超过该上限的语句会被取消(Cancel)从而导致得不到执行。该参数要根据具体硬件情况进行合理的评估，从OS层面来说，物理内存的容量用完之后，根据OS的配置，可能会使用SWAP，对于普通磁盘来说，不到万不得已，强烈建议不要使用SWAP。
- gp_vmem_idle_resource_timeout 与 gp_vmem_protect_segworker_cache_limit -- 用于释放Instance上空闲DB进程的内存。为了提高并发量，管理员可以考虑调整这些配置。一般不需要修改这些参数。

3. 以下参数与查询优先级有关。注意，这些参数都是本地化(LOCAL)的参数，必须修改所有Instance的postgresql.conf文件：

- gp_resqueue_priority -- 缺省状态下查询优先级特性开启。
- gp_resqueue_priority_sweeper_interval -- 设置CPU为所有活动语句重新计算CPU资源分配的时间间隔。缺省值通常已经可以满足要求。
- gp_resqueue_priority_cpucores_per_segment -- 设置每个Instance使用的CPU core数。缺省情况下Instance为4而Master为24。该参数是LOCAL参数。该参数对于Master也是有影响的，需要配置一个较高的值。例如，在一个集群中，每个Host主机有8个CPU Core并且每个Instance Host有4个Instance，那可以按照如下配置：

Master和Standby

```
Master:
gp_resqueue_priority_cpucores_per_segment = 8
Instance:
gp_resqueue_priority_cpucores_per_segment = 2
```

提示：官方文档中会提到：如果每个Instance Host主机上配置的Instance数量低于CPU核数，确保将该参数调整到一个合适的值，过低的值可能会导致CPU资源利用不足。编者认为，实际上，往往可能不需要过于关注这个问题。

4. 要查看和修改这些参数，尽可能使用gpconfig命令来统一修改，很少有人直接去改postgresql.conf文件，不过如果由于参数修改不当导致GP系统无法启动，可能需要手动修改或者进入Master Only模式进行gpconfig配置。

5. 例如，要查看一个参数的配置情况：

```
$ gpconfig -s gp_vmem_protect_limit
```

在 SQL 中使用 show 命令也可以查看一个参数的值，但其不能体现 Master 与 Instance 之间的差异，仅显示 Master 的值。

6. 例如，要修改一个参数的值，且Master的值与Instance不同：

```
$ gpconfig -c gp_resqueue_priority_cpucores_per_segment -v 2 -m 8
```

7. 重启GP以确保修改的参数生效(本节介绍的参数都需要重启生效)：

```
$ gpstop -r
```

创建资源队列

创建资源队列涉及到Name、cost、最大活动语句数量、执行优先级等。通过CREATE RESOURCE QUEUE命令来创建新的资源队列。

创建含活动语句数量限制的资源队列

资源队列通过设置ACTIVE_STATEMENTS控制活动语句的数量。例如，创建一个名称为adhoc，最大活动语句数量为3的资源队列：

```
=# CREATE RESOURCE QUEUE adhoc WITH (ACTIVE_STATEMENTS=3);
```

这意味着，分配到adhoc资源队列的所有ROLE，在同一时刻最多只能有3个语句处于执行状态。如果该队列当前已经有3个语句正在执行，在该队列的ROLE提交第4个语句时，其将处于等待状态，直到前面3个语句有一个执行完。

创建含内存限制的资源队列

资源队列通过设置MEMORY_LIMIT控制该队列所有语句可以使用的内存总量。每个主机上所有Instance可以获得的物理内存总数不得超过该主机的物理内存总数。建议将MEMORY_LIMIT控制在该Instance可以得的的物理内存总数的90%以下。例如，一个Host主机有512GB的物理内存，有6个Instance，那么每个Instance可以获得的物理内存为85GB。这样就可以简单的得到MEMORY_LIMIT=0.9*85GB=76GB。如果存在多个资源队列，他们的MEMORY_LIMIT总和应被控制在不超过76GB。

当与ACTIVE_STATEMENTS结合使用时，缺省每个语句获得的内存为：MEMORY_LIMIT / ACTIVE_STATEMENTS。当与MAX_COST结合使用时，缺省的内存分配为：MEMORY_LIMIT * (query_cost / MAX_COST)。推荐与ACTIVE_STATEMENTS结合使用而不是与MAX_COST结合使用，因为cost的评估很多时候会严重失准，这样将会严重影响内存分配的合理性。

例如，创建一个活动语句数量为10，内存限制为2000MB的资源队列(每个语句在执行时在每个Instance上将获得200MB的内存)：

```
=# CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=10,
```

```
MEMORY_LIMIT='2000MB');
```

缺省的内存分配可以在每个语句通过设置statement_mem参数复写，但不可超过MEMORY_LIMIT和max_statement_mem设定的值。例如，分配更多的内存：

```
=> SET statement_mem='2GB';
=> SELECT * FROM my_big_table WHERE column='value' ORDER BY id;
=> RESET statement_mem;
```

通常来说，MEMORY_LIMIT的设置建议所有的资源队列的总和不要超过Instance可以获得的物理内存总量。如果不同类型语句之间交错执行，可能真实的使用量会远小于资源队列限制的总量，所以，并不是绝对的说MEMORY_LIMIT的总和不能炒作Instance可以获得的物理内存总量，这取决于如何安排和优化，但仍需注意的是，如果某个Instance超出了内存限制超，相关语句会被取消而导致失败。

创建包含成本限制的资源队列

资源队列通过设置MAX_COST限制被执行的语句可消耗的最大成本(Cost)。Cost以一个浮点数(如100.0或使用科学计数法如1e+2)来指定。

Cost是查询优化器(如使用EXPLAIN查看)评估出来的总预估成本。因此管理员在设置时需要对该系统执行的查询很熟悉才可以得到一个恰当的Cost阈值。Cost意味着对磁盘的操作数量。1.0等于获取一个磁盘页(disk page)。

例如，创建一个Cost阈值为100000.0 (1e+5)，名称为webuser的资源队列：

```
=# CREATE RESOURCE QUEUE webuser WITH (MAX_COST=100000.0);
```

或者

```
=# CREATE RESOURCE QUEUE webuser WITH (MAX_COST=1e+5);
```

这意味着，分配到webuser资源队列的所有ROLE执行的全部语句Cost总和不能超过100000.0的限制。例如，有20个Cost为5000.0的语句正在执行，第21个Cost为1000.0的语句提交后只能等到空闲的Cost足够时才能得到执行。

允许在系统空闲时执行语句

若一个资源队列配置了Cost阈值，管理员可以设置允许COST_OVERCOMMIT(这是缺省设置)。在系统没有其他语句执行时，超过资源队列Cost阈值的语句可以被执行。

而当有其他语句在执行时，Cost阈值仍被强制执行。如果COST_OVERCOMMIT被设置为false，超过Cost阈值的语句将永远被拒绝。这个特性听起来很不错，可是仔细想一想，没有其他语句在执行的时间肯定是极少的。

允许小查询绕过队列限制

可能存在一些工作负载很小的查询，管理员希望其不占用资源队列的活动语句数量而直接被允许执行。例如，一些检索系统表的语句不涉及大的资源消耗甚至不需要与Instance进行交互。管理员可以设置MIN_COST指明低于指定的开销被认为是小查询。那些低于MIN_COST的语句将立即被执行。MIN_COST不仅可以同MAX_COST一起使用，还可以和活动语句数量一起使用。例如：

```
=# CREATE RESOURCE QUEUE adhoc WITH (ACTIVE_STATEMENTS=10, MIN_COST=100.0);
```

设置优先级级别

为了控制CPU资源的使用，管理员可以设置合适的优先级。在并发争抢CPU资源时，高优先级资源队列中的语句将可以获得比低优先级资源队列中的语句更多的CPU资源。

优先级可以在CREATE RESOURCE QUEUE和ALTER RESOURCE QUEUE的时候通过WITH来设置。例如，为adhoc和reporting队列指定优先级，管理员可以使用下面的命令：

```
=# ALTER RESOURCE QUEUE adhoc WITH (PRIORITY=LOW);
=# ALTER RESOURCE QUEUE reporting WITH (PRIORITY=HIGH);
```

创建最高优先级的队列executive，管理员可以使用下面的命令：

```
=# CREATE RESOURCE QUEUE executive WITH (ACTIVE_STATEMENTS=3, PRIORITY=MAX);
```

在优查询优先级特性开启时，资源队列的优先级缺省为MEDIUM。如果关闭，则不会对优先级的设置进行评估。

重要提示：要使得资源队列的优先级设置在执行语句中强制生效，必须确保优先级特性的相关参数已经设置好。

分配ROLE(User)到资源队列

一旦资源队列被创建好了，就需要把ROLE(User)分配到合适的资源队列。如果ROLE没有被明确的分配到一个资源队列，其将被分配到缺省的资源队列pg_default。缺省的资源队列有20个活动语句数量限制和MEDIUM的优先级。

使用ALTER ROLE或者CREATE ROLE命令来分配ROLE到资源队列。例如：

```
=# ALTER ROLE name RESOURCE QUEUE queue_name;
=# CREATE ROLE name WITH LOGIN RESOURCE QUEUE queue_name;
```

每个ROLE同一时间只能被分配到一个资源队列，可以使用ALTER ROLE命令修改ROLE的资源队列。

资源队列的分配必须通过逐个User的方式进行。如果有一个层级较高的ROLE(例如GROUP ROLE)，将该ROLE分配到一个资源队列并不会将其包含的User分配到该资源队列。

SUPERUSER总是不受资源队列的限制。SUPERUSER的查询总是可以立即得到执行，而不管资源队列的限制如何设置。

从资源队列中移除ROLE

所有ROLE都需要分配到资源队列。如果没有被明确分配到指定的资源队列，该ROLE将会进入缺省资源队列pg_default。如果想将ROLE从现有资源队列中移除并放到缺省队列中，可将其资源队列分配到none。例如：

```
=# ALTER ROLE role_name RESOURCE QUEUE none;
```

修改资源队列

在创建资源队列后，可以使用ALTER RESOURCE QUEUE命令来改变或者重置队列的限制。还可以使用DROP RESOURCE QUEUE命令删除资源队列。

变更资源队列

使用ALTER RESOURCE QUEUE命令来改变资源队列的限制。一个资源队列必须包含ACTIVE_STATEMENTS或者MAX_COST(或者都包含)。变更资源队列，设置新的参数值。例如：

```
=# ALTER RESOURCE QUEUE adhoc WITH (ACTIVE_STATEMENTS=5);
=# ALTER RESOURCE QUEUE exec WITH (MAX_COST=100000.0);
```

要将活动语句数量或者内存限制重置为无限制，可以使用-1值。要重置Cost阈值为无限制，可以设置为-1值。例如：

```
=# ALTER RESOURCE QUEUE adhoc WITH (MAX_COST=-1.0, MEMORY_LIMIT='2GB');
```

可以使用ALTER RESOURCE QUEUE命令改变查询优先级。例如，设置一个资源队列的优先级为最低级别：

```
=# ALTER RESOURCE QUEUE webuser WITH (PRIORITY=MIN);
```

删除资源队列

使用DROP RESOURCE QUEUE命令删除资源队列。要删除一个资源队列，该资源队列不能与任何ROLE相关联，或者队列中有语句正等待执行。删除一个资源队列：

```
=# DROP RESOURCE QUEUE name;
```

检查资源队列状态

检查资源队列状态涉及以下内容：

- 查看排队语句和资源队列状态

- 查看资源队列统计信息
- 查看分配到资源队列的ROLE
- 查看资源队列中等待的语句
- 清除资源队列中等待的语句
- 查看活动语句的优先级
- 重置活动语句的优先级

查看排队语句和资源队列状态

管理员可以通过查看视图gp_toolkit.gp_resqueue_status来查看资源队列的状态。该视图展示系统中每个资源队列有多少个语句在等待执行，多少语句正在执行。查看资源队列在系统中当前的状态：

```
=# SELECT * FROM gp_toolkit.gp_resqueue_status;
```

查看资源队列统计信息

如果要追踪资源队列的统计信息和性能，需要为资源队列开启统计信息收集的配置。这可以通过配置Master上postgresql.conf文件的这个参数来实现：

```
stats_queue_level = on
```

一旦该配置开启，就可以使用系统视图pg_stat_resqueues来查看资源队列使用的统计信息。注意，开启该配置会带来轻微的资源开销，每个通过资源队列被执行的语句都会被追踪。开启统计信息收集对于初期的资源队列诊断是有帮助的，而后期的运行应该关闭该参数。

查看分配到资源队列的 ROLE

要查看ROLE与资源队列之间的关联关系，可以使用系统视图pg_roles和gp_toolkit.gp_resqueue_status来获得：

```
=# SELECT rolname, rsqname FROM pg_roles, gp_toolkit.gp_resqueue_status
WHERE pg_roles.rolresqueue=gp_toolkit.gp_resqueue_status.queueid;
```

或者可以创建一个视图来简化以后的使用。例如：

```
=# CREATE VIEW role2queue AS
SELECT rolname, rsqname FROM pg_roles, pg_resqueue
WHERE pg_roles.rolresqueue=gp_toolkit.gp_resqueue_status.queueid;
```

这样就可以直接查询视图了：

```
=# SELECT * FROM role2queue;
```

查看资源队列中等待的语句

当语句通过资源队列执行时，其会被记录在pg_locks系统表中。这里可以查看到所有的活动语句和等待语句。要检查处于等待状态的语句，可以使用gp_toolkit.gp_locks_on_resqueue视图。例如：

```
=# SELECT * FROM gp_toolkit.gp_locks_on_resqueue WHERE lorwaiting='true';
```

若该查询没有结果返回，意味着此时没有语句在资源队列中等待执行。

清除资源队列中等待的语句

有时候可能需要清除资源队列中处于等待状态的语句。例如，想要清除在资源队列中处于等待状态还没有得到执行的语句。还有可能想要终止一个已经开始而需要很长时间才能完成的语句，或者该语句处于空闲的事务状态而希望其把资源让给其他需要的ROLE。要达到这些目的，首先需要知道哪些语句需要被清除，确定该进程的ID，然后使用pg_cancel_backend函数来终止该进程。

例如，查看当前处于活动状态或者等待状态的语句：

```
=# SET from_collapse_limit TO 1;
SELECT rolname, rsqname, a.pid, granted, a.query, datname
```

```
FROM pg_roles r, gp_toolkit.gp_resqueue_status s, pg_locks l,
pg_stat_activity a
WHERE r.rolresqueue = l.objid AND r.rolname = a.usename
AND l.objid=s.queueid AND a.pid=l.pid;
```

若没有结果返回，意味着当前没有语句处于资源队列中。例如有两个语句在资源队列中可能是这种样子的：

	rolname name	rsqname name	pid integer	granted boolean	query text	datname name
1	name2	test queue	1855	t	select pg sleep(1000);	postgres
2	name1	test queue	1794	f	select 1;	postgres

根据输出结果确定需要清除语句的进程ID(pid)。通过下面的方式清除语句：

```
=# SELECT pg_cancel_backend(1855);
```

注意：尽量不要使用OS的KILL命令。当然也不是完全不可以用，除非有把握确保不会导致数据库损坏。

查看活动语句的优先级

在gp_toolkit模式中有个视图gp_resq_priority_statement，其包含了所有正在执行的语句的优先级，会话ID等信息。该视图只能通过gp_toolkit模式访问。

重置活动语句的优先级

SUPERUSER可以在语句运行期间通过内置函数gp_adjust_priority(session_id, statement_count, priority)调整优先级。通过该函数SUPERUSER可以提升或者降低任何语句的优先级。例如：

```
=# SELECT gp_adjust_priority(752, 24905, 'HIGH');
```

该函数需要获取语句的SESSION ID和Statement Count两个参数，SUPERUSER可以通过gp_toolkit.gp_resq_priority_statement视图获得，session_id和statement_count两个参数分别对应rqpsession和rqpcommand。该函数只对指

定的语句有效，同一个资源队列随后的语句仍然使用其预先设定的优先级。

第七章：定义数据库对象

本章介绍GP的数据定义语言(DDL)以及如何创建和管理数据库对象。

- 创建与管理数据库
- 创建与管理表空间
- 创建与管理模式
- 创建与管理表
- 分区大表
- 创建与使用序列
- 在GP中使用索引
- 创建与管理视图
- 创建与管理物化视图

创建与管理数据库

一个GP系统可以有多个数据库(Database)。这与一些DBMS不同(例如Oracle)，它们的Instance就是Database。在GP系统中，虽然可以创建多个DB，但是客户端程序一次只能连接一个DB，而且不可以跨越DB执行查询语句。

关于数据库模版

每个新的数据库都是基于一个模版库创建的，这种创建可以理解为模板库的复制，如果基于一个非空的模版库来创建，那么该模板库中的所有对象和数据都会一模一样的复制到新创建的数据库中。缺省的数据库模版为template1，在初始化GP系统初期可以连接到该库，在没有明确指定模版的情况下创建新的数据库将缺省使用该DB作为模版库，除非你希望之后创建的DB包含你所创建的对象，不然的话，不要在该DB中创建任何对象。

除了template1之外，每个新建的GP系统还包含另外两个模版template0和postgres，这两个DB是系统内部使用的，最好不要删除或者修改。template0模版库可以用来创建仅仅包含标准对象的完全干净的数据库。如果想避免从template1中拷贝任何的对象，可以考虑使用该模版。

注意：在创建新的数据库时，模板库上必须没有任何连接，否则创建操作将会报错失败。

创建数据库

通过CREATE DATABASE命令来创建一个新的数据库。例如：

```
=# CREATE DATABASE new_dbname;
```

要创建一个数据库，必须具备创建数据库的权限或者是SUPERUSER身份。若没有正确的权限是无法创建数据库的。需要联系GP管理员授予必要的权限或者帮助创建一个数据库。

还有一个客户端程序createdb可以用来创建数据库。例如，通过命令行终端执行下面的命令将会在指定的Host上创建名为mydatabase的数据库：

```
$ createdb -h masterhost -p 5432 mydatabase
```

克隆一个数据库

缺省状态下，创建数据库是通过克隆数据库模版template1的方式完成的。然而任何的DB都可以作为模版来创建一个新的数据库，因此可以通过指定DB的方式克隆或者拷贝出一个新的数据库，新的DB包含模版库中的所有对象和数据。例如：

```
=# CREATE DATABASE new_dbname TEMPLATE old_dbname;
```

查看数据库列表

在psql客户端程序中，直接使用\l指令查看GP中包含模版库在内的所有DB的列表。使用其他客户端程序时，可以通过查询pg_database系统表来得到。例如：

```
=# SELECT datname from pg_database;
```

修改数据库

使用ALTER DATABASE命令来改变DB的属性，例如Owner、Name以及缺省配置等。必须是该DB的Owner或者SUPERUSER才可以执行这样的操作。下面的例子演示修改缺省的搜索路径：

```
=# ALTER DATABASE mydatabase SET search_path TO myschema, public, pg_catalog;
```

删除数据库

使用DROP DATABASE命令来删除DB。该操作将从系统表中删除该DB的信息记录，并删除该DB包含的全部磁盘数据。必须是该DB的Owner或者SUPERUSER才可以执行删除DB操作，其他用户无法删除该DB，在删除DB时，不能有任何用户连接到该DB，包括当前的操作用户，因此，可以先连接到template1(或者其他DB)，然后再删除需要删除的DB。例如：

```
=# \c template1
=# DROP DATABASE mydatabase;
```

这里同样有一个客户端程序叫做dropdb用以删除DB。例如，通过命令行终端执行下面的命令将会在指定的Host上删除名为mydatabase的数据库：

```
$ dropdb -h masterhost -p 5432 mydatabase
```

警告：删除数据库操作是无法回滚的。慎用该操作！

创建与管理表空间

表空间(tablespace)允许DB管理员使用多个文件系统来存储数据库对象，从而可以决定如何更好的利用他们的物理储存设备。表空间的使用是有好处的，例如在访问频度不同的数据库对象上使用不同性能的磁盘。例如，将经常使用的表放在高性能磁盘的文件系统上(例如SSD固态盘)，而将其他表放在普通硬盘的文件系统上。

一个表空间，在GP集群中，对应的是一组分布式的操作系统目录，在每个Instance上都有一个目录，这些目录的集合，组成了一个表空间，表空间创建成功之后，用户在使用这些表空间时，不需要再关心这些目录的具体位置，只需要在建表时指定表空间名称，或者设置缺省的表空间即可，缺省表空间通过参数default_tablespace来设置。

在GP6版本之前，表空间还不是一个完全独立的概念，其需要依赖文件空间对象，在6版本之前的文件空间，实际上也是一组分布式的操作系统目录，在每个Instance上都有一个目录，这些目录的集合，组成了一个文件空间。这一段看起来和刚刚介绍表空间的几乎一样，是的，没错，在6版本之前，虽然说表空间是依赖文件空间的，但是，如果这样来描述表空间也是正确的，为什么一般不这样描述呢，因为，文件空间已经把这个定义做好了，表空间的目录就是这些文件空间的那些目录的子目录，表空间在文件空间的目录下建立以<tablespace_id>/<database_id>组成的子目录，不同的数据库中的对象，用到对应的表空间时，数据文件则存储到对应的子目录。

在6版本中，表空间的概念，在单个Instance上来看，几乎与PostgreSQL完全一致，当创建一个表空间时，需要为该表空间指定一系列的路径，而这些路径将会通过软连接的方式关联到Instance工作目录的pg_tblspc目录下，以<tablespace_id>为名称，其中的目录结构为：GPDB_大版本号_系统表版本号/<database_id>。

这里，还将继续介绍文件空间的内容，实际上，如果使用6之前的GP版本，很多时候，我们的客户并不需要为创建文件空间头疼，编者在为各位客户安装初始化GP集群时会自动创建一个名为gpfs的文件空间，当需要使用表空间时，只需要使用这个文件空间来创建表空间即可。

创建文件空间

注意：6版本开始，已经不再有文件空间的概念，此处所说的是6版本之前的概念。

要创建文件空间，首先需要在所有相关的GP Host主机上准备好需要的目录。文件系统位置对于Master和所有的Primary和Mirror来说都是必须的。在准备好了文件系统之后，使用gpfilespace命令来定义文件空间。只有SUPERUSER才能进行该操作。

注意：GP并不直接知晓文件系统的界限，只是把文件存向指定的目录位置。因此，在一个逻辑磁盘位置定义多个文件空间是没有意义的，因为基于文件空间创建的表空间就是多个子目录。如果有多个逻辑磁盘，则需要创建创建多个文件空间，虽然一般不会这样规划磁盘，。

使用gpfilespace创建文件空间

1. 使用gpadmin用户登录到GP系统的Master主机。

```
$ su - gpadmin
```

2. 创建一个文件空间的配置文件：

```
$ gpfilespace -o gpfilespace_config
```

3. 将会提示输入一个文件空间的名称，Primary Instance的文件系统位置，Mirror Instance的文件系统位置，Master的文件系统位置。例如，若每个Instance Host配置了2个Primary和2个Mirror，将会提示输入5个文件系统位置(包括Master)。就像这样：

```
Enter a name for this filespace> fastdisk
primary location 1> /gpfs1/seg1
```

```
primary location 2> /gpfs1/seg2
mirror location 1> /gpfs2/mir1
mirror location 2> /gpfs2/mir2
master location> /gpfs1/master
```

4. 该命令将会输出一个配置文件。请再次检查该文件确保其按照期望的那样反映出了想要使用的文件系统位置。

5. 再次执行该命令，基于之前生成的配置文件创建文件空间：

```
$ gpfilespace -c gpfilespace_config
```

转移临时文件或事务文件的位置

注意：此处所说的是6版本之前的概念。

可以选择将临时文件或事务文件转移到一个特殊的文件空间从而改善 DB 的查询性能、备份性能、数据读写的性能。

临时文件和事务文件缺省都是存储在每个 Instance(包括 Master、Standby、Primary 和 Mirror)目录下。只有 SUPERUSER 可以移动该位置。只有 gpfilespace 命令可以修改临时文件和事务文件的位置。

关于临时文件和事务文件

除非另有指明，临时文件和事务文件和用户数据放在一起。缺省的临时文件位置为：

```
<filespace_directory>/<tablespace_oid>/<database_oid>/pgsql_tmp
```

使用gpfilespace --movetempfiles命令来修改临时文件的位置。

关于临时文件和事务文件，需要注意以下几点：

- 虽然可以使用同一个文件空间存储不同类型文件,但只能为临时文件或者事务文件指定一个文件空间。
- 如果文件空间被临时文件使用，该文件空间将不能被删除。
- 文件空间必须提前被创建好才能使用。

使用 gpfilespace 移动临时文件

1. 确保文件空间存在，且与存储其他用户数据的文件空间不同。

2. 将 GP 系统停掉，保持停机状态。

注意：任何活动的连接都会导致 gpfilespace --movetempfiles 操作的失败。

3. 把 GP 启动为限制模式，确保其他用户无法连接，执行下面的命令：

```
$ gpfilespace --movetempfilespace filespace_name
```

注意：临时文件位置在 Instance 中配合共享内存使用，在创建、打开、删除临时文件时用到。如果查询用到了临时文件，表明当前的可用内存已经无法满足该查询对内存的需求。很多时候我们宁愿使用临时文件也不选择使用 SWAP 作为内存扩展方案。除非 SWAP 的性能比临时文件目录所在文件系统的性能还要高。

使用 gpfilespace 移动事务文件

1. 确保文件空间存在，且与存储其他用户数据的文件空间不同。
2. 将 GP 系统停掉，保持停机状态。

注意：任何活动的连接都会导致 gpfilespace --movetransfiles 操作的失败。

3. 把 GP 启动为限制模式，确保其他用户无法连接，执行下面的命令：

```
$ gpfilespace --movetransfilespace filespace_name
```

注意：事务文件位置在 Instance 中配合共享内存使用，在创建、打开、删除事务文件时用到。

如果要移动临时文件或事务文件的目录到缺省路径，使用如下命令：

```
$ gpfilespace --movetempfilespace default
```

和

```
$ gpfilespace --movetransfilespace default
```

在编者看来，这一块的功能几乎不会有人用到，一般来说，Instance 的工作目录所在的磁盘，就是整个主机上性能最好的磁盘了。

注意：处所说的是 6 版本的概念。

可以选择将临时文件或事务文件转移到一个特殊的表空间从而改善 DB 的查询性能、备份性能、数据读写的性能。GP 数据库通过 temp_tablespaces 参数来控制，用于 Hash Agg，Hash Join，排序操作等临时溢出文件的存储位置。这个目录缺省为 <data_dir>/base/pgsql_tmp，这与 6 之前的版本不同，6 之前的版本，这个目录

缺省为：<data_dir>/base/<database_id>/pgsql_tmp。

当使用 CREATE 命令创建临时表和临时表上的索引时，如果没有明确的指定表空间，temp_tablespaces 所指向的表空间将存储这些对象的数据文件。

使用 temp_tablespaces 时需要注意：

1. 只能为临时文件或事务文件指定一个临时表空间，该表空间还可以用于存储其他数据对象的表空间。
2. 如果表空间被用作临时表空间，该表空间将不能被删除。

创建表空间

注意：此处所说的是6版本之前的概念。

一旦文件空间创建好了，就可以使用该文件空间定义表空间了。要定义一个表空间，使用CREATE TABLESPACE命令，例如：

```
=# CREATE TABLESPACE fastspace FILESPACE gpfs;
```

表空间必须由SUPERUSER才可以创建，不过在创建好之后可以允许普通的DB User来使用该表空间。可以将CREATE权限授予相应的用户。例如：

```
=# GRANT CREATE ON TABLESPACE fastspace TO admin;
```

注意：此处所说的是6版本的概念。

通过CREATE TABLESPACE来创建表空间。例如：

```
=# CREATE TABLESPACE fastspace LOCATION '/data/gpfs';
```

表空间必须由SUPERUSER才可以创建，不过在创建好之后可以允许普通的DB User来使用该表空间。可以将CREATE权限授予相应的用户。例如：

```
=# GRANT CREATE ON TABLESPACE fastspace TO admin;
```

正如看到的这样，在6版本中的表空间的概念发生了很大的变化，不再需要依赖文件空间的概念，而是直接在表空间的定义中指定路径的信息，这既带来了便捷，也带来了困扰，以往，Primary和Mirror的文件空间的路径可能是不同的，而在6版本中，Primary和Mirror的路径必须相同，因为没有地方可以指定相同Content值的Primary和Mirror拥有不同的路径信息。当集群中不同的Content之间的路径也不相同时(实际上只要一台机器多于一个数据目录，肯定就会有不同)，则需要使用这样的

语法来完成表空间的定义：

```
=# CREATE TABLESPACE gpts LOCATION '/data/master_ts'
WITH(content0='/data1/gpts', content1='/data2/gpts');
```

不管是LOCATION还是WITH中的路径都必须指定绝对路径，而且长度不能超过100个字符，否则会有问题，如果没有指定WITH，则所有Content的Instance都需要有LOCATION指定的路径，且gpadmin用户拥有该路径的权限，该目录必须为空目录，建议LOCATION指定Master的目录，其他的Content目录全部在WITH中列出。不要试图通过修改pg_tblspc目录下软连接的指向去设置Primary和Mirror指向不同的目录，因为这样会有一个很大的问题，当出现Primary和Mirror的故障切换时，在做gprecoverseg全量恢复时，GP数据库并不清楚软连接的目标是不同的，这个软连接的信息是不会存储在系统表中的，做全量恢复时，只是按照对应Content活着的Instance的信息去重建。

使用表空间存储 DB 对象

表、索引、甚至整个DB都可以存储在特定的表空间。需要拥有对应的表空间的CREATE权限的ROLE才可以在该表空间上创建对象。例如，在space1表空间上创建一张名为foo的表：

```
=# CREATE TABLE foo(i int) TABLESPACE space1;
```

或者使用缺省表空间参数default_tablespace来设定：

```
=# SET default_tablespace = space1;
=# CREATE TABLE foo(i int);
```

在将default_tablespace设置为一个非空字符串后，其相当于给CREATE TABLE和CREATE INDEX等命令添加一个TABLESPACE的子句，但却不必明确写出。

如果一个表空间与DB关联，那么其将存储所有该DB的系统表、临时文件等。此外，其也是在该DB上创建表、索引等缺省的表空间(除非通过TABLESPACE或者default_tablespace参数重新指定)。如果DB在创建的时候没有与特定的表空间相关联，那么该DB与其使用的模版DB使用相同的表空间。

一旦表空间被创建了，将可以被任何有访问权限的用户在任意的DB中使用。

查看现有的表空间和文件空间

每个GP系统都有两个缺省的表空间：pg_global(用以存储全局系统表的数据，系统保留，不可使用)和pg_default(缺省表空间，存储template1和template0模版数据库和postgres数据库，也是其他数据库的缺省表空间)。

注意：此处所说的是6版本之前的概念。

在6之前的版本中，这些缺省的表空间使用的是缺省的文件空间pg_system(这是GP集群在初始化的时候建立的，使用的是Instance的工作目录)。

要获取文件空间的信息，可以通过系统表pg_filespace和pg_filespace_entry进行关联查询。例如：

```
=# SELECT fsname AS filespace, fsedbid AS dbid,fselocation AS datadir
FROM pg_filespace f, pg_filespace_entry e
WHERE e.fsefsoid = f.oid ORDER BY filespace,dbid;
```

	filespace name	dbid smallint	datadir text
1	gpfs	1	/data/greenplum5/gpfs/gpseg-1
2	gpfs	2	/data/greenplum5/gpfs/gpseg0
3	gpfs	3	/data/greenplum5/gpfs/gpseg1
4	pg_system	1	/data/greenplum5/master/gpseg-1
5	pg_system	2	/data/greenplum5/primary/gpseg0
6	pg_system	3	/data/greenplum5/primary/gpseg1

可通过上述两个系统表与pg_tablespace关联查看表空间的完整定义。例如：

```
=# SELECT spcname AS tablespace, fsname AS filespace, fsedbid AS dbid,
fselocation ||
decode(spcname,'pg_default','/base','pg_global','/global','/'||t.oid) AS
datadir
FROM pg_tablespace t, pg_filespace f, pg_filespace_entry e
WHERE t.spcfsoid = e.fsefsoid AND e.fsefsoid = f.oid
ORDER BY tablespace, dbid;
```

	tablespace name	filespace name	dbid smallint	datadir text
1	gpts	gpfs	1	/data/greenplum5/gpfs/gpseg-1/86118
2	gpts	gpfs	2	/data/greenplum5/gpfs/gpseg0/86118
3	gpts	gpfs	3	/data/greenplum5/gpfs/gpseg1/86118
4	pg_default	pg_system	1	/data/greenplum5/master/gpseg-1/base
5	pg_default	pg_system	2	/data/greenplum5/primary/gpseg0/base
6	pg_default	pg_system	3	/data/greenplum5/primary/gpseg1/base
7	pg_global	pg_system	1	/data/greenplum5/master/gpseg-1/global
8	pg_global	pg_system	2	/data/greenplum5/primary/gpseg0/global
9	pg_global	pg_system	3	/data/greenplum5/primary/gpseg1/global

注意：此处所说的是6版本的概念。

在6版本中，这些缺省的表空间，使用的是系统初始化时指定的Master和Instance的工作目录。

要获取表空间的信息，可以查询pg_tablespace系统表。例如：

```
=# SELECT oid,* FROM pg_tablespace;
```

	oid oid	spcname name	spcowner oid	spcacl aclitem[]	spcoptions text[]
1	1663	pg_default	10		
2	1664	pg_global	10		
3	86891	gpts	10		

然后使用gp_tablespace_location()函数来查看具体某个表空间的目录信息。例如：

```
=# SELECT * FROM gp_tablespace_location(86891) ORDER BY gp_segment_id;
```

	gp_segment_id integer	tblspc_loc text
1	-1	/data/tbsp
2	0	/data/tbsp
3	1	/data/tbsp

通过该函数无法查询到缺省表空间pg_global和pg_default的目录信息，不过，这些信息，可以通过查询gp_segment_configuration系统表的datadir字段来获取。

删除表空间和文件空间

在表空间相关的所有对象被删除之前，该表空间是不能被删除的。同样的，对于6之前的版本来说，在相关的表空间被删除之前，文件空间，也是不能被删除的。

要删除表空间，可通过DROP TABLESPACE命令完成。表空间只能被其Owner和SUPERUSER删除。

在6版本之前，要删除文件空间，可通过DROP FILESPACE命令完成。只有SUPERUSER可以删除文件空间。

注意：在6版本之前，如果文件空间被临时文件或者事务文件使用，该文件空间将不能被删除。在6版本开始，如果表空间被临时文件或者事务文件使用，该表空间将不能被删除。

创建与管理模式

模式(Schema)是在DB内组织对象的一种逻辑结构。模式可以允许用户在一个DB内不同的模式之间使用相同Name的对象(例如Table)。

缺省"Public"模式

每个新创建的DB都有一个缺省的模式public。如果没有创建其他的模式，在创建DB对象时将缺省使用public模式。缺省情况下所有的ROLE(User)都有public模式下的CREATE和USAGE权限。而在创建其他模式时，需要将该模式授权给相关的ROLE(User)。

创建模式

使用CREATE SCHEMA命令来创建一个新的模式。例如：

```
=# CREATE SCHEMA myschema;
```

要访问某个模式中的对象，可以通过模式名加小数点加对象名来指明对象所属的模式。例如：

```
schema.table
```

还可以在创建模式的时候将Owner设置为其他ROLE(User)。语法如下：

```
=# CREATE SCHEMA schemaname AUTHORIZATION username;
```

模式搜索路径

当需要查询特定模式下的对象时，可以通过明确指定模式名的方式来实现。例如：

```
=# SELECT * FROM myschema.mytable;
```

若不想通过指定模式名称的方式来实现，可以通过设置search_path参数来完成。该参数告诉DB在哪些可用的模式中搜索对象。在不指明模式名称的情况下，搜索路径(search_path)列表中的第一个模式将成为缺省模式，例如创建对象等操作时，对象将会自动创建到search_path的第一个模式中。

设置模式搜索路径

search_path用于设置模式的搜索顺序。该参数可以通过ALTER DATABASE命令修改DB的模式搜索路径。例如：

```
=# ALTER DATABASE mydatabase SET search_path TO myschema, public, pg_catalog;
```

还可以通过ALTER ROLE命令修改特定ROLE(User)的模式搜索路径。例如：

```
=# ALTER ROLE sally SET search_path TO myschema, public, pg_catalog;
```

设置了模式搜索路径之后，在未明确指明模式名称的情况下访问DB对象，将会按照search_path列表的顺序依次在相应的Schema中查找对应的Object，直到找到为止，若在不同的Schema中存在相同Name的Object，DB优先匹配search_path中靠前的Schema下的Object。

查看当前的模式

有时不确定当前所在的模式或者搜索路径。要查看这些信息，可以通过使用current_schema()函数或者SHOW命令来查看。例如：

```
=# SELECT current_schema();
=# SHOW search_path;
```

删除模式

使用DROP SCHEMA命令来删除模式。例如：

```
=# DROP SCHEMA myschema;
```

缺省状态下，只有空的模式才可以被删除。若想要直接删除模式及相关的所有Object(Table、Index、Function等)。使用如下命令：

```
=# DROP SCHEMA myschema CASCADE;
```

系统模式

下面的这些系统模式在所有的DB中都存在：

- pg_catalog模式存储着系统表(System Catalog Table)、内置类型(Type)、函数(Function)和运算符(Operator)。该模式无论是否在search_path中指明，都存在search_path中，因为没有这个模式的话，SQL将无法执行，数据库将无法使用。
- information_schema模式由一组标准化视图构成，这些视图用于以标准化的方法从系统表中查看对象信息，不过，该模式中的很多视图定义复杂且臃肿，当库中对象较多时，这些视图的性能可能会很差，有时候只需要查询必要的信息时，可以考虑写SQL重新实现。
- pg_toast模式是一个储存大对象的地方(那些超过页面尺寸(page size)的记录)。该模式仅供GP系统内部使用，通常不建议管理员或者任何用户访问。
- pg_bitmapindex视图是一个储存bitmap index对象的地方。该模式仅供GP系统内部使用，通常不建议管理员或者任何用户访问。
- pg_aoseg视图是一个储存append-optimized表辅助信息的地方。该模式仅供GP系统内部使用，通常不建议管理员或者任何用户访问。
- gp_toolkit是一个管理用的模式，可以查看和检索系统日志文件和其他的系统信息。gp_toolkit视图包含一些外部表、视图、函数，可以通过SQL的方式访问它们。gp_toolkit视图对于所有DB User都是可以访问的。

创建与管理表

GP中的Table除了数据是分布在不同Instance这点外，和其他关系型数据库的Table是很类似的。在创建Table时，需要一个额外的SQL语法来指明Table的分布策略。

创建表

CREATE TABLE命令用于创建一张新的Table和定义其结构。在创建Table时，通常需要定义如下几个方面的信息：

- 都有哪些字段(Column)以及对应的数据类型。
- Table或者Column的约束(Constraint)，其限定了Table或者Column可以储存什么样的数据。

- Table的分布策略，其决定了Table的Data如何被分割存储在GP的各个Instance上。
- Table在Disk上的存储方式。例如压缩、按列存储等选项。
- 大表的分区策略(Partition Table)。

选择 Column 的数据类型

Column的Data Type决定了其可以储存什么类型的数据值。通常应该考虑使用最小的空间储存数据，不是为了节省空间，重要的是，要考虑Data Type对数据范围的约束。例如，使用Character类型储存字符串，Date或者Timestamp储存日期，Numeric储存数字等，如果用TEXT来存储VARCHAR(32)，将会允许任意数据范围，这样会导致很多异常数据不容易被发现。

对于Character类型来说，CHAR、VARCHAR和TEXT之间不存在性能差异，当然是在不考虑填补空白导致的储存尺寸增加的情况下。然而在其他的DB系统中，可能CHAR会表现出最好的性能，但在GP中是不存在这种性能差异的。在多数情况下，应该选择使用TEXT或者VARCHAR而不是CHAR。

对于Numeric类型来说，应该尽量选择更小的数据类型来存储数据。例如，选择BIGINT类型来存储SMALLINT类型范围内的数值，会造成空间的大量浪费。另外，过于宽泛的类型定义会容许异常的数值，造成隐藏的问题，且不容易被发现，应该选择合适的类型，当有异常数据尝试插入表中时，会因为类型不匹配而报错。

对于打算用来做Table Join的Column来说，应该考虑选择相同的数据类型。如果做Join的Column具有相同的数据类型(例如主键Primary Key与外键Foreign Key)，其工作效率会更高。如果两者的数据类型不同，DB还需要将其中一个类型做转换才可以做关联比较，这种开销是不必要的浪费。

设置 Table 和 Column 的约束

数据类型用来限制在Table中可以存储的数据的性质。但对于很多应用来说，数据类型提供的限制粒度太大。SQL标准允许在Table和Column上定义约束。约束将允许在Table的数据上使用更多的限制。如果User试图在Table上储存违反约束的数据将会报错。在GP中使用约束是有一些限制的，最为显著的是外键(Foreign Key)、主键(Primary Key)和唯一约束(Unique Constraint)。对其它类型约束的支持与PostgreSQL相同。

检查约束

检查约束是最常见的约束类型。其通过指定数据必须满足一个布尔表达式来约束。例如，要求产品的价格必须为正数，可以这样：

```
=# CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

非空约束

非空约束简单的理解就是不可以存在空(NULL)值。非空约束是一种Column类型的约束。例如：

```
=# CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

唯一约束

唯一约束可以确保包含某些Column的数据在整个Table中是唯一的。在GP中使用唯一约束存在强制限制条件，Table必须是HASH分布的或者复制分布的(而不是DISTRIBUTED RANDOMLY)，如果是HASH分布的，唯一约束的Column集合必须完整包含所有的DK Column，也就是说，分布键字段的集合必须是唯一约束的字段集合的子集。例如：

```
=# CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
) DISTRIBUTED BY (product_no);
```

复制分布的表，对唯一约束没有限制条件，因为每个Instance都包含了完整的数据，无论如何都不会发生跨Instance检查唯一性的情况。如同之前介绍GP的分布式概念中提到的，GP是Share-Nothing结构的，每个Instance只负责自己的数据和计算，对于唯一约束也一样，假如唯一约束的字段不能包含全部的分布键字段，就可能需要跨Instance检查数据的唯一性，这是无法做到的，也违反Share-Nothing的设计初衷。

主键约束

主键约束就是唯一约束与非空约束的结合体。在GP中使用主键约束存在强制条件，Table必须是HASH分布的或者复制分布的(而不是DISTRIBUTED RANDOMLY)，并且主键约束的Column集合必须完整包含所有的DK Column。如果一个Table包含主键，且在建表时没有指定分布键，那么主键将被用作分布键，也就是说，会把主键包含的所有字段用作分布键。例如：

```
=# CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
) DISTRIBUTED BY (product_no);
```

创建主键和创建唯一约束+非空约束的效果是完全等价的。

外键约束

在目前版本的GP中外键约束是没有被支持的。可以定义外键约束，但参照完整性是无效的，就是说DB不会理会定义参照完整性限制。

外键约束要求一个Table中某些Column的值必须在另外一个Table中出现。这样可以保持两个相关表之间的数据参照完整性。在当前版本的GP中，在分布式的Table之间的数据参照完整性检查是无效的，如同唯一约束中所述，要支持该功能，将无法避

免的需要跨Instance检查数据，性能会失控。

选择表的分布策略

GP的所有Table都是分布式存储的。在CREATE TABLE和ALTER TABLE的时候有个DISTRIBUTED BY(HASH分布)、DISTRIBUTED RANDOMLY(随机分布)或DISTRIBUTED REPLICATED(复制分布)子句用以决定Table的Row数据如何分布。

在选择表的分布策略时需要重点考虑以下几点(依次更重要)：

- **平坦的数据分布** -- 为了尽可能达到最好的性能，所有的 Instance 应该尽量储存等量的数据。若数据的分布不平衡或倾斜，那些储存了较多数据的 Instance 在处理自己那部分数据时将需要耗费更多的时间。为了达到数据的平坦分布，可以考虑选择唯一性较高的 DK。虽然，主键往往是唯一性最好的字段，但是，不建议为了选择一个分布键而去增加一个主键，这是一种逻辑颠倒的做法，通常，应该选择一个常用于大表之间关联的某个唯一性较高的字段作为分布键，一般这个字段可能在其他某个表中具有主键特征，例如，客户 ID，例如会员卡号，例如手机号码，例如身份证号码，等等，在选择分布键时，仅需要考虑大表与大表之间的关联，任何涉及到小表关联的场景均不应作为选择分布键的考虑因素。

- **本地操作与分布式操作** -- 在处理查询时，很多处理如关联、排序、聚合等算子，如果能够在Instance本地完成，其效率将高于需要从其他Instance获取数据的操作。当不同的Table使用相同的DK时，在DK上的关联或者排序操作将会以最高效的方式把绝大部分工作在Instance本地完成。若Table使用随机分布策略，将会出现更多的数据移动(Motion)，虽然随机分布策略可以绝对确保数据分布的平坦性，但也只是确保了数据分布的平坦性。

- **平坦的查询处理** -- 在一个查询正被处理时，我们希望所有的Instance都能够处理等量的工作负载，从而尽可能达到最好的性能。有时候查询场景与数据分布策略很不吻合，这时很可能导致工作负载的倾斜。例如，有一张交易流水表。该表的DK为公司名称，那么数据分布的HASH算法将基于公司名称的值来计算，假如有一个查询以某个特定的公司名称作为查询条件，该查询任务将仅在一个Instance上执行。这里需要解释一下，这种交易流水表，特定公司名称可能会对应大量的记录，这就导致，针对某些交易规模很大的公司进行查询时，繁重的计算任务由一个Instance来完成，这种情况是不应该发生的。编者认为，这个例子是用以说明查询倾斜是如何发生的，这个例子讲述的倾斜与表的分布键有关，虽然该表整体上可能没有显著的倾斜。但不等于说不能使用查询条件字段作为分布键，在真实的应用中，有时为了提高索引查询的并发性能，会特意按照查询字段作为分布键，当然这种字段的重复度较低，由单个Instance完成任务更经济。

复制表(DISTRIBUTED REPLICATED)，应该仅用于小表，将数据复制到所有Instance 的是代价极高的，而且，随着集群规模的增大，这种代价愈加的凸显，应该绝对禁止将尺寸较大的事实表使用复制分布策略！

复制表的主要应用场景：

- 在复制表上没有UDF无法在Instance上查询的限制，以前，UDF如果访问了用户表，则不允许在Instance上执行，而如果访问的是复制表，UDF在Instance上允许对该表进行只读的查询，当然，修改数据的操作仍然是不被允许的。
- 对于与其他表关联时需要被广播的小表来说,使用复制分布策略可以避免广播操作，从而提升查询的性能，实际上，数据相当于提前广播好了。

注意：隐藏的系统字段（ctid、cmin、cmax、xmin、xmax和gp_segment_id）在复制表上是不可用的，如果试图查询这些字段，将会得到一个字段不存在的报错信息。虽然，这些系统字段对于从Master的访问来说没有意义，或者可能会带来歧义或者混淆，但是，不等于说这些字段是真的不存在的，因为复制表在Instance上也是普通的表，也是有这些字段的，所以，如果直接连接到Instance上去查询，是可以查到这些字段信息的。

声明分布键

在创建Table时有一个额外的子句用以指明分布策略。如果在创建Table时没有指明DISTRIBUTED BY、DISTRIBUTED RANDOMLY或者DISTRIBUTED REPLICATED子句，GP将会选择使用HASH分布，并依次考虑使用主键(假如该Table有的话)或者第一个字段作为HASH分布的DK。几何类型或者自定义类型的Column是不适合作为GP的DK的。如果一个Table没有一个合适类型的Column作为DK,该表将使用随机分布策略。另外，如果设置了gp_create_table_random_default_distribution参数的值为on，在不指定任何分布策略的情况下，将会等同于DISTRIBUTED RANDOMLY，这很重要，至少可以避免自动选择DISTINCT值很少的第一个字段作为分布键。

复制表是没有分布键的，因为每一条记录都会分布到整个集群的所有Instance上。

为了确保数据的平坦分布，可能需要选择一个具有较高唯一性的Column作为DK，如果达不到平坦的效果，也可以选择DISTRIBUTED RANDOMLY策略，但这只应该作为最后的选择。例如：

```
=# CREATE TABLE products (
    name varchar(40),
    prod_id integer,
    supplier_id integer
```

```
)DISTRIBUTED BY (prod_id);
=# CREATE TABLE random_stuff (
    things text,
    doodads text,
    etc text
) DISTRIBUTED RANDOMLY;
```

选择表的存储模式

GP提供几种灵活的存储模式(或者混合模式)。在创建一张新的TABLE时，有几个选项来决定数据如何储存在磁盘上。本节介绍这几种选项，以及出于工作负载的考虑如何实现最佳的储存模式。

注意：为了简化建表时定义存储模式，可以设置缺省的存储选项，可以通过参数gp_default_storage_options定义缺省存储选项。

堆存储

缺省情况下GP使用与PostgreSQL相同的存储模式--堆存储。堆存储模式在OLTP类型工作负载的DB中很常用 -- 更适合数据在初始装载后经常变化的场景。UPDATE和DELETE操作需要对ROW级别做版本控制从而确保DB事务处理的可靠性。堆表更适合一些小表，例如维表，这种表可能会在初始化装载后经常更新数据。

创建堆表

行存堆表是缺省的存储模式，创建堆表时不需要额外的CREATE TABLE语法。例如：

```
=# CREATE TABLE foo (a int, b text) DISTRIBUTED BY (a);
```

在6版本中，引入了新的概念，全局死锁检测，以实现降低UPDATE和DELETE锁级别的目标，在6以前的版本中，无论是Heap表还是AO(Column表也是AO表的一种，虽然有人喜欢称为CO表)表，UPDATE和DELETE操作都是表级锁，也就是说，在6之前的版本，一张表上，同时只能有一个UPDATE或者DELETE的语句正在被执行，其他的UPDATE或者DELETE语句需要等待前面的语句执行完成之后才能获得所需要的锁。

在6版本开始，对于Heap表，打开全局死锁监测并关闭Orac优化器，Heap表的

UPDATE和DELETE操作的锁将降低为行级排他锁：

```
$ gpconfig -c gp_enable_global_deadlock_detector -v on
$ gpconfig -c optimizer -v off
$ psql postgres -c "CHECKPOINT"
$ gpstop -af
$ gpstart -a
```

另外，如果要进行大并发的INSERT，DELETE和UPDATE操作，建议关闭log_statement参数，因为过多的日志输出也会影响这种操作的极限性能。编者认为，不应该多度追求这种OLTP型的性能，如果可以通过批量或者微批的形式来处理业，将可以更好的发挥和利用GP的MPP优势，不要总是热衷于跟技术较劲。

追加优化存储

在数据仓库等分析型场景，追加优化(AO)表会表现出更好的性能，这种存储模式非常适合事实表，事实表通常都是规模很大的表，一般都是批量数据操作和只读查询操作，另外，AO表不再维护MVCC信息，可以节省一些存储空间，不仅如此，AO表一般还会选择压缩存储，将可以大大节省存储空间。不过，AO表不适合单行INSERT操作，这是强烈建议应该避免的操作。

创建追加优化表

通过CREATE TABLE的WITH子句来定义存储选项，缺省不指定WITH子句时，创建的是行存堆表(如果设置了gp_default_storage_options参数，存储模式将与该参数的设置一致)。例如，这是一个创建不带压缩选项的追加优化表的例子：

```
=# CREATE TABLE bar (a int, b text)
   WITH (APPENDOPTIMIZED=true) DISTRIBUTED BY (a);
```

注意：APPENDOPTIMIZED是以前的APPENDONLY的别称，在系统表中仍然存储着APPENDONLY关键字，在显示存储信息时也将显示APPENDONLY。CLUSTER，DECLARE . . . FOR UPDATE不适用AO表。

选择行存或列存

GP支持在CREATE TABLE时选择行存或者列存，或者在分区表中为不同的分区做不同的选择，可以有的分区用行存，有的分区用列存。本节提供一些关于正确选择行存或列存的常规指导。不过具体情况还需根据业务场景进行确切的评估，**编者认为，绝大部分情况下，不要选择列存，因为现在的列存技术，文件数膨胀极其严重，后果更严重**。

从一般性的角度来说，行存具有更广泛的适用性。列存对于一些特定的业务场景可以大量节省IO资源以提升性能，也可以提供更好的压缩效果。在考虑选择行存还是列存时，可以参考如下几点：

- **更新操作：**如果一张表在装载完之后有频繁的更新操作，那么就选择行存Heap表。列存表必须是AO表，所以没有列存的Heap表。
- **INSERT操作：**如果有频繁的INSERT操作，那么就选择行存表。列存表不擅长频繁的INSERT操作，因为列存表在物理存储上每一个字段都对应一个文件，频繁的INSERT操作将需要每次都写很多个文件。
- **查询涉及的COLUMN数量：**如果在SELECT列表或者WHERE条件中经常涉及大量的字段，那么就选择行存表。对于大数据量的单字段AGG查询，列存表将会表现的更好。例如：

```
=# SELECT SUM(salary)...
=# SELECT AVG(salary)... WHERE salary > 10000
```

或者在WHERE条件中使用单独的字段进行条件过滤且返回相对少量的记录数。例如：

```
=# SELECT salary, dept ... WHERE state='CA';
```

- **表中的字段数量：**当查询涉及的字段数量很多，或者表中总的字段数量很少，这时，行存表将更有优势。对于表中字段数量很多，而查询只涉及很少的字段时，列存表将更有优势。

- **压缩：**对于列存来说，因为是相同的数据类型连续存储在一起，压缩效果会更好，而行存则无法具备这种优势，所以，同样的数据，同样的压缩选项，往往列存可以获得更好的压缩效果。当然，越好的压缩效果就意味着越困难的随机访问，因为数据的读取都需要解压缩，不过，在6版本引入了ZSTD压缩算法，有非常优秀的压缩和解压缩效率。

创建列存表

在CREATE TABLE时使用WITH子句来指明TABLE的存储模式。如果没有指明，该表将会是缺省的行存堆表。使用列存的TABLE必须是AO表。例如，创建一张不带压缩选项的列存储表：

```
=# CREATE TABLE bar (a int, b text)
    WITH (appendonly=true, orientation=column) DISTRIBUTED BY (a);
```

编者再次提醒，就目前来说，如无必要，请不要在生产中使用列存，否则后果自负！

使用压缩(必须是AO表)

在GP中，AO表可以选择表级别的压缩，也可以针对列存表选择不同的列不同的压缩选项，前者应用到整个TABLE，后者应用到指定的COLUMN。在选择COLUMN级别压缩时，可以为不同的COLUMN选择不同的压缩算法。下表是可用的压缩算法：

行或列	可用压缩类型	支持压缩算法
行	表级别	ZLIB、ZSTD 和 QUICKLZ(开源版本不可用)
列	列级别 和 表级别	RLE_TYPE、ZLIB、ZSTD 和 QUICKLZ(开源版本不可用)

使用库内压缩要求Instance所在的机器具备较强的CPU来压缩和解压缩数据，不过，就目前的主流配置来说，应付ZSTD压缩算法绝对没有问题。不过，不要在使用了压缩文件系统的GP集群使用压缩表，因为这样只会带来额外的CPU消耗，而不会带来文件尺寸的压缩。

在选择AO表的压缩方式和级别时，需要考虑以下几点因素：

- **CPU性能：**机器需要有足够的CPU资源来压缩和解压数据。
- **压缩比和磁盘尺寸：**既要考虑压缩比以减少数据文件的尺寸，也需要考虑CPU的能力，因为越高级别的压缩需要消耗更多的CPU资源来压缩和解压数据。这就要求，我们需要找到一个适中的压缩选项来兼顾压缩比和压缩解压的性能。
- **压缩速度：**虽然说，quicklz与zlib相比，有更好的压缩解压速度，相对来说压缩率差一些，但是，已经有ZSTD可以用了，谁还在乎这些呢。例如，quicklz与zlib level 1的压缩相比，具有相当的压缩率，会有更好的性能，与zlib level 6相比，zlib level 6会有好很多的压缩率，当然性能也会有显著的下降。选择ZSTD，通过选择压缩级别，就可以兼顾压缩率与性能。所以，在6版本之前，一般选择zlib level 5或者6，而在6版本开始，就直接选择ZSTD好了。
- **解压速度或扫表效率：**压缩表的查询性能不仅取决于压缩选项，还与硬件的配比，查询优化等因素有关。应该进行压缩选项的测试以选择适合的压缩选项。

quicklz压缩只有1个压缩级别可以选择，zlib有1~9压缩级别可以选择，ZSTD有1~19压缩级别可以选择，RLE有1~4压缩级别可以选择。

一般来说，在6版本之前，使用zlib 5级或者6级压缩就可以了，6版本使用ZSTD是最好的选择。经过编者在虚拟机中的简单测试发现，ZSTD在9级以上，压缩效率几乎不再有明显的优势，但INSERT的耗时会有很大的增加，例如19级，INSERT的时间可能会是1级的很多倍，不过，各个压缩级别的表扫描性能都表现优良。所以，可以考虑适当的选择ZSTD 6~9级的压缩选项作为通用压缩选项。

RLE可能是一个最没有存在感的压缩算法，因为真的没有什么实用价值，另外，列存表在不同的列上设置不同的压缩选项往往也是多余的，实在多于，毫无价值，因为这样做并不会得到什么有意义的效果，所以，还不如忘了这个事情，直接在表级别设置压缩属性就好了，编者编写的ddl备份脚本就直接忽略了字段级别的压缩属性，因为这毫无意义，列存表的使用就应该是非常谨慎的事情，编者再次提醒，就目前来说，如无必要，请不要在生产中使用列存，否则后果自负！编者见过太多因为滥用列存导致的灾难，如果再结合了滥用分区表，那将是无尽的折磨！

创建压缩表

在CREATE TABLE时使用WITH子句来指明TABLE的存储模式。使用压缩模式的TABLE必须是AO表。例如，要创建一张5级ZSTD压缩的AO表：

```
=# CREATE TABLE foo (a int, b text)
    WITH (appendonly=true, compresstype=zstd, compresslevel=5);
```

检查 AO 表的压缩与分布情况

GP提供了内置的函数用以检查AO表的压缩率和分布情况。这两个函数可以使用对象ID或者TABLE的NAME作为参数。表名可能需要带模式名。

函数	返回类型	解释
get_ao_distribution(name) get_ao_distribution(oid)	集合类型 (dbid, tuplecount)	展示 AO 表的分布情况，每行对应 segid 和记录数。
get_ao_compression_ratio(name) get_ao_compression_ratio(oid)	float8	计算出 AO 表的压缩率。如果该信息未得到，将返回-1 值。

压缩率得到的是一个常见的比值类型。例如，6.19，意思是该TABLE未压缩状态下的储存尺寸是压缩状态下的储存尺寸的6.19倍。

分布信息展示的是每个Instance存储该TABLE的记录数量。例如，在一个有着4个Instance的系统，其dbid范围为0 - 3，该函数返回类似下面的结果集：

```
=# SELECT * FROM get_ao_distribution('lineitem_comp');
 segmentid | tupcount
-----------+----------
         0 |  7500721
         1 |  7501365
         2 |  7499978
         3 |  7497731
(4 rows)
```

支持运行长度编码

GP已支持COLUMN级别的运行长度编码(Run-length Encoding /RLE)压缩算法。RLE是一种将连续重复的数据作为一种计数方式存储的压缩算法。RLE对于重复元素是很有效的。例如，在一个表中有两个COLUMN，一个日期COLUMN和一个描述COLUMN，其中包含200000个date1和400000个data2，RLE压缩处理这种数据为类似data1 200000 data2 400000这样的效果。对于那些没有很多重复值的数据RLE是不适合的，而且还可能会显著的增加存储文件的尺寸。

RLE压缩有4种级别。级别越高，压缩效率越高，但压缩速度也会越低。

使用了RLE压缩的行存表对4.2.1之前的版本是不兼容的。若需要将这些表备份并在之前的版本上恢复，可在恢复操作前，先将这些表ALTER为无压缩或者旧版本兼容的压缩模式(ZLIB或QUICKLZ)，再执行恢复操作，因为RLE模式的表定义在4.2.1之前的版本不兼容。

实际上，正如前面所述，RLE压缩算法并没有什么实用意义，忘记这个事情就好了，好好使用ZSTD就对了。

在列上设置压缩

注意：编者不希望读者浪费很多时间来学习这部分的知识，所以，先把观点列出来，编者根据10年的经验判断，除了作为一块知识来学习外，可能永远也不需要在每个字段

上设置压缩，因为那是极其多余和毫无意义的。在真实的使用环境中，往往列存储的选择都应该是极其少见的，因为列存储的选择需要满足多方面条件，选择列存的往往是那种尺寸很大的表，追求较高的压缩率，字段很多，而往往只需要访问很少的字段，且追求很好的查询性能。另外，列存表，在底层的数据文件层面，每个字段都单独存放一个文件，随意大量的使用列存储会造成文件数量巨大，文件尺寸过于零碎，影响文件系统的性能，从而导致数据库性能下降甚至稳定性变差。即便是在列存表上，通常也只需要在TABLE层面设置统一的压缩选项即可，为不同的COLUMN指定不同的压缩选项往往是徒劳的，因为这样并不会得到显著的压缩率的提升或者性能的提升。所以，如果您是为了扩展知识，可以继续阅读，如果是为了实用，可以跳过此节，不要浪费生命。

在CREATE列存表时，允许针对每个字段设置单独的存储选项，包括如下三种属性可选：

- 压缩算法
- 压缩级别
- 列数据文件的块尺寸(Block Size)

在CREATE TABLE、ALTER TABLE和CREATE TYPE命令中包含对COLUMN设置压缩类型、压缩级别和块尺寸(Block Size)的选项(注意，ALTER TABLE时不能修改这些选项，仅在增加分区时可选)，这些参数统称为列存储参数。下面列举这3种存储参数及每种参数的可选值：

名称	解释	可选值
COMPRESSTYPE	使用的压缩类型	ZSTD(最好的压缩选择) ZLIB(更高压缩) QUICKLZ(更快压缩) RLE_TYPE(运行长度编码) none(无压缩、缺省值)
COMPRESSLEVEL	压缩级别	ZSTD为1~19级可选
		ZLIB为1~9级可选
		QUICKLZ仅1个级别可选(缺省不需指定)
		RLE_TYPE为1~4级可选
BLOCKSIZE	表的存储块大小	8192 ~ 209715(8K ~ 2M)该值必须是8192的倍数

对列存表单独的字段使用存储参数的格式如下：

```
[ ENCODING ( storage_options [, . . .] ) ]
```

这里ENCODING关键字是必须的，存储参数包含3个部分：参数名称、等于号、参数值。要指定多个存储参数，用逗号(,)分割即可。存储参数可以应用在单独的COLUMN上，还可以作为所有COLUMN的缺省设置，如下面的CREATE TABLE语句所示：

一般用法：

```
column_name data_type ENCODING ( storage_options [, . . . ] ),. . .
COLUMN column_name ENCODING ( storage_options [, . . ] ), . . .
DEFAULT COLUMN ENCODING ( storage_options [, . . . ] )
```

例如：

```
C1 char ENCODING (compresstype=quicklz, blocksize=65536)
COLUMN C1 ENCODING (compresstype=quicklz, blocksize=65536)
DEFAULT COLUMN ENCODING (compresstype=quicklz)
```

列级缺省压缩属性

在使用了列压缩，没有为字段设置ENCODING选项，又没有设置DEFAULT COLUMN ENCODING的情况下，该字段将不会使用压缩存储，而块大小使用系统配置参数block_size指定的值，此时，该字段的存储属性，与普通AO表一致。

列级压缩设置的优先级

COLUMN的压缩设置通过TABLE向分区再向子分区传递。在越低级别的设置具有越高的优先级。

- 表层面的COLUMN压缩选项的设置，将覆盖TYPE层面的设置。这里可能会有点糊涂，不过，后续会介绍到通过TYPE来设置COLUMN级别的压缩选项。
- 子分区的表层面的COLUMN压缩设置将覆盖其父级表的任何COLUMN和TABLE层面的设置。
- 子分区的COLUMN层面的COLUMN压缩设置将覆盖任何分区层面的压缩设置、父表层面的COLUMN设置、父表层面的设置。
- ENCODING的设置高于WITH子句的设置。

这里关于优先级的说明，源文档中有5句，但是，实在难以说清楚，原文的英文说明理解起来可能还比较清晰，但要用中文讲清楚实在是困难，这里可以简单的一句话来总结：越靠近具体字段的压缩设置，优先级越高。具体的可以参考后续的例子来学习和理解。

注意：COLUMN表不允许使用INHERITS语法，若使用LIKE子句来创建TABLE，表层面

的存储参数以及COLUMN层面的存储参数都会被忽略，不会被继承。

列压缩设置的优选

最佳的方法是根据不同的数据设置特定的列压缩。下面的例5展示了2级分区表在子分区上使用RLE_TYPE压缩。

列级压缩的例子

下面的例子展示了在使用CREATE TABLE语句时使用列级压缩设置。

例1

该例子中，COLUMN c1使用ZSTD压缩并使用系统定义的块尺寸。COLUMN c2使用QUICKLZ压缩并使用块尺寸为65536。COLUMN c3不使用压缩且使用系统定义的块尺寸。

```
=# CREATE TABLE T1 (
   c1 int ENCODING (compresstype=zstd),
   c2 char ENCODING (compresstype=quicklz, blocksize=65536),
   c3 char
) WITH (appendonly=true, orientation=column);
```

例2

该例子中，COLUMN c1使用ZLIB压缩并使用系统定义的块尺寸。COLUMN c2使用QUICKLZ压缩并使用块尺寸为65536。COLUMN c3使用RLE_TYPE压缩并使用系统定义的块尺寸。

```
=# CREATE TABLE T2 (
   c1 int ENCODING (compresstype=zlib),
   c2 char ENCODING (compresstype=quicklz, blocksize=65536),
   c3 char,
   COLUMN c3 ENCODING (compresstype=RLE_TYPE)
)WITH (appendoptimized=true, orientation=column);
```

例3

该例子中，COLUMN c1使用ZLIB压缩并使用系统定义的块尺寸。COLUMN c2使用QUICKLZ压缩并使用块尺寸为65536。COLUMN c3使用RLE_TYPE压缩并使用系统定义的块尺寸。值得注意的是在子分区的定义中，COLUMN c3使用了ZLIB(不是RLE_TYPE)压缩，由于分区中的COLUMN存储设置的优先级比父表层面的COLUMN存储设置的优先级高，实际上c3使用的是ZLIB压缩而非RLE_TYPE压缩。

```
=# CREATE TABLE T3 (
    c1 int ENCODING (compresstype=zlib),
    c2 char ENCODING (compresstype=quicklz, blocksize=65536),
    c3 text, COLUMN c3 ENCODING (compresstype=RLE_TYPE)
)WITH (appendoptimized=true, orientation=column)
PARTITION BY RANGE (c3)(
    START ('2020-01-01'::DATE) END ('2100-12-31'::DATE),
    COLUMN c3 ENCODING (compresstype=zlib)
);
```

例4

该例子中，创建TABLE时，COLUMN c1是ZLIB压缩并使用系统定义的块尺寸。而COLUMN c2没有明确指定存储设置，因此其将从DEFAULT COLUMN ENCODING子句继承压缩方式(QUICKLZ)和块尺寸(65536)。COLUMN c3指定了压缩方式为RLE_TYPE，而块尺寸(65536)从DEFAULT COLUMN ENCODING子句继承而来。COLUMN c4没有压缩。因为缺省列存储设置指定了压缩模式，COLUMN c4明确指定了不压缩，其块尺寸从DEFAULT COLUMN ENCODING子句继承而来为65536。

```
=# CREATE TABLE T4 (
    c1 int ENCODING (compresstype=zlib),
    c2 char,
    c3 text,
    c4 smallint ENCODING (compresstype=none),
    DEFAULT COLUMN ENCODING (compresstype=quicklz,blocksize=65536),
    COLUMN c3 ENCODING (compresstype=RLE_TYPE)
)WITH (appendoptimized=true, orientation=column);
```

例5

该例子中，创建一个2层分区表。p1分区的子分区sp1的i字段采用zlib压缩，块尺寸为65536，p2分区的子分区sp1的字段i采用rle_type压缩并使用系统定义的块尺寸，p2分区的子分区sp1的字段k使用块尺寸为8192。

```
=# CREATE TABLE T5(
    i int,
```

```
    j int,
    k int,
    l int
)WITH (appendoptimized=true, orientation=column)
PARTITION BY range(i) SUBPARTITION BY range(j)(
    partition p1 start(1) end(2) (
        subpartition sp1 start(1) end(2),
        column i encoding(compresstype=zlib, blocksize=65536)
    ),
    partition p2 start(2) end(3) (
        subpartition sp1 start(1) end(2),
        column i encoding(compresstype=rle_type),
        column k encoding(blocksize=8192)
    )
);
```

关于如何在现有TABLE上新增COLUMN并设置压缩，请参考ALTER TABLE命令。

通过 TYPE 来设置列的压缩

在创建新类型时，可以定义该类型的默认压缩属性。例如，下面的CREATE TYPE命令定义了一个名为int33的类型(这只是个例子，实际上这个SQL不能成功执行，因为没有定义类型所需的函数)，该类型设置了quicklz压缩：

```
=# CREATE TYPE int33 (
    internallength = 4,
    input = int33_in,
    output = int33_out,
    alignment = int4,
    default = 123,
    passedbyvalue,
    compresstype="quicklz",
    blocksize=65536,
    compresslevel=1
);
```

在CREATE TABLE时将int33指定为字段类型时，将使用在类型定义中指定的压缩选项：

```
=# CREATE TABLE t2 (
```

```
    c1 int33
)WITH (appendoptimized=true, orientation=column);
```

注意：编者不建议使用这种方式来定义列压缩，虽然在定义TABLE时看起来精简了不少，但对于别人来说，阅读和理解可能都存在障碍。另外替代原生TYPE的定义未必适应所有情况。建议慎用。

选择块尺寸

在一个TABLE中，每个块尺寸意味着相应数量byte的存储。块尺寸必须在8192到2097152之间，并且必须是8192的整数倍。缺省值为32768。需要注意的是，指定大的块大小会消耗更多的内存资源。GP会为每个分区、列存表的每个列维护一块BUFFER，因此重度分区表和列存储表也会消耗更多的内存。

根据以往的经验，修改blocksize往往并不会带来任何的好处，所以，没必要在这一块花很多的心思，因为不会有什么有价值的收获。

修改表定义

ALTER TABLE命令用于改变现有表的定义。通过ALTER TABLE命令可以改变TABLE的各种属性，如：列定义、分布策略、存储模式和分区结构(可参见"维护分区表"章节)等。例如，将TABLE的一个COLUMN添加一个非空限制：

```
=# ALTER TABLE address ALTER COLUMN street SET NOT NULL;
```

修改表的分布策略

ALTER TABLE命令提供了改变分布策略的选项。在修改TABLE的分布策略时，表中的数据要在磁盘上做重分布，该操作可能需要密集的资源消耗。还有一个按照现有策略重新分布数据的选项。对于分区表来说，修改分布策略会递归的应用于所有的子分区。该操作不会改变表的OWNER以及其他TABLE属性。例如，下面的SQL是修改sales表的DK为customer_id并重分布数据：

```
=# ALTER TABLE sales SET DISTRIBUTED BY (customer_id);
```

在修改TABLE为HASH分布时，表数据会自动重新分布。然而，将分布策略改为随机分布时不会同时进行数据重新分布操作。例如：

```
=# ALTER TABLE sales SET DISTRIBUTED RANDOMLY;
```

当修改分布策略为DISTRIBUTED REPLICATED或者从DISTRIBUTED REPLICATED修改为其他类型时，将会自动进行数据重分布。

重分布表数据

对于由HASH分布改为随机分布策略的表来说，因为数据并不会自动完成重分布操作，所以，很可能需要有数据重分布的操作，或者在进行扩容时，数据的重分布操作也是很有必要的，使用REORGANIZW=TRUE来对数据进行重分布。例如：

```
=# ALTER TABLE sales SET WITH (REORGANIZE=TRUE);
```

该命令会在Instance之间按照现有的分布策略(包括随机分布策略)重新分散表中数据。

当修改分布策略为DISTRIBUTED REPLICATED或者从DISTRIBUTED REPLICATED修改为其他类型时，即便指定了REORGANIZE=FALSE，也一定会自动进行数据重分布。

修改表的存储模式

存储选项只能在CREATE TABLE时指定，包括Heap和AO的选择，压缩的选择，行和列的选择等，CREATE TABLE时WITH子句中的属性都只能在CREATE TABLE时指定。要修改这些属性，必须采用重新建表的方式进行，即，按照预期的属性设置，创建一张新表，把数据插入新表，删除旧的表，修改新表的名称为旧的表名。另外，还需要重建权限和依赖关系，假如旧的表上有视图，则需要先删除视图，等修改了表名之后再重建视图。例如：

```
=# CREATE TABLE sales2 (LIKE sales)
```

```
WITH (appendoptimized=true,compresstype=zstd,compresslevel=9);
=# INSERT INTO sales2 SELECT * FROM sales;
=# DROP TABLE sales;
=# ALTER TABLE sales2 RENAME TO sales;
=# GRANT ALL PRIVILEGES ON sales TO admin;
=# GRANT SELECT ON sales TO guest;
```

在现有表上添加压缩列

可以使用ALTER TABLE命令一个列存表上添加一个压缩字段，非列存表不能为新增的字段指定ENCODING子句，否则会报错。关于压缩列的设置，参见"在列上设置压缩"章节。下面的例子演示了在现有的T1表上增加zlib压缩列：

```
=# ALTER TABLE T1 ADD COLUMN c4 int DEFAULT 0 ENCODING (compresstype=zlib);
```

继承压缩设置

在增加一个子分区(非一级分区)时，新的分区如果没有明确指定压缩属性，将不会从父表继承，TEMPLATE是预先设置的模版，可以指定新增子分区的压缩属性。下面的例子演示了创建一个带子分区设置的表，然后增加一个分区：

```
=# CREATE TABLE ccddl (
    i int,
    j int,
    k int,
    l int
)WITH(appendoptimized = TRUE, orientation=COLUMN)
PARTITION BY range(j)
SUBPARTITION BY list (k)
SUBPARTITION template(
    SUBPARTITION sp1 values(1, 2, 3, 4, 5),
    COLUMN i ENCODING(compresstype=ZLIB),
    COLUMN j ENCODING(compresstype=QUICKLZ),
    COLUMN k ENCODING(compresstype=ZLIB),
```

```
    COLUMN l ENCODING(compresstype=ZLIB)
)
(
    PARTITION p1 START(1) END(10),
    PARTITION p2 START(10) END(20)
);
=# ALTER TABLE ccddl ADD PARTITION p3 START(20) END(30);
```

运行ALTER TABLE命令创建了ccddl表的一级分区ccddl_1_prt_p3和二级分区ccddl_1_prt_p3_2_prt_sp1。一级分区ccddl_1_prt_p3与二级分区sp1具有不同的压缩设置。这里的二级分区(sp1)其实是沿用了SUBPARTITION TEMPLATE的设置,而不是所谓的继承,基本上很多版本中,分区的存储设置不会自动从父级表继承,需要明确指定，SUBPARTITION TEMPLATE与普通的继承不同，是一种预置的模板。

关于为分区表增加分区时，新增的分区不能继承父表的压缩选项，这可能会是一个非常让人烦躁的问题,至少曾经有些版本是继承的,也有很多版本是没有继承的,不过,从目前编者的测试来看，目前最新的5版本和6版本都是不继承的，所以，养成新增分区时写好WITH子句的习惯很重要,或者设置好gp_default_storage_options参数。

删除表

可通过DROP TABLE命令从DB中删除表。例如：

```
=# DROP TABLE mytable;
```

要想在不删除表定义的情况下清空表中的记录,使用DELETE或TRUNCATE命令。例如：

```
=# DELETE FROM mytable;
=# TRUNCATE mytable;
```

DROP TABLE会删掉所有与该表相关的索引、规则、触发器、约束等。然而要一起删除与该表相关的视图VIEW，必须使用CASCADE。CASCADE会删除所有依赖该TABLE的VIEW。如果不使用CASCADE，当表上有依赖时，DROP操作将会报错失败。例如：

```
=# DROP TABLE mytable CASCADE;
```

DELETE操作是将表中的数据标记为丢弃，不会真正的释放存储的空间，而TRUNCATE会直接清空数据文件，前者的数据找回相对容易，后者只能通过文件系统来恢复，会很困难。

分区大表

表分区用以解决特别大的表的管理问题，例如事实表，可将这种数据量很大的表分成一些尺寸适中且更容易管理的分区表。分区表在执行特定的查询语句时只扫描相关分区的数据而不是全部的数据从而可以提升查询性能。分区表对于数据库的管理也有帮助，可以更好的进行数据周期管理。**但是，编者提醒，不要滥用分区表，如果在不分区的情况下，性能可以满足需求，那么就不要分区，如无必要，不要分区，尤其不要滥用分区，尤其要杜绝过度分区，否则后果自负！**

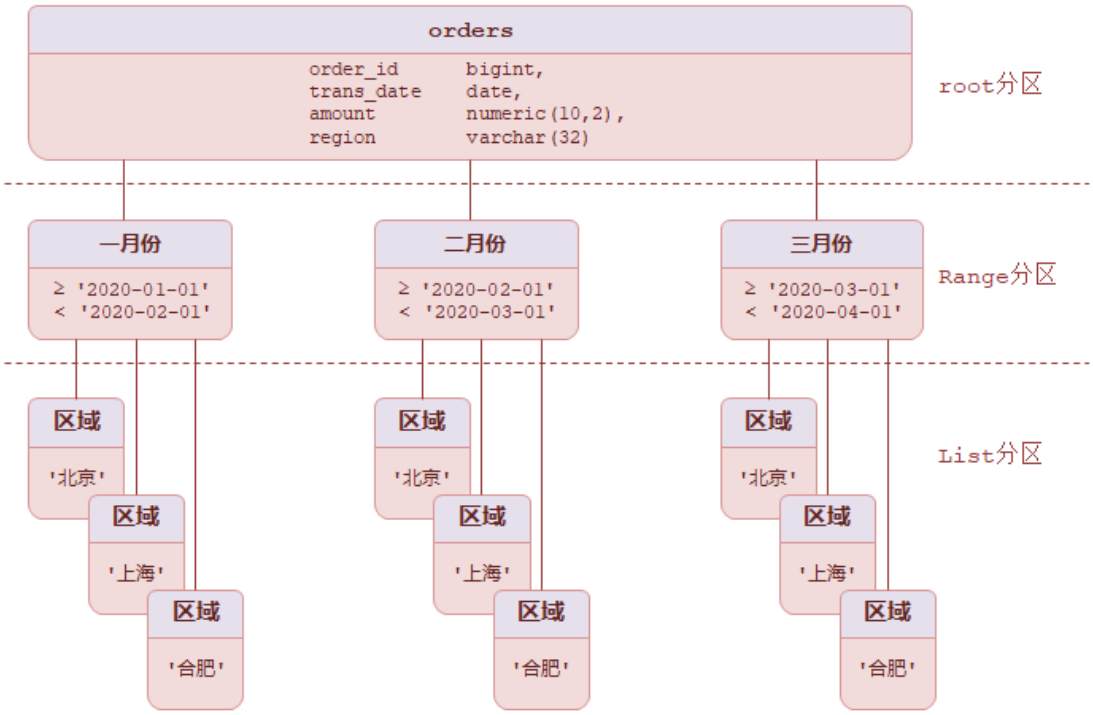
理解 GP 的表分区

在CREATE TABLE时使用PARTITION BY(以及可选的SUBPARTITION BY)子句来对表进行分区。在GP中对一张表做分区，实际上是创建了一张根(ROOT表)表和多个子表。在内部，GP在ROOT表与子表之间创建了继承关系(类似于PostgreSQL中的继承/INHERIT功能)，不过，**切忌想当然的去使用INHERIT语法来创建分区表，那样创建出来的并不是分区表，**GP的分区表不仅仅有继承关系，还有一些其他的系统表信息来决定分区关系。

根据分区创建时定义的准则，每个分区在创建时都带有一个不同的检查(CHECK)约束，其限制了该表可以包含的数据。该检查约束还用于查询优化器在执行特定的查询语句时决定扫描哪些分区。

分区层级关系被储存在GP系统表中，因此插入到ROOT表的数据将被存储到相应的叶子节点的分区中。任何对分区结构的修改或者TABLE定义的修改都需要通过对ROOT表使用ALTER TABLE命令来完成，修改分区结构需要再结合PARTITION子句来完成。

GP支持范围(根据数值型字段的取值范围分割数据，可以比较大小的数据类型都可以作为范围分区的字段，例如日期或价格)分区和列表(根据值列表分区，例如区域或生产线)分区，或者两种类型的结合(在不应该被使用的多级分区表中，不同层级的分区可以选择不同的分区类型)。



分区表和其他非分区表一样都是在GP的Instance之间分布的。表在GP的Instance之间物理上的分散可以确保并行查询处理。表分区是一种大表逻辑切分的手段。分区本身不会改变Instance间物理上的数据分布规律。

决定表是否分区的原则

并不是任何TABLE都适合做分区。如果下列问题的全部或者大部分的答案是Yes，这样的表可以通过分区来提高查询性能。如果大部分的答案是No，分区不是好的选择：

- **表是否足够大？**大的事实表才适合做分区。对于一张很大的表，从逻辑上把表分成较小的分区将可以改善性能。而对于较小的表，对分区的管理和维护的开销可能已经超过了性能的改善程度。那么对于是否选择分区这个问题来说，什么样的表算大表，什么样的表算小表，根据编者十多年的经验来看，不能仅仅从表的尺寸或者记录数来简单的区分，还应该结合集群规模来考虑，一般建议每个分区在每个Instance上的数据量可以控制在100万到1000万左右的范围(落实到项目中的具体标准可以视情况而定，例如有100个Primary的集群，可能会规定记录数在一亿条以下的表不允许分区)会比较合适，这样的分区粒度是适中的。如果对于列存储的表来说，这个范围还可以再放大10倍甚至更高，因为列存储的表是按照每个字段一个数据文件来存储的。

- **对目前的性能不满意？**作为一种调优方案，应该在查询性能低于预期时再考虑对表进行分区。分区不是万能的优化手段，GP已经是MPP架构，对于很多不是很大的表，不分区的性能已经完全满足预期的情况下，分区是多于的。

- **查询条件是否能匹配分区条件？**检查查询语句的WHERE条件是否与考虑分区的COLUMN一致。例如，如果大部分的查询使用日期条件，那么按照月或者周进行分区设计也许很有用，而如果查询条件更多的是使用地区条件，可以考虑使用地区字段将表做列表类型的分区。不过，一般来说，List分区的使用场景极少，尤其要避免多级分区，因为维护更复杂。

- **数据仓库是否需要滚动历史数据？**历史数据的滚动需求也是分区设计的考虑因素。例如，数据仓库中仅需要保留过去两个月的数据。如果数据按月进行分区，将可以很容易的删除掉两个月之前的数据(TRUNCATE分区或者删除分区)，而最近的数据存入最近月份的分区即可。

- **按照某个规则数据是否可以被均匀的分拆？**应该选择尽量把数据均匀分拆的规则。若每个分区储存的数据量相当或者与分区跨度成比例，那么查询性能的改善将与分区的数量或者条件的范围相关。例如，把一张表分为10个分区，命中单个分区条件的查询性能可能会比未分区的情况下高10倍。编者认为，我们不应该简单的说查询性能是10倍，因为多数的查询不是count(*)这样简单的计数，对于其他复杂的算子，除了扫描过滤数据这部分可以提升外，后续的处理部分是不会有性能提升的，因为计算量是不变的。

注意：千万不要随意的创建大规模分区表，因为分区的维护和管理也是需要消耗资源和精力的，例如全表查询，VACUUM，gprecoverseg，gpexpand等操作，都是需要一个一个的叶子分区去操作。另外，如果是全表扫描，分区不仅无法带来性能的提升，反而会更慢。因此，如果不太会用到分区消除的查询场景，应尽量避免分区，当然，有时为了数据周期管理，需要进行分区，此时应考虑更粗粒度的分区。尤其应该杜绝使用多级分区，多级分区一般并不会比只有一级分区的表带来更显著的性能提升，同时会带来大量的系统信息维护的压力，以及大量的数据文件管理压力。正如前面所述，应该按照每个Instance存储的记录数(理想情况是按照尺寸来衡量，但一般界定和落实比较难)来制定标准界定合理的分区粒度。编者的一键式集群安装初始化脚本中会缺省将gp_max_partition_level参数设置为1，不允许建多级分区。

创建分区表

TABLE只能在执行CREATE TABLE命令时被分区。

表做分区的第一步是选择分区类型(范围分区或列表分区)和分区字段，决定分区的层数，分区字段的取值范围，分区的粒度等。例如，先按照日期范围划分一级月分区，再按照区域做二级列表分区。本节通过示例演示如何创建多种分区表。编者再次提醒，如无必要，杜绝在生产中创建多级分区表，否则后果自负！

- 定义时间范围分区表
- 定义数字范围分区表

- 定义列表分区表
- 定义多级分区表
- 将现有表分区

定义时间范围分区表

日期范围分区，可以使用单个date或者timestamp类型的字段作为分区字段。如果需要，还可以使用同样的字段做子分区(例如按月分区后再按日做子分区，当然，实际使用中应杜绝这样做，此处只是举例而已)。使用日期分区时优先考虑直接使用最细粒度的分区。例如，一年365天，直接按日每日一个分区，而不是先按年分区、然后按月分区、再按日分区。多级分区会降低执行计划的性能，但水平的分区设计对执行计划的影响会好一些(只是比多级分区好一些，但过多的分区仍然会有很大的影响，要杜绝过度分区！)。

可以通过使用START和END指定分区边界，还可以结合EVERY子句定义分区步长让数据库自动在分区边界内，产生多个范围分区，不过这是一次性的，当表创建成功后，数据库不会自动增减分区，GP中目前没有自动维护分区的概念(自动增加不存在的分区)。缺省情况下，START值总是被包含而END值总是被排除。例如：

```
=# CREATE TABLE sales (
    id int,
    date date,
    amt decimal(10,2)
)DISTRIBUTED BY (id) PARTITION BY RANGE (date) (
    START (date '2020-01-01') INCLUSIVE
    END (date '2021-01-01') EXCLUSIVE EVERY (INTERVAL '1 month')
);
```

也可以为每个分区单独指定名称。例如：

```
=# CREATE TABLE sales (
    id int,
    date date,
    amt decimal(10,2)
) DISTRIBUTED BY (id) PARTITION BY RANGE (date) (
    PARTITION Jan20 START (date '2020-01-01') INCLUSIVE ,
    PARTITION Feb20 START (date '2020-02-01') INCLUSIVE ,
    PARTITION Mar20 START (date '2020-03-01') INCLUSIVE ,
    PARTITION Apr20 START (date '2020-04-01') INCLUSIVE ,
    PARTITION May20 START (date '2020-05-01') INCLUSIVE ,
```

```
    PARTITION Jun20 START (date '2020-06-01') INCLUSIVE ,
    PARTITION Jul20 START (date '2020-07-01') INCLUSIVE ,
    PARTITION Aug20 START (date '2020-08-01') INCLUSIVE ,
    PARTITION Sep20 START (date '2020-09-01') INCLUSIVE ,
    PARTITION Oct20 START (date '2020-10-01') INCLUSIVE ,
    PARTITION Nov20 START (date '2020-11-01') INCLUSIVE ,
    PARTITION Dec20 START (date '2020-12-01') INCLUSIVE
       END (date '2021-01-01') EXCLUSIVE
);
```

注意：由于分区的范围限制是连续的，这种情况下，可以不用为每个分区指定END值，而只需要为最后一个分区指定即可，这个时候，上一个分区的END值就是下一个分区的START值。但如果分区的范围不是连续的，应该为每个不连续的上一个分区指定END值。

定义数字范围分区表

数字范围分区表使用单个数字列作为分区字段。例如：

```
=# CREATE TABLE rank (
    id int,
    rank int,
    year int,
    gender char(1),
    count int
)DISTRIBUTED BY (id) PARTITION BY RANGE (year) (
    START (2020) END (2021) EVERY (1),
    DEFAULT PARTITION extra
);
```

关于默认分区的更多信息，参见”添加默认分区”相关章节。

定义列表分区表

列表分区表可以使用任何数据类型的列作为分区字段，分区规则使用等值比较。列表分区可以使用多个COLUMN(组合起来)作为分区字段，而范围分区只允许使用单独COLUMN作为分区字段。对于列表分区，必须为每个分区指定相应的值。例如：

```
=# CREATE TABLE customer (
   id int,
   name varchar(32),
   birthday date,
   gender char(1),
   viplevel int
) DISTRIBUTED BY (id) PARTITION BY LIST (gender,viplevel) (
   PARTITION girls1 VALUES (('F',1)),
   PARTITION girls2 VALUES (('F',2)),
   PARTITION boys1 VALUES (('M',1)),
   PARTITION boys1 VALUES (('M',2)),
   DEFAULT PARTITION other
);
```

这里只是展示一下多字段LIST分区的语法，实际使用中可能永远也不需要用到这种多字段分区的语法，LIST分区的使用场景本来就少，而且这种分区表会很难维护。

关于默认分区的更多信息，参见”添加默认分区”相关章节。

定义多级分区表

当需要多级分区时(应该永远都不需要)，可以使用多级分区的设计。使用SUBPARTITION TEMPLATE来确保每个分区具有相同的子分区结构，尤其是对那些后增加的分区来说。例如，创建一个两层的分区表：

```
=# CREATE TABLE sales (
    trans_id int,
   date date,
   amount
   decimal(9,2),
   region text
) DISTRIBUTED BY (trans_id)
PARTITION BY RANGE (date)
SUBPARTITION BY LIST (region)
SUBPARTITION TEMPLATE (
   SUBPARTITION usa VALUES ('usa'),
   SUBPARTITION asia VALUES ('asia'),
   SUBPARTITION europe VALUES ('europe'),
   DEFAULT SUBPARTITION other_regions
)
```

```
(
    START (date '2020-01-01') INCLUSIVE
    END (date '2021-01-01') EXCLUSIVE
    EVERY (INTERVAL '1 month'),
    DEFAULT PARTITION outlying_dates
);
```

下面是一个3级分区表的例子，这里sales表被按照年、月、区域进行三级分区。第一个SUBPARTITION TEMPLATE子句确保每年的一个一级分区都有12个月的子分区和1个默认分区，第二个SUBPARTITION TEMPLATE子句确保每个月的二级分区都有3个LIST分区和1个默认分区：

```
=# CREATE TABLE sales (
    id int,
    year int,
    month int,
    day int,
    region text
) DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
SUBPARTITION BY RANGE (month)
SUBPARTITION TEMPLATE (
    START (1) END (13) EVERY (1),
    DEFAULT SUBPARTITION other_months
)
SUBPARTITION BY LIST (region)
SUBPARTITION TEMPLATE (
    SUBPARTITION usa VALUES ('usa'),
    SUBPARTITION europe VALUES ('europe'),
    SUBPARTITION asia VALUES ('asia'),
    DEFAULT SUBPARTITION other_regions
)
(
    START (2020) END (2026) EVERY (1),
    DEFAULT PARTITION outlying_years
);
```

注意：当使用多级RANGE分区时，很容易产生大量的子分区，如前面所说的，这会带来极大的性能问题和系统表压力。应该尽可能杜绝创建多级分区表。

将现有表分区

对已经创建的未分区的表是不能直接修改为分区表的。只能在CREATE TABLE的时候进行分区定义。要想对现有的表做分区，只能重新创建一个分区表、重新装载数据到新的分区表中、删掉旧表然后把新的分区表改为旧表的名称。另外，还需要重建权限和依赖关系，例如旧的表上有视图，则需要先删除视图，等修改了表名之后再重建视图。例如：

```
=# CREATE TABLE sales2 (LIKE sales)
PARTITION BY RANGE (date) (
    START (date 2020-01-01') INCLUSIVE
    END (date '2021-01-01') EXCLUSIVE
    EVERY (INTERVAL '1 month')
);
=# INSERT INTO sales2 SELECT * FROM sales;
=# DROP TABLE sales;
=# ALTER TABLE sales2 RENAME TO sales;
=# GRANT ALL PRIVILEGES ON sales TO admin;
=# GRANT SELECT ON sales TO guest;
```

分区表的限制

对于一个分区表中任何一个层级的某个分区来说，其最多只能有32767个子分区，这是因为pg_partition_rule系统表的parruleord字段是smallint类型。

分区表上的主键或者唯一约束，必须包含所有的分区字段。唯一索引可以不包含分区字段，但其只对叶子分区有效，不能对整个分区表有效，所以，这种不包含分区字段的唯一索引不能在整个分区表层面保证数据的唯一性。

使用DISTRIBUTED REPLICATED分布策略的复制表不能进行分区。

Orca支持规整(每个分区的子分区结构完全相同)的多级分区表，Orca是缺省打开的，如果Orca启用且查询的是不规整的多级分区表，该查询将会自动切换到Legacy(PostgreSQL优化器)的优化器。

当叶子分区存在外部表时，分区表还会有以下的限制(看到这么多的限制，估计想一想就不太会用外部表做分区表的分区了)：

- 对包含外部表的分区表进行查询时，将只能使用PostgreSQL优化器。
- 当分区是外部表时，一些访问或者修改外部表数据的操作将会报错，例如：
  - INSERT、DELETE和UPDATE命令试图修改外部表分区的数据会报错。
  - TRUNCATE命令涉及外部表分区时会报错。
  - 当COPY命令向分区表COPY数据且需要修改外部表时会报错。
  - 用COPY命令从一个包含外部表的分区表COPY数据是不允许的，如果使用IGNORE EXTERNAL PARTITIONS语法是可以的，这样可以过滤掉外部表叶子分区。可以通过COPY一个SQL查询的方式，从一个包含外部表的分区表COPY数据，例如这个例子，将数据COPY到标准输出：

```
=# COPY (SELECT * from my_sales ) TO stdout;
```

  - VACUUM命令会忽略外部表分区。
- 下面的操作，如果没有数据被修改，是允许的，否则将会报错：
  - 增加或删除字段。
  - 修改字段类型。
- 如果分区表包含外部表，一些ALTER PARTITION操作也是不支持的：
  - 设置子分区模板。
  - 修改分区属性。
  - 创建缺省分区。
  - 设置分布策略。
  - 修改字段NOT NULL约束。
  - 增加或删除约束。
  - 分割外部表分区。
- 如果分区表的叶子分区是外部表，备份工具将不会备份该分区的数据。

插入数据到分区表

一旦创建了分区表，所有的非叶子分区，永远是没有数据的。数据只储存在最底层的表中(叶子分区)。在多级分区表中，仅仅在最底层的叶子分区中有数据。

如果有记录无法匹配到叶子分区，该数据将会被拒绝并导致插入失败。若希望在任何时候插入时都不出现失败，可以增加默认分区。这样所有不能匹配分区CHECK约束的数据将插入到默认分区。可参考相关的"添加默认分区"章节。

在生成执行计划时，优化器会根据查询条件的情况，检查分区表的CHECK约束来决定有哪些分区需要被扫描。对于PostgreSQL优化器来说，该分区表中的所有默认分区(只要该层级中存在)总是会被扫描，如果默认分区中包含数据，其一定会影响处理时间。对于Orca优化器来说，如果查询条件不涉及默认分区，则不会扫描默认分区，如果分区条件不是常量，Orca还会进行动态分区裁剪。

在使用COPY或者INSERT向ROOT表装载数据时，这些数据会默认自动路由到正确的叶子分区。因此，可以像使用普通的未分区表一样插入数据到分区表。

如果有必要，可以直接把数据装载到叶子分区中，实际上很多工具都需要这样做，例如，备份恢复操作，集群之间数据同步操作等，因为把整个分区表作为一个整体来操作可能是一个特别巨大的任务，按照不同的叶子分区来操作，可以将巨大的任务分拆为多个较小的任务，还可以通过叶子分区之间的并发来加速。对于需要直接插入数据到叶子分区的情况，在业务实现中，还可以先创建一个临时的表、插入数据、然后与相应的分区表进行分区交换，假如分区表上有索引，直接插入数据的性能会受到影响，这种分区交换的方式，可以在数据准备结束之后再创建索引，整个数据处理过程对分区表没有任何影响，总体性能高于直接的COPY和INSERT。参考相关的"交换分区"章节。

验证分区策略

表分区的目的是减少查询的数据扫描量。若一张表基于相应的查询条件做分区，可以使用EXPLAIN查看执行计划来验证是否只扫描了相关的分区而不是扫描全表。

例如，有一张表sales已经根据日期按月做了范围分区，同时按区域做了子分区，参见"定义多级分区表"章节的例子。对于下面的查询：

```
=# EXPLAIN SELECT * FROM sales WHERE date='2020-01-07' AND region='usa';
```

下图是PostgreSQL优化器生成的执行计划：

```
Gather Motion 2:1  (slice1; segments: 2)  (cost=0.00..2734.00 rows=5 width=54)
  ->  Append  (cost=0.00..2734.00 rows=3 width=54)
        ->  Seq Scan on sales_1_prt_outlying_dates_2_prt_usa  (cost=0.00..683.50 rows=1 width=54)
              Filter: ((date = '2020-01-07'::date) AND (region = 'usa'::text))
        ->  Seq Scan on sales_1_prt_outlying_dates_2_prt_other_regions  (cost=0.00..683.50 rows=1 width=54)
              Filter: ((date = '2020-01-07'::date) AND (region = 'usa'::text))
        ->  Seq Scan on sales_1_prt_2_2_prt_usa  (cost=0.00..683.50 rows=1 width=54)
              Filter: ((date = '2020-01-07'::date) AND (region = 'usa'::text))
        ->  Seq Scan on sales_1_prt_2_2_prt_other_regions  (cost=0.00..683.50 rows=1 width=54)
              Filter: ((date = '2020-01-07'::date) AND (region = 'usa'::text))
Optimizer: Postgres query optimizer
```

下图是Orca生成的执行计划

```
Gather Motion 2:1  (slice1; segments: 2)  (cost=0.00..431.00 rows=1 width=24)
  ->  Sequence  (cost=0.00..431.00 rows=1 width=24)
        ->  Partition Selector for sales (dynamic scan id: 1)  (cost=10.00..100.00 rows=50 width=4)
              Partitions selected: 1 (out of 52)
        ->  Dynamic Seq Scan on sales (dynamic scan id: 1)  (cost=0.00..431.00 rows=1 width=24)
              Filter: ((date = '2020-01-07'::date) AND (region = 'usa'::text))
Optimizer: Pivotal Optimizer (GPORCA)
```

从以上的的两个执行计划可以发现，不管是PostgreSQL优化器生成的执行计划，

还是Orca生成的执行计划，都只选择了部分分区，不过，Orca生成的执行计划是不同的，因为其分区选择只选择了一个分区，分区表中是有默认分区的，Orca认为，默认分区中存储的数据与其他普通的分区不应该有重叠，当然，在正常的插入数据时也是这样检查的。经过编者测试，直接插入非法数据到默认分区是会报错的，只能通过设置gp_enable_exchange_default_partition参数为on然后交换默认分区WITHOUT VALIDATION的形式完成，所以，如果要强制交换分区，确保数据的正确性很重要，后面还会提到。

对于这个执行计划应该只显示对下列表的扫描：

- 默认分区返回0-1条数据(可能不止一个默认分区，PostgreSQL优化器一定会扫描默认分区，Orca根据条件判断，不一定会扫描默认分区)
- 符合日期的USA地区叶子分区(sales_1_2_prt_usa)返回一些记录

从PostgreSQL优化器生成的执行计划发现，有3个默认分区被扫描了，分别是第一级一月份分区的其他地区子分区，第一级其他时间分区的usa子分区，第一级其他时间分区的其他地区子分区。而Orca只扫描了条件完全匹配的一个分区。

注意：要确保执行计划只扫描了必要的叶子分区。

分区选择性的诊断

如果执行计划显示没有出现分区过滤，而是扫描全表，可能和以下的限制有关：

- 执行计划仅可以对稳定(函数根据易失性可以分为VOLATILE、IMMUTABLE和STABLE，IMMUTABLE是事务稳定的，STABLE是永远稳定的，这些概念可以参考PG的文档，在函数章节也会继续介绍)的比较运算符执行选择性扫描，如：

```
=
<
<=
>
>=
<>
```

- 执行计划不识别非稳定函数来执行选择性扫描。例如，WHERE子句中使用如date > CURRENT_DATE会使得执行计划有选择性的只扫描部分叶子分区，而time > TIMEOFDAY不会。这里涉及的是函数的易失性问题，VOLATILE类型的函数，对于完全相同的输入参数，不能保证输出的结果不变，例如random()函数，在创建函数时如果没有明确指定，就是VOLATILE的。IMMUTABLE类型的函数，在整个事务范围内，相同的输入参数，一定会得到相同的输出结果，例如now()函数。STABLE类型的函数，任何时候，输入的参数相同，输出的结果就相同，例如sin()

函数。如果分区字段匹配的条件是VOLATILE类型函数，那么将无法替换为常量值，因为这类函数的输出值随时可能发生改变，而IMMUTABLE和STABLE类型的函数，可以在事务开始时计算出一个结果并一直使用，这样就可以进行分区过滤了，所以，对于那些可能用于分区过滤的函数，请注意其VOLATILE属性。

查看分区设计

要查看分区表的设计情况，可以通过查询pg_partitions视图来查看。例如，查看sales表的分区情况：

```
=# SELECT partitionboundary, partitiontablename, partitionname,
partitionlevel, partitionrank
FROM pg_partitions
    WHERE tablename='sales';
```

另外还有如下系统表和视图可以查看分区表的信息：

- pg_partition -- 记录分区表的层级关系。
- pg_partition_rule -- 记录各个层级的分区的规则和隶属关系。
- pg_partition_templates -- 展示SUBPARTITION TEMPLATE信息的视图，实际上，SUBPARTITION TEMPLATE信息也是存储在pg_partition和pg_partition_rule系统表中的，通过paristemplate字段来标识。
- pg_partition_columns -- 分区表的分区字段信息。

关于这些系统表和视图的更详细的信息，可以参照相关章节。

维护分区表

必须使用ALTER TABLE命令从ROOT表来维护分区。最常见的场景是根据日期范围维护数据，例如，删除旧的分区并添加新的分区。还有一种可能就是把旧的分区转换为压缩AO表以节省空间。若在父表中存在默认分区，添加分区的操作只能是从默认分区拆分出一个新的分区。

- 添加新分区
- 修改分区名称
- 添加默认分区
- 删除分区
- 清空分区数据
- 交换分区
- 拆分分区

- 修改子分区模版
- 交换叶子分区为外部表

重要提示：在定义和更改分区时，使用分区名称而不是分区TABLE的relation name。虽然可以在CREATE分区表时通过WITH子句中的tablename属性的方式为分区指定个性化的relation name，但是建议永远不要这样做，这是一个违反规范的做法(也许可以作为一个考题，例如，如何创建一个10级分区表，因为按照缺省的分区命名规则，到8级分区时就会出现表名重复的报错)。虽然可以使用SQL命令直接针对分区表进行查询和装载操作，但只能通过ALTER TABLE . . . PARTITION的方式来修改分区。

从语法限制的角度来说，分区不强制要求定义名称，若分区没有名称，下面的两种表达式仍可用于选择一个分区：

```
PARTITION FOR (value);
PARTITION FOR(RANK(number));
```

注意：编者建议，在真实的生产中，务必为每个分区指定一个名称，这样对分区表的维护等都会带来便利。永远不要使用FOR(RANK(number))的方式来选择分区，例如，在删除分区时，rank是会不断发生变化的，这种方式很容易导致删错分区。

添加新分区

可以使用ALTER TABLE命令在现有的分区表上添加新分区。如果现有的分区表包含了SUBPARTITION TEMPLATE定义，新增的分区将根据该模版创建子分区。例如：

```
=# CREATE TABLE sales (
    trans_id int,
    date date,
    amount
    decimal(9,2),
    region text
) DISTRIBUTED BY (trans_id)
PARTITION BY RANGE (date)
SUBPARTITION BY LIST (region)
SUBPARTITION TEMPLATE (
    SUBPARTITION usa VALUES ('usa'),
    SUBPARTITION asia VALUES ('asia'),
    SUBPARTITION europe VALUES ('europe')
)
(
```

```
    START (date '2020-01-01') INCLUSIVE
    END (date '2021-01-01') EXCLUSIVE
    EVERY (INTERVAL '1 month')
);
=# ALTER TABLE sales ADD PARTITION p202101
    START (date '2021-01-01') INCLUSIVE END (date '2021-02-01') EXCLUSIVE;
```

如果在创建TABLE时没有SUBPARTITION TEMPLATE，在新增分区时需要定义子分区：

```
=# CREATE TABLE sales (
    trans_id int,
    date date,
    amount
    decimal(9,2),
    region text
) DISTRIBUTED BY (trans_id)
PARTITION BY RANGE (date)
SUBPARTITION BY LIST (region)
(
    START (date '2020-01-01') INCLUSIVE
    END (date '2021-01-01') EXCLUSIVE
    EVERY (INTERVAL '1 month')
    (
        SUBPARTITION usa VALUES ('usa'),
        SUBPARTITION asia VALUES ('asia'),
        SUBPARTITION europe VALUES ('europe')
    )
);
=# ALTER TABLE sales ADD PARTITION p202101
    START (date '2021-01-01') INCLUSIVE END (date '2021-02-01') EXCLUSIVE
(
    SUBPARTITION usa VALUES ('usa'),
    SUBPARTITION asia VALUES ('asia'),
    SUBPARTITION europe VALUES ('europe')
);
```

如果要在现有分区上添加子分区，可以指定分区执行ALTER命令。例如：

```
=# ALTER TABLE sales ALTER PARTITION FOR ('2020-12-01')
    ADD PARTITION africa VALUES ('africa');
```

注意：这种方式会导致该多级分区表变的不规整，Orca无法处理这种多级分区表。

注意：若在有默认分区的分区表中添加新的分区，只能从默认分区拆分出一个新的分区。

参见"拆分分区"相关章节。编者建议，尽可能避免添加默认分区，因为维护更困难。另外，在真实的生产中，所有的新增分区都要明确指定分区名称，因为不指定的情况下，数据库将会自动产生一个随机的名称，当需要查找分区表的relation name或者在不同的集群之间比对分区结构时，将是很大麻烦。

修改分区名称

子表的名称同样是受唯一性约束和长度限制的，GP中对象长度限制为63个字符，pg_class系统表的唯一索引pg_class_relname_nsp_index限制了名称不能重复。若名称超过长度限制，表名的超长部分会被截断，如果截断之后的relation name存在重复就会报表已存在的错误。子表的表名格式如下：

```
<父表名称>_<分区层级>_prt_<分区名称>
```

例如：

```
sales_1_prt_jan20
```

对于未指定分区名称而自动产生的范围分区表来说可能是这样的：

```
sales_1_prt_1
```

子表的relation name不能通过直接执行ALTER表名来实现。但修改ROOT表的relation name时，该修改将会影响所有相关的分区表。例如：

```
=# ALTER TABLE sales RENAME TO globalsales;
```

该操作将会把相关的分区表名称改为如下的格式：

```
globalsales_1_prt_1
```

只修改分区名称的操作如下所示：

```
=# ALTER TABLE sales RENAME PARTITION FOR ('2020-01-01') TO jan20;
```

该操作将会把相关分区表的表名改为：

```
sales_1_prt_jan20
```

注意：在使用ALTER TABLE命令修改分区时，使用的是分区名称(如jan8)，而不是

分区表的relation name(如sales_1_prt_jan08)。对于使用FOR (value)指定分区的方式，GP是使用FOR指定的value来匹配分区表的CHECK约束，换言之，只要FOR()的条件在分区条件内即可匹配。

添加默认分区

可以使用ALTER TABLE命令为现有分区表添加默认分区(当有多级子分区且没有SUBPARTITION TEMPLATE时，添加默认分区也需要定义子分区)：

```
=# ALTER TABLE sales ADD DEFAULT PARTITION other;
```

如果是多级分区表，同一层次中的每个分区都需要一个默认分区。例如：

```
=# ALTER TABLE sales ALTER PARTITION FOR (RANK(1))
   ADD DEFAULT PARTITION other;
=# ALTER TABLE sales ALTER PARTITION FOR (RANK(2))
   ADD DEFAULT PARTITION other;
=# ALTER TABLE sales ALTER PARTITION FOR (RANK(3))
   ADD DEFAULT PARTITION other;
```

RANK指的是范围分区同一层级中的顺序，在不涉及rank可能会发生变化的操作时不会造成误操作。可参见pg_partition_rule表的parruleord字段。若分区表没有默认分区，无法匹配到叶子分区CHECK约束的新增记录将被拒绝，如果有默认分区，无法匹配到叶子分区CHECK约束的新增记录将进入默认分区。

删除分区

可以使用ALTER TABLE命令删除分区表中的分区。如果被删除的分区有子分区，其所有的子分区(包括所有的数据)会一起被删除。对于范围分区的表来说，在滚动数据时，通常是删除最老的数据。例如：

```
=# ALTER TABLE sales DROP PARTITION FOR (RANK(1));
=# ALTER TABLE sales DROP PARTITION FOR ('2020-01-01');
```

注意：在将RANK(1)的分区删除后，其余分区的rank值仍然是从1开始的连续编号。编号的顺序按照分区字段的值由小到大从1开始排序，不管分区是否连续(所有的

START和END之间没有空缺即为连续)，所以，在真实的项目场景中，要绝对杜绝这种删除分区的方式，要使用partition name或者value的方式来匹配分区。

清空分区数据

可以使用ALTER TABLE命令来清空分区。在清空一个包含子分区的分区时，其所有相关子分区的数据都自动被清空。命令如下：

```
ALTER TABLE sales TRUNCATE PARTITION FOR (RANK(1));
ALTER TABLE sales DROP PARTITION FOR ('2020-01-01');
```

注意：建议采用partition name或者value的方式来匹配分区。

交换分区

交换分区是用一个普通的TABLE与现有的分区交换身份。使用ALTER TABLE命令来交换分区。另外只能交换叶子分区(多级分区中的非叶子分区不可以被交换)。交换分区对于数据加载是有帮助的。例如，先将数据装载到一个临时的表中，然后与目标分区进行交换，例如分区表上有索引，则可以在临时的表上重建好索引之后再进行交换，将对分区表的影响降到最低，这种模式在索引查询场景有着广泛的应用。还可以使用交换分区将旧的分区储存为AO表。

交换分区，不可以与复制表交换(DISTRIBUTED REPLICATED)，也不可以与其他分区表或者分区表的子表进行交换，只能与普通表进行交换。

例如：

```
=# CREATE TABLE sales (
   id int,
   date date,
   amt decimal(10,2)
)DISTRIBUTED BY (id) PARTITION BY RANGE (date) (
   START (date '2020-01-01') INCLUSIVE
   END (date '2021-01-01') EXCLUSIVE EVERY (INTERVAL '1 month')
);
=# BEGIN;
```

```
=# CREATE TABLE jan20y (LIKE sales) WITH (appendonly=true);
=# INSERT INTO jan20y SELECT * FROM sales_1_prt_1 ;
=# ALTER TABLE sales EXCHANGE PARTITION FOR (DATE '2020-01-01')
   WITH TABLE jan20y;
=# DROP TABLE jan20y;
=# END;
```

注意：该例子涉及的是一级分区表sales。编者认为能使用一级分区的情况下，最好不要选择多级分区这种复杂的结构，避免维护管理时不必要的麻烦。

警告：如果使用了WITHOUT VALIDATION语法进行分区交换，将不会检查数据的合法性，必须自行确保数据符合分区的约束检查，否则可能会在查询时得到错误的结果。

警告：GP的参数gp_enable_exchange_default_partition用以控制是否可以使用EXCHANGE DEFAULT PARTITION语法来交换默认分区，这个参数的缺省值是off，也就是说，缺省情况下，尝试交换默认分区是会失败报错的。正如之前所说，如果使用WITHOUT VALIDATION语法强制交换默认分区，而该分区中存在本应该存储在其他叶子分区的数据，这可能会导致Orca的查询得到错误的结果。

拆分分区

拆分分区是将现有的一个分区分成两个分区。使用ALTER TABLE命令来拆分分区。只能拆分叶子分区(多级分区中的非叶子分区不可以被拆分)，这也是建议不要使用多级分区的一个因素，维护多级分区将是极大的麻烦。指定的分割值对应的数据将进入后面一个分区(就是START为INCLUSIVE)。

例如，将一月份的分区拆分成一个1-15日的分区和另一个16-31日的分区：

```
=# CREATE TABLE sales (
   id int,
   date date,
   amt decimal(10,2)
)DISTRIBUTED BY (id) PARTITION BY RANGE (date) (
   START (date '2020-01-01') INCLUSIVE
   END (date '2021-01-01') EXCLUSIVE EVERY (INTERVAL '1 month')
);
=# ALTER TABLE sales SPLIT PARTITION FOR ('2020-01-01')
AT ('2020-01-16') INTO (PARTITION jan20y01to15, PARTITION jan20y16to31);
```

如果分区表有默认分区，要添加新的分区只能从默认分区拆分。而且只能从叶子分

区的默认分区拆分(多级分区中的非叶子分区不可以被拆分)。在使用INTO子句时，第2个分区名称必须是已经存在的默认分区。例如，从默认分区中拆分出一个新的月份分区：

```
=# CREATE TABLE sales (
    id int,
    date date,
    amt decimal(10,2)
)DISTRIBUTED BY (id) PARTITION BY RANGE (date) (
    START (date '2020-01-01') INCLUSIVE
    END (date '2021-01-01') EXCLUSIVE EVERY (INTERVAL '1 month'),
    DEFAULT PARTITION other
);
=# ALTER TABLE sales SPLIT DEFAULT PARTITION
    START ('2021-01-01') INCLUSIVE END ('2021-02-01') EXCLUSIVE
    INTO (PARTITION jan21, default partition);
```

注意：如何在有默认分区的多级分区表上增加新的分区？SPLIT也是不行的，因为SPLIT只能针对叶子分区，所以，有默认分区的情况下，不能直接添多级分区表的一级子分区，只能先删掉一级分区的默认分区，添加新的分区之后再重新添加默认分区。再次重复之前的观点，不要使用多级分区，除了会带来麻烦，可能什么好处也不会带来。同时，最好也不要设置默认分区，不要图一时方便，遗留万分苦难。

修改子分区模版

使用ALTER TABLE SET SUBPARTITION TEMPLATE命令来修改现有分区表的子分区模版。在修改了子分区模版之后添加的分区，其子分区将按照新的模版产生。已经存在的分区不会被修改。例如：

```
=# CREATE TABLE sales (
    trans_id int,
    date date,
    amount
    decimal(9,2),
    region text
) DISTRIBUTED BY (trans_id)
PARTITION BY RANGE (date)
SUBPARTITION BY LIST (region)
SUBPARTITION TEMPLATE (
    SUBPARTITION usa VALUES ('usa'),
```

```
    SUBPARTITION asia VALUES ('asia'),
    SUBPARTITION europe VALUES ('europe'),
    DEFAULT SUBPARTITION other_regions)
(
    START (date '2020-01-01') INCLUSIVE
    END (date '2021-01-01') EXCLUSIVE
    EVERY (INTERVAL '1 month')
);
=# ALTER TABLE sales SET SUBPARTITION TEMPLATE (
    SUBPARTITION usa VALUES ('usa'),
    SUBPARTITION asia VALUES ('asia'),
    SUBPARTITION europe VALUES ('europe'),
    SUBPARTITION africa VALUES ('africa'),
    DEFAULT SUBPARTITION other
);
```

当有更多级的子分区模版时，可以通过下面的语法形式来逐级修改子分区模版：

```
=# ALTER TABLE sales ALTER PARTITION FOR (RANK(1)) ALTER PARTITION
    FOR (RANK(1)) SET SUBPARTITION TEMPLATE
=# ALTER TABLE sales ALTER PARTITION FOR (RANK(1)) SET SUBPARTITION TEMPLATE
=# ALTER TABLE sales SET SUBPARTITION TEMPLATE
```

在使用新的模版后为表sales新增一个分区时，其将包含Africa地区的子分区，下面的命令将创建子分区usa、asia、europe、africa和默认分区other：

```
=# ALTER TABLE sales ADD PARTITION jan21
    START ('2021-01-01') INCLUSIVE END ('2021-02-01') EXCLUSIVE;
```

注意：这个例子在一级分区有默认分区时是不能执行的。要删除子分区模版，使用SET SUBPARTITION TEMPLATE并使用空的参数来完成。例如，将上面例子中的sales表的子分区模版清空：

```
=# ALTER TABLE sales SET SUBPARTITION TEMPLATE ();
```

交换叶子分区为外部表

不记得具体是从什么版本(大约是 4.3 的某个版本吧)开始，GP 允许将分区表的叶子分区和可读外部表进行交换，而外部表的数据可以存储在数据库之外的地方，可能是

文件服务器，NFS，或者 HDFS 等，这样做显然会有不少限制，所以，如果确定没有使用的必要，可以跳过该部分内容。

举个例子来说，假如有一张分区表，按照月份来分区，而绝大部分的查询只针对最近几个月的数据来查询，这样就可以通过外部表分区的方式将早期数据以半离线的状态存储到 GP 集群之外更廉价的存储空间。当对该表进行查询时，通过分区条件对分区进行过滤，这样可以避免扫描外部表的分区，而当一些查询需要用到外部表分区的数据时，数据将被从外部存储读取，其性能跟库内的分区相比会有很大的差异，但数据是在线可查的，这就需要平衡性能与成本。

如果分区表包含 CHECK 约束或者 NOT NULL 约束的字段，将不能跟外部表进行交换分区。

关于包含外部表的分区表的限制，请参考”分区表的限制”章节。

外部表与分区表交换示例

这个例子将展示，如何将一张基于时间的分区表的历史数据转移到数据库之外并变成一个外部表的分区。我们将创建一个 2015 年到 2020 年按年分区的表，之后将 2015 年到 2018 年的分区转存到外部表中。首先，我们创建一张分区表：

```
=# CREATE TABLE sales (
    id int,
    year int,
    qtr int,
    day int,
    region text
)DISTRIBUTED BY (id)
PARTITION BY RANGE (year) (
    PARTITION yr START (2015) END (2020) EVERY (1)
);
```

我们创建了一个含有 5 个分区的分区表，然后我们向表中插入 5 条数据，每个年份的分区中有一条数据：

```
=# INSERT INTO sales
SELECT generate_series(1,5),
       generate_series(2015,2019),
       generate_series(5,9),
       generate_series(10,14),
```

```
        'region'||generate_series(1,5);
```

接下来我们将 2015 年到 2018 年的数据存到外部表中。

1、 要确保外部表用到的协议在 GP 数据库中已经具备。

目前的例子，我们选择用 gpfdist 协议，例如，我们在 192.168.88.66 机器上启动 gpfdist 服务。

```
$ nohup gpfdist -p 8080 -d /../ >/dev/null 2>&1 &
```

此处说明一下，gpfdist 不允许直接在/目录上启动服务，因此可以通过/../的方式间接的从/上启动服务。

2、 创建一张可写外部表,表结构与需要交换分区的分区表保持一致(使用 LIKE 语法)：

```
=# CREATE WRITABLE EXTERNAL TABLE my_sales_ext (LIKE sales)
LOCATION ('gpfdist://192.168.88.66/data/sales_2010_2018')
FORMAT 'TEXT'
DISTRIBUTED BY (id);
```

3、 将数据从分区表导出到可写外部表,根据需要交换的数据范围指定条件进行数据过滤：

```
=# INSERT INTO my_sales_ext SELECT * FROM sales
   WHERE year >= 2015 AND year <= 2018;
```

注意：在导出数据之前，要先确保文件目录已经存在，gpfdist 不会自动创建目录，还要确保目标文件不存在或者为空，可写外部表不会清空目标文件。

4、 创建可读外部表，数据从刚刚建立的可写外部表写出的文件获取：

```
=# CREATE EXTERNAL TABLE my_sales_ext_prt (LIKE sales)
LOCATION ('gpfdist://192.168.88.66/data/sales_2010_2018')
FORMAT 'TEXT';
```

5、 因为需要交换的分区有多个，因此，需要删除外部表中需要交换的分区，再重新建立一个跨度更大的分区：

```
=# ALTER TABLE sales DROP PARTITION FOR(2015);
=# ALTER TABLE sales DROP PARTITION FOR(2016);
=# ALTER TABLE sales DROP PARTITION FOR(2017);
=# ALTER TABLE sales DROP PARTITION FOR(2018);
=# ALTER TABLE sales ADD PARTITION p2015_2018 START (2015) END (2019);
```

6、 交换分区和外部表：

```
=# ALTER TABLE sales EXCHANGE PARTITION FOR (2015)
   WITH TABLE my_sales_ext_prt WITHOUT VALIDATION;
```

注意：要确保查询的结果正确，外部表中的数据必须符合被交换的叶子分区的约束检查。

7、 删除交换出来的分区表，删除可写外部表：

```
=# DROP TABLE my_sales_ext_prt;
=# DROP EXTERNAL TABLE my_sales_ext;
```

可以将外部表的叶子分区修改分区名，使得名字中包含 ext，以助于识别该叶子分区是一个外部表分区，例如：

```
=# ALTER TABLE sales RENAME PARTITION p2015_2018
   TO p2015_2018_ext;
```

创建与使用序列

GP数据库中的序列，实质上是一种特殊的单行记录的表，用以生成自增长的数字，可用于为表的记录生成自自增长的标识。不过，不要以为使用serial类型或者bigserial类型就可以避免创建序列了，其实这两种类型会自动创建序列。所以，还不如明确的创建序列，可能还可以优化一下性能。

GP提供了创建、修改、删除序列的命令，还提供了内置的函数用于获取序列的下一个值(nextval())，重新设置序列的初始值(setval())。

注意：PostgreSQL的currval()函数lastval()在GP中是不支持的。不过，可以通过直接查询序列这个表来获取。例如：

```
=# SELECT last_value,start_value FROM myserial;
```

序列对象包括几个属性，例如，名称，步长(每次增长的量)，最小值，最大值，缓存大小等，还有一个布尔属性：is_called，该属性的含义是，nextval()先返回值还是序列的值先增长，例如序列当前的值为100，如果is_called为TRUE，则下一次调用nextval()时返回的是101，如果is_called为FALSE，则下一次调用nextval()时返回的是100，编者认为，这个属性差异不大，不必在意。

创建序列

使用CREATE SEQUENCE命令来创建并初始化一个给定名称的序列。序列的名字在Schema之下必须与其他的对象如SEQUENCE、TABLE、INDEX或VIEW都不同，因为这些对象的名称信息都是存储在pg_class系统表中的。例如：

```
=# CREATE SEQUENCE myserial START 101;
```

在创建序列时，GP 会将 is_called 设置为 FALSE，第一次调用 nextval()函数访问新创建的序列时，序列的值不增加，并将 is_called 设置为 TRUE。例如刚刚创建的序列,其 last_value 为 101,is_called 为 FALSE,第一次调用 nextval()获得的是 101，last_value 属性不变，之后，is_called 变为 TRUE，再次调用 nextval()获得的是 102，last_value 增长为 102。

使用序列

使用CREATE SEQUENCE命令创建好序列之后，就可以使用nextval函数来获取序列的值了。例如，获取序列的下一个值并插入表中：

```
=# INSERT INTO vendors VALUES (nextval('myserial'), 'acme');
```

nextval()根据序列的is_called属性来决定是否在返回数值之前先增加计数器，如果is_called为TRUE，则先增加计数器，然后返回计数器的值，如果is_called为FALSE，则先将is_called改为TRUE，然后返回计数器的值。

nextval()函数是不回滚的。只要被调用就被认为返回的值已经被使用，即便是事务在nextval()之后失败或者被回滚。这就意味着中断事务会使得有空缺的序列没有被真正的使用。同样的setval函数也是不回滚的。

注意：如果启用的Mirror镜像，那么，在UPDATE和DELETE语句中不能使用nextval()函数。

可以使用setval()函数重置一个序列计数器的值。例如：

```
=# SELECT setval('myserial', 201);
```

setval()函数有两种参数形式，setval(sequence, start_val)和setval(sequence, start_val, is_called)。setval(sequence, start_val)等同于setval(sequence, start_val, TRUE),即,设置is_called的属性为TRUE，如果不希望第一次调用setval的时候计数器增长，可以指定is_called为FALSE，编者想说，这几乎毫无意义。

setval()函数同样永远不会回滚，就是说，即便ROLLBACK了，已经修改的值不会变，等同于COMMIT了。

要查看序列当前的所有属性，可以直接查询序列对象：

```
=# SELECT * FROM myserial;
```

修改序列

使用ALTER SEQUENCE命令修改已有的序列表的属性，例如START的值，最小值，最大值，步长等，也可以设置序列从指定的值重新开始。没有在ALTER SEQUENCE命令中指定的属性值将保持不变。

ALTER SEQUENCE sequence START WITH start_value语句设置序列的START为一个新的值，但对last_value属性没有影响，nextval()函数的返回值也不受影响。

ALTER SEQUENCE sequence RESTART语句设置序列的last_value属性重新从start_value属性的值开始，同时，is_called被设置为FALSE，nextval()函数将返回start_value属性的值，序列如同新建。

ALTER SEQUENCE sequence RESTART WITH restart_value语句设置序列的last_value属性为restart_value的值，同时，is_called被设置为FALSE，nextval()函数将返回restart_value的值。效果等同于执行了函数SELECT setval('myserial', restart_value, FALSE);

下面的命令是设置myserial序列重新从105开始：

```
=# ALTER SEQUENCE myserial RESTART WITH 105;
```

删除序列

使用DROP SEQUENCE命令删除已有的序列表。例如：

```
=# DROP SEQUENCE myserial;
```

如果还有表使用了序列，序列将无法被删除，可以使用 CASCADE 进行级联删除。

设置序列为字段缺省值

除了在CREATE TABLE时使用SERIAL或者BIGSERIAL类型，可以明确的使用序列来实现自增字段。使用SERIAL或者BIGSERIAL类型会自动创建序列。例如：

```
=# CREATE TABLE tablename (
    id INT4 DEFAULT nextval('myserial'),
    name text
);
```

还可以在创建了表之后，通过ALTER TABLE命令设置字段的缺省值为序列：

```
=# ALTER TABLE tablename ALTER COLUMN id SET DEFAULT nextval('myserial');
```

序列回旋

缺省情况下,序列是不允许回旋的(就是last_value达到了max_value的时候，重新从 start_value 开始)，也就是说，当序列的值达到最大值时，netvalue()函数将会报错：

```
ERROR:  nextval: reached maximum value of sequence . . .
```

虽然说有 3 种 SERIAL 类型，SMALLSERIAL、SERIAL 和 BIGSERIAL，但是，序列的属性max_value是BIGINT类型的,所以,如果使用序列的字段类型比BIGINT小的话，没等序列出现上述报错，字段就会报 out of range 的错误。BIGINT 的最大值大约为 922 亿亿，正常的使用可能永远也不会达到。

可以设置序列允许回旋：

```
=# ALTER SEQUENCE myserial CYCLE;
```

也可以在创建序列的时候设置允许回旋：

```
=# CREATE SEQUENCE myserial CYCLE;
```

在GP中使用索引

在大多数的OLTP数据库中，索引可以显著的改善数据访问的性能。然而在分布式数据库(例如GP)中，应该谨慎使用索引。GP执行顺序扫描已经很快，而索引是通过随机寻址在磁盘上定位数据记录，两者适用场景不同。与传统的OLTP数据库不同的是，GP中数据是分布在多个Instance上的。这意味着每个Instance都扫描全部数据的一小部分来查找结果。如果使用了分区表，扫描的数据可能会更少。通常，商业智能(BI)的查询需要返回大量的数据，这种情况下使用索引未必有效。

GP建议在没有添加索引的情况下先测试一下查询的性能。索引更易于改善OLTP类型查询的性能，一般，索引查询期望返回很少量的数据。在返回少量结果的场景下，索引同样可以改善压缩AO表上查询的性能，当情况合适时优化器会把索引作为获取数据的选择，而不是一味的全表扫描。对于压缩数据来说，索引访问数据时只解压需要的记录而不是全表解压(最小单元的压缩块是要解压的)。

值得注意的是，GP会自动为主键字段创建主键索引。在分区表的ROOT表上建立索引会自动在其相关的子表上也建立索引，分区索引的命名规则与分区表的命名规则类似，但是，修改ROOT表的索引名称不会自动修改子分区的索引名称，这与表名的修改不同。

添加索引会带来一些资源开销 -- 其必定占用相当的存储空间，在更新数据时的索引维护也需要消耗计算资源。需确保索引的创建在查询中真正被使用到。同时，需要检查索引的确对于查询性能有显著的改善(与顺序扫描的性能相比)。可以使用EXPLAIN查看执行计划来确认是否使用了索引。参考相关"查询剖析"章节。

在创建索引时需要综合考虑以下因素：

- **查询的类型。**索引有助于改善OLTP型的查询，其返回很少量的数据。对于返回数据比例较大的查询，使用索引不会带来性能的改善。

- **压缩表。**在返回少量结果的情况下，索引同样可以改善压缩AO表的查询性能。对于压缩数据来说，索引访问数据的时只解压需要的记录而不是全表解压。

- **避免在频繁更新的字段上使用索引。**在频繁更新的字段上创建索引，当该字段被更新时，需要消耗大量的写盘操作，对IOPS能力的要求很高。

- **如何选择B-tree索引。**在考虑索引时，数据的唯一性指数(编者认为这样说更容易理解)，是个重要的指标，唯一性指数，是字段中DISTINCT值的数量除以表中的总记录数。例如，如果一张表中有1000条记录，某个字段有800个DISTINCT值，该字段的唯一性指数为0.8，唯一性很高。唯一索引总是具备1.0的唯一性指数，不能更高了，因为所有的值都互不相同。值得注意的是在GP中唯一索引必须包含所有的DK字段。唯一性指数高的字段更适合使用B-tree索引。

- **如何选择Bitmap索引。**PostgreSQL不支持GP中的Bitmap索引，唯一性指数很

低的字段可能更适合Bitmap索引。参照"关于位图索引"章节。

- **索引字段用于关联查询。**在经常关联查询的字段上建立索引或许可以改善关联查询的性能，因为其可以帮助优化器使用其他的关联方法。这可能会涉及到Orca的optimizer_enable_indexjoin参数或者PostgreSQL优化器的enable_nestloop参数。

- **索引字段经常用在查询条件中。**对于大表来说，查询语句WHERE条件中经常用到的字段上，才适合考虑创建索引，因为查询用得到，但不是说WHERE条件中经常用到的字段，就需要建索引，这又是一种逻辑颠倒的理解，还需要综合考虑其他因素。

- **避免索引重叠。**在一个或多个顺序相同的字段上创建多个索引是多于的。不是说不能将一个字段用于多个索引中，如果的确有必要也是可以的，但应该避免功能相似的重复索引。

- **批量数据加载前删除索引。**对于表中已有大量数据，需要批量加载数据的情况，应该考虑先删除索引，加载数据之后再重新创建索引，这样可能会比带索引加载更快。索引的维护代价是很高的，所以，有时可以通过分区交换的方式提前把数据和索引准备好，尽可能将对目标表的影响降到最低，可参见"交换分区"章节。

- **聚集索引。**聚集索引的意思是，表中的数据记录按照索引字段在磁盘上排序存储。如果需要查询的数据在磁盘上的存储是无序的，数据库需要在磁盘文件上进行离散扫描来获取，如果数据是有序的，数据库可以在连续的磁盘存储上获取数据，所以对聚集索引字段的单条件查询的性能会更高效。

在 GP 中使用聚集索引

对于大表来说，使用CLUSTER(该命令只可以作用于Heap表)命令来排序物理记录以创建聚集索引可能需要耗费极长的时间。要快速达到同样的效果，可以通过创建一张中间表的方式来手动排序数据，由于CLUSTER命令只能用于Heap表，对于AO表，要达到聚集索引的效果，也只能通过数据排序插入的方式实现。例如：

```
=# CREATE TABLE new_table (LIKE old_table) AS
   SELECT * FROM old_table ORDER BY myixcolumn;
=# CREATE INDEX myixcolumn_ix ON new_table;
=# ANALYZE new_table;
=# DROP old_table;
=# ALTER TABLE new_table RENAME TO old_table;
```

注意：从语法上来说，GP不支持CREATE CLUSTER INDEX语法，因此，上述方法是常被用于实现类似聚集索引的方法，除了重建表，对于分区表，还可以通过交换分区的

方式来实现对分区的聚集索引效果。

索引类型

GP支持的PostgreSQL索引类型包括：B-tree、GiST、SP-GiST、GIN。不支持HASH索引，每种索引使用不同的算法，适应不同类型的查询场景。缺省情况下CREATE INDEX命令将创建B-tree索引，其适用于大多数查询场景。关于索引类型的详细说明，可以参照PostgreSQL相关文档。

注意：GP在使用唯一索引时有特殊考虑。唯一索引必须包含所有的DK键。唯一索引不支持AO表。在分区表上，唯一索引可以不包含分区字段，但其只对叶子分区有效，不能对整个分区表有效，所以，这种不包含分区字段的唯一索引不能在整个分区表层面保证数据的唯一性。

关于位图索引

除了PostgreSQL提供的索引类型之外，GP还提供了位图索引类型的支持。位图索引对于数据仓库系统和决策分析系统可能会有帮助。这些应用通常拥有海量数据，日常需要处理很多ad-hoc类型的查询，但少有数据修改的操作。

索引是一系列按照指定字段排序并包含指向表中记录指针的集合。普通的索引每个Key对应一组数据表中相同字段值Row的tuple ID。而Bitmap索引，为表中的指定字段的每一个不同的值存储一个位图，用二进制的1和0来标识是否有该值的记录。普通的索引，尺寸有时可能会比表中的数据尺寸大几倍，而位图索引的尺寸可能只有表中数据尺寸的N分之一。当然，如果未来GP引入了BRIN索引，可能其尺寸会更小，对于特定场景的性能也会更高。

位图的每个bit对应表中记录的tuple ID，被标记的bit意味着对应的tuple ID的记录包含这个位图的字段值。数据的实际位置可以通过映射函数得到。位图索引以压缩的方式存储位图，位图索引字段DISTINCT值的数量越小，位图索引的尺寸就越小，压缩效果也会越好，与其他索引相比节省空间方面也更有优势。位图索引的尺寸与表中记录数和索引字段DISTINCT值的数量正相关。

在WHERE子句中包含多个条件的查询时，相比较WHERE子句中只包含个别条件的场景，位图索引可能更有效。

何时使用位图索引

Bitmap索引更适合只读的查询场景，不适合数据更新的场景。当索引字段的DISTINCT值的数量介于100到10万之间，并常与其他索引字段一同查询时，Bitmap索引可能会表现的更好。DISTINCT值的数量少于100的字段往往可能不适合使用任何类型的索引，例如，性别字段通常只有两种DISTINCT值：男、女，不适合使用索引。一个DISTINCT值的数量超过10万的字段，Bitmap索引的尺寸优势和性能优势都将严重下降。

Bitmap索引可以提升ad-hoc类型查询的性能。对于在WHERE子句中使用AND和OR的多条件查询，可以直接在位图索引上进行位图运算，而不用先转换为tuple ID，性能可以得到很大的提升。当需要返回的记录数很小时，查询可以快速得到结果，而不需要全表扫描。

注意：任何索引都不是万能的，这里说了很多Bitmap索引的优势，不等于就可以随意的创建，适用有条件，选择需谨慎。此处所述的100~10万之间的DISTINCT值的数量不是万能公式，实际上绝大部分场景不适合使用Bitmap索引，另外，衡量DISTINCT值的数量时，要以单个Instance的情况来评估，因为索引也是分布式的。绝大部分情况下，从全表的角度来看唯一性指数的话，可能会比较低，但从单个Instance的角度来看，可能又很高。例如，一个100个Primary的集群，一张1亿条记录的表，某个字段的DISTINCT值的数量为10000，整体计算，唯一性指数为0.0001。而从单个Primary来看，其DISTINCT值的数量是10000，记录总数为100万，唯一性指数则为0.01，跟集群角度相比，提高了100倍，也就是说，每个值对应大约100条记录，此时B-tree索引可能会比Bitmap更适合。

何时不宜使用位图索引

位图索引不适合用于唯一性字段和DISTINCT值的数量很高的字段，例如客户名称、电话号码。位图索引在DISTINC值的数量超过10万后，不管表中的记录数是多少，Bitmap索引的尺寸优势和性能优势都将严重下降。

位图索引主要适用数据仓库应用 -- 大量的分析查询但却极少的数据修改。不适合大量并发事务更新数据的OLTP类型应用。

和B-tree相比，Bitmap索引的使用应该更保守。建议在建立Bitmap索引之后做必要的测试以证明其可以对查询性能有改善(相对于做全表扫描查询)。另外，最好跟其他索引类型做必要的对比，就编者的经验来看，正如前面[何时使用位图索引]章节的[注意]部分所述，可能在很多需要使用索引的时候，直接选择B-tree就足够了，使

用GP时，需要时刻清醒，这是个MPP数据库，数据是分散的，不是集中的，要用分布式的眼光来看待问题。

创建索引

使用CREATE INDEX命令在表上创建新的索引。缺省情况下，没有明确指定索引类型时创建的是B-tree索引。例如，在表films的title字段上创建B-tree索引：

```
=# CREATE INDEX title_idx ON films(title);
```

在表employee的gender字段上创建位图索引：

```
=# CREATE INDEX gender_bmp_idx ON employee USING bitmap(gender);
```

表达式索引

得益于PostgreSQL的特性，GP中的索引不是只能建立在表中的字段之上，还可以是表中的字段组成的表达式或者函数。这对于那些使用表达式或者函数作为条件的查询也可以使用索引来加速。

表达式索引的维护代价比普通索引要高一些，因为在插入和更新索引时都需要计算表达式的值，如果表达式的计算代价不大，这种差异可能也不会很显著。但在查询期间并不需要重新计算，因为索引存储的是表达式计算的结果，查询时，只需要将条件中的表达式算出并进行索引扫描即可，所以对于数据库来说，表达式索引在查询时的效果等同于WHERE indexedcolumn = 'constant'，查询的性能与普通的非表达式索引完全相同。因此，对于查询性能很重要，而插入和更新的性能相对不那么重要的情况，表达式索引将很有用。

下面是一个做小写转换查询的例子：

```
=# SELECT * FROM test1 WHERE lower(col1) = 'value';
```

如果希望该查询可以实现索引扫描，可以这样创建索引：

```
=# CREATE INDEX test1_lower_col1_idx ON test1(lower(col1));
```

假如经常执行下面这种查询：

```
=# SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

可以这样创建索引为这种查询提供索引扫描：

```
=# CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

注意：如果是一个表达式，CREATE INDEX 语法要求表达式要用括号括起来，如果是一个简单的函数，括号可以省略。

索引检验

虽然在GP中索引一般不需要维护和调优，但检查索引是否被使用到，还是很重要的，因为索引会带来很大的资源开销，没有被查询使用到的索引是对资源的极大浪费。可以使用EXPLAIN命令来检验查询是否使用了索引。

执行计划显示出不同的执行步骤以及时间评估等信息。在EXPLAIN的输出中寻找下面的步骤和操作以确认索引的使用情况：

- **Index Scan** -- 扫描索引。
- **Bitmap Heap Scan** -- 根据BitmapAnd、BitmapOr或BitmapIndexScan得到的Bitmap结果到数据表中获取相关的记录。
- **Bitmap Index Scan** -- 根据查询语句中的某个字段的多个索引条件扫描相关字段的索引并生成位图。
- **BitmapAnd or BitmapOr** -- 将多个来自BitmapIndexScan的结果进行AND或者OR操作，生成一个新的位图。

没有一个通用的公式可以计算出哪些场景会使用索引或者不会使用索引，最好的方法还是进行验证。应考虑以下因素：

- 在创建和更新索引后运行ANALYZE。该命令将收集优化器需要的统计信息，优化器将利用这些统计信息以评估不同类型执行计划的成本，只有更准确的统计信息才能更有助于优化器选择更合理的执行计划。
- 使用真实数据进行测试。用测试数据进行测试，这的确可以测出哪些索引是有用的，但这样的结论对测试数据是没错的，但这个结论对于真实的数据可能没有任何帮助。
- 不要使用很少的数据来进行测试，因为跟真实的场景可能偏差太大。当测试的数据量很小时，特定的条件匹配到的记录数将会非常少，此时优化器很可能会选择使用索引扫描，而真实数据量很大，同样的条件可能匹配到的记录数会非常多，此时优化器反而会选择全表扫描。
- 要小心制造测试数据（通常由于安全等因素，获取真实的数据很难）。数值相似，完全随机，或者排序的数据等，这些都会与真实的数据特点有很大的差异。
- 当索引没有被使用，可以强制索引生效（修改参数的类型或者评估的系数让优化器

尽可能选择索引扫描)。有些运行时的参数可以关闭一些执行计划类型。例如，对于PostgreSQL优化器来说，关闭顺序扫描(enable_seqscan)和打开嵌套循环(enable_nestloop)等，使用EXPLAIN ANALYZE命令对使用索引前后进行比较。对于SSD磁盘，还可以减小random_page_cost(缺省值为100，含义是一次随机访问磁盘的代价是100个page，seq_page_cost的缺省值是1，含义是一次顺序访问磁盘的代价是1个page)参数的值来强制使用索引。对于Orca优化器来说，可能会涉及optimizer_nestloop_factor和optimizer_enable_tablescan等参数的设置。

维护索引

当索引的性能变差，可通过REINDEX命令来重建索引。重建索引时，将使用数据表中的数据重新建立一个全新的索引，然后取代旧的索引。

重建表上的全部索引

```
=# REINDEX TABLE my_table;
```

重建特定的索引

```
=# REINDEX INDEX my_index;
```

删除索引

使用DROP INDEX命令来删除一个索引。例如：

```
=# DROP INDEX title_idx;
```

在装载大量数据时，应该先删除索引、再装载数据、然后再重新创建索引，这样可能比直接装载数据要快很多。编者建议使用这样的操作。编者提醒，有些版本存在不会自动删除分区索引的情况，即，在删除ROOT表的索引时，分区的索引不会自动删除(目前6版本会自动删除)，如果没有自动删除，可能需要逐个分区删除。

创建和管理视图

对于那些使用频繁或者比较复杂的查询，通过创建视图(VIEW)可以把其当作一张表来使用SELECT语句访问。视图不会存储真实的数据(后面提到的物化视图会存储数

据)。每当视图被访问时，创建视图的查询语句会被作为子查询而执行。

使用视图可能需要考虑以下几个因素：

- **创建视图的最佳实践** -- 如何创建视图才最合理。
- **视图的依赖关系** -- 查看视图信息，查看视图依赖哪些对象，在GP中视图有强依赖关系，这些依赖信息存储在系统表中。
- **视图是如何被存储的** -- 描述视图依赖的机制。

创建视图

使用CREATE VIEW命令将查询语句定义为一个视图。例如：

```
=# CREATE VIEW comedies AS SELECT * FROM films WHERE kind = 'comedy';
```

注意：官方文档总是这样说：视图会忽略ORDER BY或者SORT操作，虽然在定义视图的语句中可以使用ORDER BY子句，但该子句不会得到执行，除非有LIMIT子句同时出现。但实际上，很多时候，在VIEW中定义了ORDER BY子句之后，查询的时候，真的会是有排序的(也许有版本差异，但6版本真的会执行)。

删除视图

使用DROP VIEW命令删除已有的视图。例如：

```
=# DROP VIEW topten;
```

如果视图之上还有其他视图依赖此视图，使用DROP VIEW . . . CASCADE命令可以将这些依赖的对象一同删除，在此视图被其他视图依赖时，没有CASCADE选项的话，DROP VIEW的操作将会报错失败。

创建视图的最佳实践

在创建视图时需要时刻谨记，视图其实就是一个SQL，实际执行时，其与直接写出SQL没有任何区别，优化器可能会把视图中的SQL拆开进行优化调整计算的顺序。

视图的常见用处：

- 视图可以把一个复杂的SQL作为一个简单的对象来重复使用。
- 视图可以把一张表以另一种形式呈现，例如设置不同的数据过滤条件，不同的访问权限等，这样，不需要再创建一张表。

如果一种SQL查询只是在个别语句中用到，可以使用SELECT命令的WITH子句来实现，可能不需要为此而创建一张很少用到的视图。另外，WITH子句中的查询还可以将执行计划分离，WITH子句中的查询，优化器不会尝试进行拆解，而是直接当做一个整体来执行，因此往往会是一个非常好的选择。

通常，不要创建多层视图 -- 就是基于其他视图来创建视图，这样会极大增加视图的管理难度，因为在GP中视图是有强依赖关系的，当需要删除并重建(CREATE OR REPLACE命令不可以修改视图的字段定义)某个视图时，所有依赖该视图的上层视图，都需要被删除。

有两类使用视图的方式是应该避免的，刚刚讲的多层视图还有其他弊端：

- **定义了很多层的视图，最后的查询语句看起来很简单** -- 这样的设计看起来一点都不酷，因为当遇到问题需要排查时，执行计划可能会变得很复杂，导致无从下手。以前有不少人以写得出一个巨大无比的单条SQL搞定一个复杂的问题而自我陶醉，带来的后续维护问题是痛苦的，反而拆分为多个相对简单的步骤更便于排查问题和维护。编者认为，优雅而高效的解决问题才最重要，故意把问题复杂化不值得提倡，那不能证明能力。

- **定义一个大而全的视图，涉及很多表，然后可以用于各种场景** -- 这种设计也是极其糟糕的，乍一看很酷，实际上，因为适用的场景多，就很难兼顾到每个场景，所以，可能有的场景SQL执行的比较优化，而有的场景SQL执行的很糟糕，这样的视图想要优化到每个场景都用起来都能表现出很好的性能，那几乎是不可能的。

视图的依赖关系

如果要删除一张表，而该表上有视图依赖，则必须使用CASCADE来删除，或者先把依赖的视图全部删掉。如下面的例子所示：

```
=# CREATE TABLE t (
    id integer,
    name text
);
=# CREATE VIEW v AS SELECT * FROM t;
=# DROP TABLE t;
```

```
ERROR: cannot drop table t because other objects depend on it
DETAIL: view v depends on table t
HINT: Use DROP ... CASCADE to drop the dependent objects too.
=# ALTER TABLE t DROP id;
ERROR: cannot drop column id of table t because other objects depend on it
DETAIL: view v depends on column id of table t
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

这个例子表明，当视图对表有依赖，两者的CREATE必须是有顺序的，必须先创建了表，才能基于该表创建视图。不可能在创建好需要的表之前创建视图。

如果要修改一张表，例如要把字段的数据类型从integer改为bigint，这种需求很正常，可能因为业务的调整，需要存储更大的数字，但是，因为表上有视图使用了这个字段，那就没办法直接完成修改，可以先删除这些依赖的视图，然后修改字段类型，之后再重建这些被删掉的视图。此时就需要用到视图依赖信息。

查看视图的依赖关系

为了便于阅读和学习实践，下面将使用举例的方式来展示表及字段与视图之间的依赖关系。这里会介绍以下内容：

- **查看一张表上直接依赖的视图**
- **查看一个字段上的直接依赖**
- **查看视图与依赖表的模式信息**
- **查看视图的定义**
- **查看视图的多层依赖关系**

首先创建一些表和视图，用于本节的示例使用：

```
=# CREATE TABLE t1 (
    id integer PRIMARY KEY,
    val text NOT NULL
);
=# INSERT INTO t1 VALUES (1, 'one'), (2, 'two'), (3, 'three');
=# CREATE FUNCTION f() RETURNS text
    LANGUAGE sql AS 'SELECT ''suffix''::text';
=# CREATE VIEW v1 AS SELECT max(id) AS id FROM t1;
=# CREATE VIEW v2 AS SELECT t1.val FROM t1 JOIN v1 USING (id);
=# CREATE VIEW v3 AS SELECT val || f() FROM t1;
=# CREATE VIEW v5 AS SELECT f();
```

```
=# CREATE SCHEMA mytest;
=# CREATE TABLE mytest.tm1 (
    id integer,
    val text NOT NULL
);
=# INSERT INTO mytest.tm1 VALUES (1, 'one'), (2, 'two'), (3, 'three');
=# CREATE VIEW vm1 AS SELECT id FROM mytest.tm1 WHERE id < 3;
=# CREATE VIEW mytest.vm1 AS SELECT id FROM public.t1 WHERE id < 3;
=# CREATE VIEW vm2 AS SELECT max(id) AS id FROM mytest.tm1;
=# CREATE VIEW mytest.v2a AS SELECT t1.val FROM public.t1
    JOIN public.v1 USING (id);
```

查看一张表上直接依赖的视图

要查看哪些视图直接依赖于t1表，使用一个关联多张系统表的SQL来查询，这些系统表中包含了依赖关系，并限制查询只返回视图类的依赖：

```
=# SELECT v.oid::regclass AS view,d.refobjid::regclass AS ref_object
FROM pg_depend AS d JOIN pg_rewrite AS r ON r.oid = d.objid
JOIN pg_class AS v ON v.oid = r.ev_class
WHERE v.relkind = 'v' AND d.classid = 'pg_rewrite'::regclass
    AND d.deptype = 'n' AND d.refclassid = 'pg_class'::regclass
    AND d.refobjid = 't1'::regclass
GROUP BY 1,2 ORDER BY 1,2;
```

	view regclass	ref_object regclass
1	v1	t1
2	v2	t1
3	v3	t1
4	mytest.vm1	t1
5	mytest.v2a	t1

查询中使用了对象标识符类型的强制类型转换，相关信息可以参考PostgreSQL相关文档。由于原英文手册展示比较复杂，体现出了多字段依赖关系，这里已经进行了分组去重。对上述SQL进行适当的修改之后可以查询哪些视图依赖于f函数：

```
=# SELECT v.oid::regclass AS view,d.refobjid::regproc as ref_object
FROM pg_depend AS d JOIN pg_rewrite AS r ON r.oid = d.objid
JOIN pg_class AS v ON v.oid = r.ev_class
WHERE v.relkind = 'v' AND d.classid = 'pg_rewrite'::regclass
    AND d.deptype = 'n' AND d.refclassid = 'pg_proc'::regclass
    AND d.refobjid = 'f'::regproc
GROUP BY 1,2 ORDER BY 1,2;
```

	view regclass	ref_object regproc
1	v3	f
2	v5	f

查看一个字段上的直接依赖

修改上面的查询语句，可以用于查询依赖于某个字段的视图，当需要修改表上的某个字段或者删除该字段时(编者建议，不要在一张大表上直接修改字段定义，可能会是一个性能很差的操作，应该制定规范，采用重建表的方式进行)，会用到这种查询，这里将会用到pg_attribute系统表：

```
=# SELECT v.oid::regclass AS view,d.refobjid::regclass AS ref_object,
   a.attname AS col_name
FROM pg_attribute AS a
JOIN pg_depend AS d ON d.refobjsubid = a.attnum AND d.refobjid = a.attrelid
JOIN pg_rewrite AS r ON r.oid = d.objid
JOIN pg_class AS v ON v.oid = r.ev_class
WHERE v.relkind = 'v' AND d.classid = 'pg_rewrite'::regclass
   AND d.refclassid = 'pg_class'::regclass AND d.deptype = 'n'
   AND a.attrelid = 't1'::regclass AND a.attname = 'id'
ORDER BY 1,2,3;
```

	view regclass	ref_object regclass	col_name name
1	v1	t1	id
2	v2	t1	id
3	mytest.vm1	t1	id
4	mytest.v2a	t1	id

查看视图与依赖表的模式信息

如果在多个模式中创建了视图，而视图依赖的表也分散在不同的模式中，则可以将视图以及依赖的表以及所属的模式信息一起查询出来。这里将涉及pg_namespace系统表，不过，会忽略系统模式：

```
=# SELECT v.oid::regclass AS view,ns.nspname AS schema,
   d.refobjid::regclass AS ref_object
FROM pg_depend AS d
JOIN pg_rewrite AS r ON r.oid = d.objid
JOIN pg_class AS v ON v.oid = r.ev_class
JOIN pg_namespace AS ns ON ns.oid = v.relnamespace
WHERE v.relkind = 'v' AND d.classid = 'pg_rewrite'::regclass
```

```
    AND d.refclassid = 'pg_class'::regclass
    AND d.deptype = 'n' AND (ns.oid >= 16384 OR ns.nspname = 'public')
    AND NOT (v.oid = d.refobjid)
GROUP BY 1,2,3 ORDER BY 1,2,3;
```

	view regclass	schema name	ref_object regclass
1	v1	public	t1
2	v2	public	t1
3	v2	public	v1
4	v3	public	t1
5	vm1	public	mytest.tm1
6	mytest.vm1	mytest	t1
7	vm2	public	mytest.tm1
8	mytest.v2a	mytest	t1
9	mytest.v2a	mytest	v1

查看视图的定义

通过该SQL查询出依赖t1表的视图信息，包括依赖的字段和创建视图的SQL：

```
=# SELECT v.relname AS view,d.refobjid::regclass as ref_object,
    string_agg(a.attnum||':'||a.attname,',' order by a.attnum) ref_cols,
    'CREATE VIEW ' || v.relname || ' AS ' || pg_get_viewdef(v.oid) AS view_def
FROM pg_depend AS d
JOIN pg_rewrite AS r ON r.oid = d.objid
JOIN pg_class AS v ON v.oid = r.ev_class
JOIN pg_class AS t ON t.oid = d.refobjid
JOIN pg_attribute AS a
    ON a.attrelid =  d.refobjid AND a.attnum = d.refobjsubid
WHERE NOT (v.oid = d.refobjid) AND d.refobjid = 't1'::regclass
GROUP BY 1,2,4
ORDER BY 1,2;
```

	view name	ref_object regclass	ref_cols text	view_def text
1	v1	t1	1:id	CREATE VIEW v1 AS SELECT max(t1.id) AS id
2	v2	t1	1:id,2:val	CREATE VIEW v2 AS SELECT t1.val
3	v2a	t1	1:id,2:val	CREATE VIEW v2a AS SELECT t1.val
4	v3	t1	2:val	CREATE VIEW v3 AS SELECT (t1.val || f())
5	vm1	t1	1:id	CREATE VIEW vm1 AS SELECT t1.id

查看视图的多层依赖关系

这是一个CTE语句(这不是RECURSIVE CTE，所以，几乎所有版本都支持)，WITH

中查询出所有的视图，主体查询中查出哪些视图依赖其他视图：

```
=# WITH views AS (
    SELECT v.relname AS view,d.refobjid AS ref_object,
       v.oid AS view_oid,ns.nspname AS namespace
    FROM pg_depend AS d
    JOIN pg_rewrite AS r ON r.oid = d.objid
    JOIN pg_class AS v ON v.oid = r.ev_class
    JOIN pg_namespace AS ns ON ns.oid = v.relnamespace
    WHERE v.relkind = 'v' AND (ns.oid >= 16384 OR ns.nspname = 'public')
       AND d.deptype = 'n' AND NOT (v.oid = d.refobjid)
)
SELECT views.view, views.namespace AS schema,
    views.ref_object::regclass AS ref_view,
    ref_nspace.nspname AS ref_schema
FROM views
JOIN pg_depend as dep ON dep.refobjid = views.view_oid
JOIN pg_class AS class ON views.ref_object = class.oid
JOIN pg_namespace AS ref_nspace ON class.relnamespace = ref_nspace.oid
WHERE class.relkind = 'v'
    AND dep.deptype = 'n';
```

	view name	schema name	ref_view regclass	ref_schema name
1	v2	public	v1	public
2	v2a	mytest	v1	public

实际上，编者在编写并行DDL备份恢复脚本时，已经将这些复杂的视图依赖关系都拆解了，通过编者的脚本备份出的DDL中，会自动把视图按照多层依赖关系进行分拆，确保恢复DDL的时候，视图按照依赖关系进行并行恢复，同一层级相互没有依赖关系的视图可以一起并行恢复。编者在实现集群之间DDL增量比对脚本时也实现了依赖拆解和并行恢复，对实现灾备集群提供了有力的支持。

视图是如何被存储的

视图与表相似，都是relation，名称存储在pg_class系统表中，也都有字段属性，字段属性与表一样存储在pg_attribute系统表中。下面是一些区别：

- 视图没有数据文件 -- 因为视图不存储数据。
- pg_class系统表中的relkind属性是v，而数据表是r。
- 每个视图有一个ON SELECT事件名称为_RETURN的rewrite规则。

视图的rewrite规则存储在pg_rewrite系统表中，视图的定义存储在该系统表的ev_action字段中。关于视图的更多详细信息，可以参考PostgreSQL的相关文档。

视图的定义不是以字符串的形式存储的，存储的是解析后的查询树，在视图被创建时生成的查询解析树，这样会有几方面的影响：

- 对象名称是在视图创建时解析的，所以创建时的search_path会影响到视图的定义，如果使用时的search_path与创建时不一致，可能会导致找不到表的报错。
- 视图对其他对象的引用是通过OID来实现的，因此，修改依赖的表或者字段的名称并不会影响视图的依赖关系。也就是说，如果视图依赖是表名是old，从old改为new之后，依赖的表就是new。
- GP可以精确的获取视图使用了哪些对象，所以能够存储严谨的依赖关系。

注意：GP处理视图的方式和处理函数的方式完全不同，对于函数，GP存储的是字符串，创建时不会解析为查询树，因为函数中的具体执行情况无法预知，只有具体的参数和具体的数据在执行时才能确定涉及的对象，所以，没有办法精准获取函数的依赖关系。编者在社区遇到很多次关于函数涉及的表如何查询的问题，这个的确是无能为力的，即便通过pg_proc系统表的prosrc(存储function的全部源码)字段来查找，也只能匹配明文写出的对象名称。

视图的依赖信息存储在哪里

这些表中存储着视图依赖哪些对象：

- pg_class -- 存储所有relation的信息，包括表、视图、索引、外部表和序列等。通过relkind来区分不同类型的对象。
- pg_depend -- 存储着数据库中非共享对象的依赖关系。
- pg_rewrite -- 存储着表和视图的rewrite规则。
- pg_attribute -- 存储着字段信息。
- pg_namespace -- 存储着模式信息。

需要注意的是，视图对其依赖的对象没有直接的依赖关系，而是通过rewrite规则来实现依赖关系。其实这话的意义不大，因为依赖关系仍然是明确的且强制。

创建和管理物化视图

物化视图与普通视图类似，都是将一个常用的查询保存为一个relation，之后就可以如同访问一张表一样执行SELECT操作。而不同的是，物化视图会直接将查询结果持久化为数据文件，类似数据表的存储形式，所以访问物化视图时，是直接访问持久化的数据，往往比通过视图访问数据表更快，但数据不是实时的。

物化视图的数据无法直接修改，只能通过REFRESH MATERIALIZED VIEW 命令来刷新数据，用于存储物化视图的查询语句与普通视图的查询语句的存储方式相同。例如可以这样创建一个物化视图：

```
=# CREATE MATERIALIZED VIEW sales_summary AS
SELECT seller_no, invoice_date,
    sum(invoice_amt)::numeric(13,2) as sales_amt
FROM invoice
WHERE invoice_date < CURRENT_DATE
GROUP BY seller_no, invoice_date
ORDER BY seller_no, invoice_date;

=# CREATE UNIQUE INDEX sales_summary_seller
    ON sales_summary (seller_no, invoice_date);
```

可以使用如下命令定期刷新视图的数据：

```
=# REFRESH MATERIALIZED VIEW sales_summary;
```

GP中的物化视图的属性信息与表或普通视图是一样的，都是relation，表和普通视图也是relation。当查询一个物化视图时，数据直接从物化视图的数据文件获取，就如同访问普通的数据表一样，而物化视图中的查询语句，仅用于产生数据以填充物化视图。

如果业务上可以接受定期更新物化视图的数据，将会为查询带来极大的性能提升。

物化视图还可以建立在外部表之上，以提升外部数据的访问性能，物化视图上还可以创建索引，外部表上是不能创建索引的。不过，这种场景可能只适合从固定的外部表查询固定的数据，或者外部表的数据有周期性变化，编者认为，通过物化视图来加速外部表的访问并不是物化视图特有的功能，在外部表上创建物化视图同样需要读取外部表的全部数据，这与，把数据加载到一张普通的数据表，没有任何差异。而物化视图的刷新与普通数据表的TRUNCATE并重新INSERT效果相同。

如果一种SQL查询只是在个别语句中用到，可以使用SELECT命令的WITH子句来实现，可能不需要为此而创建一张很少用到的视图。编者再次提醒，不要乱用视图，更不要随意创建多层视图，虽然编者的脚本可以处理这些难题，但日常维护会非常困难。

创建物化视图

使用CREATE MATERIALIZED VIEW命令基于一个查询语句来创建物化视图：

```
=# CREATE MATERIALIZED VIEW us_users AS
SELECT u.id, u.name, a.zone
FROM users u, address a WHERE a.country = 'USA';
```

如果查询语句中包含ORDER BY或SORT子句，只会影响物化视图的数据生成，但不会影响物化视图的查询，也就是说，生成的物化视图的数据会是有序的，但针对物化视图的查询不保证顺序，不过这不等于说排序是完全无意义的，有序的数据可以有助于物化视图创建聚集索引。

刷新或停用物化视图

使用REFRESH MATERIALIZED VIEW命令来刷新物化视图的数据：

```
=# REFRESH MATERIALIZED VIEW us_users;
```

使用WITH NO DATA子句来刷新物化视图，物化视图中的数据将被清空，并且不会产生新的数据，此时，物化视图将不能再被查询，查询这种物化视图将会得到一个报错信息。

```
=# REFRESH MATERIALIZED VIEW us_users WITH NO DATA;
=# SELECT * FROM us_users;
ERROR:  materialized view "us_users" has not been populated
HINT:  Use the REFRESH MATERIALIZED VIEW command.
```

删除物化视图

使用DROP MATERIALIZED VIEW命令来删除物化视图的定义以及数据。例如：

```
DROP MATERIALIZED VIEW us_users;
```

使用命令DROP MATERIALIZED VIEW . . . CASCADE将可以级联删除所有依赖该物化视图的对象，例如另一个物化视图依赖该物化视图，也会一同被删除，此时如果没有CASCADE子句，DROP MATERIALIZED VIEW命令将会报错失败。物化视图的依赖关系和普通视图是一样的。

例如，修改"查看视图的依赖关系"章节的相关示例SQL，可以查询物化视图的依赖信息，注意，物化视图在pg_class中存储的relkind属性为m。例如，查询依赖t1表的物化视图：

```
=# SELECT v.oid::regclass AS view,d.refobjid::regclass AS ref_object
FROM pg_depend AS d JOIN pg_rewrite AS r ON r.oid = d.objid
JOIN pg_class AS v ON v.oid = r.ev_class
WHERE v.relkind = 'm' AND d.classid = 'pg_rewrite'::regclass
    AND d.deptype = 'n' AND d.refclassid = 'pg_class'::regclass
    AND d.refobjid = 't1'::regclass
GROUP BY 1,2 ORDER BY 1,2;
```

	view regclass	ref_object regclass
1	m v	t1

第八章：数据的分布与倾斜

GP 要求数据在 Instance 上均匀分布，在 MPP Share-Nothing 数据库中，对于一个查询来说，所有操作都完成才算完成，那么这个总的耗时就是最慢 Instance 的耗时。如果有数据的倾斜，处理数据更多的 Instance，完成计算所需要的时间就越久，所以，如果所有的 Instance 处理的数据量相当，那么总体的执行时间就会保持一致，如果个别 Instance 要处理更多的数据，将可能导致严重的资源消耗且拖慢整体的处理时间。

在进行大表关联时，合理的数据分布很重要，当进行关联时，匹配的记录必须在 Instance 本地，如果不能满足这个条件，执行计划中将会加入数据移动的算子，需要将一个表或多个表做数据重分布，这样将消耗很多的网络资源，当然，在有些时候，如果其中一个表很小，还可能会选择将小表进行广播。重分布操作是，每个 Instance 按照关联字段重新计算 HASH 值得到记录应该发送到哪个 Instance 然后发送过去。

本地关联

使用 HASH 分布的情况下，表的记录均匀的分散到所有 Instance 上，所以，在进行关联查询时，表之间匹配的数据都在本地，计算将直接在本地完成，这种情况称为本地关联。本地关联将可以最大程度的避免数据移动，所有的 Instance 都独立处理本地的数据，而不需要在 Instance 之间通过内联网络交换数据。

要实现大表之间的本地关联，需要确保关联字段包含全部的分布键，这部分在”解读 GP 分布策略”章节已经做了很多详细介绍，当关联的数据都在 Instance 本地，将可以显著提升处理的性能。另外，在 CREATE TABLE 时应该确保关联的字段在不同的表中采用相同的字段类型，因为，不同的数据类型对应不同的底层数据结构，相同的记录因为底层存储的差异会分散到不同的 Instance 上，这种情况，在进行关联查询时仍然会涉及数据的重分布。正如”解读 GP 分布策略”章节所述，**尽可能只选择一个字段作为分布键 (这是非常重要的)**。

数据倾斜

数据倾斜一般是由于选择了错误的分布键而造成的结果，或者是因为在 CREATE TABLE 时没有指定分布键而自动以第一个字段作为分布键。通常可能会表现出查询性能差，甚至出现内存不足的报错。数据倾斜会直接影响表扫描的性能，同时也会影响相关的关联查询和分组汇总等计算的性能。

检验数据分布是否均匀非常重要，无论是初次加载数据之后，还是增量数据加载之后。有时，数据量不大时可能不会明显的表现出倾斜，所以需要定期检查倾斜情况。

虽然在官方文档中介绍了查询表中记录数分布情况的方法，但编者不想介绍这种方法，编者认为，这种方法是陈旧而落后的，因为其需要使用 count(*)的方式来计算表中的记录数，因为对于很大的表来说，这种操作无疑是难以忍受的。编者推荐，通过计算一张表在不同 Instance 上所占的空间尺寸来评估是否发生倾斜。例如：

```
=# SELECT gp_segment_id,pg_relation_size('t1')
FROM gp_dist_random('gp_id') ORDER BY 2 DESC;
```

在 gp_toolkit 中有可用于查看表倾斜情况的视图，虽然编者从来不用这些视图，但还是有必要介绍一下，后续编者将介绍更高效的实现方案。

- gp_toolkit.gp_skew_coefficients视图，一个非常复杂的视图，经过编者了解，该视图最终会针对每张表分为AO表和非AO表来分别计算记录数情况，AO表会通过get_ao_distribution函数来计算记录数，Heap表会通过count(*)来计算记录数，真是一个神奇的设计。该视图以Instance记录数的标准差除以平均值再乘以100来表示倾斜的严重程度，值越大倾斜越严重。

- gp_toolkit.gp_skew_idle_fractions视图，一个非常复杂的视图，经过编者了解，该视图，会计算Instance中记录数的最大值与平均值的差值，然后除以最大值，得到一个不大于1的浮点数，值越大倾斜越严重。不过，在获取表的信息时与gp_toolkit.gp_skew_coefficients视图是一模一样的，所以，没有性能优势。

编者来说说自己的实现，不去轮询查询每张表的信息，因为这些系统视图性能极差的根本原因是，都要循环获取每张表的信息，尤其是表的数量很大的时候，每个表的信息获取都会变慢，Heap 表的 count(*)操作更是致命的。我们的目的是检查倾斜情况，而反应倾斜情况的未必一定要通过记录数来体现，如前面所述，可以通过尺寸来体现。编者使用如下函数来获取文件信息：

```
CREATE OR REPLACE FUNCTION gp_toolkit.gp_table_file_info() RETURNS SETOF
VARCHAR[] AS $$
import os
_rslt = plpy.execute("""select current_database() dbname,
inet_server_port() port;""")
(_dbname, _port) = (_rslt[0]["dbname"], str(_rslt[0]["port"]))
_rslt = plpy.execute("""select oid,dattablespace from pg_database where
datname = '%s';""" % (_dbname))
(_dboid, _dbspc) = (str(_rslt[0]["oid"]), str(_rslt[0]["dattablespace"]))
def getSqlValue(_sql):
    _utility = "PGOPTIONS='-c gp_session_role=utility' psql -v
```

```
ON_ERROR_STOP=1"
    _cmd = """%s -d '%s' -p %s -tAXF '|' 2>&1 <<_END_OF_SQL\n""" % (_utility,
_dbname, _port) + _sql + "\n_END_OF_SQL"
    try:
        val = os.popen(_cmd).read()
        return val.strip()
    except Exception, e:
        plpy.error(str(e))
_dftpath = getSqlValue("""show data_directory""") + "/base/" + _dboid + "/"
_version = int(getSqlValue("""SELECT
(string_to_array((string_to_array(version(),'Greenplum Database
'))[2],'.'))[1];"""))
_rslt
if _version < 6:
    _rslt = plpy.execute("""select
t.oid,trim(n.location_1)||'/'||t.oid||'/'||'%s'||'/' path
    from pg_tablespace t,pg_filespace f,gp_persistent_filespace_node n
    where t.spcfsoid = f.oid and f.oid = n.filespace_oid;""" % (_dboid))
else:
    _dirc = getSqlValue("""show data_directory""")
    _rslt = plpy.execute("""select
t.oid,'%s'||'/pg_tblspc'||t.oid||'/'||'GPDB_*'||'/%s/' path from
pg_tablespace t;""" % (_dirc,_dboid))
_spcarray = []
_spcarray.append([str(1663), _dftpath])
for _row in _rslt:
    (_spcoid, _spcpath) = (str(_row["oid"]), _row["path"])
    if not(os.path.exists(_spcpath)):
        continue
    if os.path.isfile(_spcpath):
        continue
    _spcarray.append([_spcoid, _spcpath])
_sizemap = {}
for _spcinfo in _spcarray:
    (_spcoid, _spcpath) = (_spcinfo[0], _spcinfo[1])
    _lscmd = """ls -lL --full-time %s|awk '{print $9"\t"$5"\t"$6" "$7}'|grep
'^[0-9]'|sort -n""" % (_spcpath)
    _rslt = os.popen(_lscmd).read().strip()
    if _rslt == "":
        continue
    for _row in _rslt.split("\n"):
        (_relfile, _size, _time) = _row.split("\t")
        _relfile = _relfile.split(".")[0]
        _key = _spcoid + "-" + _relfile
```

```
        if _sizemap.has_key(_key):
            _sizemap[_key] = [_sizemap[_key][0] + int(_size),
_sizemap[_key][1] + 1, _sizemap[_key][2] + "\n" + _time]
        else:
            _sizemap[_key] = [int(_size),1,_time]
_rslt = plpy.execute("""select
n.nspname,c.relname,c.reltablespace,c.relfilenode,c.relstorage
    from pg_class c, pg_namespace n where c.relnamespace = n.oid
    and c.relkind = 'r' and c.relstorage <> 'x' and c.reltablespace <> 1664
and not c.relhassubclass
    and n.nspname not like E'pg\_temp\_%' and n.nspname not like
E'pg\_toast\_temp\_%';""")
for _row in _rslt:
    (_nspname, _relname, _relspc) = (_row["nspname"], str(_row["relname"]),
str(_row["reltablespace"]))
    (_relfile, _storage) = (str(_row["relfilenode"]), _row["relstorage"])
    if _relspc == "0":
        _relspc = _dbspc
    _key = _relspc + "-" + _relfile
    if _sizemap.has_key(_key):
        if _storage == "h":
            yield (_nspname, _relname, _sizemap[_key][0], _sizemap[_key][1],
_relspc, _sizemap[_key][2])
        else:
            yield (_nspname, _relname, _sizemap[_key][0], _sizemap[_key][1],
_relspc, None)
    else:
        yield (_nspname, _relname, 0, 0, None)
$$ LANGUAGE PLPYTHONU;
```

该函数较长，使用时请注意缩进，函数是用 Python 写的，所以，缩进不能出错。在函数中，直接通过操作系统的 ls 命令查看数据库目录下的所有文件的信息，然后与系统表进行关联。通过下面的 SQL 使用刚刚创建的函数来获取表的文件信息：

```
=# select nspname, relname, tablesize, filecount, expectfilecount,
    round(filecount / expectfilecount,1) filecountratio, minsize, maxsize,
fileflag from (
    select size[1] as nspname, size[2] as relname,
        sum(size[3]::bigint) tablesize,string_agg(size[3],','),
sum(size[4]::bigint) filecount,
        min(size[3]::bigint) minsize,
        max(size[3]::bigint) maxsize,
        md5(string_agg(size[6],E'\n' order by segment_id)) fileflag from (
        select gp_toolkit.gp_table_file_info() size, gp_segment_id
```

```
segment_id from gp_dist_random('gp_id')
    ) x group by 1,2
) x left join (
   select nspname, relname, decode(relstorage, 'c', attcount, 1) * y.segs
expectfilecount from (
       select nspname, relname, relstorage, count(*) attcount
       from pg_namespace n, pg_class c, pg_attribute a
       where n.oid = c.relnamespace and c.oid = a.attrelid
       and c.relkind = 'r' and c.relstorage <> 'x' and c.reltablespace <>
1664 and not c.relhassubclass
       and n.nspname not like E'pg\_temp\_%' and n.nspname not like
E'pg\_toast\_temp\_%'
       group by 1,2,3
   ) x, (
       select count(*) as segs from gp_segment_configuration where role =
'p' and content <> -1
   ) y
) y using(nspname, relname) order by 6 desc;
```

这段查询输出的每条记录包含 9 个属性，分别是，模式名称、表名称、该表的总尺寸、该表总的文件数、预期文件数、文件数膨胀倍数、最小的尺寸(单个 Instance)、最大尺寸(单个 Instance)、Heap 表的文件时间戳 MD5 值。

该方法经历过多次大规模集群的检验，有数万张表，目录下有几十万个文件的情况下，也能在几分钟内完成全库的信息收集，可以将查询结果导出到文件中以便于进一步的详细分析。

复制表的注意事项

在 CREATE TABLE 时指定分布策略为 DISTRIBUTED REPLICATED 就可以创建复制表。复制表，会在每个 Instance 上存储一份完整的数据，所以复制表是绝对不会倾斜的，因为每个实例的记录都完全相同。更多关于复制表的注意事项可以参考”复制表的主要应用场景”章节。

计算倾斜

当数据倾斜到个别 Instance 时，它往往是 GP 数据库性能和稳定性的罪魁祸首(编者想说的是，有无数的问题的根源都在倾斜上，当然不仅仅说的是数据倾斜，计算倾斜是更隐蔽的问题，往往可能造成更严重的影响而且难以被发现和解决)，不过，对于数据分布的倾斜，发现和处理往往不难。然而，当倾斜发生在关联、排序、聚合等各种算子的计算过程中时，事情就变的十分复杂，这种情况我们称之为计算倾斜。而要处理计算倾斜，可以说十分困难，一般的技术人员很难发现，更不用说解决这种问题，但

这也是用好GP非常关键的一门武功，此门绝技练到第十层，就是大师级的人物了。

如果单个Instance出现了故障，这有可能与计算倾斜有关(X86经不住长期超负荷的资源压榨，不堪重负必有内伤)，目前，处理计算倾斜还是一个手动的过程(编者也想不出如何实现自动，因为计算倾斜是个过程)。处理计算倾斜时，首先可以看一下溢出文件的情况，如果有计算倾斜但又没有出现溢出文件，可能这种倾斜并不会造成严重的后果。如果发现有计算倾斜现象出现，下面是一些步骤和方法可以参考。

- **gp_toolkit.gp_workfile_usage_per_segment** -- 通过该视图可以查询每个Instance目前使用的Workfile溢出文件的尺寸和文件数量。通过该视图，可以清晰的发现哪些Instance有严重的溢出文件问题。

- **gp_toolkit.gp_workfile_usage_per_query** -- 通过该视图可以查询每个Query在每个Instance上的Workfile的使用情况，显示的信息包括，数据库名称，进程号，会话ID，command count，用户名，查询语句，SegID，溢出文件尺寸，溢出文件数量。

通常，通过这两个视图就可以确定正在发生倾斜的查询，而要解决这些问题，往往需要重新优化SQL，例如确认统计信息是否严重失真，如果是，应该尝试更新统计信息，找到执行计划中不合理的算子，通过修改可能的参数来干预执行计划，使用WITH子句来分拆SQL以达到隔离执行计划的目的，使用临时表以强制拆分执行步骤，强制执行计划选择两阶段AGG或者三阶段AGG等，总之，优化的最高目标就是，让数据库生成的执行计划符合预期，最佳的预期需要基于对MPP的分布式理解和对数据的理解，如果只是从其他数据库的使用中学到了一些支离破碎的知识，可能预期还不如数据库自动生成的执行计划，当能够确切的知道什么样的执行计划才是最优的，那么距离优化出这样的执行计划已经很近了。编者认为，对于GP的学习，首先要能够完全看懂执行计划，其次是知道哪些步骤是有问题的，然后才能优化。优化的最基本前提是看懂执行计划，否则一切都是空谈。让数据库按照期望的最佳路径生成执行计划，是最高功力，不过，往往绝大部分技术人员，只是了解一些SQL语法，并不清楚数据库是如何一步一步计算得到结果的，也就不可能知道何为最佳路径，所以，缺乏这些基本的内功，是不可能练出十层绝学的。

第九章：数据增删改

本章讲述关于数据管理和GP中的并发访问。包含如下内容：

- 关于GP的并发控制
- 插入新记录
- 更新记录
- 删除记录
- 使用事务
- 全局死锁检测
- 回收空间

关于 GP 的并发控制

与事务型数据库系统通过锁机制来控制并发访问的机制不同，GP(与PostgreSQL一样)使用多版本控制(Multiversion Concurrency Control/MVCC)保证数据一致性。这意味着在查询数据库时，每个事务看到的只是数据的快照，其确保当前的事务不会看到其他事务在相同记录上的修改。据此为数据库的每个事务提供事务隔离保护。

MVCC以避免给数据库事务显式上锁的方式，最大化减少锁争用以确保多用户环境下的性能。在并发控制方面，使用MVCC而不是使用锁机制的最大优势是，MVCC机制下，查询(读)的锁与写的锁不存在冲突，并且读与写操作之间从不会互相阻塞。

GP提供了各种锁机制来控制对表数据的并发访问。大多数GP的SQL命令可以自动获取适当模式的锁以确保在命令执行时相应的表不会被删除或者修改。对于不能适应MVCC锁的应用来说，可以使用LOCK命令来显式的获取必要的锁。然而，恰当的使用MVCC比使用LOCK有更好的性能表现。

GP中的锁模式：

锁模式	相关 SQL 命令	冲突
ACCESS SHARE	SELECT	ACCESS EXCLUSIVE
ROW SHARE	SELECT FOR UPDATE SELECT FOR SHARE	EXCLUSIVE、ACCESS EXCLUSIVE
ROW EXCLUSIVE	INSERT、COPY	SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、ACCESS EXCLUSIVE
SHARE UPDATE EXCLUSIVE	VACUUM (no FULL) ANALYZE	SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE ACCESS EXCLUSIVE
SHARE	CREATE INDEX	ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE、ACCESS EXCLUSIVE
SHARE ROW EXCLUSIVE		ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、ACCESS EXCLUSIVE
EXCLUSIVE	DELETE, UPDATE、	ROW SHARE、ROW EXCLUSIVE、SHARE

	SELECT . . . FOR UPDATE, REFRESH MATERIALIZED VIEW CONCURRENTLY	UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、ACCESS EXCLUSIVE
ACCESS EXCLUSIVE	ALTER TABL、DROP TABLE、TRUNCATE、REINDEX、CLUSTER, REFRESH MATERIALIZED VIEW (without CONCURRENTLY), VACUUM FULL	ACCESS SHARE、ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、ACCESS EXCLUSIVE

注意：对于Heap表的UPDATE、DELETE和SELECT . . . FOR UPDATE操作，GP数据库缺省使用EXCLUSIVE锁。当开启全局死锁检测时，对于Heap表的UPDATE和DELETE操作将可以使用ROW EXCLUSIVE锁。对于SELECT . . . FOR UPDATE操作仍然需要使用表级别的锁。

插入新记录

在表刚被创建时，是没有数据的。在数据库进行更多使用之前的第一步是插入数据。要插入新的记录，使用INSERT命令。该命令需要表名和该表每个字段的值。数据的值按照字段在表中出现的顺序排列，使用逗号分割。例如：

```
=# INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

如果不知道字段在表中的顺序，还可以将字段显式的列出来。很多用户认为总是列出字段名称是一种比较好的习惯。例如：

```
=# INSERT INTO products (name, price, product_no)
   VALUES ('Cheese', 9.99, 1);
```

将一张表中符合条件的记录插入另一张表中：

```
=# INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

使用一个命令插入多条记录。例如：

```
=# INSERT INTO products (product_no, name, price) VALUES
(1, 'Cheese', 9.99),
(2, 'Bread', 1.99),
(3, 'Milk', 2.99);
```

在同时插入大量数据时，应该考虑使用外部表(CREATE EXTERNAL TABLE)或者COPY命令。在装载大量记录时，这些装载机制比使用INSERT VALUES更高效(是高效好几个数量级)。更多关于批量装载数据的信息参见”装载和卸载数据”相关章节。

AO表已经为批量装载做了优化。不建议在AO表上使用单条的INSERT VALUES语

句。GP数据库最多支持单个AO表上127个并发INSERT数据，不过，编者建议尽量避免AO表上的并发INSERT，因为底层的数据文件会分裂最多达到127倍!。

更新记录

UPDATE意味着对数据库中现有的数据进行修改。可以修改表中一条记录、一部分记录或者全部记录。每个字段都可以被单独更新而不影响其他字段。

要执行更新，需要如下3方面的信息：

1. 要被更新的表和字段的名称。
2. 字段的新值。
3. 用于过滤需要被更新字段的记录的条件。

例如，下面的命令更新products表中所有price为5的记录的price为10：

```
=# UPDATE products SET price = 10 WHERE price = 5;
```

在GP中使用UPDATE命令有如下的限制：

- Orca支持更新分布键字段，6版本之前的PostgreSQL优化器不支持(编者实测如此，而如果断然的说不支持，跟后续的[全局死锁检测]中所述冲突)。
- 在启用了Mirror的情况下，UPDATE语句中不能有STABLE或者VOLATILE类型的函数。
- PostgreSQL优化器不能UPDATE分区字段，Orca可以UPDATE分区字段。

删除记录

使用DELETE命令从指定的表中删除符合WHERE条件的记录。如果没有使用WHERE子句,将会删除该表的所有记录。例如,从products表中删除所有price为10的记录：

```
=# DELETE FROM products WHERE price = 10;
```

或者删除表中所有记录：

```
=# DELETE FROM products;
```

在GP中使用DELETE操作的限制：

- 在启用了Mirror的情况下，DELETE语句中不能有STABLE或者VOLATILE类型的

函数。

清空表

若想要快速删除表中的所有记录，应该考虑使用TRUNCATE命令。例如：

```
=# TRUNCATE mytable;
```

该命令一次清空表中的全部记录。值得注意的是，TRUNCATE不扫描表，直接将数据文件清空，继承此表的其他表不会被清空，只是被TRUNCATE的表会受到影响。分区表会被视作一个整体，在父表上执行TRUNCATE操作会清空所有相关叶子分区的数据。

使用事务

事务允许将多个SQL语句放在一起当作一个整体来执行，所有SQL一起成功或失败。

在GP中执行事务的SQL命令为：

- 使用BEGIN或START TRANSACTION开始一个事务。
- 使用END或COMMIT提交事务。
- 使用ROLLBACK回滚事务(放弃所有修改)(序列的增长不会被回滚)。
- 使用SAVEPOINT选择性的保存事务点。
- 使用ROLLBACK TO SAVEPOINT回滚到之前保存的事务点。
- 使用RELEASE SAVEPOINT来释放之前保存的事务点。

更多信息参考相关SQL说明，编者认为在GP中应用场景不多。

事务隔离级别

GP数据库兼容标准的SQL事务级别的情况如下：

- READ UNCOMMITTED和READ COMMITTED的表现类似标准的READ COMMITTED。
- REPEATABLE READ和SERIALIZABLE的表现类似REPEATABLE READ。

下面描述GP的不同事务隔离级别的特征：

读未提交和读已提交

GP数据库不允许任何命令看得到另一个并行事务未提交的更新(其实，对于heap表，设置gp_select_invisible参数为on之后是可以看到其他事务未提交的数据的，也看得到未被VACUUM回收的UPDATE和DELETE的数据，INSERT回滚数据也看的到，一旦VACUUM就看不到了，不过，**这个参数要特别慎用！**)。所以，READ UNCOMMITTED的效果与READ COMMITTED一致，READ COMMITTED提供了简单高效的事务隔离机制。相当于SELECT、UPDATE和DELETE命令在开始执行时(注意，是命令开始时，不是事务开始时)在数据库上做了个快照。

READ COMMITTED事务的SELECT命令将会：

- 看得到查询语句开始之前所有已提交的数据(不是事务开始前)。
- 看得到当前事务已经修改的数据。
- 看不到其他事务未提交的数据。
- 看得到当前事务启动之后其他并发事务已经提交的修改。

如果其他事务在当前事务的不同查询之间提交修改，当前事务中的不同SELECT会看到不同的数据。UPDATE和DELETE命令只能看得到命令启动时已经提交的数据，不过，不是事务启动的时间，在事务期间，有其他事务提交的修改，UPDATE和DELETE依然可见，这就是[读已提交]，只要命令开始时是已经提交的数据，就可见。而不用关心当前的事务是何时开始的。

READ COMMITTED事务隔离级别，允许一个事务中的UPDATE或DELETE操作开始之前，其他并发的事务同样可以查询和修改记录。也就是说，READ COMMITTED不能保证在事务期间，所涉及的数据不会发生变化，所以，对于一些要求事务期间必须保证数据库完全一致的应用来说，READ COMMITTED事务隔离级别是不够的。

可重复读取和可串行化

根据SQL标准定义的SERIALIZABLE事务隔离级别，其可以保证并发事务的运行结果与一个接一个的运行结果完全相同。如果指定了SERIALIZABLE事务隔离级别，GP数据会将其降级到REPEATABLE READ事务隔离级别。REPEATABLE READ事务隔离级别不需要重量级的锁就可以避免脏读、不可重复读和幻读，但GP数据库不会检测并发事务期间所有可能的可串行化的相互影响(就是不能保证得到可串行化的结果)。所以，可以通过检查并行事务之间可能的影响来排查这种可串行化的相互影响，可以通过显式的LOCK操作来避免并发事务之间的影响(在事务开始的时候就显式的把相关的表加上必要的锁)。这段很难懂，英文解释也很难懂，总之，GP数据库本身无法保证可串行化的效果，就是说，并发事务不能保证和串行执行一样的结果。

REPEATABLE READ事务的SELECT命令将会：

- 在事务开始时(而不是在查询开始时)确定数据的快照。
- 只能看得到事务开始之前已经提交的数据。
- 看得到当前事务已经修改的数据。
- 看不到其他事务未提交的数据。

- 看不到当前事务启动之后其他事务提交的修改。
- 在当前事务中多次执行SELECT命令总是得到相同的数据。
- UPDATE、DELETE、SELECT FOR UPDATE和SELECT FOR SHARE命令只能看得到事务开始之前的记录，如果其他并发的事务已经修改、删除或LOCK了同一条记录，REPEATABLE READ事务(修改同一条记录的操作)需要等待该事务提交或者回滚这些修改，如果其他并发的事务提交了修改，当前的REPEATABLE READ事务将会失败回滚，如果其他并发事务回滚了修改，当前的REPEATABLE READ事务将可以提交当前的修改。

GP数据库缺省的事务隔离级别是READ COMMITTED，要修改事务隔离级别，可以在开始事务时显式的指定事务隔离级别，或者在事务开始之后通过SET TRANSACTION来设置。例如：

```
=# BEGIN ISOLATION LEVEL REPEATABLE READ;
=# END;
=# BEGIN;
=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
=# END;
```

关于事务隔离级别，这一段真的很难懂，编者觉得用中文讲，这样算是比较易懂了吧。简单总结来说，读已提交，就是只要其他事务COMMIT了，就你看得到，多次查询的结果可能是会变的(结果不可重复)。可重复读，就是在一个事务的生命周期内，多次查询看到的数据是不变的(结果可重复)。

全局死锁检测

GP 数据库的全局死锁检测，通过后台进程收集所有 Instance 的锁信息，通过检测算法来检测死锁情况。这样，就可以放宽 Heap 表的并发 UPDATE 和 DELETE 操作的锁限制。对于 AO 表，UPDATE、DELETE 和 SELECT . . . FOR UPDATE 仍然需要表级锁。

缺省情况下，全局死锁检测并未开启，GP 执行 Heap 表的 UPDATE 和 DELETE 命令需要进行串行化操作。

可以通过设置参数 gp_enable_global_deadlock_detector 来开启全局死锁检测，这样就可以对 Heap 表执行并发 UPDATE 和 DELETE 操作了。

在开启了全局死锁检测后，相关的后台进程会跟随数据库启动一起启动，通过设置 gp_global_deadlock_detector_period 参数来指定收集和分析锁信息的时间间隔。

如果全局死锁检测发现了死锁，会中断相关事务中最新的一些事务来解除死锁。

如果全局死锁检测发现下列的死锁类型，将只会有一个事务成功，其他事务会失败，并且会收到报错信息。

```
concurrent updates to the same row is not allowed
```

- 同一条记录上的并发事务，第一个事务是UPDATE，后面的的事务是UPDATE或者DELETE并且其执行计划包含一个Motion算子。
- 由PostgreSQL优化器执行的并发UPDATE操作在一张Heap表的同一个分布键上。
- 由Orca优化器生成的在一张Hash分布的Heap表上同一条记录的并发UPDATE。

注意： GP 数据库使用 deadlock_timeout 参数来判断本地死锁，由于本地死锁与全局死锁的检测算法不同，可能会被本地死锁检测发现，也可能被全局死锁检测发现，这取决于哪个先发现。

注意： 如果 lock_timeout 开启且设置的值比 deadlock_timeout 和 gp_global_deadlock_detector_period 小，可能一个查询在被检测到死锁之前就因为锁等待时间超时而被中断。

通过执行 pg_catalog.gp_dist_wait_status()函数可以查看所有 Instance 的锁等待信息，通过输出信息，可以确认，哪些事务在等待锁，哪些事务在持有锁，锁资源的类型(锁什么样的对象，例如 relation、row 等)和模式(锁的等级)，等待锁的 SessionID，持有锁的 SessionID，锁的 SegID。例如：

```
=# SELECT * FROM pg_catalog.gp_dist_wait_status();
```

```
-[ RECORD 1 ]----+--------------------
segid            | -1
waiter_dxid      | 28
holder_dxid      | 26
holdTillEndXact  | t
waiter_lpid      | 3196
holder_lpid      | 3109
waiter_lockmode  | AccessExclusiveLock
waiter_locktype  | relation
waiter_sessionid | 18
holder_sessionid | 17
```

当全局死锁检测解除一个死锁，会有如下报错信息：

```
ERROR: canceling statement due to user request: "cancelled by global
deadlock detector"
```

全局死锁检测对 UPDATE 和 DELETE 的并发容许

全局死锁检测可以管理 Heap 表上这些类型的 UPDATE 和 DELETE 命令的并发执行(这些类型之间的容许情况见本节最后的表格)：

- 单表的简单UPDATE。由PostgreSQL优化器执行的，没有FROM子句或者子查询的，非分布键字段的UPDATE。

```
=# UPDATE t SET c2 = c2 + 1 WHERE c1 > 10;
```

- 单表的简单DELETE。命令的FROM或者WHERE子句中没有子查询。

```
=# DELETE FROM t WHERE c1 > 10;
```

- 裂变更新(Split UPDATE -- 在执行计划中的算子叫Split)。PostgreSQL优化器，可以通过UPDATE命令更新分布键字段。

```
=# UPDATE t SET c = c + 1; -- c is a distribution key
```

对于Orca优化器，可以通过UPDATE命令更新分布键字段和分区字段。

```
=# UPDATE t SET b = b + 1 WHERE c = 10; -- c is a distribution key
```

- 复杂更新。UPDATE命令包含多表关联。

```
=# UPDATE t1 SET c = t1.c+1 FROM t2 WHERE t1.c = t2.c;
```

或者包含子查询的 UPDATE。

```
=# UPDATE t SET c = c + 1 WHERE c > ALL(SELECT * FROM t1);
```

- 复杂删除。类似于复杂UPDATE，包含多表关联或者子查询。

```
=# DELETE FROM t USING t1 WHERE t.c > t1.c;
```

下表列举了全局死锁检测管理的 UPDATE 和 DELETE 命令的互相容许情况。涉及哪些命令之间有没有冲突。例如，在同一条记录上的并发简单更新是容许的，并发复杂更新和简单更新，将只有一个 UPDATE 会被执行，另一个 UPDATE 会失败报错。

	简单更新	简单删除	裂变更新	复杂更新	复杂删除
简单更新	YES	YES	NO	NO	NO
简单删除	YES	YES	NO	YES	YES
裂变更新	NO	NO	NO	NO	NO
复杂更新	NO	YES	NO	NO	NO
复杂删除	NO	YES	NO	NO	YES

回收空间

由于MVCC事务并发模型的原因，已经删除或者更新的记录仍然占据着磁盘空间，虽然其对于新的事务来说已经不可见。如果数据库有大量的更新和删除操作，其将会产生大量的过期记录。定期的运行VACUUM命令可以回收这些过期的记录空间。例如：

```
=# VACUUM mytable;
```

VACUUM命令还会收集表级别的统计信息，例如，记录数、占用磁盘页面数，所以在装载数据之后对全表执行VACUUM是有必要的，这条规律同样适用AO表。

编者认为，应该更多的了解VACUUM命令。例如对于系统表来说，应该定期执行VACUUM操作，系统表会因为数据库的DDL操作逐渐膨胀，如果长时间不做VACUUM，系统表的空间会严重膨胀，带来性能问题，甚至导致集群异常(这绝非危言耸听，编者遇到不止一次因为系统表膨胀严重导致的性能问题或集群故障)，所以编者会为客户的集群配置每日定时任务 -- 执行系统表的VACUUM操作。对于太长时间没有做VACUUM的系统表，可能会面临需要花费数小时来运行VACUUM FULL的麻烦。

对于业务Heap表(如果不是的确有必要，尽可能用AO表)，虽然目前的版本VACUUM命令的性能已经有了极大的提升，但还是建议做必要的对比测试，如果有可能，也许做REORGANIZE会有更好的效率。

对于AO表，gp_appendonly_compaction_threshold参数决定了AO表的VACUUM操作的实际行为，对于膨胀比例没有超过阈值的情况，不会真的执行VACUUM操作，不过，如果执行了VACUUM FULL，则会忽略该参数。所以，对于AO表，也可以定期执行VACUUM命令来尝试回收空间，而且在AO表上执行VACUUM命令时，空间将得到回收，并将空间返回给文件系统，这与Heap表不同，Heap表是记录垃圾空间以便重新利用。

配置自由空间映射

注意：从6版本开始，已经不再需要自由空间映射，该章节所讲述的内容是6版本之前的特性，在6版本开始，因为PostgreSQL版本升级，Heap表中可用tuple的信息将在文件中进行存储，6版本之前这些信息是存储在内存中的。

在6版本之前，对于Heap表来说，过期的记录会被存在叫做自由空间映射的地方。自由空间映射的大小必须足够容纳数据库中的所有过期记录。如果尺寸不够大，超出自由空间映射的过期记录占用的空间将无法被VACUUM命令回收。

VACUUM FULL命令将回收Heap表中所有无效记录，并将空间返回给文件系统，但这是一个很昂贵的操作，对于一张大表，该操作可能会花费无法接受的时间长度。如果自由空间映射已经溢出，最好的做法是及时的使用CREATE TABLE AS命令来重建数据表并删除旧的表，也可以使用REORGANIZE来整理表，两者的效率可能是相当的，后者的好处在于，不需要重建表，也不会涉及依赖关系的处理。

最好将自由空间映射设置为一个合适的值。自由空间映射由下面的参数来设置：

```
max_fsm_pages
max_fsm_relations
```

更多信息可参考PostgreSQL相关文档。

第十章：数据查询

本章讲述在GP中如何使用SQL语言对数据库中的数据进行查询。可以使用标准的PostgreSQL命令行工具psql来执行SQL语句，或者使用其他客户端工具来执行SQL语句。

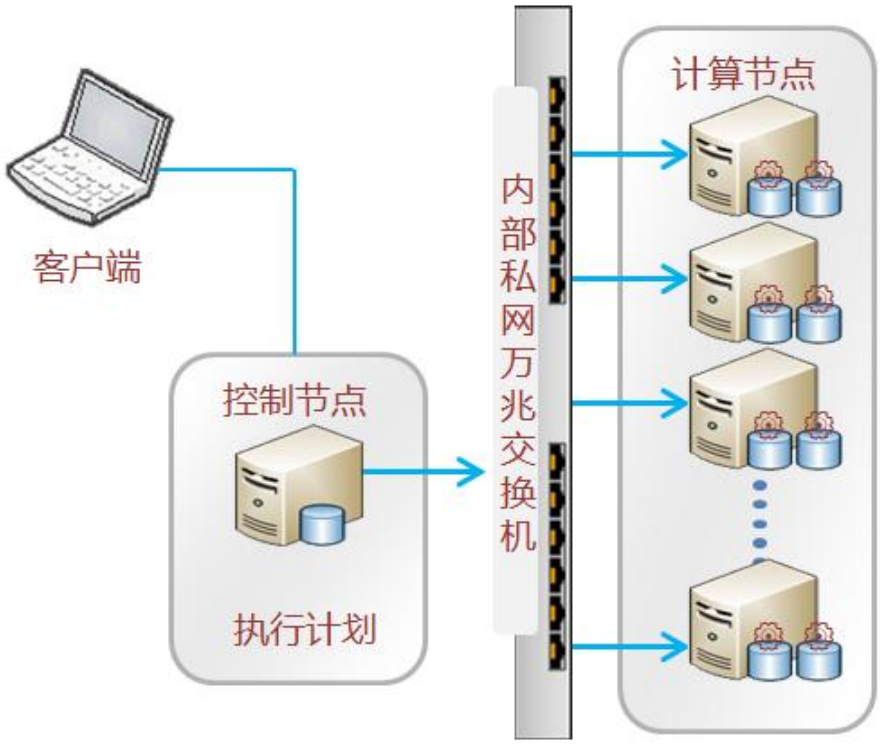
理解 GP 的查询处理

本节讲述 GP 数据库是如何处理查询的。深入理解这些概念，对于写出高效的 SQL 和 SQL 调优将是至关重要的。用户向 GP 数据库提交 SQL 查询的方式与其他数据库是相似的，通过客户端工具(例如 psql)连接到 Master 机器，执行 SQL 语句。

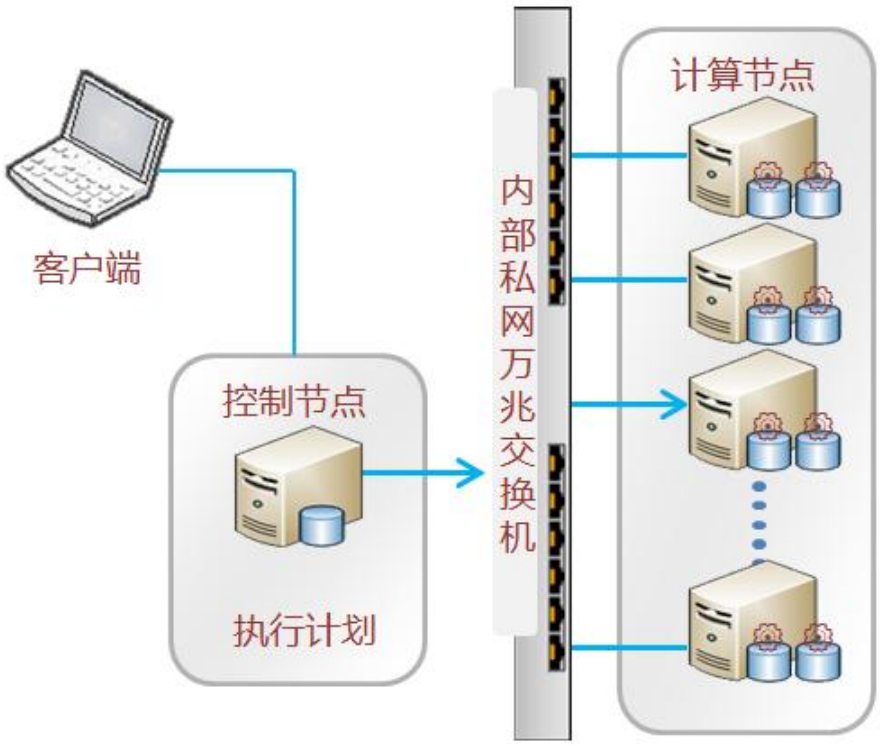
理解执行计划与分发

查询被 Master 接收、处理、优化、创建一个并行的或者定向的执行计划(根据查询语句决定)。之后 Master 将执行计划分发到相关的 Instance 去执行，每个 Instance 只负责处理自己本地的那部分数据,如果需要用到其他 Instance 的数据，执行计划会在每一步的处理之前进行数据移动(Motion)。

大部分的算子 -- 例如表扫描(Scan)、关联(Join)、聚合(Aggregation)、排序(Sort)都是在 Instance 本地被执行，每个 Instance 同时独立执行，不涉及其他 Instance 的资源，当某个算子需要使用其他 Instance 的数据时，执行计划中会被加入一个称为移动(Motion)的算子，以帮助其他算子准备需要的数据。



一些特定的语句可能只使用到一个 Instance，例如单行的 INSERT、UPDATE、DELETE 或者 SELECT 操作，还有一些直接过滤 DK 的查询。这些语句不会被分发到全部 Instance，而是定向的发送到包含该 DK 的 Instance。当一个语句直接查询一张复制表时，该语句只需要得到一个 Instance 的响应。



理解执行计划

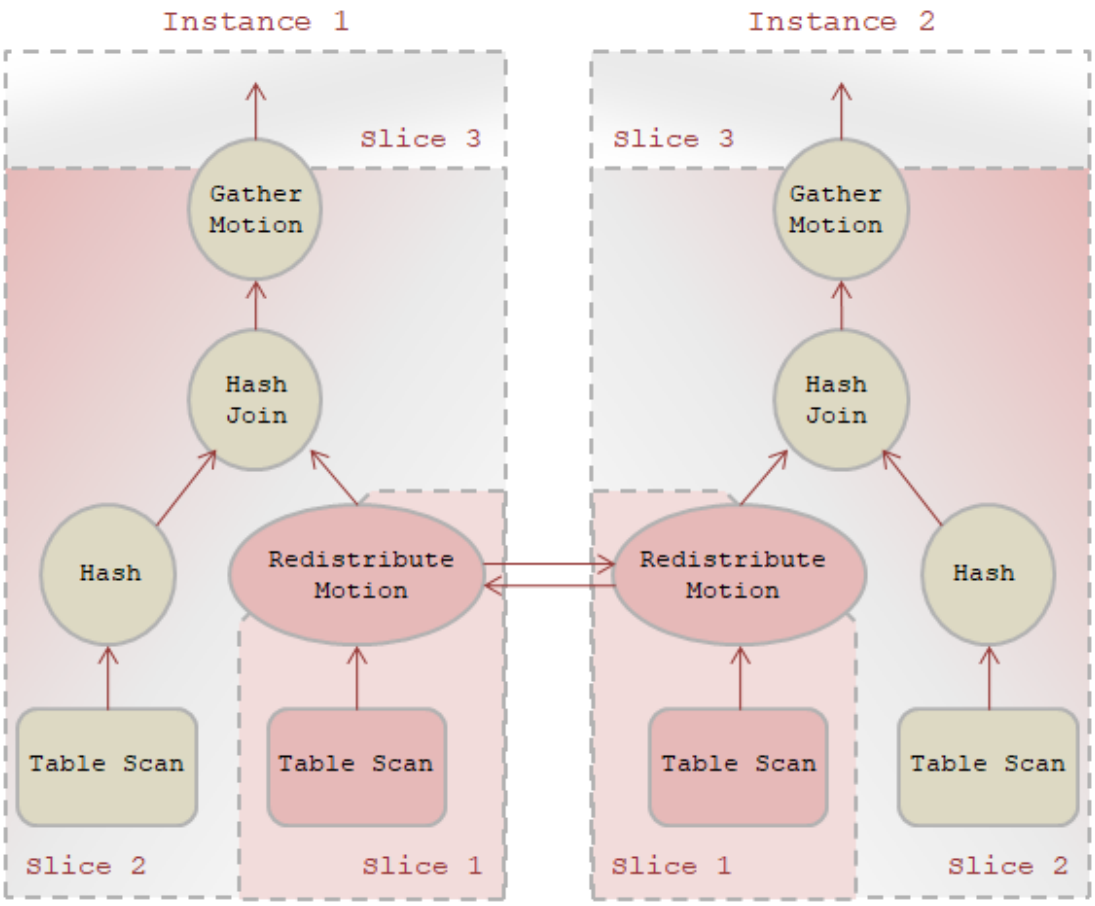
执行计划是GP根据特定的查询语句的处理逻辑，生成的一系列算子的有序集合，执行计划是一个安排，一个规划，将Instance要做哪些运算，这些运算的顺序等安排出来。执行计划的每一步代表着特定的算子，例如：扫表、关联、聚合、排序等。执行计划被从下向上执行。与典型的数据库的处理不同的是，GP有一个特有的算子：移动(Motion)。移动算子(Motion)涉及到查询处理期间在Instance之间移动数据。不过并非所有的查询都需要移动数据。例如针对系统表(在Master上)的查询不会涉及通过内联网络移动数据，因为这些数据只需要从Master获取，不需要访问Instance。

为了最大限度的实现并行化处理，GP会将执行计划分割为多个步骤。每个步骤是执行计划的一部分，其可以在Instance上被独立执行。当需要移动数据时，执行计划会被分割开，两个算子分处在数据移动算子的两侧，即：先执行一步算子，然后执行数据移动，再执行下一步算子。

例如，下面两个Table的关联查询：

```
=# SELECT customer, amount FROM sales JOIN customer USING (cust_id)
   WHERE sale_date = '2020-06-06';
```

下图解释了执行计划。每个Instance获取到执行计划的副本然后并行开始执行。对于该执行计划来说，有一个重分布算子(Redistribute motion)，这是为了完成连接(Join)而执行的数据移动。执行计划被重分布算子分割为两步(slice 1与slice 2)。该执行计划还有另外一种数据移动称为汇总移动(Gather motion)，汇总是Instance将计算结果反馈到Master从而可以反馈给客户端的一种操作。由于这是一个SELECT操作，所以会有汇总算子(slice 3)。不是所有的查询都有汇总算子，例如：CREATE TABLE . . . AS SELECT . . . 语句就不需要汇总算子。



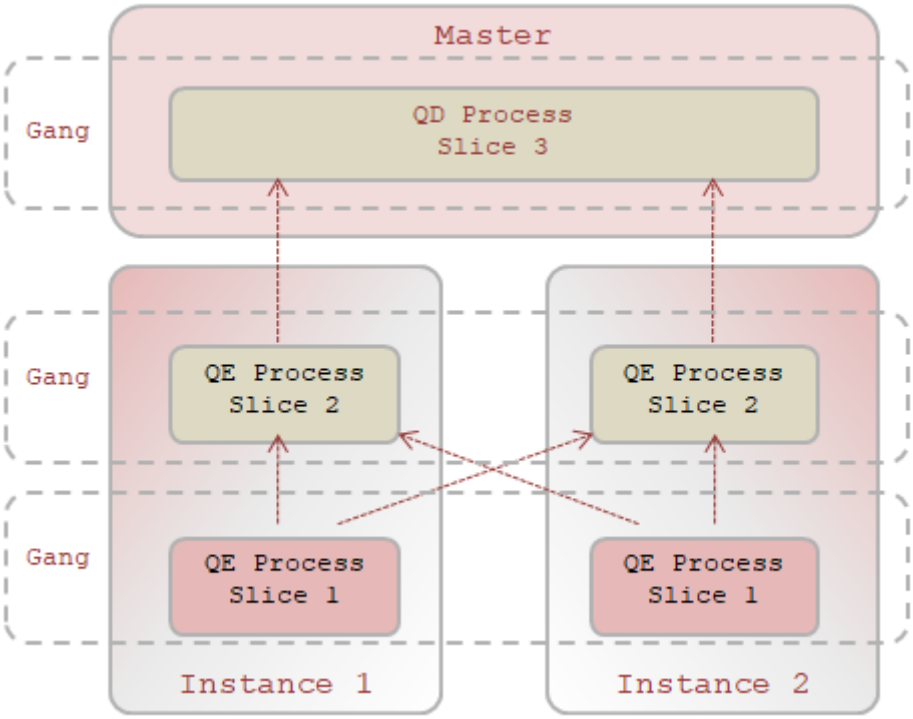
理解并行执行

GP 会创建多个 DB 进程来处理查询。在 Master 上被称为查询分发器(Query Dispatcher/QD)。QD 负责创建、分发执行计划，汇总反馈最终结果。在 Instance 上，处理进程被称为查询执行器(Query executor/QE)。QE 负责完成自身部分的处理工作以及与其他 QE 之间交换可能需要移动的中间结果数据。

执行计划的每个处理部分都至少涉及一个处理工作。执行进程只处理属于自己部分的工作。在查询被执行期间，每个 Instance 会并行的执行一系列的处理工作。

同一部分相关的处理工作被称为簇(Gangs，这是编者的翻译，想不出更好的翻译，所以一直沿用至今，实际上这个概念对于日常使用来说并不重要)。在一部分处理完成后，数据将从当前处理向上传递，直到执行计划被完成。Instance 之间的通信涉及到 GP 的内联网络组件。

下图展示查询处理如何在 Master 和 2 个 Instance 之间被逐步执行的：



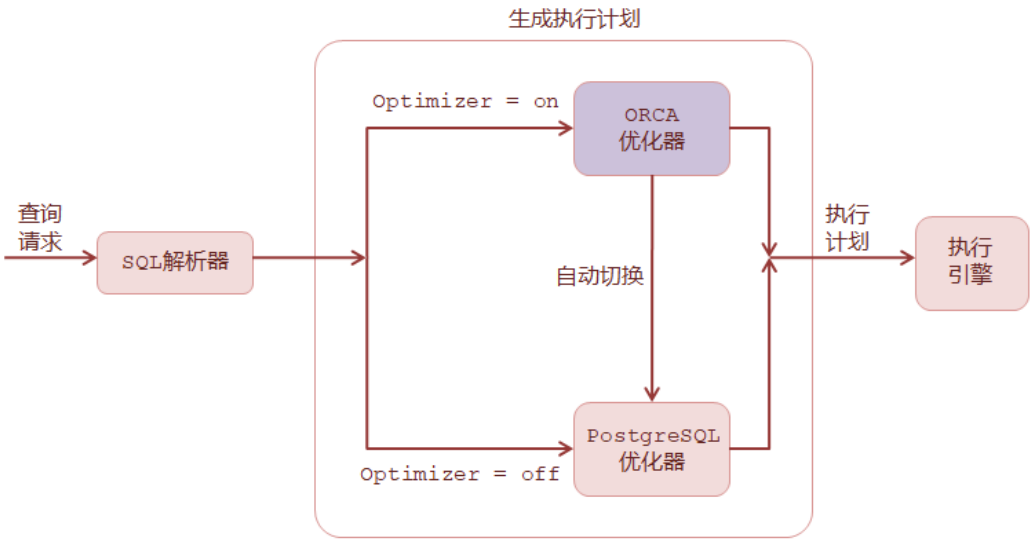
关于 ORCA 优化器

在 GP 数据库中，目前 Orca(optimizer 参数控制)优化器与 PostgreSQL 优化器共存，不是取代关系(因为取代不了，以前可能想过要取代)。在一些复杂 SQL 场景，还有一些多分区的分区表场景，Orca 有时会更有优势。目前，在 5 版本和 6 版本中，Orca 是缺省打开的，数据库会尽可能使用 Orca 优化器，对于 Orca 优化器还不能处理的场景，会自动切换到 PostgreSQL 优化器。

Orca 优化器在以下场景会表现出更好的性能：

- 针对分区表的查询(如果是多级分区表，必须是规整的)。
- 包含子查询的查询。
- 包含CTE的查询。
- DML操作(主要是功能的增强)。

目前，Orca 优化器与 PostgreSQL 优化器共存，缺省使用 Orca 优化器，如果 Orca 优化器不适用，会自动切换到 PostgreSQL 优化器。下图显示，Orca 优化器和 PostgreSQL 优化器的共存逻辑：



注意：在使用 Orca 优化器时，Orca 优化器在生成执行计划时会忽略所有 PostgreSQL 的执行计划参数，Orca 有自己的一套参数。对于 Orca 搞不定的 SQL，自动切换到 PostgreSQL 优化器时，PostgreSQL 的执行计划参数将会被使用，这些 PostgreSQL 的执行计划参数一般是以 enable_开头或者 gp_enable_开头，一般分别在 guc.c 和 guc_gp.c 文件中可以找到这些参数。

启用或禁用 Orca

在 5 版本和 6 版本中，Orca 是缺省启用状态，可以通过参数 optimizer 来设置启用或者禁用 orca 优化器。

可以在不同的级别来设置是否开启 Orca，例如，系统级别，数据库级别，ROLE 级别，会话级别，语句级别，按照这些级别来看，级别越低，优先级越高，例如，在 Database 级别的设置会覆盖系统级别的设置，依次类推，所有的参数(当前优先级可以设置的话)都遵循这种优先级的规则。

注意：可以通过参数 optimizer_control 来禁止修改 optimizer 的设置，这个参数的缺省值为 on，只有 SUPERUSER 可以修改。不过编者想说，慎用这个参数。

在系统级别开启 Orca

1、使用 gpadmin 用户登录 Master Host 主机。

2、使用 gpconfig 命令来设置 optimizer 参数：

```
$ gpconfig -c optimizer -v on --masteronly
```

编者认为，是否需要指定--masteronly 选项，并不重要。

3、重新加载 postgresql.conf 配置文件，使修改生效，此处不是真正停止数据库的操作，这类似 PostgreSQL 的 pg_ctl reload 操作。

```
$ gpstop -u
```

要在系统级别禁用 Orca，类似以上的操作，将 optimizer 参数设置为 off 即可。

在数据库级别开启 Orca

要在指定的数据库范围来设置启用 Orca，可以使用 ALTER DATABASE name SET 命令来实现。例如：

```
=# ALTER DATABASE test_db SET OPTIMIZER = ON ;
```

要在数据库级别禁用 Orca，将 optimizer 参数设置为 off 即可。例如：

```
=# ALTER DATABASE test_db SET OPTIMIZER = OFF ;
```

还可以恢复缺省设置，即，该级别不做设置，而是从系统级别继承。例如：

```
=# ALTER DATABASE test_db RESET OPTIMIZER;
```

在 ROLE 级别开启 Orca

要在指定的 ROLE 来设置启用 Orca，可以使用 ALTER ROLE name SET 命令来实现。例如：

```
=# ALTER ROLE user1 SET OPTIMIZER = ON ;
```

要在 ROLE 级别禁用 Orca，将 optimizer 参数设置为 off 即可。例如：

```
=# ALTER ROLE user1 SET OPTIMIZER = OFF ;
```

还可以恢复缺省设置，即，该级别不做设置，而是从更高的别继承。例如：

```
=# ALTER ROLE user1 RESET OPTIMIZER;
```

另外，在 6 版本中，还可以在 ROLE 级别针对特定的数据库进行参数设置，这些信息存储在 pg_db_role_setting 系统表中。例如：

```
=# ALTER ROLE user1 IN DATABASE test_db SET OPTIMIZER TO OFF;
```

在会话级别或者语句级别启用 Orca

除了之前的多个级别可以设置参数外，和很多其他参数一样，可以在会话和语句级别进行设置，通过 SET 命令来完成，可以在一组 SQL 语句的任何一个语句之前进行设置，SET 设置之后，将影响后续的语句。例如：

```
=# SET OPTIMIZER TO ON;
```

收集 ROOT 分区的统计信息

对于分区表，Orca 必须使用 ROOT 分区的统计信息来生成执行计划，这些信息非常重要，这将决定了 Orca 会如何评估每个算子的代价(Cost)，选择 Join 的顺序，如何执行 AGG 操作。但对于 PostgreSQL 优化器而言，使用的是叶子分区的统计信息。

如果使用 Orca 来查询分区表，ROOT 分区上需要有统计信息，并且这些信息需要保持更新，这样 Orca 才能生成更准确的执行计划。如果 ROOT 分区的统计信息过于陈旧或者缺失，ORCA 仍然会选择动态分区评估，但可能会生成很差的执行计划。这不等于说必须定期在 ROOT 分区上收集统计信息，因为这个代价往往会很大，后面会介绍关于叶子分区统计信息上透的内容。

执行 ANALYZE 命令

缺省情况下，直接在ROOT分区上执行ANALYZE命令，会自动为所有叶子分区收集统计信息，也会为ROOT分区收集统计信息，同时还会收集叶子分区的频度统计信息(HyperLogLog)，个人理解，是一种可增量更新的统计信息，后面会提到有关叶子分区的统计信息，自动上透到ROOT分区。ANALYZE ROOTPARTITION命令将只收集ROOT分区的统计信息。通过参数optimizer_analyze_root_partition可以控制是否需要ROOTPARTITION关键字来收集ROOT分区的统计信息，不过，这个情况，从编者的测试来看，不同的版本表现的情况不同，optimizer_analyze_root_partition参数和gp_statistics_pullup_from_child_partition参数关闭之后，在6版本中没有使用关键字ROOTPARTITION来ANALYZE ROOT分区，同样会更新ROOT分区的统计信息，而在5版本，关闭这两个参数之后，的确不会更新ROOT分区的统计信息，在明确使用了ROOTPARTITION关键字之后，会更新ROOT分区的统计信息。不过编者认为，这里不需要过多的考虑，使用缺省的配置即可，这样，叶子分区的统计信息更新会自动上透到ROOT分区，如果这时Orca生成的执行计划有问题，再尝试收集ROOT分区的统计信息或关闭Orca。

ANALYZE 在收集 ROOT 分区统计信息时会扫描全部叶子分区的数据，如果该分区表非常大，这将是一个极其耗时的操作。ANALYZE 需要 ShareUpdateExclusive 锁，与很多操作是冲突的，例如 TRUNCATE 和 VACUUM，和 ANALYZE 本身也是冲突的，所以，应该合理安排 ANALYZE 的时机，例如在业务空闲时期，最佳的选择是在业务处理过程中，在数据发生大规模修改之后及时进行 ANALYZE 操作。

以下是一个关于在分区表上执行 ANALYZE 和 ANALYZE ROOTPARTITION 的最佳实践：

- 一个分区表在完成数据初始化后执行ANALYZE <root_partition>命令。一个新的叶子分区在完成数据初始化后，或者发生大规模数据变化后，执行ANALYZE <leaf_partition>命令。缺省情况下，在叶子分区收集统计信息，会自动上透到ROOT分区。
- 在长期没有更新ROOT分区的统计信息之后，如果在执行计划中发现执行计划变差了，或者叶子分区有大规模的数据变化之后，可能自动上透的统计信息会有失准，此时应该更新ROOT分区的统计信息。例如，在上一次收集ROOT分区统计信息之后，增加了很多叶子分区，导致了大规模的数据变化，此时应该考虑执行ANALYZE 或者ANALYZE ROOTPARTITION命令来更新ROOT分区的统计信息。
- 对于超级大表，应该把ANALYZE或者ANALYZE ROOTPARTITION的间隔设置很长，例如每次间隔一周或者更久。所有表的统计信息收集间隔都取决于该表中的数据变化比例和频繁程度。
- 避免执行ANALYZE命令而不带任何参数，因为这样会对数据库中的所有表收集统计信息，包括所有分区，对于一个大规模的数据库，这种操作的耗时是不可控的。
- 如果IO资源充足，可以并行执行ANALYZE <table_name>或者ANALYZE ROOTPARTITION <table_name>命令来加速收集统计信息的速度。
- 可以使用GP自带命令analyzedb来更新表的统计信息，该命令会自动判断一张表

从上次收集了统计信息之后到目前为止是否发生了变化，如果没有发生变化，就不用重新收集统计信息，这个自动判断的粒度可以精确到每个叶子分区。

Orca和叶子分区的统计信息

对于在分区表上使用Orca优化器来说，维护ROOT分区的统计信息非常重要，这将直接决定了Orca生成的执行计划的优劣，不过，维护叶子分区的统计信息同样很重要。有些查询可能无法适用Orca优化器，此时就会使用PostgreSQL优化器，而PostgreSQL优化器是必须使用叶子分区的统计信息来生成执行计划的。对于直接查询叶子分区的查询，Orca也需要使用叶子分区的统计信息来生成执行计划。例如，当明确知道需要查询的数据在特定的叶子分区时，可以直接查询该叶子分区表，这种情况下，Orca也需要使用该叶子分区的统计信息。

禁用自动收集ROOT分区统计信息

如果不打算使用Orca来查询分区表(设置参数optimizer为off)，则可以禁用ROOT分区的统计信息自动收集，通过optimizer_analyze_root_partition参数来控制是否需要明确的指定ROOTPARTITION关键字来收集ROOT分区的统计信息。缺省值为ON，ANALYZE一个分区表的ROOT分区时，不需要指定ROOTPARTITION关键字，会自动收集ROOT分区的统计信息，通过设置该参数为OFF来关闭这种自动收集。如果关闭了这个参数，则需要指定ROOTPARTITION关键字才能收集ROOT分区的统计信息。不过，正如编者在前面提到的，这个情况还是以实际的测试为准，至少目前测试下来，5版本的表现符合这个表述，6版本的表现不符合这个描述。

1、 使用 gpadmin 用户登录 Master Host 主机。
2、 使用 gpconfig 命令来设置 ptimizer_analyze_root_partition 参数：

```
$ gpconfig -c ptimizer_analyze_root_partition -v off --masteronly
```

编者认为，是否需要指定--masteronly 选项，并不重要。

3、 重新加载 postgresql.conf 配置文件，使修改生效，此处不是真正停止数据库的操作，这类似 PostgreSQL 的 pg_ctl reload 操作。

```
$ gpstop -u
```

使用 Orca 的注意事项

要用好Orca，需要注意以下事项：

- 分区表中不能有多字段分区键。
- 多级分区表必须是规整的(每个分区的子分区结构完全相同，例如第一个分区有3个子分区，第二个分区有4个子分区，这种就不是规整的多级分区表)。
- 需要确保optimizer_enable_master_only_queries参数设置为ON，才能在查询只在Master上存在的系统表时使用Orca优化器。实际上，并不需要启用该参数，因为带来的坏处比好处更大。
- ROOT分区上已经收集了统计信息。

注意：打开 optimizer_enable_master_only_queries 参数会降低针对系统表的短查询性能，请只在必要时在会话中打开。

如果一个分区表包含了超过 20000 个分区，应该考虑重新规划表的设计。编者认为超过 500 个分区就已经很不正常了。分区粒度要适中，不能太随意，否则后患无穷。

以下这些参数将会影响 Orca 如何生成执行计划：

- optimizer_cte_inlining_bound -- 编者没有理解这个参数的具体含义，编者查找了github上的测试用例，对每个测试场景进行了测试，不管该参数设置为多少，所有测试用例的执行计划是否走Orca和该参数的开关没有任何关系。
- optimizer_force_multistage_agg -- 强制Orca选择多阶段聚合，该参数在5版本的缺省值为TRUE，在6版本的缺省值为FALSE。为FALSE时，由Orca根据Cost评估，选择一阶段聚合或二阶段聚合。编者认为，三阶段聚合的适用面更广。
- optimizer_force_three_stage_scalar_dqa -- 强制Orca选择三阶段聚合。该该参数缺省值为TRUE，建议不要修改。
- optimizer_join_order -- 设置Orca对Join的顺序调整的激进程度，有四个可选级别，分别为query、greedy、exhaustive、exhaustive2，缺省的级别为exhaustive，各个级别大概的含义分别是，不调整、尝试、努力、穷举。通常，不建议降低该参数的级别，因为可能会导致很多查询无法找到最优的执行计划。
- optimizer_join_order_threshold -- Orca对Join尝试调整顺序的Join数量的上限。缺省值为10，编者认为，Join的表的数量超过该参数设置的值之后，Orca不会再尝试调整Join的顺序，这与一些人的理解有出入，有些人的理解是，超过10张表Join时，Orca将会按照每组不超过10张表进行分组优化，编者的测试显示，超过10张表的Join，生成执行计划的结果和耗时，与把该参数设置为1是完全一致的。而对于PostgreSQL的类似参数join_collapse_limit，则更

符合分组优化的解释。Join的表数量过多时，的确无法穷举所有排序的可能性，因为N张表Join顺序的穷举数量是N的阶乘，数量级增长过快，CPU无法处理。

- optimizer_nestloop_factor -- 设置Orca评估Nestloop Join时的Cost因子。缺省值为1024，该值越小，越容易选择Nestloop Join，例如在IOPS能力非常好的SSD系统，可以缩小该参数提高选择索引关联的可能性。
- optimizer_parallel_union -- 是否对UNION和UNION ALL执行并行扫描。缺省值为OFF，当设置为ON时，Orca将可以针对UNION和UNION ALL的不同部分执行并行扫描，任务会被拆分成更多的Slice，执行计划也会更复杂，有时未必会带来性能的提升，建议不要随意修改。
- optimizer_sort_factor -- 设置Orca选择排序时的成本因子，缺省值为1。最小值为0，越大的值越可能避免走排序。
- gp_enable_relsize_collection -- 设置当一张表没有统计信息时，Orca和PostgreSQL优化器如何处理该表。缺省情况下，Orca使用一个默认值来(一般为1条记录)作为没有统计信息的表的统计信息评估结果，如果该参数设置为ON，Orca会通过该表的尺寸来评估统计信息。

  对于一个分区表来说，Orca查询ROOT分区时，如果ROOT分区上没有统计信息，Orca并不会去评估表的尺寸，因为ROOT表中没有数据，而是继续使用默认值。PostgreSQL优化器会获取叶子分区的尺寸。这种情况下可能需要使用ANALYZE ROOTPARTITION命令来收集ROOT分区的统计信息。

以下参数控制着Orca的日志信息的输出：

- optimizer_print_missing_stats -- 控制在Orca执行一个查询时，其中的字段缺失统计信息时是否显示字段的信息。不过编者实测，目前的5版本和6版本都不会有相关信息或者日志输出，而且，也的确不需要这个信息。
- optimizer_print_optimization_stats -- 控制Orca是否输出对执行计划进行优化的过程计量信息，包括一些资源的消耗和耗时等，主要用于跟踪Orca的性能。缺省为FALSE。

使用 Orca 时还可以通过 optimizer_minidump 参数来控制生成 Orca 的 minidump 文件，该文件会存储在 Master 工作目录下的 minidumps 子目录中，该文件不是一个平面文件，主要用于原厂支持人员分析诊断问题，例如：

```
Minidump_20200618_222043_11_33.mdp
```

该参数的缺省值为ONERROR，设置为ALWAYS则会每次生成minidump文件。该参数可以在Session中设置生效。

使用 Orca 执行 EXPLAIN 或者 EXPLAIN ANALYZE 命令时，执行计划只会显示分区消除的数量，并不会显示详细的分区信息，号称设置参数 gp_log_dynamic_partition_pruning 为 ON 就可以显示，经过编者实测目前的 5 版本和 6 版本，并不会显示这些信息。

Orca特性与增强

Orca，号称是GP的下一代查询优化器，在某些查询和算子的场景中优势明显。

- 针对分区表的查询。
- 包含子查询的查询。
- 包含CTE的查询。
- DML操作的增强。

在有些方面Orca也有改进：

- Join顺序的调整。
- 关联聚合的顺序调整。
- Sort顺序的优化。
- 对数据倾斜的评估。

针对分区表的查询

Orca对于分区表的查询有如下的增强：

- 分区消除得到改善。
- 可以支持规整的多级分区表(一般不存在多级分区表，所以无所谓是否规整)。
- 执行计划可以包含一个算子作为动态分区消除的条件，对于常量分区条件，执行计划会给出分区消除的数量信息，非常量分区条件则不会给出分区消除的数量信息。
- 执行计划中不会枚举所有分区(动态分区扫描，整体作为一个算子)。

  对于常量分区过滤条件，Orca会在执行计划中列出分区消除的数量。例如：

```
->  Partition Selector for sales (dynamic scan id: 2)  . . .
    Partitions selected: 1 (out of 12)
->  Dynamic Table Scan on sales (dynamic scan id: 2)  . . .
    Filter: date = '2020-01-02'::date
```

  对于分区过滤条件是一个算子的情况，具体的分区消除的数量只有在执行时才能知道，所以，Orca不会在执行计划中列出分区消除的数量，但在EXPLAIN ANALYZE的输出中会有体现。例如，下面是EXPLAIN ANALYZE的一部分：

```
->  Dynamic Table Scan on sales (dynamic scan id: 1)  . . .
Rows out:  Avg 105.0 rows x 2 workers.  . . .
Partitions scanned:  Avg 1.0 (out of 12) x 2 workers.  Max 1 parts (seg0).
```

- 执行计划的尺寸与分区数量无关。
- 大大缓解了因为分区数量引起的内存不足的报错。

例如，使用 CREATE TABLE 命令创建了一张 RANGE 分区表：

```
CREATE TABLE sales (
    id int,
    date date,
    amt decimal(10,2)
)DISTRIBUTED BY (id) PARTITION BY RANGE (date) (
    START (date '2020-01-01') INCLUSIVE
    END (date '2021-01-01') EXCLUSIVE EVERY (INTERVAL '1 month')
);
```

Orca 改进了该分区表的这些查询场景：

- 全表扫描的查询，执行计划中不会枚举所有分区。

```
=# SELECT * FROM sales;
```

- 使用常量作为分区过滤条件，执行计划中显示了分区消除的信息。

```
=# SELECT * FROM sales WHERE date = '2020-01-02';
```

- 使用范围过滤条件，执行计划中显示了分区消除的信息。

```
=# SELECT * FROM sales WHERE date BETWEEN '2020-01-02' AND '2020-01-03';
```

- 通过一个子查询来指定分区过滤条件，执行计划中，只显示动态分区消除，而不显示分区消除的数量。

```
=# SELECT * FROM sales WHERE date =
    (SELECT max(date+100) FROM sales WHERE date='2020-01-02');
```

包含子查询的查询

Orca可以更好的处理子查询，子查询是嵌套在一个查询中的查询语句，例如在WHERE查询中的SELECT就是一个子查询。例如：

```
=# SELECT * FROM part
   WHERE price > (SELECT avg(price) FROM part);
```

Orca可以更好的处理关联子查询，关联子查询是，在子查询中引用了外部查询的字段，例如下面这个查询，brand字段在子查询中被引用：

```
=# SELECT * FROM part p1
   WHERE price > (
      SELECT avg(p2.price) FROM part p2
      WHERE p2.brand = p1.brand
   );
```

Orca为以下类型的子查询生成更高效的执行计划：

- 查询的字段列表中包含关联子查询。

```
=# SELECT *,
    (SELECT min(p2.price) FROM part p2 WHERE p1.brand = p2.brand) AS foo
FROM part p1;
```

- 在OR条件中包含关联子查询。

```
=# SELECT FROM part p1 WHERE p1.p_size > 40 OR p1.p_retailprice > (
   SELECT avg(p2.p_retailprice)
   FROM part p2
   WHERE p2.p_brand = p1.p_brand
);
```

- 跨级关联的嵌套子查询(内层的子查询引用了更外层查询的字段 -- 跨级引用)。

```
=# SELECT * FROM part p1 WHERE p1.p_partkey IN (
   SELECT p2.p_partkey FROM part p2
   WHERE p2.p_retailprice = (
      SELECT min(p3.p_retailprice) FROM part p3
      WHERE p3.p_brand = p1.p_brand
   )
);
```

注意：PostgreSQL优化器不支持这种越级关联的嵌套子查询。

- 包含AGG和非等值关联的关联子查询。

```
=# SELECT * FROM part p1 WHERE p1.p_retailprice = (
    SELECT min(p2.p_retailprice) FROM part p2
    WHERE p2.p_brand <> p1.p_brand
);
```

- 关联子查询中的子查询对于每个关联值只能返回单条记录。

```
=# SELECT p_partkey,
(SELECT p2.p_retailprice FROM part p2 WHERE p2.p_brand = p1.p_brand)
FROM part p1;
```

注意：如果子查询对于每个关联值返回多条记录，将会在运行时期间报错，SQL语法并不会报错。

包含 CTE 的查询

Orca 增强了对包含 WITH 子句的查询的支持。这种查询也被称为 CTE，WITH 中的语句块可以类似临时表一样来使用。例如：

```
=# WITH v AS (SELECT a, sum(b) as s FROM T where c < 10 GROUP BY a)
SELECT *FROM v AS v1 , v AS v2
WHERE v1.a <> v2.a AND v1.s < v2.s;
```

Orca 还可以将条件下推到 CTE 中。例如：

```
=# WITH v AS (SELECT a, sum(b) as s FROM T GROUP BY a)
SELECT * FROM v as v1, v as v2, v as v3
WHERE v1.a < v2.a AND v1.s < v3.s
    AND v1.a = 10 AND v2.a = 20 AND v3.a = 30;
```

Orca 还可以处理这些类型的 CTE：

- 定义了多个AS的CTE。例如：

```
=# WITH cte1 AS (
```

```
    SELECT a, sum(b) as s FROM T where c < 10 GROUP BY a
), cte2 AS (
    SELECT a, s FROM cte1 where s > 1000
)
SELECT * FROM cte1 as v1, cte2 as v2, cte2 as v3
WHERE v1.a < v2.a AND v1.s < v3.s;
```

- 嵌套CTE

```
=# WITH v AS (
    WITH w AS (
        SELECT a, b FROM t WHERE b < 5
    )
    SELECT w1.a, w2.b FROM w AS w1, w AS w2
    WHERE w1.a = w2.a AND w1.a > 2
)
SELECT v1.a, v2.a, v2.b
FROM v as v1, v as v2
WHERE v1.a < v2.a;
```

注意：编者提醒，这些 CTE 像不像小学生写的，切记不要在业务中写出一堆这种非等值关联的查询，如果不懂数据库就不要乱给业务写 SQL，害人害己。

DML 操作的增强

Orca 还增强了 DML 操作，例如 INSERT、UPDATE 和 DELETE。

- 在执行计划中，DML也是一个算子。
  - 作为一个常规的算子，可以出现在执行计划的任何地方(目前只在顶端)。
  - 可以有后续算子(这与上一条的注释矛盾啊，可能目前还不存在吧)。
- UPDATE操作通过Split算子来支持如下操作：
  - 在分布键字段执行UPDATE命令。
  - 在分区字段执行UPDATE命令。

  例如，这是一个 Split 算子的例子：

```
Update  (cost=0.00..431.13 rows=1 width=1)
  ->  Result  (cost=0.00..431.00 rows=1 width=34)
        ->  Split  (cost=0.00..431.00 rows=1 width=30)
              ->  Result  (cost=0.00..431.00 rows=1 width=30)
```

```
                  ->  Seq Scan on t  (cost=0.00..431.00 rows=1 width=26)
Optimizer: Pivotal Optimizer (GPORCA)
```

官方文档说，引入了一个新的算子Assert用于约束检查，编者就目前最新的5和6版本测不出这种算子，而是一堆Result算子，编者认为，这一块可能已经改了。

Orca带来的改变

跟PostgreSQL优化器相比，Orca优化器带来了一些新的变化(有些是依然没有支持)。

- 允许在分布键字段上执行UPDATE命令(在6版本，PostgreSQL优化器也已经支持了，在6版本之前，只有Orca优化器可以支持)。
- 允许在分区字段上执行UPDATE命令。
- 支持对规整(每个分区的子分区结构完全相同)的多级分区表的查询。
- 分区表的叶子分区有外部表的，Orca会自动切换成PostgreSQL优化器。
- 除了INSERT命令，不允许直接对分区表进行DML操作。

  可以直接对叶子分区执行 INSERT 操作，仍需要满足分区条件的约束，非叶子分区不允许执行 INSERT 操作。

- ROOT分区需要统计信息来帮助Orca生成正确的执行计划。
- 在执行计划中引入了新的算子：
  - Partition selector算子 -- 支持动态分区扫描。
  - Split算子 -- 支持对分布键字段和分区字段的UPDATE。
- 使用Orca生成的执行计划与PostgreSQL优化器不同(最后一行会标识)。

的确带来了很多改变，有时不知道该用 Orca 优化器，还是用 PostgreSQL 优化器，二者皆有长短，Orca 可以在一些场景有提升，总归是一件好事吧。

Orca的限制

Orca 不是万能的，目前，GP 数据库中，Orca 优化器和 PostgreSQL 优化器并存，因为，Orca 在一些场景有优势，但是又不能完全搞定所有问题，所以，只能共存，编者也不知道 Orca 的未来在何处。

- 不支持的SQL类型。
- 性能差异。

不支持的 SQL 类型

Orca 不支持的 SQL 类型：

- 参数化的Prepared statement(用Java连过数据库的都清楚这是啥)。
- 表达式索引(Orca不会为此选择索引扫描，也不切换到PostgreSQL优化器)。
- SP-GiST索引。Orca支持的索引类型为：B-tree、bitmap、GIN和GiST。对于不支持的索引，Orca是当作其不存在，例如上一条。
- 不规整的多级分区表(每个分区的子分区结构不完全相同)。
- 叶子分区有外部表的分区表。
- 表名前带有ONLY关键字的SELECT、UPDATE和DELETE操作。

注意：在官方文档中提到的一些其他的限制，编者觉得微不足道或者没能测试验证的，此处不做列举。

性能差异

一些 Orca 和 PostgreSQL 优化器已知的性能差异：

- Orca的短查询性能不好，因为Orca生成执行计划的开销比起PostgreSQL优化器来说，大很多，虽然在5.20左右的版本有了较大提升，但相比于后者还是有极大的差异，虽然这些差异对于复杂计算(例如耗时在分钟级别或以上)不算什么，但对于原本只需要几十毫秒的短查询来说，不可忽略。
- ANALYZE -- 对于Orca，针对ROOT分区的ANALYZE会收集ROOT分区的统计信息，而PostgreSQL优化器不会，因为其不需要这个统计信息。
- DML操作 -- Orca增强了对分布键字段和分区字段的更新支持，但也需要更多的前置(overhead姑且就是这个含义吧)开销。

注意：Orca 生成执行计划的耗时，的确比 PostgreSQL 优化器多很多，这个是一个已知的问题。

验证查询是否使用了 Orca

可以通过检查EXPLAIN的输出来验证,当执行一个GP的查询时,是否使用了Orca优化器。

例如在Orca开启的情况下，下面的例子表示，查询使用了Orca优化器：

```
Settings: optimizer=on
Optimizer status: Pivotal Optimizer (GPORCA) version . . .
```

例如下面的例子表明，在Orca开启的情况下，自动切换到了PostgreSQL优化器：

```
Settings: optimizer=on
Optimizer status: Postgres query optimizer
```

例如下面的例子表明，在Orca关闭的情况下，使用了PostgreSQL优化器：

```
Settings: optimizer=off
Optimizer status: Postgres query optimizer
```

注意：关于Orca，编者就讲到这里，想了解更多知识，请参考官方文档。

定义查询

查询是一个查看、修改或者分析数据库中数据的命令。本节介绍如何在GP中构造SQL查询。

- SQL修辞
- SQL值表达式

SQL 修辞

SQL(结构化查询语言)是用来访问数据库的一种语言。SQL语言有特定的修辞和词法(单词、特征等)，据此构造数据库引擎可以理解的查询或命令。

SQL由一系列的命令组成。命令由一系列按照语法规范编写的修辞组成，以分号(;)结尾。

GP基于PostgreSQL，并遵循相同的SQL结构和语法(一些MPP相关的有差异)。大多情况下，GP的语法与PostgreSQL对等，不过在GP中有些命令可能会有增量或者语法限制(由MPP的特性决定的)。

关于PostgreSQL中SQL规则和概念的完整解释，参考相关PostgreSQL的相关文档。

SQL值表达式

SQL值表达式，由数值、符号、运算符、SQL函数和数据组成。使用表达式来做数据的比较，或执行运算。运算包括逻辑运算、算术运算和集合运算等。

以下这些都是值表达式(这一块编者也不能逐个确定示例，姑且先这样)：

- 聚合表达式
- 数组构造函数
- 字段的引用
- 常量或字符串
- 关联子查询
- 字段选择表达式
- 函数调用
- 插入或更新字段的新值
- 涉及字段引用的运算符
- 位置参数引用 -- 例如在函数定义中或者Prepared statement中
- 行构造函数
- 标量子查询
- WHERE子句中的查询条件
- SELECT命令的字段列表
- 类型转换
- 括号中的值表达式 -- 用于分组子表达式或设置优先级
- 开窗表达式

还有一些SQL结构，例如函数和运算符，也是表达式，但其遵循一般的语法规则。可参考”使用函数和运算符”章节。

字段的引用

引用字段的格式为：

```
correlation.columnname（对象名.字段名）
```

对象名可以是一张表(有时还需要模式名)，或视图，或者一个FROM子句中的对象

的别名，或者在RULE中的NEW、OLD关键字，用于表示对新数据、旧数据的引用。当字段名在当前查询的所有表中是唯一的，”对象名.”这部分可以省略，因为不会有歧义。

位置参数

位置参数指的是SQL语句或者函数的参数，通过参数的位置来引用参数。 例如，$1指的是第一个参数，$2是第二个参数，$3是第三个参数，以此类推。这些参数的值，是在SQL之外设置的(例如Prepared statement对象的一系列setValue的操作)，或者在调用函数时指定的。例如在sql语言的function中，参数的值是在调用函数时指定，此时，位置参数指向的函数定义之外(调用时指定)的值，参数引用的格式为：

```
$number
```

例如：

```
=# CREATE FUNCTION dept(text) RETURNS dept
AS $$
SELECT * FROM dept WHERE name = $1
$$ LANGUAGE sql;
```

这里，在函数被调用时$1就是函数的第一个参数的值。

下标表达式

在访问数组类型数据的特定位置的元素时，可以使用这样格式的表达式：

```
expression[subscript]
```

还可以直接获取数组中的连续元素子集。例如：

```
expression[lower_subscript:upper_subscript]
```

每个下标，必须是一个结果为整数值的表达式，可以是一个常量，或者一个函数，一个子查询，只要结果是一个整数即可。

通常，数组表达式必须在括号中，但是，当使用下标的表达式是字段的引用或位置

参数时，可以省略括号。当原始数组是多维数组时，可以连续使用多个下标。例如：

```
mytable.arraycolumn[4]
```

```
mytable.two_d_column[17][34]
```

```
$1[10:42]
```

```
(arrayfunction(a,b))[42]  --这里的括号不可以省略
```

字段选择表达式

当一个表达式产生的是复合类型(row type)，可以使用字段选择表达式来获取该复合类型的特定属性(或者称为字段也可以)。例如：

```
expression.fieldname
```

通常，复合类型表达式必须被括起来，但是，当表达式是字段的引用或位置参数时，可以省略括号。例如：

```
mytable.mycolumn
```

```
$1.somecolumn
```

```
(rowfunction(a,b)).col3
```

当省略括号可能会带来歧义时，括号不能被省略，例如，一个带表名的字段引用：

```
(mytable.arraycolumn).somefield
```

因为此时如果省略了括号，将会和如下用法混淆：

```
mydatabase.mytable.mycoloumn
```

运算符调用

运算符调用有3中可能的语法：

```
expression operator expression(二目运算符)
```

```
operator expression(一目前置运算符)
```

```
expression operator(一目后置运算符)
```

这里的operator可以是一些符号(例如+、-、*、/、>、<等等)，还可以是AND、OR、NOT等SQL关键字。例如：

```
OPERATOR(datatype,datatype)
```

存在哪些特定的运算符以及他们的目数，取决于系统预定义的情况或用户的定义。

函数调用

调用函数的语法是函数名(有时还需要模式名)，后面跟着相关的参数，这些参数用括号括起来：

```
function ([expression [, expression ... ]])
```

例如，下面是调用求平方根的函数来计算2的平方根：

```
sqrt(2)
```

聚合表达式

聚合表达式，对查询多行记录应用聚合函数，并得到一个结果，例如，求和、求平均值。聚合表达式的语法如下：

```
aggregate_name(expression [ , ... ] ) [ FILTER ( WHERE filter_clause ) ]
```

对所有输入的非空记录(NOT NULL)进行聚合运算。

```
aggregate_name(ALL expression [ , ... ] ) [ FILTER ( WHERE filter_clause ) ]
```

与第一种等价，因为第一种就是ALL的缺省情况。

```
aggregate_name(DISTINCT expression [ , ... ] ) [ FILTER ( WHERE
filter_clause ) ]
```

对所有输入的记录的唯一值(且NOT NULL)进行聚集运算。

```
aggregate_name(*) [ FILTER ( WHERE filter_clause ) ]
```

对所有输入的记录(NULL也包含在内)进行聚合运算。通常用于count(*)函数。

其中aggregate_name是一个预先定义好的聚合函数，expression是一个除聚合表达式以外的任意的值表达式，也就是说，聚合函数表不可以嵌套调用。

这里的FILTER子句的作用是，为聚合函数指定特定的条件以过滤输入的记录，只有满足FILTER子句中WHERE条件的记录才会被作为聚合函数的的输入。例如：

```
=# SELECT count(*) AS unfiltered,count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
```

	unfiltered bigint	filtered bigint
1	10	4

GP数据库提供了中值函数MEDIAN，该函数返回排序在50%位置点的数值，还提供了百分比插值函数PERCENTILE_CONT和PERCENTILE_DISC,返回数据集中排序后指定百分比位置的线性插值的结果，MEDIAN和PERCENTILE_CONT返回的结果是线性插值，PERCENTILE_DISC返回的结果是距离线性插值最近的输入值。例如：

```
=# SELECT MEDIAN(i),
    PERCENTILE_CONT(0.22) WITHIN GROUP (ORDER BY i),
    PERCENTILE_DISC(0.22) WITHIN GROUP (ORDER BY i)
FROM generate_series(0,15) AS i;
```

	median double precision	percentile_cont double precision	percentile_disc integer
1	7.5	3.3	3

聚合表达式的限制

以下是一些聚合表达式的限制：

- 一些高级的聚集函数不能和ALL、DISTINCT、FILTER或OVER等关键字一起使用，这里需要澄清，不是GP不支持这些关键字。

- 一些聚合表达式不能与分组规范一起使用：CUBE、ROLLUP和GROUPING SETS。

- 聚合表达式只能作为结果字段或者在HAVING子句中出现。不能出现在其他位置(例如WHERE条件中)，因为这些位置的条件过滤或者运算是在得到聚合结果之前。

- 当一个聚合表达式出现在子查询中，常用于评估记录数量(例如count(*))或者求极值(例如max、min等)等，在子查询中，如果该聚集函数的参数包含了外部的字段引用，该子查询的聚合表达式的结果需要以一个常量的形式出现在外部相关的结果中。

- GP数据库不支持将聚合函数作为另一个聚合函数的参数(不可以嵌套调用)。

- GP数据库不支持将开窗函数作为聚合函数的参数。

开窗表达式

开窗函数的支持，使得应用开发人员，可以使用标准SQL命令，方便的构造复杂的在线分析处理(OLAP)查询。例如，可以计算移动平均值，或者不同时间段总数，根据不同的字段值进行分组聚合，得到聚合结果或者等级关系。

开窗表达式是在开窗分组(OVER()子句)的基础上执行开窗函数的计算。开窗分组将一个数据集合分割为多个部分，开窗函数对每个分组进行独立处理，这与聚合函数和GROUP BY的配合有相似之处，不同的是，聚集函数为每组记录返回一个结果，而开窗函数为每条记录返回一个结果，这些计算只是在一个开窗分组内完成。如果没有指定开窗分组条件，开窗函数会将全部数据作为一个开窗分组来进行计算。

GP数据库不支持将开窗函数作为另一个开窗函数的参数(不可以嵌套调用)。

开窗表达式的语法为：

```
window_function ( [expression [, ...]] ) [ FILTER ( WHERE filter_clause ) ]
OVER ( window_specification )
```

window_function是开窗函数，后续还会介绍，有哪些常见的开窗函数，还可以是用户自定义的开窗函数，expression是一个值表达式，但不能是开窗表达式。其中window_specification的语法格式为：

```
[window_name]
[PARTITION BY expression [, ...]]
[[ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}]
   [, ...]
      [{RANGE | ROWS}
         { UNBOUNDED PRECEDING
         | expression PRECEDING
         | CURRENT ROW
         | BETWEEN window_frame_bound AND window_frame_bound }]]
```

其中window_frame_bound可以是下面的一种格式：

```
      UNBOUNDED PRECEDING    -- 第一行
      expression PRECEDING   -- 前expression行
      CURRENT ROW            -- 当前行
      expression FOLLOWING   -- 后expression行
      UNBOUNDED FOLLOWING    -- 最后一行
```

一个开窗表达式，只能出现在SELECT语句的结果字段列表中。例如：

```
=# SELECT count(*) OVER(PARTITION BY customer_id), * FROM sales;
```

如果要指定FILTER子句，则只有满足FILTER中WHERE条件的记录会被开窗函数处理，其他不满足的记录不会被开窗函数处理(只是影响计算的结果，不影响SELECT语句输出的记录数)。在开窗表达式中，FILTER子句只能与本身为聚合函数的开窗函数一起使用。例如：

```
=# SELECT count(*) filter(WHERE id < 10)
   OVER(PARTITION BY customer_id), * FROM sales;
```

下面这种是不允许的：

```
=# SELECT rank() filter(WHERE id < 10)
   OVER(PARTITION BY customer_id), * FROM sales;
```

在开窗表达式中，表达式必须包含OVER子句，通过OVER子句来指定开窗分组的方式，这也是开窗表达式与普通的函数和聚合函数的显著区别。

开窗分组的规范如下：

- PARTITION BY子句，其决定了开窗函数通过该表达式对数据进行开窗分组。如果没有指定PARTITION BY子句，将会把全部数据作为一个开窗分组。

- ORDER BY子句定义了在一个开窗分组如何对记录进行排序。值得注意的是，开窗分组中的ORDER BY仅对开窗分组内的数据进行局部排序。对于计算Rank的开窗函数来说需要有ORDER BY子句，不然Rank值就是随机排序的结果。对于OLAP聚合来说，在使用ROWS或RANGE子句的开窗分组时，也要有ORDER BY子句，不然开窗函数计算得到的也是随机排序的结果。

- ROWS/RANGE子句用于定义开窗分组内的动态分组。PARTITION BY子句定义了数据如何进行开窗分组。当开窗分组的方式确定了以后，使用了ROWS/RANGE子句的情况下，开窗函数将进行步进式的动态计算，而不是将整个开窗分组内的数据进行整体计算。ROWS关键字对应基于行偏移的动态计算，RANGE关键字对应基于值偏移的动态计算，ROWS偏移基于每一条记录进行动态计算，不关心ORDER BY字段的值是否有差异，RANGE偏移基于ORDER BY字段的值来进行动态计算，相同的值得到相同的结果，后面的例子中会有展示。

开窗表达式例子

下面的示例演示如何将开窗函数与开窗分组一起使用。

首先，为了这些例子，创建一张 empsalary 表用于演示：

```
=# CREATE TABLE empsalary (
    depname varchar(10),
    empno int,
    salary numeric(10,0)
) ;
INSERT INTO empsalary VALUES
('研发',1,20000),('研发',2,20000),('研发',3,20000),
('研发',4,30000),('研发',5,30000),('研发',6,30000),
('客户部',7,40000),('客户部',8,40000),
('客户部',9,50000),('客户部',10,50000);
```

例一，如何将开窗函数与开窗分组一同使用。

PARTITION BY 子句将相同值的字段进行分组。

该例是将员工的工资与部门的平均工资进行对比：

```
=# SELECT depname, empno, salary, avg(salary)
    OVER(PARTITION BY depname) FROM empsalary;
```

	depname character varying(10)	empno integer	salary numeric(10,0)	avg numeric
1	客户部	10	50000	45000.000000000000
2	客户部	9	50000	45000.000000000000
3	客户部	8	40000	45000.000000000000
4	客户部	7	40000	45000.000000000000
5	研发	5	30000	25000.000000000000
6	研发	6	30000	25000.000000000000
7	研发	4	30000	25000.000000000000
8	研发	3	20000	25000.000000000000
9	研发	2	20000	25000.000000000000
10	研发	1	20000	25000.000000000000

前面 3 个字段来自 empsalary 表中，表中的每一条记录都有一行输出。第四列是每个部门的平均工资，具有相同部门名称的记录为同一部门，一起算出平均值。例子中有 2 个部门，avg 函数与一般的 avg 函数没有区别，但计算平均值的范围是由 OVER 子句中的 PARTITION BY 字段决定的。

还可以将开窗分组的定义放到 WINDOW 子句中，在开窗分组的 OVER 位置引用这个开窗分组，下面的 SQL 与上面的 SQL 等价：

```
=# SELECT depname, empno, salary, avg(salary) OVER(mywindow)
FROM empsalary WINDOW mywindow AS (PARTITION BY depname);
```

当 SELECT 的字段列表中有多个相同的开窗分组方式时，这种引用的方式比较有用。

例二，带有 ORDER BY 子句的开窗分组。

OVER 子句中的 ORDER BY 子句指定了开窗分组内的记录顺序，此处的 ORDER BY 与输出的顺序没有直接关系。此例使用 rank()函数来对员工的工资进行部门内的排序：

```
=# SELECT depname, empno, salary,
    rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
```

	depname character varying(10)	empno integer	salary numeric(10,0)	rank bigint
1	客户部	9	50000	1
2	客户部	10	50000	1
3	客户部	7	40000	3
4	客户部	8	40000	3
5	研发	4	30000	1
6	研发	6	30000	1
7	研发	5	30000	1
8	研发	1	20000	4
9	研发	2	20000	4
10	研发	3	20000	4

例三，基于记录偏移的动态开窗。

ROWS/RANGE 子句用于定义开窗分组内的动态分组 -- 开窗函数的输出基于开窗分组内的一部分记录进行计算。例如，从分组开始的位置到当前行的所有行。

这个例子计算每个部门，从开始到当前员工的薪水总和：

```
=# SELECT depname, empno, salary,
    sum(salary) OVER (PARTITION BY depname ORDER BY salary
    ROWS between UNBOUNDED PRECEDING AND CURRENT ROW)
FROM empsalary ORDER BY depname, sum;
```

	depname character varying(10)	empno integer	salary numeric(10,0)	sum numeric
1	客户部	7	40000	40000
2	客户部	8	40000	80000
3	客户部	9	50000	130000
4	客户部	10	50000	180000
5	研发	1	20000	20000
6	研发	2	20000	40000
7	研发	3	20000	60000
8	研发	4	30000	90000
9	研发	6	30000	120000
10	研发	5	30000	150000

例四，基于值偏移的动态开窗。

RANGE 根据 OVER 子句中 ORDER BY 表达式的值来决定动态开窗的情况。这个例子演示了 RANGE 与 ROWS 之间的区别。这个例子与上面的例子类似，不同的的是，薪水相同的员工算出的薪水总和是相同的，并且包含所有这些薪水相同的员工的薪水：

```
=# SELECT depname, empno, salary,
    sum(salary) OVER (PARTITION BY depname ORDER BY salary
```

```
    RANGE between UNBOUNDED PRECEDING AND CURRENT ROW)
FROM empsalary ORDER BY depname, sum;
```

	depname character varying(10)	empno integer	salary numeric(10,0)	sum numeric
1	客户部	8	40000	80000
2	客户部	7	40000	80000
3	客户部	10	50000	180000
4	客户部	9	50000	180000
5	研发	2	20000	60000
6	研发	3	20000	60000
7	研发	1	20000	60000
8	研发	5	30000	150000
9	研发	6	30000	150000
10	研发	4	30000	150000

类型转换

类型转换指的是，从一种数据类型转换为另外一种数据类型。类型转换是运行时进行的。必须定义了合适的类型转换函数，才能进行对应的类型转换操作。对于字符串的类型转换，是将其作为对应类型的字符表现进行转换的，如果转换合法，则可以成功。

GP数据库支持三种可用于值表达式的强制类型转换：

- 显式的强制转换 -- 明确的指定两个数据类型之前的强制转换，有两个等效的语法形式：

```
CAST ( expression AS type )
expression::type
```

- 赋值转换 -- GP数据库可以在进行目标赋值时进行自动类型转换。在CREATE CAST时，通过AS ASSIGNMENT子句来创建一个赋值转换。例如，tbl1.f1字段是TEXT类型，则在执行如下INSERT时，INT类型将会自动转换为TEXT类型：

```
=# INSERT INTO tbl1 (f1) VALUES (42);
```

- 隐式转换 -- 根据表达式的上下文情况自动进行强制类型转换。在CREATE CAST时，通过AS IMPLICIT子句来创建一个隐式转换。隐式转换会根据表达式的上下文情况自动进行类型的转换。例如，tbl1.c1是int类型的字段，下面的SQL中，c1会自动进行int到decimal类型的转换：

```
=# SELECT * FROM tbl1 WHERE tbl1.c2 = (4.3 + tbl1.c1);
```

对于自动转换不会出现歧义的情况，往往可以进行隐式转换，有些可能会造成歧义的类型转换往往需要显式的转换，例如5版本开始，一些类型到TEXT类型的隐式转换被废除了，这种时候往往需要显式的类型转换。例如，字段为TEXT类型时使用INT类型的WHERE条件，将会收到如下的报错信息：

```
ERROR:  operator does not exist: text = integer
```

可以通过psql的命令\dC来查看类型转换，类型转换的元数据信息存储在pg_cast系统表中，类型信息存储在pg_type系统表中。

标量子查询

标量子查询是一个在括号中的普通SELECT查询，其返回单行单列的结果。该SELECT查询被执行，其返回的标量值，将作为其他表达式的一部分。作为一个标量子查询，不能使用返回多行或者多字段的子查询。如果标量子查询引用了子查询之外的字段，该子查询被称为关联标量子查询。例如：

```
=# SELECT *,
    (SELECT min(p2.price) FROM part p2 WHERE p1.brand = p2.brand) AS foo
FROM part p1;
```

关联子查询

关联子查询提供了一种使用其他查询结果来组织结果的方法。GP支持关联子查询，其为很多已有的应用提供了兼容性。关联子查询是一个普通的SELECT查询，其WHERE子句或目标列表包含了外部子句的引用。关联子查询可以是标量子查询，也可以是非标量子查询，区别在于其返回的是单个数值还是多条记录或多个字段。例如，这是一个跨级非标量关联子查询：

```
=# SELECT * FROM part p1 WHERE p1.p_partkey IN (
    SELECT p2.p_partkey FROM part p2
    WHERE p2.p_retailprice = (
       SELECT min(p3.p_retailprice) FROM part p3
```

```
        WHERE p3.p_brand = p1.p_brand
    )
);
```

关联子查询例子

例1 -- 标量关联子查询

```
=# SELECT * FROM t1 WHERE t1.x > (SELECT MAX(t2.x) FROM t2 WHERE t2.y = t1.y);
```

例2 -- EXISTS在的关联子查询

```
=# SELECT * FROM t1 WHERE EXISTS (SELECT 1 FROM t2 WHERE t2.x = t1.x);
```

GP执行关联子查询有两种方式，如下：

1. 关联子查询可以被拆解为关联(JOIN)操作，这是高效的。大部分的关联子查询查询属于这种，包括所有TPC-H基准测试的查询。

2. 外部查询的每条记录都执行一次关联子查询的查询语句，这是极其低效的。在SELECT列表中或者使用OR连接的子句中出现的关联子查询属于这种。

接下来的例子将介绍如何改写这种SQL语句来提升性能。

例3 -- 在SELECT列表中的关联子查询

原始的SQL：

```
=# SELECT T1.a,
    (SELECT COUNT(DISTINCT T2.z) FROM t2 WHERE t1.x = t2.y) dt2
FROM t1;
```

实际上，Orca已经已经能够很好的优化该SQL，正确的将这种关联子查询转化为HASH JOIN，已经很高效，而PostgreSQL优化器无法优化这种SQL，需要进行等价的改写。

改写后的SQL：

```
=# SELECT t1.a, t2.count FROM t1 LEFT JOIN (
```

```
    SELECT t2.y AS y, COUNT(DISTINCT t2.z) AS count
    FROM t2 GROUP BY t2.y
) t2 ON (t1.x = t2.y);
```

例4 -- 带有OR子句的关联子查询

原始的SQL：

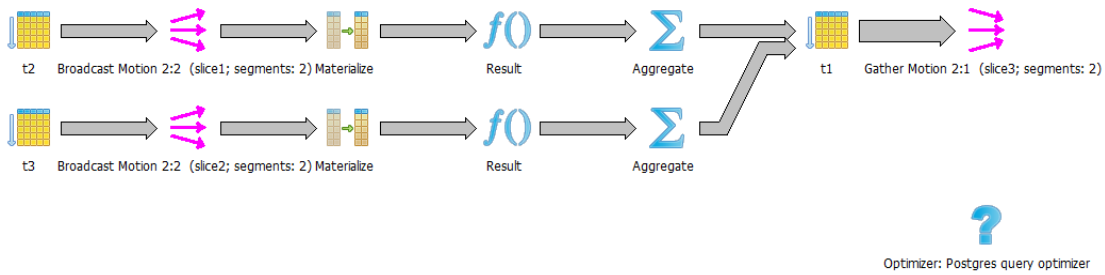
```
=# SELECT * FROM t1 WHERE
    x > (SELECT COUNT(*) FROM t2 WHERE t1.x = t2.x)
    OR
    x < (SELECT COUNT(*) FROM t3 WHERE t1.y = t3.y);
```

实际上，Orca已经已经能够很好的优化该SQL，正确的将这种关联子查询转化为HASH JOIN，已经很高效，而PostgreSQL优化器无法优化这种SQL，需要进行等价的改写。

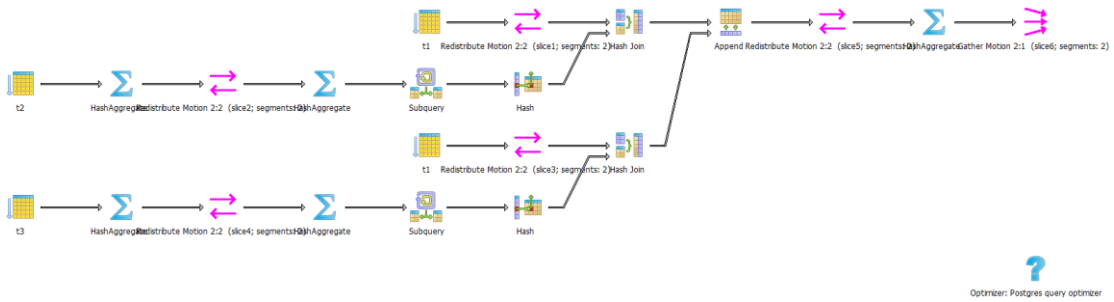
改写后的SQL：

```
=# SELECT * FROM t1
WHERE x > (SELECT count(*) FROM t2 WHERE t1.x = t2.x)
    UNION
SELECT * FROM t1
WHERE x < (SELECT count(*) FROM t3 WHERE t1.y = t3.y);
```

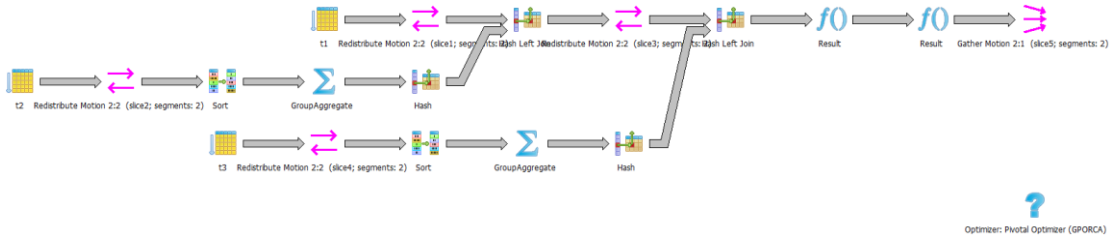
要检查对比改写前后的SQL，优化器生成的执行计划是如何处理这些关联子查询的，可以通过EXPLAIN命令查看执行计划来确认。例如例4中的SQL，关闭Orca后，原始的SQL的执行计划为：



改写后的执行计划执行计划为：



开启Orca时，原始SQL的执行计划为：



显然，对于Orca来说，原始的SQL生成的执行计划已经够优化，无需进行改写。

数组构造函数

数据构造函数，通过数据元素的值，构造出一个数组。简单的来说，数组构造函数由一个ARRAY关键字，一个左中括号([)，一串由逗号分隔的值表达式，一个右中括号(])组成。例如：

```
=# SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

数据元素的类型是成员表达式的类型，其与UNION或CASE结构使用相同的规则。

多维数据可以通过嵌套数据构造函数的方式构造。在构造函数内部，ARRAY关键字可以被省略。例如，这两个SELECT语句是等价的：

```
=# SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
```

```
 {{1,2},{3,4}}
(1 row)
=# SELECT ARRAY[[1,2],[3,4]];
    array
---------------
 {{1,2},{3,4}}
(1 row)
```

多维数据必须是规整的，内部同一层级上的构造函数必须生成完全相同维度的子数组。

多维数据构造函数，还可以适用于各种能正确的产生多维数据的用法。例如

```
=# WITH arr AS(
   SELECT ARRAY[[1,2],[3,4]] AS f1, ARRAY[[5,6],[7,8]] AS f2
)
SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                  array
-----------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

还可以将一个子查询的结果构造成数组。这种情况下，数组构造函数被写为关键字ARRAY和一个括号括起来的子查询。例如：

```
=# SELECT ARRAY(SELECT generate_series(1,10));
        array
-----------------------
 {1,2,3,4,5,6,7,8,9,10}
(1 row)
```

子查询必须返回单字段的结果集(多字段可以使用ROW函数构造成复合类型)。生成的单维数组将子查询得到的每行作为一个元素，元素的类型与子查询输出的字段类型一致。数组的下标值始终从1开始。

行构造函数

行构造函数，使用字段的值，构造出一个ROW对象。例如：

```
=# SELECT ROW(1,2.5,'this is a test');
```

行构造函数可以包含rowvalue.*语法，就像在SELECT中使用.*那样，.*会被自动展开。假如t表包含f1和f2两个字段，下面两句是等价的：

```
=# SELECT ROW(t.*, 42) FROM t;
=# SELECT ROW(t.f1, t.f2, 42) FROM t;
```

缺省情况下，使用ROW表达式构造匿名record类型。如果有必要，其可以转换为命名的复合类型 -- 或者一张表的行类型，或者一个通过CREATE TYPE AS创建的复合类型。有时为避免歧义可以进行明确的类型转换。例如：

```
=# CREATE TABLE mytable(f1 int, f2 float, f3 text);
=# CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1'
   LANGUAGE SQL;
```

在下面的查询中，不需要强制转换，因为只有一个getf1()函数，没有歧义：

```
=# SELECT getf1(ROW(1,2.5,'this is a test'));
getf1
-------
1
=# CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);
=# CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT
$1.f1' LANGUAGE SQL;
```

现在需要强制转换以明确调用哪个函数：

```
=# SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR: function getf1(record) is not unique
```

```
=# SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
getf1
-------
1
=# SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
getf1
-------
11
```

在表中有符合类型时，行构造函数，可用于构造这种符合类型的字段值。或者，在调用符合类型参数的函数时，为函数构造参数。

表达式评估规则

表达式的评估顺序没有定义。通常，一个运算符或者函数的输入参数不能保证按照固定的顺序从左到右进行评估。此外，如果一个表达式的结果可以通过只评估其一部分来确定，其它部分的表达式可能完全没必要被评估。例如，一种写法为：

```
=# SELECT true OR somefunc();
```

函数somefunc()可能根本不会被调用。另一种写法仍然不能保证somefunc()函数一定会被调用：

```
=# SELECT somefunc() OR true;
```

注意：这与一些程序语言布尔操作的从左到右"固定执行顺序"是不同的，在GP中这种表达式会被优化器优化掉。

不要在复杂的表达式中利用函数的副作用，更不要在WHERE和HAVING子句中利用函数的副作用，因为在生成执行计划时，这些表达式可能会被优化掉，或者被重新组织顺序或者逻辑结构。

如果一定要强制评估的顺序，可以选择CASE结构。例如，这样在WHERE子句中避免被0除是靠不住的：

```
=# SELECT . . .  WHERE x <> 0 AND y/x > 1.5;
```

但这样做是安全的：

```
=# SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

优化器不会对CASE结构进行重新组织，因此请在必要的时候这么做。

WITH 语句(CTE)

WITH 子句，为复杂的 SELECT 和修改数据的操作，提供了不同的表达方式，除了常用于 SELECT 语句外，还可以用于 INSERT、UPDATE 和 DELETE 语句。

使用 WITH 子句的限制：

- 包含WITH子句的SELECT命令，最多只能包含一个修改表数据(INSERT、UPDATE或DELETE)的WITH子句。

- 包含WITH子句的数据修改命令(INSERT、UPDATE或DELETE)，WITH子句中只能是SELECT命令，不能在WITH子句中再包含数据修改的命令。

缺省情况下，WITH 子句的 RECURSIVE 关键字是可用的，其是否可用，通过参数 gp_recursive_cte 来控制。

WITH 子句中使用 SELECT 命令

WITH 子句通常被称为 CTE(中文可以叫通用表表达式)，CTE 的效果和创建一张临时表是很像的，虽然在执行查询时，并不会在数据库中真的创建一张临时表。下面的示例显示了在执行 SELECT 命令时使用了 WITH 子句，这些示例中的 WITH 子句同样可以与 INSERT、UPDATE 和 DELETE 一起使用，因为，在命令主体看来，这个 WITH 就相当于一个普通的子查询。

在每次执行整个 SQL 时，WITH 子句中的 SELECT 命令只会被执行一次，如果在 WITH 之外的语句中多次引用了 WITH 子查询，也只执行一次。因此，从这个角度来说，WITH 子句与临时表很像，对于代价较大的需要多次引用的子查询，可以使用 WITH 来避免重复的计算。为了避免有副作用的功能被多次执行，也可以使用 WITH 子句来实现。优化器不会将 WITH 中的查询拆解然后与外部的查询进行整体优化，优化器可以将外部的条件下推到 WITH 子句中(这一点，官方文档的表述有误)，

一个常见的用途是将复杂的查询分解为多个简单的部分。例如：

```
=# WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region, product, SUM(quantity) AS product_units,
    SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

这个查询也可以不使用 WITH 子句，但需要使用二级嵌套子查询，而使用 WITH 子句，则可以把 SQL 的逻辑结构简化很多。

启用了RECURSIVE关键字之后，WITH子句就可以完成普通SQL无法完成的递归查询，这时，WITH子句中的查询可以引用WITH子句自己的输出。下面的这个例子，是计算从1到100的整数求和：

```
=# WITH RECURSIVE t(n) AS (
    VALUES (1)
    UNION ALL
    SELECT n + 1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

递归WITH子句，一般由一个非递归项，后面UNON或者UNION ALL一个递归项组成，只有递归项才包含WITH子句本身的引用。

```
non_recursive_term UNION [ ALL ] recursive_term
```

包含UNION或者UNION ALL的递归WITH查询按照如下方式执行：

1、计算非递归项。对于UNION（但不是UNION ALL），去除重复的记录，把递归查询中其余的记录，存储在一个临时工作表中。
2、重复如下步骤，直到这个临时的工作表被清空为止：
   a、 计算递归项，使用临时工作表中的数据来进一步填充递归自引用，意思是，根据递归关联条件，把临时工作表中与递归自引用能关联(递归项WHERE条件)上的记录进一步填充到递归自引用中，对于UNION(但不是UNION ALL)，放弃重复的记录(包括与之前的任意记录重复)，把剩余的记录(没有关联上的)存储到到一个临时中间表中。
   b、 清空临时工作表，将临时中间表的记录插入临时工作表，清空临时中间表。

这是一个迭代的过程，不过，RECURSIVE这个关键字是SQL标准定义的。

递归查询通常用于处理多层次或者有树状关系的数据。例如，此查询，用于查找产品的所有直接和间接组成部分，表中的数据只有直接的包含关系：

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
    UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part;
```

这个 SQL 对应的表结构应该是这样的，记录的父子关系由父一级的记录来存储，而且可能会有多条记录，因为其不只是存储父子关系，还需要存储计件数量。

在使用递归查询时，必须确保查询的递归部分最终一定会有找不到记录的情况，否则查询将无限循环。在前面计算整数求和的例子中，工作表在每个步骤中包含一行记录，并且在连续的步骤中包含的值从 1 到 100。在第 100 步中，由于 WHERE 子句而没有输出，查询结束。

对于某些查询，使用 UNION 而不是 UNION ALL 可以确保查询的递归部分最终不会无限循环，方法是丢弃与先前输出行有重复的记录。但是，有时候，决定重复关系的只是一个或几个字段，而不是全部的输出字段。通过检查相关的字段，可以确定是否达到了相同的点，标准的方法是通过数组来记录访问的路径，在递归时进行检验。例如下面的查询，搜索图表的路径：

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 1
    FROM graph g
        UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

如果存在环状关系，此查询将无限循环下去，因为查询需要输出 depth 属性，depth 属性随着循环不断增长，此时修改 UNION ALL 为 UNION 也无法解决无限循环问题。所以，需要识别递归是否到达了同一条记录。修改 SQL，加入 path 和 cycle 以帮助检查是否到达了同一条记录：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
        ARRAY[g.id],false
    FROM graph g
        UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
        path || g.id,g.id = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

除了检测环状关系外，path 数组的值还显示了关联关系的路径信息。

在需要检查多个字段以识别环状关系时，可以使用 record 数组。例如，如果需

要比较字段 f1 和 f2：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
       ARRAY[ROW(g.f1, g.f2)],false
    FROM graph g
       UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
       path || ROW(g.f1, g.f2),ROW(g.f1, g.f2) = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

在不确定查询是否存在无限循环时，一种有效的测试方法是在主体查询中设置一个限制条件。例如，此查询如果不使用 LIMIT 子句，将会无限循环：

```
WITH RECURSIVE t(n) AS (
    SELECT 1
    UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

该方法之所以奏效，是因为 WITH 只计算了外部查询实际需要的记录数。但应该注意，这个方法不是任何时候都奏效的，例如先排序再取 LIMIT 的情况，或者递归查询的结果还需要与其他表进行关联，这种方式就失效了。

WITH 子句中使用数据修改命令

在 SELECT 命令的 WITH 子句中可以使用修改数据的命令：INSERT、UPDATE 或者 DELETE，从而实现在一个查询中执行不同类型的操作。

WITH 子句中的数据修改命令只执行一次，但也不是一定会被执行的，如果主查询压根就没有引用到 WITH 子句，该 WITH 子句可能会被优化掉，从而得不到执行。

下面这个 CTE 查询，在 WITH 子句中，从 products 表中删除了一些记录，并通过 RETURNING 子句返回被删除的记录。

```
WITH deleted_rows AS (
```

```
    DELETE FROM products
    WHERE
       "date" >= '2010-10-01' AND
       "date" < '2010-11-01'
    RETURNING *
)
SELECT * FROM deleted_rows;
```

WITH 子句中修改数据的命令，必须有一个 RETURNING 子句，RETURNING 子句返回的是被修改的记录。如果 WITH 子句中有数据修改的命令，则必须有 RETURNING 子句，否则会报错。

启用了 RECURSIVE 关键字的情况下，不允许在数据修改语句中进行递归操作。不过，可以通过访问递归 WITH 的输出来完成相似的的效果。例如，此查询将删除产品的所有直接和间接组成部分。

```
WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
       UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
DELETE FROM parts WHERE part IN (SELECT part FROM included_parts);
```

WITH 子句中的语句与主查询同时执行。因此，在 WITH 中使用数据修改语句时，该语句实际上是在快照上执行。语句的修改效果在目标表上并不可见。RETURNING 子句是在 WITH 子句和主查询之间传递修改后数据的唯一方式。在下面的例子中，外部直接在 products 表上的 SELECT 在 WITH 子句中的 UPDATE 操作之前返回了原始 price 值。查询结果中，将会出现原始 price 值和被修改后的 price 值。这部分的内容，经编者测试，与官方文档的介绍完全不一致。

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t
UNION ALL
SELECT * FROM products;
```

下面的语句，效果完全等同于没有 WITH 子查询。因为 WITH 子查询被优化掉了。

```
WITH t AS (
    UPDATE products SET price = price * 1.05
```

```
    RETURNING *
)
SELECT * FROM products;
```

在同一个语句中，一条记录不能被更新两次，因为顺序是不确定的，所以，如果允许更新两次，结果将是不确定的。这个限制是从 5 版本开始的，也是 4 版本升级需要注意的事项。

使用函数和运算符

- 在GP中使用函数
- 自定义函数
- 内置函数和运算符
- 开窗函数
- 高级聚合函数

在 GP 中使用函数

在GP中调用函数时，函数的一些属性，决定了该函数的运行差异。函数的易变性(VOLATILE)属性决定了函数的行为差异，EXECUTE ON属性决定了函数可以在什么地方被调用。VOLATILE属性原自PostgreSQL，该属性很重要，之前在"分区选择性的诊断"章节也提到过，VOLATILE属性有三种类型IMMUTABLE、STABLE和VOLATILE。EXECUTE ON属性是GP数据库特有的属性。

对于IMMUTABLE(不变的)型的函数，其特点是，只要输入的参数相同，输出的结果一定相同，在任何时间调用该函数，输出结果与输入参数保持绝对的匹配关系。这就允许这种函数在生成执行计划时被执行，因为其输出不会发生变化。STABLE(稳定的)型的函数，其特点是，在事务范围内，效果与IMMUTABLE一样，在事务期间输出是不变的，所以，这种函数也可以在生成执行计划时被执行。因为可以在生成执行计划时被执行，IMMUTABLE和STABLE这两种函数，可以在生成执行计划时直接被优化为函数结果的常量表达式。而VOLATILE(易变)型的函数，其特点是，任何时间执行，都不能保证输出的不变，所以，这种函数不能被优化器优化为常量，必须为查询中的每条记录执行一次函数。

对于EXECUTE ON属性，EXECUTE ON MASTER型的函数只能在Master上被执行，EXECUTE ON ALL SEGMENTS型的函数只能在所有的Primary上被执行，不可以在Master上执行。

下面的表格总结了GP数据库中函数的不同属性对应的行为特征。

VOLATILE属性：

VOLATILE 属性	描述	备注
IMMUTABLE	在任何时候执行，只要输入的参数不变，函数的结果就不变	可以被优化器优化为常量
STABLE	在事务过程中，只要输入的参数不变，函数的结果就不变	可以被优化器优化为常量
VOLATILE	函数的结果与输入参数之间没有关系，随时会发生变化	必须为每一条记录执行一次

EXECUTE ON属性：

EXECUTE ON 属性	描述	备注
EXECUTE ON ANY	缺省的属性，允许在 Master 或任何 Primary 上执行。	由数据库决定在哪里执行
EXECUTE ON MASTER	只能在 Master 上执行	对于需要访问用户表的自定义函数，应该选择这种类型
EXECUTE ON ALL SEGMENTS	只能在 Primary 上执行	
EXECUTE ON INITPLAN	函数包含了一个在所有 Primary 上执行的 SQL 命令，并希望在 Master 上调用时有特殊的处理，这种函数不能用于 CTE 子句	编者还没有没有完全理解这种用法

在GP中使用VOLATILE型函数是受限的。VOLATILE表明，即便是单表的扫描，函数值也可能发生变化。相对来说，很少有数据库内置函数属于这种类型，一些常见例子如random()、currval()、timeofday()等。不过需要提醒的是，所有有副作用的函数都必须是VOLATILE的，即便其返回值是可预测的(例如setval())。

在GP中，数据分散存储在各Instance上 -- 每个Instance是一个独立的PostgreSQL数据库。为了防止节点之间的数据出现不一致，任何含有SQL语句或者修改数据库的VOLATILE函数都不可以在Instance上执行。

在6版本，因为引入了复制表，函数可以对Instance上的复制表(DISTRIBUTED

REPLICATED)执行只读查询命令，但任何修改数据的命令都必须在Master上执行。

注意：隐藏的系统字段（ctid、cmin、cmax、xmin、xmax和gp_segment_id）在复制表上是不可用的，如果试图查询这些字段，将会得到一个字段不存在的报错信息。

为确保数据的一致性，VOLATILE和STABLE函数在Master上执行是安全的。例如，下面的语句在Master上被执行(没有FROM子句的语句)：

```
=# SELECT setval('myseq', 201);
=# SELECT foo();
```

简单的来说，在函数中，可以在Instance上执行对复制表的只读查询。除此之外，在函数中对任何表的访问和修改操作，都是不允许在Instance上执行的，只能在Master上执行。

函数的 VOLATILE 属性与执行计划缓存

从生成执行计划的角度来说，IMMUTABLE、STABLE 和 VOLATILE 三种类型最明显的区别是，VOLATILE 不能进行常量替换的优化，IMMUTABLE 和 STABLE 一种是永远保持稳定，一种是在事务过程中保持稳定，都可以通过先计算出函数的结果来进行常量替换的优化。

但是，IMMUTABLE 和 STABLE 在执行计划缓存时，就有本质的区别，因为 IMMUTABLE 是永远稳定的，所以，多次执行可以用相同的结果来进行常量替换，而 STABLE 只是事务稳定的，在其他事务中执行不能保证之前的计算结果是正确的，所以，使用了这种类型的函数的 SQL，执行计划不能被缓存。

自定义函数

GP像PostgreSQL一样支持自定义函数的使用。更多信息可以参看PostgreSQL文档的。

可以使用CREATE FUNCTION命令来创建自定义函数。就像"在GP中使用函数"章节描述的那样，缺省状态下，函数被声明为VOLATILE，因此，如果自定义函数是IMMUTABLE或者STABLE的，在创建函数时应该明确指定其VOLATILE属性，这一点很重要，不然，VOLATILE的函数用于查询条件时，将无法被优化，会影响分区裁剪等。注意，对于有副作用的函数(例如修改表中数据，执行Linux命令等)，必须指定为VOLATILE类型，否则，即便创建时不会报错，也将无法正常调用。

CREATE FUNCTION时缺省的EXECUTE ON属性是EXECUTE ON ANY。函数中，除了可以在Instance上对复制表的进行只读查询，函数只有在Master上被执行时，才能访问表中的数据或者修改数据。对于需要在函数中访问非复制表的情况，应该将该函数设置为EXECUTE ON MASTER属性，在没有指定该属性的情况下，执行计划可能会把函数下推到Instance去执行，可能会遭遇报错。

在创建自定义函数时，要避免出现FATAL错误或者有破坏性的操作，其可能会导致数据库宕机或者重启等异常。

在GP中，自定义函数的共享库文件在每个GP主机(Master和Instance所在机器)上的库路径必须相同。

从5版本开始，支持匿名代码块功能，这些匿名代码块，使用GP的函数过程语言编写，匿名代码块会像函数一样被执行。更多信息，可以参考DO命令。

内置函数和运算符

下表列出了PostgreSQL支持的内置函数和运算符的类别。除了STABLE和VOLATILE函数外，所有PostgreSQL的函数和运算符在GP中都支持，所受限制如"在GP中使用函数"中的描述。

关于内置函数和运算符的更多信息参照PostgreSQL文档。

运算符/函数类型	不稳定函数	稳定函数
逻辑运算符		
比较运算符		
数学函数和运算符	random、setseed	
字符串函数和运算符	所有内置转换函数	convert、pg_client_encoding
二进制字符串函数和运算符		
按 bit 位字符串函数和运算符		
模式匹配		
日期类型格式化函数		to_char、to_timestamp
日期/时间函数和运算符	timeofday	age、current_date、current_time current_timestamp、localtime localtimestamp、now
枚举类型的支持函数		
几何函数和运算符		
网络地址函数和运算符		

序列处理函数	nextval、setval	
条件表达式		
数组函数和运算符		所有数据函数
聚合函数		
子查询表达式		
行比较和数组比较		
集合返回函数	generate_series	
系统信息函数		所有会话信息函数 所有访问权限查询函数 所有模式可见性查询函数 所有系统表信息函数 所有备注信息函数 所有事务 ID 和快照
系统管理函数	set_config、 pg_cancel_backend pg_reload_conf、 pg_rotate_logfile pg_start_backup、 pg_stop_backup pg_size_pretty、 pg_ls_dir pg_read_file、 pg_stat_file	current_setting 所有数据库对象尺寸函数
XML 函数		xmlagg(xml) xmlexists(text, xml) xml_is_well_formed(text) xml_is_well_formed_document(text) xml_is_well_formed_content(text) xpath(text, xml) xpath(text, xml, text[]) xpath_exists(text, xml) xpath_exists(text, xml, text[]) xml(text) text(xml) xmlcomment(xml) xmlconcat2(xml, xml) 更多内容参考官方文档

开窗函数

以下内置的开窗函数是GP数据库对PostgreSQL的扩展。所有开窗函数都是IMMUTABLE的。有关开窗函数的详细信息，请参见"开窗表达式"章节。

函数	返回值类型	完整语法	描述
cume_dist()	double precision	CUME_DIST() OVER ([PARTITION BY expr] ORDER BY expr)	计算一组值的累计分布(0到1的浮点数)，相同值的记录得到相同的分布
dense_rank()	bigint	DENSE_RANK () OVER ([PARTITION BY expr] ORDER BY expr)	计算一组值的排名。相同值的排名相同，有并列排名且排名连续
first_value(expr)	与输入表达式类型相同	FIRST_VALUE(expr) OVER ([PARTITION BY expr] ORDER BY expr [ROWS|RANGE frame_expr])	计算一组值的第一个值
lag(expr [,offset] [,default])	与输入表达式类型相同	LAG(expr [,offset] [,default]) OVER ([PARTITION BY expr] ORDER BY expr)	一种跨行访问方式。按照排序的位置，LAG将一行记录向后偏移。如果不指定offset，缺省值为1。如果不指定default缺省值为null
last_value(expr)	与输入表达式类型相同	LAST_VALUE(expr) OVER ([PARTITION BY expr] ORDER BY expr [ROWS|RANGE frame_expr])	计算一组值的最后一个值
lead(expr [,offset] [,default])	与输入表达式类型相同	LEAD(expr [,offset] [,default]) OVER ([PARTITION BY expr] ORDER BY expr)	一种跨行访问方式。lead将一行记录向前偏移。如果不指定offset，缺省值为1。如果不指定default缺省值为null
ntile(expr)	bigint	NTILE(expr) OVER ([PARTITION BY expr] ORDER BY expr)	将一组值，按照排序后的顺序分割为expr个部分，生成从1到expr的序

			号，同一部分内的序号相同
percent_rank()	double precision	PERCENT_RANK () OVER ([PARTITION BY expr] ORDER BY expr)	计算一组值的百分比排名。相同值的排名相同
rank()	bigint	RANK () OVER ([PARTITION BY expr] ORDER BY expr)	计算一组值的排名。相同值的记录排名相同且排名可能不连续
row_number()	bigint	ROW_NUMBER () OVER ([PARTITION BY expr] ORDER BY expr)	为排序的一组值，每行分配一个唯一的连续编号

高级聚合函数

以下内置的高级聚合函数是GP数据库对PostgreSQL的扩展。所有这些函数都是IMMUTABLE的。

函数	返回值类型	完整语法	描述
MEDIAN (expr)	timestamp, timestampz, interval, float	MEDIAN (expression) 例如: SELECT MEDIAN(i) FROM generate_series(0,15) AS i;	返回一个中间值或者线性插值。空值被忽略。
PERCENTILE_CONT (expr) WITHIN GROUP (ORDER BY expr [DESC/ ASC])	timestamp, timestamptz, interval, float	PERCENTILE_CONT(percentage) WITHIN GROUP (ORDER BY expression) 例如: SELECT PERCENTILE_CONT(0.22) WITHIN GROUP (ORDER BY i) FROM generate_series(0,15) AS i;	根据给定的百分比计算排序集合的线性插值。空值会被忽略。
PERCENTILE_DISC (expr) WITHIN GROUP (ORDER BY expr [DESC/ASC])	timestamp, timestampz, interval, float	PERCENTILE_DISC(_percentage_) WITHIN GROUP (ORDER BY _expression_) 例如: SELECT PERCENTILE_DISC(0.22) WITHIN GROUP (ORDER BY i) FROM generate_series(0,15) AS i;	根据给定的百分比计算排序集合的最接近的值。空值会被忽略。
sum(array[])	smallint[]int[], bigint[], float[]	sum(array[[1,2],[3,4]]) 例如: WITH a AS(SELECT ARRAY[[1,2],[3,4]] AS arr UNION ALL SELECT ARRAY[[5,6],[7,8]] AS arr)	矩阵求和，可以使用二维数组作为输入参数。

		SELECT sum(arr) FROM a; sum ----------------- {{6,8},{10,12}} (1 row)	
pivot_sum(label[], label, expr)	int[], bigint[], float[]	pivot_sum(array['A1','A2'], attr, value) 例如: WITH a AS(SELECT generate_series(1,3) i, generate_series(1,7) j)SELECT i, pivot_sum(array['j'],'j',2) FROM a GROUP BY 1; 在编者看来就是分组计数，编者没有找到更多有用的信息	编者也没搞明白怎么使用这个函数。
unnest (array[])	任何元素集合	unnest(array['one', 'row', 'per', 'item'])	将一维数组转换为ROW记录。

查询性能

GP数据库会动态消除查询中不相关的分区，并且为执行计划中不同的算子优化内存分配。为非内存密集型算子分配固定尺寸的内存，剩余的内存分给内存密集型算子。这些增强，使得查询扫描更少的数据，内存得到更优化的分配，加速处理性能，可以提升并发的支持能力。

- 动态分区消除

  在GP中，在运行过程中才能获取到的值，将会被用于动态分区消除，这样可以提升查询的性能。通过配置gp_dynamic_partition_pruning设置为ON或OFF来启用或禁用动态分区消除。缺省情况下是ON的。

- 内存优化

  GP数据库为查询中的不同算子，根据是否为内存密集型算子，优化内存的分配，为非内存密集型算子分配固定尺寸的内存，剩余的内存分给内存密集型算子，并在处理的过程中，及时释放已经完成计算的算子的可释放内存并重新分配给后续的算子。

注意：默认情况下，GP数据库使用ORCA优化器。ORCA扩展了PostgreSQL优化器的功

能。关于ORCA的特点和限制，参见"关于ORCA优化器"章节。

控制溢出文件

GP 在执行 SQL 时，如果分配的内存不足，则会将文件溢出到磁盘上，通常称为 workfile，workfile 是 GP 内的标准称呼，因为相关的参数，视图，函数的名字都是以 workfile 来命名的。gp_workfile_limit_files_per_query 参数用于控制最大的溢出文件的数量，缺省值为 100000，可以满足大多数的场景，一般不需要修改这个参数。

如果溢出文件的数量超过了这个参数的值，数据库会返回一个报错：

```
ERROR: number of workfiles per query limit exceeded
```

有时，数据库可能会产生大量的溢出文件：

- 存在严重的数据倾斜。关于数据倾斜的检查，可以参见"数据倾斜"章节。
- 为查询分配的内存太少。可以通过max_statement_mem 和statement_mem参数来控制查询可用的最大内存尺寸或者通过资源组或者资源队列来控制。

可以通过修改查询语句 -- 优化 SQL 以降低内存的需求，更改数据分布 -- 避免数据倾斜，或修改内存配置来成功运行查询命令。gp_toolkit.gp_workfile_*视图可以用来查看溢出文件的信息，这些视图，对于查询性能的排查非常有帮助。

查询剖析

可以通过检查性能不符合预期的查询的执行计划，来确定可能存在的性能优化机会。

GP会为每个查询语句生成一个执行计划。要获得好的性能，选择正确的执行计划来适配数据的情况，是最重要的条件。执行计划决定了SQL命令在GP数据库中如何被执行。如果SQL本身的逻辑非常糟糕，可能数据库无论如何也无法产生好的执行计划，例如大表之间的非等值关联。

优化器使用统计信息(存储在系统表中)来选择一个成本更低的执行计划(根据优化器的实际情况，有时可能不一定是成本最低的)。成本衡量的标准是IO的代价，每个磁盘PAGE为一个单位。优化的目标是，选择成本最小的执行路径，但往往，生成符合预期的执行计划才是优的结果。

可以使用EXPLAIN命令来查看SQL命令的执行计划。例如：

```
=# EXPLAIN SELECT * FROM names WHERE id=22;
```

EXPLAIN ANALYZE命令会真正的执行SQL语句，而不仅仅是生成执行计划。这对于检验优化器评估的是否接近实际运行情况比较有用。例如：

```
=# EXPLAIN ANALYZE SELECT * FROM names WHERE id=22;
```

查看 EXPLAIN 输出

执行计划是一棵有很多个算子构成的树，其中每个算子是一个独立的计算操作，例如表扫描、关联、聚合或排序等。

从下到上来看执行计划，每个算子的计算结果作为上面一个算子的输入。执行计划最底部的算子，往往是表扫描算子：Seq Scan、Index Scan、Bitmap Index Scan。如果查询有关联、聚合或者排序，在扫描算子之上会有其他算子来执行这些操作。最顶端的算子往往是GP的移动算子(重分布、广播或汇总)。移动算子负责将处理过程中产生记录在Instance之间移动。

EXPLAIN的输出中每个算子都有一行，其显示基本的算子类型和该算子的成本估算：

- **cost** -- 访问的磁盘页数量，就是说，1.0等于一个连续的磁盘页操作。第一个值是获得第一条记录的成本，第二个值是获得所有记录的总成本。总成本是假设会检索所有的记录，但有时并不会真的检索所有记录，比如使用了LIMITX子句，可能不会真的检索所有记录。例如：

```
=# EXPLAIN SELECT * FROM pg_class;
Seq Scan on pg_class  (cost=0.00..17.19 rows=1019 width=265)
Optimizer: Postgres query optimizer
=# EXPLAIN SELECT * FROM pg_class LIMIT 1;
Limit  (cost=0.00..0.02 rows=1 width=265)
  ->  Seq Scan on pg_class  (cost=0.00..17.19 rows=1019 width=265)
Optimizer: Postgres query optimizer
```

注意：Orca优化器和PostgreSQL优化器生成的执行计划中，cost不具有可比性。这两个优化器，使用不同的成本估算模型和算法来评估执行计划的成本。对比两个优化器之间的cost值是没有实际意义的。

另外，对于任意优化器生成的执行计划的cost值来说，只对当前的查询和当前的统计信息有意义，不同的语句会生成不同cost的执行计划。即便如此，如果看到一个数量级非常大的cost，可能执行计划的确是有问题的。有时，cost的值会严重失真，

例如统计信息失真的情况下，这时的cost将变的不再真实。

- **rows** -- 该算子输出的记录数。该值可能与真实数量有较大的出入，其会反映WHERE子句的条件对记录的过滤。顶端算子评估的数量，在理想状态下与真实返回的、更新的或者删除的数据量接近。

- **width** -- 该算子产生的每条记录的尺寸(字节数)。这里不一定能真实体现计算的数据每条记录的尺寸，这里会去除掉表中没有被涉及到的字段的尺寸，对于列存表，这样做是准确的，但对于行存表，因为真实处理时，行存表的一条记录是一个tuple，不会因为只使用了少量字段而把tuple拆解。

注意：一个上层算子的cost包含其所有子算子的cost，最顶端算子的cost包含了整个执行计划的总cost，这就是优化器要试图减小的数字。另外，cost仅仅反映了优化器所在意的代价。除了这些，cost不包含结果集传输到客户端的开销或耗时的预估。

EXPLAIN示例

要说明如何阅读EXPLAIN得到的执行计划，参考一下下面这个简单的例子：

```
=# EXPLAIN SELECT * FROM names WHERE name = 'Joelle';
QUERY PLAN
------------------------------------------------------------
Gather Motion 2:1 (slice1) (cost=0.00..20.88 rows=1 width=13)
-> Seq Scan on 'names' (cost=0.00..20.88 rows=1 width=13)
Filter: name::text ~~ 'Joelle'::text
```

从下向上查看这个执行计划，执行计划从顺序扫描names表开始。注意，WHERE子句被用作一个filter条件。这意味着，扫描操作将根据条件检查扫描的每一行，并只输出符合条件的记录。

扫描算子的输出传递给汇总移动算子。在GP中，汇总移动是Instance向Master发送记录的操作，该场景下，有2个Instance向1个Master发送(2:1)记录。每个算子都在执行计划的一个Slice中。在GP中，一个执行计划可能会被分为多个Slice，以确保计算任务可以在Instance之间并行工作，往往不同的Slice可能会被Motion算子分开。

评估的开始成本为00.00(无cost)且总成本为20.88个磁盘页。优化器评估这个查询将返回一行记录。单条记录的尺寸为13个字节。

查看 EXPLAIN ANALYZE 输出

EXPLAIN ANALYZE会真正的执行语句，而不仅仅是生成执行计划。EXPLAIN ANALYZE依然会输出优化器的评估cost，同时会输出真实执行的cost。据此，可以评估优化器生成的执行计划与真实的执行情况是否接近。EXPLAIN ANALYZE还会额外输出如下信息(Orca和PostgreSQL优化器会有差异)：

- 执行该查询总的耗时(以毫秒计)。
- 执行计划的每个Slice使用的内存，以及分配给该查询的总的内存量。
- 参与一个算子计算的Instance数量，只统计有记录返回的Instance。
- 算子中输出记录数最多的Instance输出的记录数。如果有多个Instance输出的记录数相同，则显示耗时最长的Instance的信息。
- 算子的内存使用情况，对于工作内存不足的算子，将显示性能最低的Instance的溢出文件的数量。例如：

PostgreSQL 优化器：

```
Extra Text: (seg0) . . . ; 100038 spill groups.
 . . .
* (slice2)    Executor memory: 2114K bytes avg x 2 workers, 2114K bytes max
    (seg0).  Work_mem: 925K bytes max, 6721K bytes wanted.
Memory used:  2048kB
Memory wanted:  13740kB
```

Orca优化器：

```
Sort Method:  external merge  Disk: 1664kB
 . . .
* (slice2)    Executor memory: 2256K bytes avg x 2 workers, 2256K bytes max
    (seg0).  Work_mem: 2105K bytes max, 5216K bytes wanted.
Memory used:  2048kB
Memory wanted:  5615kB
```

- 算子中输出记录数最多的Instance，输出第一条记录所用的时间(以毫秒计)，输出最后一条记录所用的时间。如果两个时间相同，开始时间会被省略。随着执行计划从下向上被执行，时间可能是有重叠的。

EXPLAIN ANALYZE 示例

我们使用一个相对复杂一点的查询来说明。

先看一下Orca优化器的输出：

```
EXPLAIN ANALYZE
SELECT customer_id,count(*) FROM sales GROUP BY 1;

Gather Motion 2:1  (slice2; segments: 2)  (cost=0.00..476.54 rows=85709
   width=12) (actual time=376.394..452.434 rows=99351 loops=1)
  ->  HashAggregate  (cost=0.00..471.92 rows=42855 width=12) (actual
   time=377.094..421.520 rows=49765 loops=1)
       Group Key: customer_id
       Extra Text: (seg0)   49765 groups total in 32 batches; 1 overflows;
   169919 spill groups.
(seg0)   Hash chain length 2.0 avg, 16 max, using 42789 of 72704 buckets;
   total 8 expansions.
       ->  Redistribute Motion 2:2  (slice1; segments: 2)
   (cost=0.00..441.24 rows=250500 width=4) (actual time=2.788..171.396
   rows=250602 loops=1)
            Hash Key: customer_id
            ->  Seq Scan on sales  (cost=0.00..436.24 rows=250500 width=4)
   (actual time=0.019..46.502 rows=250755 loops=1)
Planning time: 31.606 ms
  (slice0)    Executor memory: 87K bytes.
  (slice1)    Executor memory: 58K bytes avg x 2 workers, 58K bytes max (seg0).
* (slice2)    Executor memory: 3106K bytes avg x 2 workers, 3106K bytes max
   (seg0).  Work_mem: 1849K bytes max, 4737K bytes wanted.
Memory used:  2048kB
Memory wanted:  5036kB
Optimizer: Pivotal Optimizer (GPORCA)
Execution time: 466.982 ms
```

从下往上看，将看到每个算子的额外信息。花费的总时间为466.982毫秒。

顺序扫描表的操作，输出记录数最多的Instance，执行计划评估的记录数是250250条，实际输出的是250755条，输出第一条的用时是0.019毫秒，输出最后一条的用时是46.502毫秒。重分布算子，输出第一条的用时是2.788毫秒，输出最后一条的用时是171.396毫秒。总的内存使用量是2048kB，而wanted是5036kB，在HASH聚合算子中因为内存不足，使用了spill溢出文件。

下面在再看一下PostgreSQL优化器的输出：

```
EXPLAIN ANALYZE
SELECT customer_id,count(*) FROM sales GROUP BY 1;

Gather Motion 2:1  (slice2; segments: 2)  (cost=11066.78..11923.86
   rows=85708 width=12) (actual time=567.520..653.070 rows=99351 loops=1)
  ->  HashAggregate  (cost=11066.78..11923.86 rows=42854 width=12) (actual
   time=566.833..619.733 rows=49765 loops=1)
      Group Key: sales.customer_id
      Extra Text: (seg0)   49765 groups total in 32 batches; 1 overflows;
   218258 spill groups.
(seg0)   Hash chain length 1.7 avg, 12 max, using 38835 of 69632 buckets;
   total 2 expansions.
      ->  Redistribute Motion 2:2  (slice1; segments: 2)
   (cost=8067.00..9781.16 rows=42854 width=12) (actual
   time=12.020..363.668 rows=227461 loops=1)
            Hash Key: sales.customer_id
            ->  HashAggregate  (cost=8067.00..8067.00 rows=42854 width=12)
   (actual time=10.862..212.384 rows=227625 loops=1)
                  Group Key: sales.customer_id
                  Extra Text: (seg0)   Hash chain length 4.4 avg, 15 max, using
   52054 of 53248 buckets; total 2 expansions.

                  ->  Seq Scan on sales  (cost=0.00..5562.00 rows=250500
   width=4) (actual time=0.018..66.742 rows=250755 loops=1)
Planning time: 0.113 ms
  (slice0)    Executor memory: 87K bytes.
  (slice1)    Executor memory: 1076K bytes avg x 2 workers, 1076K bytes max
    (seg0).
* (slice2)    Executor memory: 2114K bytes avg x 2 workers, 2114K bytes max
    (seg0).  Work_mem: 925K bytes max, 4673K bytes wanted.
Memory used:  2048kB
Memory wanted:  9644kB
Optimizer: Postgres query optimizer
Execution time: 666.468 ms
```

这里不再详细解读PostgreSQL优化器的输出。需要注意，不同的算子的耗时是有交叉和重叠的，这是因为，GP的执行器是流水线操作，下一步操作并不一定需要等待上一步完全执行完才开始执行，有些操作需要等待上一步的完成，例如Hash Join必须要等Hash操作完成才能开始。

检查执行计划排查问题

若一个查询表现出很差的性能，查看执行计划可能会有助于找到问题所在。下面是一些需要查看的事项：

- **执行计划中是否有某些算子耗时特别长？**找到占据大部分查询时间的算子。例如，如果一个索引扫描比预期的时间长，可能该索引已经过期，需要考虑重建索引。还可尝试使用enable_之类的参数(对于PostgreSQL优化器来说，这些参数很重要)，检查是否可以强制优化器选择不同的执行计划，这些参数可以设置特定的算子为开启或关闭状态。

- **优化器的评估是否接近实际情况？**执行EXPLAIN ANALYZE查看优化器评估的记录数与真实运行时的记录数是否一致。如果差异很大，可能需要在相关表的某些字段上收集统计信息。不过，如果SQL本身已经完全无法运行出结果，EXPLAIN ANALYZE将无法进行，该方法仅对运行慢的SQL有效。

- **选择性强的条件是否较早出现？**选择性越强的条件应该越早被使用，从而使得在计划树中向上传递的记录越少。如果执行计划在选择性评估方面没有对查询条件作出正确的判断，可能需要在相关表的某些字段上收集统计信息。不过，收集了准确的统计信息仍可能无法使得选择性的评估更准确，因为GP的选择性评估是基于MCV模型的，没有被统计信息记录的值，需要通过线性插值算法得到其存在概率，这种评估本身误差就较大，当需要同时对多个条件进行评估时，这种误差会呈几何倍数放大。有时，将太过复杂的SQL进行必要的拆解会更有效。

- **优化器是否选择了最佳的关联顺序？**如查询使用多表关联，需要确保优化器选择了选择性最好的关联顺序。那些可以消除大量记录的关联应该尽早的被执行，从而使得在计划树中向上传递的记录快速减少。如果优化器没有选择最佳的关联顺序，可以尝试设置join_collapse_limit=1(Orca由optimizer_join_order_threshold参数控制)并在SQL语句中构造特定的关联顺序，从而可以强制优化器选择指定的关联顺序。还可以尝试在相关表的某些字段上收集统计信息。

- **优化器是否选择性的扫描分区表？**如果使用了分区，优化器是否只扫描了查询条件匹配的相关分区。关于执行计划中是否选择了分区扫描，看参见"验证分区策略"章节。

- **优化器是否恰当的选择了HASH聚合或HASH关联算子？**HASH操作通常比其他类型的关联和聚合要快。记录在内存中进行比较和排序比在磁盘上操作要快很多。要使得优化器能选择HASH算子，必须确保有足够的内存来存放记录。可以尝试增加工作内存来提升性能(当缺省的内存配置不充裕时，如果已经足够，再增加不会提升性能，所以，不要盲目的以为增加内存就一定可以提升性能，内存只是一个通常不太会出问题的因素)。如果可能，执行EXPLAIN ANALYZE，可以发现哪些算子会用到溢出文件，使用了多少内存，需要多少内存。例如：

```
 . . .
       Extra Text: (seg0)   49765 groups total in 32 batches; 1 overflows;
   218258 spill groups.
 . . .
* (slice2)    Executor memory: 2114K bytes avg x 2 workers, 2114K bytes max
   (seg0).  Work_mem: 925K bytes max, 4673K bytes wanted.
Memory used:  2048kB
Memory wanted:  9644kB
```

需要注意的是wanted信息只是一个提示，是基于溢出文件尺寸来评估的。实际需要的内存，可能与实际情况有出入。

第十一章：数据导入与导出

本章讲述，在GP数据库中，如何将数据导入到常规的数据表中，如何从常规的数据表中将数据导出，以及数据文件的格式和异常处理等问题。

GP数据库支持高速并行数据导入和导出，对于数据量很小的导入和导出场景，也可以选择非并行的方式(原自PostgreSQL的COPY命令)。

GP支持导入和导出多种外部数据，比如，文本文件，Hadoop文件系统文件，Amazon S3，Web数据源等。

- SQL命令中的COPY命令，可以支持从psql的客户端，Master服务器，Instance服务器等位置，将文件导入到数据库中，或者从数据库导出到文件。
- 可读外部表，支持通过SQL直接对外部数据进行查询，除了SELECT外，还可以进行条件过滤、关联和排序等操作，也可以在外部表之上创建视图。不过，外部表最常见的使用场景是，将数据导入到常规的数据表中，而且，一般建议复杂的查询和处理操作，应先将数据导入数据库内，再进行复杂运算，因为，外部表查询虽然也是并行的，但性能还是远比不上常规的数据表。例如：

```
=# CREATE TABLE table AS SELECT * FROM ext_table;
=# INSERT INTO table SELECT * FROM ext_table;
```

- WEB型的外部表提供了更灵活的数据访问方式，可以通过访问http协议的URL来获取动态数据，或者通过执行GP集群内主机上的Linux脚本或命令来获取数据。编者编写的gpdbtransfer命令，就是通过这种外部表来实现GP集群之间的并行数据传输的，实现了灵活丰富的功能支持，编者一直自称为目前最先进的跨集群数据同步方案。
- GP数据库还提供了并行文件分发程序gpfdist，gpfdist命令与外部表配合，基于HTTP协议实现并行的文件数据导入和导出，支持多机部署gpfdist服务，从而实现，GP的Primary和gpfdist服务之间直接通过网络高速并行数据传输。
- gpload命令，通过YML格式文件进行参数控制，通过对gpfdist命令和外部表的包装(只是包装)，具备一定程度的自动化，实现将文件数据导入到GP数据库中。实际上，编者从未真正使用过gpload命令，因为直接使用外部表更灵活，过于追求傻瓜式，并不利于问题的发现和解决，编者不会介绍gpload命令。
- 还可以使用PXF协议来创建可读外部表或者可写外部表，来实现数据的并行导入和导出。编者也不会介绍PXF，如有需要，请按照官方说明来操作，编者想说，学会读文档和看HELP真的很重要，编者能做的是，帮助各位入门和理解，但做不到详细教学所有知识点。在6版本之前，访问Hadoop文件的主要方式是gphdfs协议，在6版本才正式切换为PXF协议，实际上gphdfs的配置和调试更加的麻烦。
- 商业版本还提供了gpkafka工具，实现从kafka高速并行导入数据到GP数据库中。
- 商业版本还提供了gpsc(Greenplum-Spark Connector)工具，实现GP和Spark之间的高速数据交互。
- 商业版本还提供了Greenplum-Informatica Connector工具，实现GP和

Informatica PowerCenter之间的高速数据传输。

根据编者的理解，以上这些方法，除了PostgreSQL的COPY命令，都是通过外部表来实现的，理论上来说，通过基于命令的WEB型外部表，完全可以自行实现与任何外部数据的高速交互。

选择什么样的数据交互方式，取决于数据的具体情况，比如，数据在哪里，数据规模的大小，需要什么样的转换和处理等。

对于数据量不大，不追求很高的性能的情况下，在可以运行psql的环境，可以简单的通过COPY命令来实现数据的导入和导出。COPY操作的文件尺寸限制，取决于psql所在环境的文件系统容量，COPY操作的性能限制，取决于psql所在环境的文件系统的性能，一般情况下，单个COPY命令就可以达到350MB/S甚至更高的数据导入和导出的性能，是一个很不错的选择。不过，通过COPY命令导入或者导出数据时，数据需要通过Master进行处理。

对于大规模的数据导入和导出，更高效的方法，是选择充分利用MPP特点的方式，让Instance同时从多个文件服务，利用多个网络端口，进行并行数据处理。可读外部表，对于查询操作来说，访问可读外部表，就如同访问常规的数据表一样，可以执行SELECT等操作。外部表与gpfdist服务配合，所有Instance与gpfdist服务之间进行全并行的数据传输，这是目前为止性能最高的数据导入导出的方式。

GP可以利用HDFS的并行架构实现与HDFS之间的高速并行文件访问，比如PXF协议。

创建外部表

GP的外部表，是数据存储在数据库之外的一种表，通过创建外部表来定义数据的位置和数据的格式。外部表根据数据的流向，可以分为可读外部表和可写外部表，可读外部表，主要用于将数据导入GP数据库，可写外部表主要用于将数据从GP数据库导出。对于外部表的使用，可以如同常规的数据库表一样来执行相应的SQL命令。比如对于可读外部表，可以通过SELECT来查询数据，对于可写外部表可以通过INSERT来导出数据，如果有必要，可读外部表还可以与其他表进行关联查询。

要创建外部表，首先需要确定数据的位置，文件的格式，根据这些内容来创建外部表的定义，之后，就可以通过外部表来实现与外部数据的交互了。

数据格式

不管是导入数据到GP数据库中，还是从GP数据库导出数据，都需要指定数据的格式。COPY和CREATE EXTERNAL TABLE(gpload实际上只是外部表的包装，不再单

独介绍，有需要的话，可以查阅相关资料)命令都可以指定数据的格式，数据可以是带分隔符的TEXT文本，逗号分隔的CSV格式等。只有正确的定义了数据的格式，在操作这些数据时才能正确的处理。

行分隔符

GP数据库可以识别的行分隔符包括：换行(LF | 0x0A)、回车(CR | 0x0D)或回车换行(CR+LF | 0x0D 0x0A)。换行符，是Unix类操作系统使用的标准行记录分隔符。Windows等操作系统可能会使用回车或回车换行作为行记录分隔符。这些行记录分隔符，GP数据库都可以支持。

字段分隔符

对于TEXT格式的文件，缺省的字段分隔符是水平制表符(0x09)。对于CSV格式的文件，缺省的字段分隔符是英文逗号(0x2C)。在使用COPY命令或者创建外部表时，都可以通过DELIMITER关键字来指定一个单字节的分隔符，作为字段之间的分隔符。字段分隔符，指的是数据文件中，以这个字符作为两个字段之间的分隔。但记录的开头和结尾位置不能有多余的字段分隔符(第一个字段或最后一个字段为空，属于正常现象)。例如，使用管道符(|)作为分隔符的一行记录：

```
data value 1|data value 2|data value 3
```

下面的例子展示，使用管道符(|)作为字段分隔符来创建外部表：

```
CREATE EXTERNAL WEB TABLE test_ext (a int, b int)
EXECUTE E'echo "08|25"'
FORMAT 'text' (DELIMITER E'\x7c');
```

不过，还是建议选择一个ASCII值较小的字符作为字段分隔符，常见的中文编码GBK编码中，低字节的范围在40 ~ FE之间，如果选择ASCII小于40的字段分隔符，将可以更有效的处理半个中文的吃字和乱码问题。在GP中，字符串类型的长度是以字符为单位的，但像Oracle等数据库，经常是以字节为单位计数的，就容易出现存储了半个中文的现象，如果半个高字节的中文和ASCII小于40的字段分隔符遇到一起，就可以在编码转换时明确的知道这是一个非法中文，如果和一个ASCII大于40的字段分隔符遇到一起，将无法界定高字节是半个中文，因为这是一个合法的GBK编码。

NULL值的定义

NULL值表示字段的值未被设定，在GP数据库中，NULL值和空字符串是不同的概念，NULL只能用IS NULL或者IS NOT NULL来判断，空字符串可以使用=''来判断。在使用COPY命令或者创建外部表时，可以通过NULL关键字来指定，将指定的字符串当作NULL值。仅当一个字段的字符串和NULL关键字指定的字符串相同时(不能有任何多余的字符)，该字段将会被作为NULL来识别。

在GP数据库导入外部数据时，缺省情况下，TEXT格式的NULL字符串为\N(反斜杠+N)，CSV格式的NULL字符串是没有双引号的空字符串，就是两个逗号之间什么都没有。例如，当定义NULL是'0'这个字符串时，将会得到下面的效果：

```
CREATE EXTERNAL WEB TABLE test_ext (a int, b int, c int)
EXECUTE E'echo "01|00|02
03|0|04"' ON MASTER
FORMAT 'text' (delimiter E'\x7c' NULL '0');
SELECT *,b IS NULL as isnull FROM test_ext;
```

	a integer	b integer	c integer	isnull boolean
1	1	0	2	f
2	3		4	t

从查询的输出可以看出，第二行的第二个字段被识别为NULL。而其他字段中只是包含了'0'，并不是整个字段为'0'，不会被当做NULL来对待。

如果在GP数据库之间导出和导入数据，可以直接使用缺省的设置。

转义符

对于GP数据库导入导出数据来说，有两类字符是特殊字符，分别是行分隔符和字段分隔符。如果数据本身包含这两类字符，则需要对字段本身进行转义，否则会造成歧义。缺省情况下，TEXT格式的转义符是反斜杠(\)，CSV格式的转义符是双引号(")。编者想说的是，转义符是解决歧义的最根本的方法，包含转义的数据格式，可以从根本上确保绝对不会有歧义，除此之外，不管是CSV，XML，JSON，还是多字节分隔符，只要没有转义，都不能绝对保证数据文件中一定不会有歧义。所以，在导入和导出数据时，使用转义符，可以绝对解决所有数据歧义问题。

对于文本中的字符转义，也是可以识别的，比如，对于与运算符(&)，在文本中，其16进制转义表示和八进制转义表示分别为[\x26]和[\046]。例如：

```
CREATE EXTERNAL WEB TABLE test_ext (a int, b int, c text)
EXECUTE 'echo "01|00|\x26 \046"' ON MASTER
FORMAT 'text' (delimiter E'\x7c');
```

	a integer	b integer	c text
1	1	0	& &

如果要关闭转义功能，可以使用ESCAPE 'OFF'，这样，文本中的所有字符就表现为原本的值，不过，建议不要这么做，尤其是中文环境，没有转义根本无法正常工作，最早的gptransfer工具就是这么做的，根本用不了。

字符编码

字符编码系统，是将字符集中的符号和一堆数字进行一对一映射，以便于传输和存储。GP服务端，虽然也支持多种编码集，比如ISO8859系列的单字节编码，还有UTF8等多字节编码，但一般不建议修改GP数据库的服务端编码，尤其在中文环境，UTF8是目前最佳的选择，也是缺省选择，建议不要修改。对于客户端的编码支持非常丰富，但GP服务端，并不支持所有编码格式，当从客户端输入数据时，数据库将根据客户端设定的编码格式自动进行编码转换，在数据返回给客户端时，数据库又会将数据自动转换为客户端的编码格式。

数据文件必须采用GP支持的字符编码，在导入数据时，如果数据文件中包含不支持的编码字符，加载将会遇到报错信息。对于少量的中文乱码，半个中文等问题，可以请求专业服务适当解决。

外部表协议

外部表主要用于大规模并行导入或导出数据，目前已经支持多种外部表协议。例如，file，gpfdist，PXF，S3等。本节将主要介绍常用的几种，关于其他协议，因为编者目前无法进行测试，不做介绍，如有需要，可参考官方文档。

---

file协议

file协议，通过URI来指向操作系统的文件。URI由主机名、文件所属的路径和文件名组成，不需要指定端口号，因为访问的是GP各个计算节点的本地文件。文件所在的位置，必须是初始化GP数据库的用户(按照管理，通常为gpadmin)可以访问的位置。主机名，必须存在于系统表gp_segment_configuration中的hostname字段中，或者address字段中，如果在这两个字段中都找不到这个主机名，则会在查询外部表时报如下的错误：

```
ERROR:  could not assign a segment database for
DETAIL:  There isn't a valid primary segment database on host . . .
```

LOCATION子句中可以有多个URI属性，用逗号分隔。例如：

```
CREATE EXTERNAL TABLE test_ext (a int, b int, c text)
LOCATION(
'file://mdw/tmp/file1*',
'file://mdw/tmp/file[2-4,6]'
)
FORMAT 'text' (delimiter E'\x7c');
```

在file协议中，URI中可以使用通配符或者C格式的模式匹配，来表示多个文件，比如上例中，file1*表示所有以file1开头的文件，[2-4,6]表示2、3、4、6。

在LOCATION中指定的URI的数量，是Primary访问文件时的并行数，对于每一个URI属性，数据库会根据主机名匹配gp_segment_configuration系统表中的hostname字段和address字段，并决定由哪个Primary来执行这条URI。理论上，为每个Primary指定一个URI，可以确保数据读取时的最大并行度。不过，每个主机上的最大URI的数量，取决于主机上有多少个Primary，也就是说，同一个Primary最多只能分配一个URI。另外，如果匹配的是address字段，则，相同address值的数量决定了URI中该主机名的数量，例如，有一台计算主机，主机名叫sdw01，主机上有两个Primary，对应address名称分别为sdw01-1和sdw01-2，下面的外部表在查询时会报错：

```
=# SELECT hostname,address FROM gp_segment_configuration WHERE hostname =
'sdw01' AND content >= 0 AND role = 'p';
 hostname | address
----------+---------
 sdw01    | sdw01-1
 sdw01    | sdw01-2
(2 rows)
=# CREATE EXTERNAL TABLE test_ext (a int, b int, c text)
```

```
LOCATION(
'file://sdw01-1/tmp/file1',
'file://sdw01-1/tmp/file2'
)
FORMAT 'text' (delimiter E'\x7c');
=# SELECT * FROM test_ext;
ERROR:  could not assign a segment database for "file://sdw01-1/tmp/file2"
DETAIL:  There are more external files than primary segment databases on host
"sdw01-1"
```

如果URI的数量超过了Primary的数量，还可能会有如下的报错：

```
WARNING:  number of locations (3) exceeds the number of segments (2)
HINT:  The table cannot be queried until cluster is expanded so that there
are at least as many segments as locations.
```

gpfdist协议

gpfdist协议，通过URI来指向gpfdist服务的相对路径的文件，数据库的所有Primary都必须能够访问gpfdist服务。gpfdist服务将所在机器的文件并行的分发给GP数据库的Primary。gpfdist服务的命令，在GP数据库集群的每台主机上都存在，在$GPHOME/bin目录下，source了GP的path文件之后，就可以直接使用该命令了。

在文件所在的服务器上启动gpfdist服务，对于Linux环境来说，可以直接复制Server的安装目录(缺省在/usr/local目录下，编者建议不要修改)。对于可读外部表，gpfdist会自动解压gzip(.gz后缀)压缩文件和bzip2(.bz2后缀)压缩文件。如果是压缩文件，需要确保后缀名称与压缩格式一致，否则可能无法正确的识别。对于可写外部表，如果目标文件的后缀名称是[.gz]，gpfdist会自动对目标文件进行gzip压缩，编者记得早些年好像导出的时候不会自动压缩，不过，编者目前测试了4.3.29版本，5.21版本和6.8版本，导出文件都可以自动压缩。

在gpfdist协议中，URI中可以使用通配符或者C格式的模式匹配，来表示多个文件，这与file协议是完全一致的。往往的确需要这样做，因为gpfdist协议对LOCATION中的URI的数量同样有限制，不能超过Primary的数量。与file协议不同的是，file协议的URI是绝对路径，而gpfdist协议的URI是gpfdist工作目录的相对路径。

注意：对于Windows平台的gpfdist服务，不论是可读外部表还是可写外部表，gpfdist服务均不支持解压和压缩。

和file协议一样，所有Primary并行请求URI的资源，不同的是，gpfdist协议不需要资源在集群内的机器上，更不需要跟gp_segment_configuration系统表中的hostname或者address字段匹配。不同的Primary可能会请求相同的URI资源，但同一个Primary不会请求多个URI资源，所以，URI的数量不能超过Primary的数量，否则，将有URI资源没有Primary来请求，会报错。在CREATE EXTERNAL TABLE时，指定多个gpfdist数据源，可以提升外部表获取数据的性能，不过对于规模不大的集群来说，一个gpfdist的吞吐能力就足够上百个Primary处理了，如果要提升多个外部表的并发导入能力，可以启动多个gpfdist服务供不同的外部表访问。

对可读外部表进行查询时，请求同一个gpfdist服务的Primary数量，还与gp_external_max_segs参数有关，当URI的数量小于该参数的值时，将最多有该参数指定的数量的Primary会发起请求，该参数的缺省值为64，绝大部分情况下已经可以满足需求，甚至在一些网络环境较差的集群，还需要降低该参数的值以缓解网络压力。

gpfdist还支持数据转换(transform)，通过在URI中使用transform参数来指定。在启动gpfdist服务时，需要通过-c参数来指定一个yml格式的配置文件，gpfdist接收到transform参数时，会根据参数的值来匹配yml文件中的配置，以确定执行具体的动作。例如：

```
$ pwd
/tmp
$ cat transform.yml
---
VERSION: 1.0
TRANSFORMATIONS:
  input:
    TYPE: input
    CONTENT: data
    COMMAND: /bin/bash /tmp/input.sh %filename%
  output:
    TYPE: output
    CONTENT: data
    COMMAND: /bin/bash /tmp/output.sh %filename%
$ cat input.sh
#!/bin/bash
cd $(cd "$(dirname "$0")"; pwd)
URI=$1
echo $URI
$ cat output.sh
#!/bin/bash
cd $(cd "$(dirname "$0")"; pwd)
URI=$1
more > $URI
```

```
$ gpfdist -c /tmp/transform.yml -d /../
=# DROP EXTERNAL TABLE test_ext;
=# CREATE EXTERNAL TABLE test_ext (a text)
LOCATION(
'gpfdist://mdw/tmp/file1#transform=input'
)
FORMAT 'text' (delimiter E'\x7c');
=# SELECT * FROM test_ext;
     a
-------------
 //tmp/file1
(1 row)
=# DROP EXTERNAL TABLE test_ext;
=# CREATE WRITABLE EXTERNAL TABLE test_ext (a text)
LOCATION(
'gpfdist://mdw/tmp/file0#transform=output'
)
FORMAT 'text' (delimiter E'\x7c');
=# INSERT INTO test_ext SELECT generate_series(1,3);
$ cat file0
1
2
3
```

这是一个简单的使用transform的例子，不过，在此基础之上，可以继续做很多的定制开发，例如，访问任意希望访问的数据源，做任意的数据转换。

WEB型的外部表

GP支持WEB型的外部表，以允许对动态数据或者网络数据进行访问。使用CREATE EXTERNAL WEB TABLE命令来创建WEB型外部表。WEB型外部表还包含两种类型，基于URL类型和基于Linux命令类型。

基于命令的WEB型外部表

GP支持通过Linux脚本或命令来实现对外部数据的操作。在使用CREATE

EXTERNAL WEB TABLE命令创建外部表时，通过EXECUTE子句来指定需要执行的命令。基于命令的可读WEB型外部表，读取数据的具体内容取决于执行命令的时间。基于命令的可读WEB型外部表，脚本或命令可以在GP数据库集群内的Master、Instance或者任意主机上(范围可以选择)被执行，命令或者脚本必须事先已经在这些机器上被定义好，缺省情况下，所有Primary都会执行指定的命令或脚本来产生数据，比如一个计算主机上有6个Primary，则指定的命令或脚本会在该主机上运行6次，每个Primary都会运行且只运行一次，如果要修改这种缺省的行为，可以通过EXECUTE子句的ON子句来定义。对于基于命令的可写WEB型外部表，所有Primary都会运行且只运行一次。

基于命令的WEB型外部表中的命令是由数据库执行的，所以，并不会source gpadmin用户的登录文件(.bashrc或.profile)，在执行命令时，可以手动source这些环境文件，也可以使用预先定义好的GP环境变量，这些变量主要用于识别不同的Primary信息和事务信息等。例如：

变量名称	描述
$GP_CID	执行外部表语句的事物的命令计数器，用于区分同一事物中的不同语句。
$GP_DATABASE	外部表的定义所在的 Database 的名称。
$GP_DATE	外部表运行的日期。
$GP_MASTER_HOST	GP 数据库集群的 Master 的主机名。
$GP_MASTER_PORT	GP 数据库集群的 Master 的服务端口。
$GP_QUERY_STRING	正在被执行的 SQL 语句。
$GP_SEG_DATADIR	当前 Primary 的工作目录。
$GP_SEG_PG_CONF	当前 Primary 的 postgresql.conf 文件位置。
$GP_SEG_PORT	当前 Primary 的服务端口。
$GP_SEGMENT_COUNT	GP 集群的 Primary 总个数。
$GP_SEGMENT_ID	当前 Primary 的 DBID，与 gp_segment_configuration 系统表中的 dbid 相同。其实更有用的是 content 值。
$GP_SESSION_ID	执行外部表语句的 Session ID。
$GP_SN	执行计划中外部表扫描节点的序号，用于区分同一的 SQL 中多次扫描同一张外部表。
$GP_TIME	外部表运行的时间。
$GP_USER	执行外部表的数据库 Role Name。
$GP_XID	执行外部表的事物 ID。

关于可读WEB型外部表的ON子句，不同的选项和含义如下表：

选项	描述
ON ALL	缺省值。在所有 Primary 上运行。
ON MASTER	只在 Master 上运行。
ON number_of_segments	由数据库随机挑选 number 个 Primary 运行。
ON HOST	每台有 Primary 的主机上运行一次。
ON HOST segment_hostname	指定主机名上的 Primary 都运行一次。

SEGMENT segment_id	指定 ID 的 Primary 运行。ID 与 gp_segment_configuration 系统表中的 content 字段一致。-1 是 Master，且不能在此处指定。

基于 URL 的 WEB 型外部表

基于 URL 的 WEB 型外部表，使用 HTTP 协议来获取 WEB 服务器的数据，通过在 LOCATION 子句中指定[http://]开头的位置信息来实现。与 file 协议类似，URL 的数量不能多于 Primary 的数量，在访问该类型外部表时，会把这些 URL 分配给一些 Primary，每个 URL 只能由一个 Primary 来执行，每个 Primary 最多只能执行一个 URL。基于 URL 的 WEB 型外部表，只有可读外部表，没有可写外部表。

例如，使用 http 协议来访问 gpfdist 服务(gpfdist 实际上也是一个 WEB 服务)：

```
=# CREATE EXTERNAL WEB TABLE test_ext (a text)
LOCATION(
    'http://mdw:8080/tmp/input.sh',
    'http://mdw:8080/tmp/output.sh'
)
FORMAT 'text';
SELECT * FROM test_ext;
```

	a text
1	#!/bin/bash
2	cd $(cd "$(dirname "$0")"; pwd)
3	URI=$1
4	tee $URI > /dev/null 2>&1
5	#!/bin/bash
6	cd $(cd "$(dirname "$0")"; pwd)
7	URI=$1
8	echo $URI

错误记录处理

可读外部表，通常用于将数据导入数据库内的常规数据表中。通过CREATE TABLE AS SELECT或者INSERT INTO . . . SELECT语句来实现数据的导入。缺省情况下，如果数据中包含了错误的记录格式，整个命令将会失败，不会有数据被导入目标表中。

SEGMENT REJECT LIMIT子句允许将可读外部表中格式错误的记录进行隔离，而正确的记录可以正常的导入目标表中。通过该子句可以设置错误记录隔离的阈值，允许按照百分比(PERCENT)或者记录数(ROWS)来限制，当格式错误的记录数量超过设置的阈值，整个导入操作仍然会失败，这个阈值的设定可用于控制数据质量，确保在有大量数据异常时可以失败报错。这个阈值，是针对每个Primary进行统计的，并不是全局统计。

当格式错误的记录数量没有达到设置的阈值，对外部表的查询，整体是成功的，错误的记录会被隔离，还可以选择是直接丢弃这些错误的记录，或者记录到日志信息中(通过在SEGMENT REJECT LIMIT子句前加上LOG ERRORS子句实现)以便进一步的处理(比如，找到错误的原因，可以有助于改善数据质量)。

当指定了SEGMENT REJECT LIMIT子句，GP读取可读外部表时，就开启了单行错误隔离模式，对于多字段、少字段、字段数据类型不匹配、编码错误等，都可以进行隔离，但不会对约束冲突进行隔离，数据违反目标表的约束，仍会导致数据导入的失败。不过，这种情况，在导入时，可以进行必要的过滤处理。例如：

```
=# INSERT INTO table_with_pkeys
SELECT DISTINCT * FROM external_table;
```

定义一个错误记录隔离的外部表

用一个简单的例子来说明：

```
=# CREATE EXTERNAL WEB TABLE test_ext (a int, b int, c int)
EXECUTE E'for i in {1..9};do echo "$i|$((i+100))|$((i+200))a";done
echo "1|3|5"' ON ALL
FORMAT 'text' (delimiter E'\x7c')
LOG ERRORS SEGMENT REJECT LIMIT 10 ROWS;
SELECT * FROM test_ext;
NOTICE:  found 18 data formatting errors (18 or more input rows), rejected
related input data
 a | b | c
---+---+---
 1 | 3 | 5
 1 | 3 | 5
(2 rows)
```

该例中，SEGMENT REJECT LIMIT子句前使用了LOG ERRORS子句，错误记录将会被记录到GP数据库的统一的错误记录日志中，如果没有LOG ERRORS子句，错误记录将会被直接丢弃。

使用GP自带的函数gp_read_error_log('external_table')可以查看出错

记录的日志。从5版本开始，GP的可读外部表，不再使用ERROR表来记录错误记录，而是使用统一的错误记录日志来记录，通过gp_read_error_log函数来查询错误记录日志，通过gp_truncate_error_log函数来清除指定的错误记录日志。原因是，以前的ERROR表无法保证事务的原子性，在多个外部表并发使用一张ERROR表时会报错。

将这个例子稍微改一下，用百分比来定义阈值：

```
=# CREATE EXTERNAL WEB TABLE test_ext (a int, b int, c int)
EXECUTE E'for i in {1..9};do echo "$i|$((i+100))|$((i+200))a";done
echo "1|3|5"' ON ALL
FORMAT 'text' (delimiter E'\x7c')
LOG ERRORS SEGMENT REJECT LIMIT 10 PERCENT;
SELECT * FROM test_ext;
NOTICE:  found 18 data formatting errors (18 or more input rows), rejected
related input data
 a | b | c
---+---+---
 1 | 3 | 5
 1 | 3 | 5
(2 rows)
```

定义的百分比阈值是10%，正确的记录是2条，错误的记录是18条，但是，查询依然成功了。这里需要解释一下gp_reject_percent_threshold这个参数，该参数定义了一个阈值(记为N，缺省值为300)，在使用百分比设置时，对于每个Primary来说，前N条记录不统计失败的百分比，这是一个无奈的方法，因为，如果不设置这样的阈值，当第一条记录就是错误数据(或者前几条记录的错误记录数较多)时，就直接到达了100%，无论后续的数据质量如何，都会整体失败。

还有一个参数gp_initial_bad_row_limit，该参数的缺省值为1000，当开始的数据全部为错误记录，且达到了该参数设定的值，整个导入操作就失败了，如果有一条正确的记录，整体操作不受该参数影响。

对于外部表，这两个参数都是对单个Primary作用的，因为Primary之间都是独立工作的，无法在处理的过程中互相串通这种统计数据。如果能在处理完最后一条记录时统计一下整体的成功率，是不是就不需要这两个参数了，编者觉得，设计有待改善。

查看错误记录日志

错误记录日志的格式为：

字段	类型	描述
cmdtime	timestamptz	查询外部表的 SQL 命令执行的时间。
relname	text	外部表的名称，或者 COPY 命令的目标表名称。
filename	text	错误记录所在的文件名称(可能是 URI 或者 URL)。

linenum	int	错误记录所在的文件中的行号。
bytenum	int	gpfdist是按照block来分拆文件的，无法记录行号，可以记录出错位置的字节数。
errmsg	text	错误记录的错误信息。
rawdata	text	错误记录的文本数据。
rawbytes	bytea	比如编码错误等，无法记录错误记录的文本，将记录错误记录的八进制编码。

gpfdist服务

在使用CREATE EXTERNAL TABLE命令创建外部表时，使用了gpfdist协议，可以用gpfdist服务来提供数据。当使用gpfdist服务配合gpfdist协议的外部表时，GP集群的Primary可以并行的读写外部数据。

本节主要包括以下内容：

- 关于gpfdist和外部表。
- 关于gpfdist的规划和性能。
- 控制Primary访问gpfdist的并发。
- 安装gpfdist。
- 启停gpfdist。
- gpfdist的故障排查。

关于 gpfdist 和外部表

gpfdist服务的命令，在GP数据库集群的每台主机上自带安装了，在$GPHOME/bin目录下，source了GP的path文件之后，就可以直接使用该命令了。当启动gpfdist服务时，需要指定服务监听的端口和工作目录(该目录内的数据都可以被访问)。例如，下面的命令是启动一个gpfdist服务，端口为8080，目录为/，并将服务提交后台运行：

```
$ nohup gpfdist -p 8080 -d /../ &
```

此处说明一下，gpfdist不允许直接在/目录上启动服务，因此可以通过/../的方式间接的从/上启动服务。

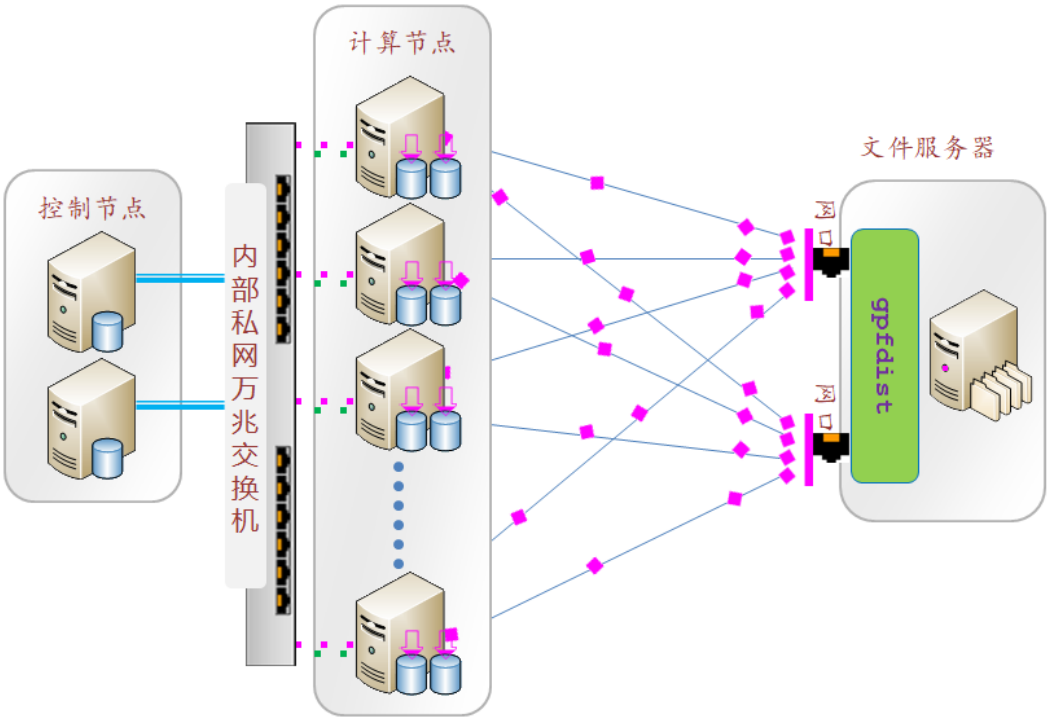
CREATE EXTERNAL TABLE命令的LOCATION子句，将外部表指向一个或者多个

gpfdist服务，如果是可读外部表，gpfdist服务将根据外部表的URI路径，在其工作的相对路径下找到对应的文件，读取这些文件，并分拆为一个一个的block，分发给GP数据库的Primary(根据Primary的请求情况)。Primary收到数据包之后，按照外表定义的ROW类型进行处理，格式化为表的tuple，然后根据具体执行语句的情况决定在哪里进一步处理这些tuple(实际上，这一步已经不属于外部表的职责，格式化为tuple之后的操作都是因为实际任务的需要，优化器为执行计划增加的处理操作)，比如要将数据INSERT到一张常规的数据表中，Primary将会根据目标表的分布键来决定将数据重新发送到哪个Primary。如果是可写外部表，Primary将会把表的tuple格式化为文本然后打包发送给gpfdist服务，gpfdist再将这些数据包写到一个或多个文件中，当写到多个文件时(LOCATION中会有多个URI选项)，根据可写外部表指定的分布键来决定发送给哪个gpfdist服务。

关于 gpfdist 的规划和性能

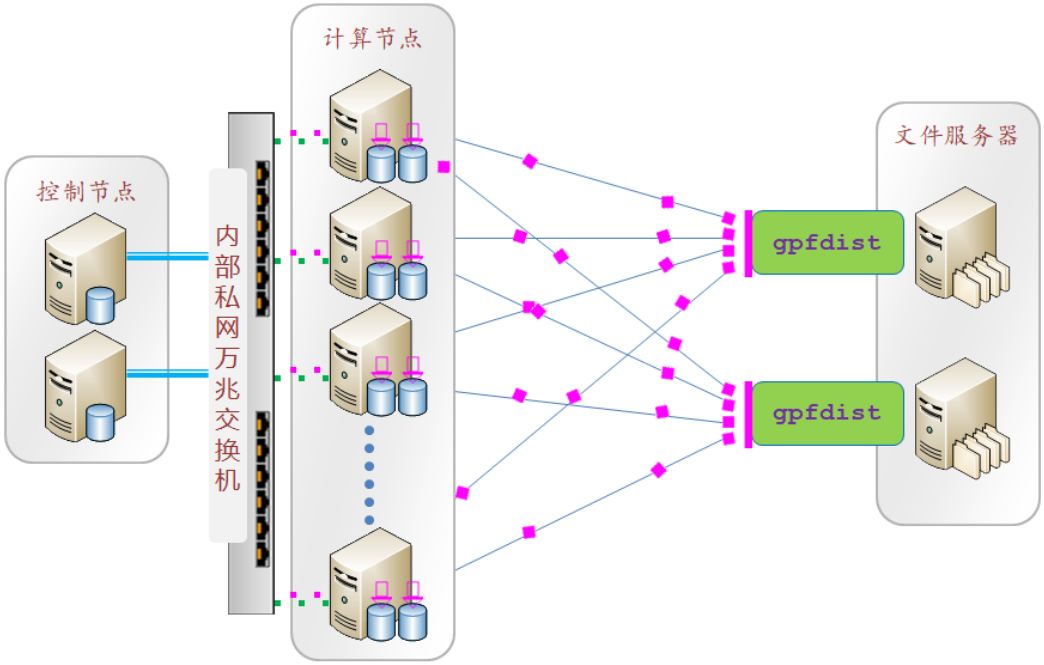
可以在不同的机器上运行多个gpfdist服务，也可以在一台机器上运行多个gpfdist服务。从而可以充分利用文件服务器的IO性能和网络带宽，以配合GP外部表高速的并行数据加载性能，获得更高的数据导入和导出的性能。

gpfdist服务，允许一个外部表，同时使用一台gpfdist文件服务器的多个网络端口，以提升数据访问的性能。这是因为，gpfdist不通过网络地址来区分是否为，对同一个资源的请求。而是通过X-GP-XID、X-GP-CID和X-GP-SN等Header信息来区分的。如下图所示，一个gpfdist服务，GP外部表，经由多个网络端口来访问。



具体到URI来说，不同的URI，通过不同的主机名，指向相同的gpfdist服务的相同目录和文件，实际上获取的是一份文件，利用了多个网络带宽，只有这种情况，才是对同一份文件利用了多个网络带宽，如果gpfdist服务、文件路径或文件名称有任何的不同，都不是同一份资源，在使用这一特性时需要注意避免重复加载数据。

可以将数据文件均匀的分散到多个gpfdist服务，同时为外部表提供数据，可以提升外部数据源的供数性能。比如，将数据均分到两台文件服务器上，每台文件服务器上启动一个gpfdist服务。例如：



对于性能很高的文件服务器来说，在多个端口(PORT)上启动gpfdist服务很有必要，毕竟一个gpfdist服务使用的CPU资源有限，当需要处理压缩、多并发等场景时，使用多个gpfdist服务可以提升高性能文件服务器的资源利用。

编者认为，分隔符对数据导出到gpfdist的性能几乎不会有任何影响，实际上，对可写外部表性能影响最大的是，Primary每次向gpfdist服务发送的数据包的尺寸，在4.3.7版本之前，Primary发送的数据包的缺省尺寸是32K，由于是HTTP协议的数据传输，大量数据导出时，更多的时间浪费在反复的HTTP连接建立和断开，将这个尺寸提升到比如16MB，导出的性能得到极大提升，可以瞬间占满gpfdist服务所在文件服务器的所有网络端口，这个尺寸由writable_external_table_bufsize参数设置，目前的版本，该参数的缺省值为64KB，还是太小了，建议直接设置到16MB。

控制 Primary 访问 gpfdist 的并发

通过gp_external_max_segs参数来控制同时访问一个gpfdist服务的Primary的数量，这样可以减轻gpfdist所在服务器的网络压力，该参数的缺省值是64，对于绝大部分的情况，不需增加这个参数，反而对于一些gpfdist所在服务器的网络环境较差的情况，需要适当的减小这个参数，当然，对于网络环境好的情况，增加这个参数是可以提升外部表导入的性能的。不过，这个参数仅对gpfdist外部表有效，其他协议不受此影响。

安装 gpfdist

可以在所有安装了GP服务端软件的机器上运行gpfdist服务，也可以在需要运行gpfdist服务的机器上复制一份GP服务端软件的安装目录。或者安装loaders客户端。

启停 gpfdist

可以在当前目录启动gpfdist服务，如果不使用-d参数指定工作目录，缺省的工作目录，是当前执行命令的目录。如果不使用-p参数指定监听的端口，缺省的监听端口为8080。例如，下面的命令是启动一个gpfdist服务，端口为8080，目录为/，并将服务提交后台运行：

```
$ nohup gpfdist -p 8080 -d /../ &
```

可以通过-l参数来指定gpfdist服务的日志输出，如果不需要gpfdist的输出，可以这样启动gpfdist服务：

```
$ nohup gpfdist -p 8080 -d /../ >/dev/null 2>&1 &
$ nohup gpfdist -p 8081 -d /../ >/dev/null 2>&1 &
```

正如上面的例子，可以在相同的目录上启动多个gpfdist服务，只要监听端口不同即可。这样就可以为不同的外部表配置不同的gpfdist服务，更充分的利用CPU资源。

要停止后台运行的gpfdist服务，可以先找到gpfdist服务的进程ID：

```
$ ps ax|grep gpfdist
```

使用kill命令杀掉gpfdist的进程。例如(加入2345是gpfdist的进程ID)：

```
$ kill 2345
```

---

gpfdist的故障排查

Primary在外部表被访问时才会真正的访问LOCATION中的资源指向，在定义外部表时，并不会检查资源是否存在或者是否可以访问。如果查询外部表时有访问方面的报错信息，首先要确保gpfdist已经正确的启动，然后确定所有Primary所在的机器可以正常的访问gpfdist服务。可以通过下面的步骤来测试gpfdist服务的可访问性：

1、 首先确定gpfdist已经启动：

```
$ ps ax|grep gpfdist
```

根据输出信息确定gpfdist的进程ID(pid)，比如pid为5134。

2、 如果已经启动，确认服务的监听端口和工作目录(上一步输出中已经显示的信息，可以不用确认)。

```
$ netstat -nltp|grep gpfdist
$ lsof -p 5134|grep cwd
```

根据输出信息确定gpfdist服务的监听端口和工作目录。

3、 确定工作目录下有一个小尺寸文件，用于测试可访问性，比如有个file0文件，监听的端口为8080。先在gpfdist服务器本地执行如下测试命令：

```
$ wget --delete-after --header 'X-GP-PROTO:0' http://localhost:8080/file0
```

如果可以成功获取测试数据文件，再从GP集群的所有主机测试该命令，确保所有主机都可以成功获取测试数据文件。这一步需要修改URL的主机名信息为真实的主机名或者IP地址。

使用外部表导入数据

使用外部表导入数据到GP数据库中，涉及以下几方面的前期准备工作：

1、 启动gpfdist服务，或者配置PXF服务。
2、 确定数据文件所在的目录位置。
3、 根据文件的数据格式创建外部表。

在这些工作准备就绪之后，就可以直接通过外部表来查询外部数据了。剩下的事情，就像使用常规的数据表一样来进行数据导入操作。例如，使用INSERT INTO命令实现从外部表ext_expenses导入类型为'travel'的数据到expenses_travel数据表中：

```
=# INSERT INTO expenses_travel
   SELECT * from ext_expenses where category='travel';
```

正如该例中所示，在查询外部表时，可以像使用常规的数据表那样使用各种过滤条件。

再例如，新建一张表并将外部表的数据全部导入：

```
=# CREATE TABLE expenses AS SELECT * from ext_expenses;
```

使用外部表导出数据

可写外部表，用于将库内的数据导出。可以是文件，可以是命名管道，可以是其他的应用程序(比如WEB服务)，GP MapReduce已经毫无意义了，不提也罢。不过，如果对性能和稳定性有很高的要求，导出到命令管道，可能会是一项极其复杂的工作，因为命名管道的状态无法精确的控制和获取，这就导致，在编程时需要设计很多的迂回措施来解决这些问题，最典型的场景就是以前的gptransfer命令，目前该命令已经废除，不过，该命令从一开始出现就注定是失败的，可以欣赏一下这段代码注释：

```
# Make sure all named pipes get an EOF on them or empty tables
# will hang.
# TODO: currently no way to ensure gpfdist on the write side has
#      written all data available to the pipe.  Closing the pipe
#      while data is remaining causes and EPIPE on the writable
#      gpfdist side.  Need a better way to coordinate this.
time.sleep(self._wait_time)
```

在gptransfer的Python代码中，充斥着各种sleep操作，因为无法精确控制命

名管道的状态，导致的结果是，一个表中哪怕只有几条记录，也需要花费至少十多秒来完成数据的传输。在不追求极致性能的场景，很多时候，命名管道是非常不错的方案，这样就可以与其他程序进行无缝对接，实现不落地的数据导入和导出。

使用 gpfdist 协议外部表导出数据

使用gpfdist协议的可写外部表导出数据时，数据库的Primary将打包好的文本数据发送给gpfdist服务，gpfdist服务将数据写入目标文件中。必须所有Primary都可以访问gpfdist服务，数据是在Primary和gpfdist服务之间直接传输的，不经过Master(断开Master和gpfdist之间的网络并不影响gpfdist外部表的使用)。

要将导出的数据分拆到多个文件中，需要指定多个URI属性，每个URI指定一个文件，并结合可写外部表的DISTRIBUTED子句来决定数据的分布情况。URI中也可以使用通配符，但是，不得匹配多个文件，所以，虽然可以用，实际无意义。URI的数量也不能超过Primary的数量。关于DISTRIBUTED子句，缺省使用的是随机分布的策略，这也是建议的最佳选项，如果一定要选择HASH分布的策略，建议选择需要导出数据的表的分布键作为可写外部表的分布键，如果这样做了，再次使用这些文件导入同构的GP集群时，也需要每个文件都列出来，否则会影响导入的性能，因为通配符匹配的文件是一个一个读取的，这样会大致，实际匹配的Primary是一个一个匹配的。例如：

```
=# CREATE WRITABLE EXTERNAL TABLE test_ext
(a integer, b integer, c integer)
 LOCATION (
    'gpfdist://mdw:8080/tmp/file1.gz',
    'gpfdist://mdw:8080/tmp/file2.gz'
)
FORMAT 'text' (delimiter '|');
=# INSERT INTO test_ext SELECT i+10,i+20,i+30
    FROM generate_series(1,10) i;
```

使用基于命令的 WEB 型外部表导出数据

定义基于命令的WEB型外部表，将数据发送至脚本或命令。当向基于命令的可写WEB型外部表INSERT数据时，数据将会以输入流的形式发送给这些脚本或命令。这些脚本或命令将由GP集群内的所有Primary通过gpadmin用户来执行，这与基于命令的可读WEB型外部表不同，基于命令的可读WEB型外部表可以指定执行脚本或命令的

Primary范围或HOST范围，基于命令的WEB型可写外部表，不行，哪怕Primary上没有数据，脚本或命令也会被调用，因为每个Primary都有自己需要处理的任务，自己的数据不能由其他Primary来处理。外部表中的脚本或命令是由数据库执行的，所以，并不会source gpadmin用户的登录文件(.bashrc或.profile)，在执行命令时，可以手动source这些环境文件，也可以使用预先定义好的GP环境变量。例如：

```
=# CREATE WRITABLE EXTERNAL WEB TABLE test_ext
(a integer, b integer, c integer)
EXECUTE 'more > /tmp/test_ext.out'
FORMAT 'TEXT'
DISTRIBUTED RANDOMLY;
=# INSERT INTO test_ext SELECT i+10,i+20,i+30
   FROM generate_series(1,10) i;
```

关于可用的GP环境变量，可参考"基于命令的WEB型外部表"章节的介绍。

使用 COPY 命令导入导出

COPY是PostgreSQL的命令，非常有用，可以实现非并行的数据导入与导出。通过COPY命令，可以实现把数据从文件导入常规的数据库表中，或者将表中的数据或一个查询的结果导出到文件中，不仅是文件，还可以是标准输入输出。COPY是很多工具的功能基础，比如，备份恢复，集群之间的数据同步，基本上都是在COPY的功能基础上构建的。虽然COPY命令本身是串行的(后来因为gpcopy和gpbackup的需要，引入了ON SEGMENT子句，从命令的层面实现了并行，不过这种并行是针对所有Primary实例的，操作的都是Primary实例的本地文件，与一般意义上的COPY不同)，但如果并行调用，那就是并行了，以前的备份恢复都是通过并行调用COPY命令来实现并行的。

COPY命令每次执行，不论是导入还是导出，都只能指定一个文件，不可以使用通配符匹配多个文件 -- 不认通配符。

还需要注意6版本的语法差异，6版本中可以把格式选项放在WITH()中，在6版本之前是不能这么写的，不过，6版本仍然可以省略WITH关键字，格式选项也可以按照之前版本的格式，连着写。

COPY 导入

使用COPY FROM命令可以将数据从文件或者标准输入追加到目标表中。COPY不是并行的，是串行的，每条数据都需要经过Master串行的处理和分发，因此，COPY应该

只用于数据量不大的导入导出的场景。

COPY的文件必须在Master的主服务进程可以访问的位置。文件路径必须是Master工作目录的相对路径，或者绝对路径。

当COPY数据到STDOUT或从STDIN COPY数据数据时，实际上是GP的Master和客户端之间的数据复制。这样就实现了远程流数据的复制，比如要从一个集群复制少量数据到另一个集群，可以采用如下的命令：

```
$ psql -h src -d srcdb -c 'COPY test TO STDOUT'|psql -h des -d desdb -c 'COPY test FROM STDIN'
```

COPY 导入文件

使用COPY导入文件，实际上是数据库的postgres主服务进程在服务器上打开了指定的文件并完成数据的导入。所以，数据库的运行用户必须拥有该文件的读取权限，文件路径必须是Master工作目录的相对路径，或者绝对路径。例如：

```
=# COPY test FROM '/tmp/file0' DELIMITER '|';
```

COPY 从 STDIN 导入

为了避免需要先将文件复制到Master主机上，使用COPY FROM STDIN命令通过标准输入，将数据输送到Master服务。如果COPY FROM STDIN命令运行时还没有已经就绪的标准输入(比如COPY命令是在其他命令的标准输出之后的匿名管道接着执行)，从COPY FROM STDIN命令执行开始，将会开始接受每行一条记录的数据输入，直到接收到反斜杠和一个小数点(\.)结束。例如：

```
=# copy test from stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself, or an EOF signal.
>> 1    2      3
>> \.
COPY 1
```

以前的gpcrondump类的备份命令，备份出来的数据就是这种格式的文件，数据和SQL混合在一起，直接当做SQL文件执行即可。

\COPY 导入数据

\COPY是一个和COPY完全不同的命令，\COPY的工作原理和COPY FROM STDIN一致的，数据通过psql客户端发送到服务器。因此，\COPY命令操作的是psql客户端所在机器的本地文件，运行psql命令的OS用户必须拥有这些文件的访问权限。\COPY可以理解为对COPY FROM STDIN的包装，也是为了避免将数据文件复制到Master主机上。例如：

```
=# \COPY test FROM '/tmp/file0' DELIMITER '|';
```

可以在任意可以使用psql命令访问GP数据库的机器上运行\COPY命令，可以是任意有文件访问权限的OS用户，甚至是root。

COPY 数据格式

COPY FROM允许使用FORMAT子句来指定输入的数据格式，可选的选项是TEXT、CSV和BINARY(5版本和6版本支持该选项，4版本不支持)。例如：

```
=# COPY test FROM '/tmp/file0' WITH(FORMAT CSV);
```

缺省情况下，CSV格式的字段分隔符为逗号(,0x2C)，TEXT格式的字段分隔符为水平制表符(0x09)，可以使用DELIMITER来指定不同的字段分隔符。例如：

```
=# COPY test FROM '/tmp/file0' WITH(FORMAT CSV, DELIMITER '|');
```

缺省情况下，文件的编码使用的是客户端的编码，可以通过ENCODING来指定文件的编码。例如：

```
=# COPY test FROM '/tmp/file0' WITH(DELIMITER '|', ENCODING 'latin1');
```

COPY的错误记录隔离

缺省请款下，如果导入的数据包含错误的记录，在第一条错误记录发生时，整个操作就会失败退出，且不会有任何数据被导入。如果使用了错误记录隔离模式，数据库会跳过包含错误的记录，并将正确的数据记录导入目标表中。错误记录隔离，仅对格式错误的数据有效，违反约束(比如NOT NULL、CHECK和唯一性约束等)的数据仍会导致整个导入失败，不会有任何数据被导入。

在使用COPY导入数据时，使用SEGMENT REJECT LIMIT子句对错误记录进行隔离。通过该子句可以设置错误记录隔离的阈值，允许按照百分比(PERCENT)或者记录数(ROWS)来限制，当格式错误的记录数量超过设置的阈值，整个导入操作仍然会失败，这个阈值的设定可用于控制数据质量，确保在有大量数据异常时可以失败报错。这个阈值是针全局统计的(编者实测，是全局的，不是每个Primary统计自己的，因为COPY的数据处理是在Master上完成的，Primary根本不会收到错误的数据)，跟外部表不同，而且，如果设置为ROWS的限制，最小值是2，否则会报错。

另外，当使用PERCENT来控制错误记录的阈值时，和可读外部表一样，还通过gp_reject_percent_threshold参数(缺省值为300)来控制最开始不统计失败百分比的记录数，还有一个参数gp_initial_bad_row_limit，该参数的缺省值为1000，当开始的数据全部为错误记录，且达到了该参数设定的值，整个导入操作就失败了。这两个参数都是全局统计的。

和可读外部表一样，通过在SEGMENT REJECT LIMIT子句前加上LOG ERRORS子句实现对错误记录的记录，如果没有指定LOG ERRORS子句，错误记录将直接被丢弃。例如：

```
=# COPY test FROM '/tmp/file0' LOG ERRORS SEGMENT REJECT LIMIT 10;
=# SELECT * FROM gp_read_error_log('test');
=# SELECT gp_truncate_error_log('test');
```

跟可读外部表类似，使用gp_read_error_log()函数来查看隔离的错误记录，使用gp_truncate_error_log()函数来清除隔离的错误记录。不同的是，可读外部表使用外部表的名称作为函数的参数，COPY命令隔离的错误记录，使用目标表的名称作为函数的参数。

COPY导出

使用COPY TO命令，将一张表，或者一个查询语句，导出到一个文件，或者标准输出。COPY TO的数据也全部需要通过Master处理。例如：

```
=# COPY (SELECT * FROM test WHERE a < 100) TO '/tmp/file100';
```

在导出数据是同样可以使用\COPY命令，其工作原理与使用COPY导入是一样的。例如：

```
=# \COPY (SELECT * FROM test WHERE a < 100) TO '/tmp/file100';
```

与数据导入相关的优化

关于数据加载的性能优化，和数据加载后的查询性能，可以参考如下因素：

- 在加载数据之前，先删除目标表上的索引。
  在已经有数据的表上创建索引的性能，可能比带索引增量导入数据的性能，要高很多，因为带索引导入是，每条数据的导入都需要更新索引信息。可以临时增加maintenance_work_mem参数的值，来提升CREATE INDEX命令的的性能。重建索引，应该在没有用户会用到该索引时进行。
- 当需要在一张新表上创建索引时，应该在最后一步来创建索引，在导入所有的数据之后，再创建必要的索引。
- 导入数据后，应该考虑在目标表上是否应该执行ANALYZE操作收集统计信息。当表中有大比例的数据发生变化时，应该考虑执行ANALYZE操作，以更新统计信息，保持统计信息反映了最新的数据情况，有助于优化器生成正确的执行计划。当然，如果不收集统计信息也能生成最优的执行计划，可以不收集。
- 在加载失败之后，应该考虑是否需要执行VACUUM操作。如果目标表不是空表，加载失败之后，垃圾记录和有用的记录混在一起，需要考虑在合适的时间执行VACUUM操作来回收垃圾空间。如果目标表每次加载都是空表，可以通过TRUNCATE来清空目标表。更多关于VACUUM的信息，可以参见”回收空间”章节。

第十二章：安装部署与初始化

GP是一个纯软件的MPP解决方案，可以运行在多种环境，比如，物理机、虚拟机、公有云、私有云、一体机，等等。甚至一个1 Core 1GB的虚拟机中都可以安装运行，执行正常的数据计算和分析。但是，任何的性能都是由硬件保证的，所以，要获得一个计算能力超强的GP集群，一套计算能力超强的硬件是最基础的条件，没有无源之水。

本章节，会从硬件开始介绍，包括硬件的配置指标，预期的性能指标，硬件的搭配平衡，以及整体的物理架构，甚至如何规划机房的摆放等。然后是如何安装操作系统，如何配置操作系统参数，如何安装GP数据库软件，如何初始化一套符合各种安全和指标要求的GP数据库集群。

对于安装好操作系统,配置好网络之后的操作,本章节主要是为了解说相关的知识，编者不再使用这种纯手工的方法，因为效率太低，编者有一个自动化脚本来完成这些重复且容易出错的工作，目前仅在编者为客户提供实施时使用，暂不公开传播。

硬件选型

GP是一个分布式数据库软件，整体数据库的性能依赖于硬件的性能和各种硬件资源的均衡。如果过度强调某一方面硬件资源,会造成资源的不均衡,也是对资源的浪费，同时也是投资的浪费。对于OLAP应用来说，最大的瓶颈是磁盘性能(而不是磁盘容量)，因此，所有其他资源都应该围绕磁盘性能来均衡配置。这些资源包括CPU主频与Core数量、内存容量、网络带宽、RAID性能等，但基本宗旨是，IO资源必须绝对富余，CPU资源永远是被充分利用的资源，内存和网络带宽也必须有富余。

CPU 主频与 Core 数量

CPU主频，理论上来说肯定是越高越好，目前GP还不支持并行度的概念，随着PG版本的进一步合并，未来可能会支持，所以，就目前来说，一个SQL语句的执行性能，取决于单个Core的计算能力和有多少个Primary参与计算，通常，对于一个Primary来说，在执行一个任务时，只能利用一个CPU Core的计算能力。不过，主频这个事情，基本上没有太多的选择，服务器级别的CPU，相同Core数量的情况下，追求很高的主频需要很高的成本，性价比很低。

目前，主流的两路X86服务器，CPU一般都配置64 Core甚至更高的Core数量。主频基本上没有什么选择的空间，所以，要提升集群的整体计算能力，Core的数量是

个非常重要的因素。如果有可能，配置96 Core甚至128 Core以上的CPU，将会带来更好的计算能力。本章介绍的硬件搭配，是均衡的搭配，磁盘的性能是有充分保障的，所以，很多时候，CPU是最先到达瓶颈的资源，因此，如有可能，尽可能的增加CPU Core的数量，越多越好。

内存容量

目前的主流配置，单机内存至少256GB，很多已经到512GB甚至更高，因为这些年来，数据量在不断增长，超大规模数据的关联分析需求上升。GP的很多算子，比如Hash、AGG、Sort都属于内存密集型算子，可能需要消耗大量的内存。

配置了很多的内存，不等于说，所有时间段，活跃内存的比例都能达到很高，活跃内存的消耗量与计算的数据量，计算的类型，并发数，都有关系。只能说，内存多，GP在处理内存需求量很大的场景时会更高效，但不能保证活跃内存的使用率就一定能很高。因此，如果极少有内存密集型计算，可以适当的降低内存的配置。不过，编者建议一个Primary的最低内存配置不要低于30GB。

内存多，是一种能力的体现，在真正的生产运行之前，无法评估真实的活跃内存使用率。也无法根据一个生产系统的情况来评估另一个生产系统的规划，当前的情况也不代表未来的情况。总体来说，内存富余，是好事情，只是用不完而已。在GP集群中，IO资源是最珍贵的资源，如果内存不够用，就要从内存溢出到磁盘文件，把压力转移到磁盘上，这种情况是不应该出现的。

内联网络

内联网络的带宽，是极其重要的。GP是MPP架构，目的是为了解决IO问题，在MPP架构下，IO问题解决了，但带来了一个新的问题，数据的移动，这是无法避免的。但好在现在的网络能力已经足够强，万兆网卡和万兆交换机已经不再昂贵。使用24块全机械盘的主流配置，IO已经可以达到3GB左右的连续读写能力。虽然数据移动不是总会发生，但如果发生，带宽尤其重要，否则数据移动操作就会卡在网络层。

一定要选择万兆网络作为内联网络，要坚决杜绝用千兆网络作为内联网络的想法。另外，为了实现网络的高可用，最少使用两个万兆网口。如果可以选择，要采用mode4的bond，而不是mode1，从安全性角度来说，mode4与mode1是完全一样的，都有多个通道，只要不是全部通道故障，不影响连通性，而从带宽角度考虑，正常情况下，mode4的带宽更高，而mode1的带宽就相当于故障了的mode4，当然，如果循规蹈矩，就要用mode1，这是自由，但是理论和事实就是如此。

一定要确保网络的健康，长期大量的错包丢包，必定会导致非常严重的后果，影响集群的稳定性甚至是性能。

对于超大规模的集群，应该考虑为Master服务器配置更多的网口做链路聚合，因为随着Master管理的机器增多，Master本身的网络压力会跟着上升，100台以上的集群，可以考虑增加Master的网络带宽为4个万兆口做mode4的链路聚合。

RAID卡性能

这个是常常不被引起重视的问题，如果使用的是机械磁盘，RAID卡非常重要，RAID卡是缓解磁盘压力的关键，RAID卡的最大作用是，将不连续的IO操作缓存并合并为连续的IO操作。

使用RAID卡的最关键原因是，普通的机械盘的随机读写能力很差，一个10K/Min转速的机械盘，连续的磁盘读写性能不会超过200MB/S，大多数情况下就100多MB/S。随机读写的性能就更差了，一般IOPS能力都到不了200，这还是磁盘厂商给的指标，实际测试可能更低。

对于OLAP型的应用，主要是大尺寸的连续读写，如果RAID卡有Cache功能，不管是读还是写，都可以经过Raid卡的Cache进行IO合并，充分发挥机械盘的连续读写性能。

一般要求，24块盘机械盘，起码要配置2GB以上Cache的双通道Raid卡，否则Raid卡的性能可能会成为充分发挥磁盘性能的障碍。根据经验，24块机械盘，一般采用12块一组的Raid5方案，条带一般选择256KB的尺寸。

磁盘配置

GP主要是为了解决IO瓶颈而设计的，所以，磁盘性能尤其重要。对OLAP型的应用，目前主流的配置方案是，计算节点主机，配置24块机械盘，Master节点主机一般配置8到12块机械硬盘。如果选择SSD或者NVMe，可以根据容量和性能来评估磁盘数量。

不能只是单纯的追求容量，要综合考虑性能指标，单块磁盘的容量越大，故障恢复的时间就越久。一般来说，如果选择机械盘，目前主流的选择是单盘不超过1.8TB。

容量评估

在规划GP集群时，评估可用容量至关重要，这决定了需要部署什么样规模的集群。对于使用Raid5的磁盘，需要扣除一块磁盘的容量，如果有Hotspare磁盘，该盘容量也不属于可用容量，对于Raid10的磁盘，需要扣除50%的磁盘容量。比如有24块磁盘，做成2个Raid5的盘阵，在没有Hotspare的情况下，实际可用容量是22块磁盘提供的，如果有2块Hotspare，则，实际可用容量是20块磁盘提供的。由于进制换算折损，操作系统中实际显示的磁盘可用容量大约是标称容量的90%。所以：

对于Raid5来说：

```
有效磁盘数量 = 磁盘总数 - Raid数量 - Hotspare磁盘数量
```

对于Raid10来说：

```
有效磁盘数量 = (磁盘总数 - Hotspare磁盘数量) / 2
```

操作系统中看到的存储裸容量为：

```
裸容量 = 单盘容量 * 有效磁盘数量 * 90%(进制换算率)
```

由于磁盘不能使用100%的可用空间，GP建议，为了性能保障，磁盘使用量不能超过70%，所以，数据库可用的磁盘空间为：

```
可用磁盘空间 = 裸容量 * 70%
```

GP还建议为每个Primary预留25%的可用空间用作查询的工作空间，这可能会涉及，临时表，溢出文件等，为了简化计算，可以简单的将数据库可用空间按照裸容量的60%来计算(这里考虑了Mirror的因素，为了便于计算的粗略评估)：

```
数据库可用空间 = 裸容量 * 60%
```

如果集群启用了Mirror，Mirror占用的空间与Primary相同，所以，可存储的数据容量还要之前所有计算的基础上，再折损50%：

```
可存储数据容量 = 裸容量 * 60% * (有Mirror ? 50% : 100%)
```

举个例子来说，24块1.2TB的磁盘，做成两组Raid5，没有Hotspare，有Mirror的情况下，单机可存储的数据容量为：

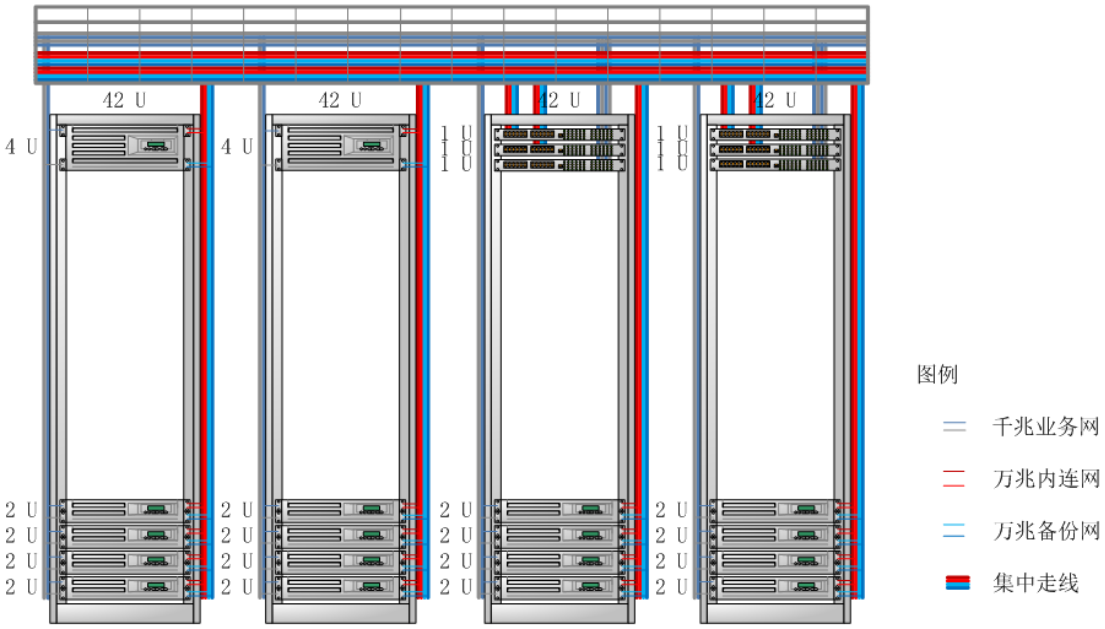
```
单机可存储数据容量 = 1.2TB * (24 - 2 - 0) * 90% * 60% * 50% ≈ 7TB
```

注意：这里得到的[单机可存储数据容量]，没有考虑库内数据压缩问题，当按照入库

前的未压缩平面文件尺寸进行评估时，一般可以考虑1:3的压缩比。

机房规划

机房规划会涉及涉及网络和供电等问题，同时还应该考虑高可用的影响。例如下图，这是一个典型的机房摆放参考图。



建议将Master和Standby分别放置在不同的机柜，将计算主机分组放到不同的机柜，因为这些机器可以形成机柜之间的备份关系。如果有条件，可以将有镜像关系的机柜分开供电，可以确保单个机柜出现断电的情况下，GP数据库可以继续提供服务，这回涉及到镜像策略问题，可以参考”Instance镜像”章节的介绍。

安装操作系统

建议按照推荐的模式安装操作系统，避免在之后安装部署GP集群时带来不必要的麻烦。GP对SWAP和数据盘的划分都有一定的讲究，但这些不是硬性规范，只是建议遵循，也可以咨询专业服务人员的意见。

开启超线程

根据实测经验，开启超线程，计算能力有几乎翻倍的提升，所以，一定要开启超线程，否则是对CPU资源的极大浪费。

Raid划分最佳实践

机械盘，建议做Raid5，一方面可以提高安全级别，另一方面，可以缓解倾斜的影响，如果不在乎成本，Raid10也是可以的。对于机械盘的Raid5，一般采取如下方式配置：

- StripSize -- Master建议32K ~ 128K，准确的选择可以根据测试情况来确定。如果无法确定，可以选择128K。计算节点，建议256K，这是经验值，具体设备的最优值可以根据测试来确定。
- WritePolicy -- 根据最佳实践的经验，需要选择[Always Write Back]或者称为[Force Write Back]。
- ReadPolicy -- 根据最佳实践的经验，需要选择[Read Ahead]。
- Raid Cache比例 -- 根据最佳实践的经验，不建议调整，因为调整了反而综合性能往往会下降。
- Drive Cache -- 根据最佳实践的经验，需要选择[Disable]。
- IO Policy -- 根据最佳实践的经验，需要选择[Direct]。
- 系统盘 -- 操作系统建议安装在单独的两块盘的Raid1上。
- Hotspare -- 如果需要，建议做Global Hotspare。如果更换磁盘的流程不是很长，比如一两天就能更换故障磁盘，建议可以不配置Hotspare。Hotspare不是一定会带来好处，因为出现故障切换时，Raid的性能会有很大下降，且这个过程是自动的，无法根据业务情况灵活安排，而更换磁盘是可以灵活安排的。

Master类主机的Raid划分参考示例：

磁盘	RAID类型	磁盘数量	设备号	可用容量	挂载点	用途
本地盘	RAID 1	2	/dev/sda	2048MB	/boot	操作系统
				剩余尺寸	/	
	RAID 5	X+1+1	/dev/sdb	内存尺寸	swap	SWAP
			/dev/sdc	剩余尺寸	/data	GP数据盘

计算节点类主机的Raid划分参考示例：

磁盘	RAID类型	磁盘数量	设备号	可用容量	挂载点	用途

本地盘	RAID 1	2	/dev/sda	2048MB	/boot	操作系统
				剩余尺寸	/	
	RAID 5	Y+1+1	/dev/sdb	内存尺寸/2	swap	SWAP
			/dev/sdc	剩余尺寸	/data1	GP数据盘
		Y+1+1	/dev/sdd	内存尺寸/2	swap	SWAP
			/dev/sde	剩余尺寸	/data2	GP数据盘

生产环境一般都需要有个Mirror的保护，一旦出现故障切换，Mirror被激活的主机将会消耗更多的内存资源，因此，SWAP是对故障切换的一种保护。

在配置GP数据库时，在健康状态下不应该使用SWAP，如果镜像模式是一一镜像，即，一台主机的镜像全部在另外一台主机，SWAP尺寸应该与物理内存相同，如果一台主机的镜像均分在另外的N台主机，一般可以设置SWAP尺寸为物理内存的1/N。

由于SWAP的性能会极大影响计算效率，Greenplum要求将SWAP设置在性能最好的磁盘上，2块盘组成的Raid1用作OS安装，其性能与数据数据盘的Raid5相比差很多，因此，不可以将SWAP设置在OS磁盘上。

对于SSD或者NVMe的磁盘来说，以上关于性能的问题需要重新衡量，比如SWAP放在哪里，比如是否要做Raid，一般来说，在不考虑Raid5安全要求的情况下，建议不做Raid5，因为Raid5的校验计算对于Raid卡的计算能力来说，会约束磁盘的性能发挥。如果条件允许，可能SSD或者NVMe会是更好的选择，这也是磁盘技术的未来趋势。机械盘在长期高压力下，故障率会高很多，而SSD技术则会稳定很多。

GP安装条件

本节主要按照6版本的情况来介绍，不过，除了GP软件包的安装方式有变化外，其他内容基本上没有太大差异。

支持的操作系统

6版本的GP Server，支持RHEL6.x_x86_64、RHEL7.x_x86_64、CentOS6.x_x86_64、CentOS7.x_x86_64和Ubuntu18.04LTS。

5版本的GP Server，支持RHEL6.x_x86_64、RHEL7.x_x86_64、CentOS6.x_x86_64、CentOS7.x_x86_64、SuSE12SP2、SuSE12SP3、SuSE11SP4。

注意：在RedHat6和CentOS6中使用资源组是有问题的，这是因为早期的cgroup有缺

陷，最好将Kernel升级到2.6.32-696或者更高的版本以修复已知问题，从而可以更好的使用资源组功能。这些问题在Redhat7和CentOS7中已经修复。

注意：Redhat7和CentOS7中，7.3之前的版本，因为有Kernel BUG会导致GP运行大负载任务时出现进程被hang，所以，建议使用7.3及之后版本，7.3及之后的版本解决了这个问题。

软件依赖

在使用rpm安装6版本GP时，下列的软件包是自动检查依赖关系的：

```
apr,apr-util,bash,bzip2,curl,krb5,libcurl,libevent,libxml2,
libyaml,zlib,openldap,openssh,openssl,openssl-libs,perl,
readline,rsync,R,sed,tar,zip
```

这个清单是根据最新的官方文档罗列的，具体的情况以实际安装时的报错提示信息为准。5版本和4版本也都有不完全相同的依赖包，而且安装的模式也各不相同。最好为GP数据库的操作系统配置yum源，在部署GP集群时，缺少的rpm包可以随时补上，不过，不建议通过yum命令来安装GP，yum难以修改安装目录，直接使用rpm命令安装即可，可以方便的修改安装目录(通过--prefix参数指定)。

硬件与网络最低要求

下面列出生产环境中，Linux系统中GP数据库服务器的最低配置要求。GP数据库中所有计算节点的服务器应该具有相同的硬件配置和软件环境。建议由GP工程师检查GP的运行环境，确保其适合GP数据库的生产运行。

最低 CPU 配置	x86_64 兼容的 CPU
最低内存配置	官方文档说每台机器 16GB，编者建议，每个 Primary 30GB
磁盘空间要求	● GP 软件安装 2GB 空间 ● GP 的数据盘需要保持使用量不超过 70%
网络要求	必须是万兆以太网，建议多个网口做 bond 如果坚持使用千以太兆网，后果请自负

文件系统要求

GP数据库运行要求使用XFS文件系统，原厂未明确支持其他文件系统。所以，GP数据库的数据目录，应该使用XFS文件系统。

对于网络文件系统或者共享存储，也必须挂载为本地XFS文件系统。非本地磁盘的文件系统，虽然支持，但不推荐，对于GP来说，都是本地目录，不会区分对待不同的存储。网络文件系统或者共享存储，虽然可以运行，但性能和可靠性无法保证。

安装 RHEL 的介绍

根据实践总结，建议按照如下方式安装RHEL7操作系统，这也是目前的主流选择：

选项	要求			
操作系统版本	rhel-server-7.4-x86_64			
目录及尺寸要求	挂载点	设备名	文件系统格式	尺寸
	/boot	/sda1	XFS	2048MB
	/	/sda2	XFS	剩余全部
	SWAP	/sdb	SWAP	内存尺寸/2
	SWAP	/sdd	SWAP	内存尺寸/2
语言选择	English(United States)			
时区选择	Asia/Shanghai			
软件选择	File and Print Server			
附加组件选择	Development Tools			
初始 root 密码	123456			

如果在安装操作系统时，图形化引导界面中无法配置超过128GB的SWAP，可以先不配置，留在操作系统装好之后再做配置。

GP数据库使用的数据盘，不需要在安装操作系统时配置(上述表格中未列出数据盘)，可以在装好操作系统之后，在准备安装部署GP数据库时再统一按照GP数据库的文件系统挂载要求进行配置。

最好为GP数据库的操作系统配置yum源，在部署GP集群时，可能会有少量的软件包需要安装，对于未按照建议安装的操作系统，可能会有大量的软件包需要安装。

修改操作系统配置

在正式开始安装和部署及初始化GP数据库集群之前，要先对操作系统进行配置修改，以满足GP的运行需要，否则，直接安装多机集群时会有各种失败报错。

需要注意的是，GP数据库的运行是依靠主机名来区分不同主机的，主机名(hostname)在整个集群中必须是唯一的。

禁用 SELinux 和防火墙

以root身份登录，或者获得root权限，比如sudo权限，执行如下命令，禁用SELinux。

RHEL6：

```
# sed s/^SELINUX=.*$/SELINUX=disabled/ -i /etc/selinux/config
# setenforce 0
# service iptables stop
# chkconfig iptables off
```

RHEL7：

```
# sed s/^SELINUX=.*$/SELINUX=disabled/ -i /etc/selinux/config
# setenforce 0
# systemctl stop firewalld.service
# systemctl disable firewalld.service
```

如果安装SuSE支持的版本，SuSE上的操作为：

```
# yast runlevel delete service=boot.apparmor runlevels=B
# yast runlevel delete service=SuSEfirewall2_init runlevels=B
# yast runlevel delete service=SuSEfirewall2_setup runlevels=B
```

修改 hostname

根据不同操作系统的配置方式，修改相应的配置文件，确保主机名被永久修改。可能会涉及如下几个文件：

```
/etc/sysconfig/network
/etc/hostname
/etc/HOSTNAME
```

修改 hosts 文件

修改/etc/hosts文件确保GP集群用到的所有主机名都进行了正确的配置，确保所有GP数据库集群会使用到的网络接口都配置了正确的名称。将修改好的文件覆盖到GP集群中的所有主机上。

修改 sysctl.conf 文件

将 GP 集群中所有主机的/etc/sysctl.conf 文件修改为如下内容：

```
kernel.shmmax = 5000000000000
kernel.shmmni = 32768
kernel.shmall = 40000000000
kernel.sem = 1000 32768000 1000 32768
kernel.sysrq = 1
kernel.core_uses_pid = 1
kernel.msgmnb = 1048576
kernel.msgmax = 1048576
kernel.msgmni = 32768
net.ipv4.tcp_syncookies = 1
net.ipv4.conf.default.accept_source_route = 0
net.ipv4.tcp_max_syn_backlog = 32768
net.ipv4.tcp_syn_retries = 3
net.ipv4.tcp_tw_recycle = 0
net.ipv4.conf.all.arp_filter = 1
net.ipv4.ip_local_port_range = 1025 65535
net.ipv4.ip_local_reserved_ports =
5432,40000-40127,41000-41127,50000-50127,51000-51127
net.ipv6.conf.all.disable_ipv6 = 1
```

```
net.ipv6.conf.default.disable_ipv6 = 1
net.core.netdev_max_backlog = 80000
net.core.rmem_default = 2097152
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
vm.overcommit_memory = 2
vm.overcommit_ratio = 95
vm.swappiness = 0
vm.zone_reclaim_mode = 0
vm.dirty_expire_centisecs = 200
vm.dirty_writeback_centisecs = 100
vm.dirty_background_bytes = 0
vm.dirty_background_ratio = 5
vm.dirty_bytes = 0
vm.dirty_ratio = 10
```

这里就不对每一项做解释了，这个配置是编者一直使用的版本，有兴趣可以对其中的配置项深入研究一下，也欢迎反馈给编者。有些配置设置的比推荐值大了很多，主要目的是为了使用多种硬件配置环境，比如共享内存等，实际上，真实的内存使用量，数据库本身仍会有限制参数。

修改 limits 文件

将 GP 集群中所有主机的/etc/security/limits.conf 文件进行如下修改：

```
# sed -e s/^ulimit/#ulimit/ -i /etc/profile
# sed -e /^[^#]/d -e /^[[:space:]]*$/d -i /etc/security/limits.conf
# echo '' >> /etc/security/limits.conf
# echo '* soft nofile 1048576' >> /etc/security/limits.conf
# echo '* hard nofile 1048576' >> /etc/security/limits.conf
# echo '* soft nproc  1048576' >> /etc/security/limits.conf
# echo '* hard nproc  1048576' >> /etc/security/limits.conf
# nprocname="/etc/security/limits.d/90-nproc.conf"
# if [ -f "/etc/security/limits.d/20-nproc.conf" ];then
#    nprocname="/etc/security/limits.d/20-nproc.conf"
# fi
# if [ -f $nprocname ];then
#    sed -e /^[^#]/d -e /^[[:space:]]*$/d -i $nprocname
#    echo '' >> $nprocname
#    echo '*    soft  nproc  1048576' >> $nprocname
```

```
#     echo 'root  soft  nproc  1048576' >> $nprocname
# fi
```

确认 XFS 挂载参数

RHEL和CentOS的XFS挂载参数为：

```
rw,nodev,noatime,nobarrier,inode64
```

Ubuntu不支持nobarrier参数，因此，其XFS挂载参数为：

```
rw,nodev,noatime,inode64
```

比如在/etc/fstab文件中的挂载配置为：

```
/dev/sdc /data xfs nodev,noatime,nobarrier,inode64 0 0
```

确认 IO 参数和 Huge Page 设置

执行如下命令来修改IO参数和Huge Page设置：

```
# localfile="/etc/rc.d/rc.local"
# if [ -f "/etc/init.d/boot.local" ];then
#     localfile="/etc/init.d/boot.local"
# fi
# chmod +x $localfile
# sed -e "/^####GP_BEGIN/,/^####GP_END/d" -i $localfile
# cat <<'END_OF_CMD'  >> $localfile
> ####GP_BEGIN'
> for i in /dev/sd*;do blockdev --setra 16384 $i;done
> for i in /sys/block/sd*/queue/scheduler;do echo deadline > $i;done
> echo never > /sys/kernel/mm/transparent_hugepage/enabled
> ####GP_END
> END_OF_CMD
# sed -n "/^####GP_BEGIN/,/^####GP_END/p" $localfile|bash
```

确认 ssh 设置

由于GP的集群操作是使用ssh来实现的，在实例多的集群中，可能会同时向一个机器发起多个ssh访问，缺省的ssh设置，可能会遭遇如下报错：

```
ssh_exchange_identification: Connection closed by remote host
```

执行如下命令来修改ssh的MaxStartups参数：

```
# sed s/.*MaxStartups.*/'MaxStartups 400:30:500'/ -i /etc/ssh/sshd_config
```

根据不同的系统，使用不同的命令来生效修改：

```
# ##RHEL7
# systemctl reload sshd.service
# ##RHEL6
# service sshd reload
# ##SuSE
# service ssh reload
```

时钟同步

请根据具体的操作系统，配置时钟同步。GP数据库要求所有机器的时间要保持同步。

创建 GP 数据库的管理员用户

一般，根据惯例，GP数据库的管理员用户使用gpadmin(例如uid为300)这个名称，例如：

```
# groupadd -r -g 300 gpadmin
```

```
# useradd -r -m -g gpadmin -u 300 gpadmin
```

安装GP软件

获取安装包，根据安装包的类型，选择合适的安装方式。比如在6版本，只有rpm包可选，建议使用rpm命令来安装，在5版本和4版本，有zip包的情况下，可以通过解压后使用bash命令执行bin文件安装，开源版本，则需要通过源码编译来安装。例如：

```
$ sudo rpm -ivh greenplum-db-<version>-<platform>.rpm
$ sudo chown -R gpadmin:gpadmin /usr/local/greenplum*
```

建立ssh互信

GP要求所有主机之间，gpadmin用户可以免密ssh访问，在6版本之前，通过命令gpssh-exkeys，可以自动完成所有步骤，而6版本，该命令有了变化，必须先确保Master和其他主机之间已经建立了互信关系，才能完成全局互信。例如：

```
$ ssh-copy-id -i ~/.ssh/id_rsa.pub gpadmin@sdw01
```

在完成Master和其他主机的互信之后，可以使用gpssh-exkeys命令建立全局互信：

```
$ gpssh-exkeys -f hostfile_exkeys
```

其中hostfile_exkeys文件中包含GP集群中所有会被GP数据库使用到的主机名和子网端口名称。例如：

```
mdw
mdw-1
mdw-2
smdw
smdw-1
smdw-2
sdw1
sdw1-1
sdw1-2
sdw2
```

```
sdw2-1
sdw2-2
```

注意：主机名不能有下划线，下划线属于非法字符，在有些系统中可能会带来麻烦。

安装确认

在所有主机上安装完GP数据库软件之后，有必要检查一下所有主机上都已经正确的安装。可以通过如下步骤来验证。

1、 使用gpadmin用户登录Master服务器：

```
# su - gpadmin
```

2、 使用gpssh工具批量检查所有主机上的安装目录：

```
# gpssh -f hostfile_exkeys -e 'ls -l /usr/local/greenplum-db/bin/gpstart'
```

如果安装成功了，该命令将会自动登录所有主机，并列出gpstart命令的文件。

GP 软件目录结构

- greenplum_path.sh -- GP数据库运行需要的环境变量文件。
- bin -- GP数据库的管理命令所在的目录。
- docs/cli_help -- GP数据库命令的帮助信息。
- docs/cli_help/gpconfigs -- gpinitsystem命令的示例文件目录。
- ext -- GP数据库命令所需的一些依赖软件，比如Python。
- include -- GP数据库的一些C语言头文件。
- lib -- GP数据库以及PostgreSQL的库文件。
- sbin -- 内部脚本和命令。
- share -- GP数据库的共享文件。

创建数据库工作目录

GP的所有PostgreSQL实例工作在独立的监听端口，同时，需要工作在独立的文

件系统目录，所以，需要在每台主机上为Primary和Mirror创建工作目录。

创建 Master 的工作目录

通常，Master和Standby的工作目录与计算节点不同，一般来说，Master的工作目录会有一层目录叫做master，以明确的界定这是Master的工作目录，在以后的日常维护工作中，可以更直观和快速的找到Master的工作目录。

Master不存储用户数据，只存储系统表信息，和一些全局信息，不过，这不等于说Master的磁盘指标就不重要。一般来说，Master目录的容量不需要像计算节点那么大，但最好还是要有一定的保障，比如1TB的尺寸还是有必要的，因为会有日志等信息的累积。另外，Master磁盘的性能也不能太差，毕竟系统表的性能影响也不能忽略。如果条件允许，可以考虑为Master目录配置NVMe，保证系统表的性能。

例如，同时在Master(mdw001)和Standby(mdw002)两台主机的/data目录下创建一个子目录master，并将owner改为gpadmin：

```
# ./usr/local/greenplum-db/greenplum_path.sh
# gpssh -h mdw001 -h mdw002 -e 'mkdir -p /data/master/default'
# gpssh -h mdw001 -h mdw002 -e 'chown gpadmin. /data/master'
# gpssh -h mdw001 -h mdw002 -e 'chown gpadmin. /data/master/default'
```

创建 Instance 的工作目录

在6版本之前，tablespace不是一个独立的概念，其必须基于filespace来创建，而filespace在创建时，允许为Primary和Mirror创建不同的路径，所以，按照惯例，为了区分Primary和Mirror，初始化路径也使用primary和mirror子目录来做区分，这也是很容易实现的。

然而，在6版本中，filespace的概念没有了，取而代之的是，在CREATE TABLESPACE时直接指定统一的路径，或者为每个content值的Instance指定一个路径，但不能为Primary和Mirror指定不同的路径。虽然，在初始化安装时，仍然可以按照惯例为Primary和Mirror设置不同的路径，但是，如果再创建tablespace，两者将会遵从不同的路径规则，可能会带来不必要的困扰，因此，编者建议不再遵从这个惯例，在6版本中，Primary和Mirror完全可以使用相同的目录路径(接下来的示例为了便于理解，有些示例可能仍会使用不同的目录路径，但这只是为了示例)。例如，在

sdw001 ~ sdw004上，在/data1和/data2目录下，创建初始化缺省的工作路径：

```
# ./usr/local/greenplum-db/greenplum_path.sh
# gpssh -hsdw{001..004} -e 'mkdir -p /data{1,2}/{default,gpfs}'
# gpssh -hsdw{001..004} -e 'chown gpadmin. /data{1,2}/{default,gpfs}'
```

对于5版本和4版本，可以按照惯例来创建Instance的工作目录：

```
# ./usr/local/greenplum-db/greenplum_path.sh
# gpssh -hsdw{001..004} -e 'mkdir -p
/data{1,2}/{primary,mirror}/{default,gpfs}'
# gpssh -hsdw{001..004} -e 'chown gpadmin. /data{1,2}/{primary,mirror}'
# gpssh -hsdw{001..004} -e 'chown gpadmin.
/data{1,2}/{primary,mirror}/{default,gpfs}'
```

系统性能检查

在真正使用GP数据库集群用于生产应用之前，应该使用GP提供的性能测试命令gpcheckperf对集群中所有主机的硬件性能进行评估检查，确保性能符合预期。

检查网络性能

通过gpcheckperf命令的-r n、-r N或-r M参数来测试网络性能，结果以MB为单位来显示。-r n是进行串行一对一测试，-r N是进行并行一对一测试，-r M是进行全矩阵交叉测试。

从指定的参与测试的网络端口列表，按照顺序进行配对，比如，-r n，按照顺序，第一个向第二个发包，发包完成后，第二个向第一个发包，然后，第三个和第四个配对，顺序相互发包，然后依次测试剩下的网络端口，直到所有的网络端口测试完。如果是-r N，匹配的顺序与-r n一致，不同的是，所有的分组测试同时开始相互发包。对于-r n和-r N的情况，如果是奇数个网络端口，最后一个会和第一个再组成一组，对于-r N的测试，可能会影响最终的显示结果。-r M则是每个网络端口都向其他网络端口发包，形成全矩阵式交叉测试。

-r n和-r N能够体现网卡之间的独立性能，而-r M则更能体现实际生产使用时的网络负载能力，是真实的工作模式。

如果一台机器上有多个网卡，会有不同的子网端口(也可以称为hostname)，在测试时，应该避免同一台机器的不同子网端口之间配对测试，因为这种情况体现的是主机内的本地网络速度，性能很高，对测试结果会是很大的混淆，如果网络有跨交换机的情况，为了测试交换机之间的性能，应该避免同一交换机内的配对，可以使用-r N模式，配置不同交换机的网络端口进行配对测试。

例如，下面是4个计算节点的子网端口情况：

主机名	一号子网端口	二号子网端口
sdw001	sdw001-1	sdw001-2
sdw002	sdw002-1	sdw002-2
sdw003	sdw003-1	sdw003-2
sdw004	sdw004-1	sdw004-2

可以根据子网端口创建两个主机名文件。例如：

hostfile_gpchecknet_ic1	hostfile_gpchecknet_ic1
sdw001-1	sdw001-2
sdw002-1	sdw002-2
sdw003-1	sdw003-2
sdw004-1	sdw004-2

例如，根据创建的主机名文件，分别进行并行配对测试：

```
# ./usr/local/greenplum-db/greenplum_path.sh
# gpcheckperf -f hostfile_gpchecknet_ic1 -r N -d /tmp
# gpcheckperf -f hostfile_gpchecknet_ic2 -r N -d /tmp
```

缺省情况下，数据包发送时间为15秒，可以通过--duration参数来自定义发送的时间长度，单位可以是秒(s)、分钟(m)、小时(h)或天(d)。一般来说，对于-r n和-r N没有必要修改时间长度，而-r M，测试几分钟就差不多了。

检查磁盘性能

通过gpcheckperf命令的-r d参数对磁盘性能做测试，结果以MB为单位来显示。。磁盘性能测试，实际上是使用Linux的dd命令在指定的目录中进行连续的大文件读写测试。

进行磁盘性能测试之前，要确定需要测试的目录，需要测试的主机名称列表，比如hostfile_gpcheckperf文件中包含了需要进行磁盘性能测试的节点的主机名：

```
sdw001
sdw002
sdw003
sdw004
```

例如，要对hostfile_gpcheckperf文件中列出的主机进行测试，测试的目录为/data1和/data2：

```
$ gpcheckperf -f hostfile_gpcheckperf -r d -d /data1 -d /data2 -D
```

缺省情况下，按照当前运行命令主机的物理内存的2倍进行磁盘性能测试，命令会根据指定的目录的数量进行均匀分割尺寸，如果磁盘性能很高，可能会出现dd命令的CPU资源100%的情况，这种情况下，可以通过指定更多的子目录，分拆更多的dd命令来进行测试，切记不要指定完全相同的-d参数，这样会导致测试结果严重失真。有时候会有用户选择使用fio命令来测试磁盘性能，fio是直接对磁盘设备进行操作的，不经过文件系统，所以，会对文件系统产生破坏，如果确实需要使用fio来测试，需要确保设备没有被文件系统使用，或者可以接受损坏的后果。另外，fio的测试结果并不能真实反映文件系统的性能，和gpcheckperf的测试结果不具有直接的可比性。

初始化 GP 数据库集群

在前面的准备工作全部做完之后，就可以开始初始化GP集群了。接下来将逐步介绍初始化一个GP集群的常规方法，定制化方案不会过多提及。实际上，编者对这一章的内容不太感兴趣，因为编者早就不这样安装部署和初始化GP数据库集群了，因为这样太费劲了，完全就是体力活，完全可以自动完成，所以，编者一直致力于优化一个更好的自动化脚本来完成本章的工作。但对于学习来说，这部分内容还是很有价值的，因为这非常有助于了解如何一步一步初始化一个GP集群，熟悉GP集群的架构关系，对日常的开发使用和运维都会有很大的帮助。

创建初始化网络端口文件

gpinitsystem命令可以通过一个网络端口清单文件来指定，在哪些主机上初始化GP集群，该文件指定的是计算节点，不包含Master和Standby主机。初始化命令根据gpinitsystem_config配置文件中指定的目录个数来决定在一个计算节点的主机上初始化多少个Instance。当一个主机上有多个子网端口时，多个Instance将会均分到所有的子网端口上。文件的格式为，每个网络端口一行。

比如，主机名如下所示：

```
sdw001
sdw002
sdw003
sdw004
```

子网端口可以采用-1、-2这种加后缀的方式来命名。要在GP集群中使用这些子网端口，需要在初始化网络端口文件中，使用这些子网端口(可以不使用主机名，gpinitsystem会自动获取主机名来分析各个子网端口属于哪个主机)，初始化时，Instance将会分散到不同的子网端口上。例如，网络端口文件hostfile_gpinitsystem中包含如下信息：

```
sdw001-1
sdw001-2
sdw002-1
sdw002-2
sdw003-1
sdw003-2
sdw004-1
sdw004-2
```

注意：不管是主机名还是子网端口，都必须已经在所有主机的/etc/hosts文件中配置好，否则gpinitsystem命令会报错说主机名无法识别，通过DNS配置也是可以的。

创建初始化配置文件

初始化配置文件，决定了gpinitsystem创建一个什么样的集群。数据库软件安装好之后，在安装目录下已经有一个初始化配置文件模板：

```
$GPHOME/docs/cli_help/gpconfigs/gpinitsystem_config
```

可以通过拷贝模板的方式来创建初始化配置文件。例如：

```
$ cat $GPHOME/docs/cli_help/gpconfigs/gpinitsystem_config >
~/gpinitsystem_config
```

根据环境的情况修改刚刚复制的文件。一个GP系统，必须要有一个Master，也必须要有Primary，如果有Mirror，一般至少需要两个计算节点主机。

通过DATA_DIRECTORY参数来指定一个主机上配置多少个Primary。当一个主机

上有多个子网端口时，多个Instance将会均分到所有的子网端口上。PORT_BASE指定了端口的开始值，这些端口最好设置在/etc/sysctl.conf文件中net.ipv4.ip_local_reserved_ports参数的范围内，这样的话，这些端口就不会被作为动态端口分配给客户端程序。

```
ARRAY_NAME="Greenplum Data Platform"
SEG_PREFIX=gpseg
PORT_BASE=40000
declare -a DATA_DIRECTORY=(/data1/primary /data1/primary /data1/primary /
data2/primary /data2/primary /data2/primary)
MASTER_HOSTNAME=mdw001
MASTER_DIRECTORY=/data/master
MASTER_PORT=5432
TRUSTED_SHELL=ssh
CHECK_POINT_SEGMENTS=8
ENCODING=UNICODE
```

SEG_PREFIX指的是，在DATA_DIRECTORY声明的目录下创建的子目录的前缀，后缀是Instance的Content(参考gp_segment_configuration系统表的content字段)值。PORT_BASE指的是，在一个主机上，Primary的监听端口从多少开始连续分配，比如上述的例子中，该参数的值为40000，而DATA_DIRECTORY参数指定了6个目录，就会在一台主机上初始化6个Primary，这6个Primary的监听端口就依次为40000 ~ 40005。DATA_DIRECTORY实际上是一个数组，指定了将在每个主机的哪些目录下初始化Primary，上述的例子中，指定了6个目录，初始化命令将会在这6个目录下初始化6个Primary，这里的目录是可以重复的，因为，真正的工作目录是这些目录下再创建的子目录，由SEG_PREFIX参数指定了子目录的前缀，由content值决定了子目录的后缀。MASTER_HOSTNAME参数指定了Master所在的主机的主机名，初始化命令会将该主机作为Master。MASTER_DIRECTORY指定了Master的工作目录，不过，Master的实际工作目录也是要创建一个子目录，规则与Primary相同，content值为-1，所以，常见的名称为gpseg-1。MASTER_PORT指定了Master的监听端口，这是整个GP集群的访问入口端口，客户端程序通过该端口来访问数据库集群。TRUSTED_SHELL指的是使用什么命令来执行远程命令，之前准备的互信在这里就开始用上了。CHECK_POINT_SEGMENTS指定了WAL检查点最多可以等待的WAL文件个数，如无必要，建议不修改，增加该参数的值，checkpoint的频繁度会降低，但崩溃后恢复的时间就会延长，数据库初始化之后，仍然可以通过checkpoint_segments参数来修改这一配置。ENCODING参数指定了服务端的编码集，建议不要修改，对于中文环境，没有更好的选择，与编码集相关的还有--locale、--lc-collate和--lc-ctype参数，这几个是gpinitsystem的命令行参数，具体可以参考PostgreSQL文档，值得注意的是，在6版本之前，CREATE DATABASE命令没有LC_COLLATE选项，缺省的排序行为遵循UTF8编码集，这会导致，ascii字符串的排序不是按照ascii编码进行排序的，很多时候不符合预期，可以通过--lc-collate参数在初始化时指定排序特征。在6版本中，要创建一个LC_COLLATE属性与模板库不同的Database是有限制的，模板库只能是template0，否则会报错失败。

如果在初始化时选择配置Mirror，还需要配置Mirror相关的参数，通过MIRROR_PORT_BASE参数指定Mirror端口的开始值，该参数的特点与PORT_BASE完全相同。例如：

```
#REPLICATION_PORT_BASE=41000
#MIRROR_REPLICATION_PORT_BASE=51000
MIRROR_PORT_BASE=50000
declare -a MIRROR_DATA_DIRECTORY=(/data1/mirror /data1/mirror /data1/
mirror /data2/mirror /data2/mirror /data2/mirror)
```

注释掉的两行，是6以前的版本需要用到的端口，REPLICATION_PORT_BASE是Primary的复制端口，MIRROR_REPLICATION_PORT_BASE是Mirror的复制端口，6版本已经不再需要这两个参数，6版本不再使用filerep进行Primary和Mirror之间的同步复制，而是使用WAL复制。

MIRROR_PORT_BASE的特点可以参考前面讲到的PORT_BASE参数的解释，MIRROR_DATA_DIRECTORY指定的是Mirror的工作目录信息，特点参考前面讲到的DATA_DIRECTORY参数。

关于Primary的工作目录和Mirror工作目录是否要相同，参见”创建Instance的工作目录”章节。

执行初始化操作

通过gpinitsystem命令，根据配置文件中设置的情况，初始化一个新的GP数据库集群。该操作需要使用gpadmin用户来进行。例如：

```
$ ./usr/local/greenplum-db/greenplum_path.sh
$ gpinitsystem -c gpinitsystem_config -h hostfile_gpinitsystem
```

如果要在初始化的时候设置Standby，可以通过-s参数指定Standby的主机名来实现，例如：

```
$ gpinitsystem -c gpinitsystem_config -h hostfile_gpinitsystem -s mdw002
```

如果在初始化时选择了设置Mirror，可以通过--mirror-mode参数来指定Mirror的策略，自带的Mirror策略有两种，group和spread，缺省为group。根据主机名的排序，所有的主机组成一个环，在环上的每个主机，其Primary对应的Mirror会分布在后续的主机上。group的意思是，一个主机上所有Primary对应的Mirror都在环中的下一个主机上。spread的意思是，一个主机上所有Primary对应的Mirror一个一个分散到后面的多个主机上，当前主机上有多少个Primary，Mirror就分散到

多少个后续主机上。因此，group镜像策略，要求最少要有两台计算节点主机，spread镜像策略，要求最少的计算节点主机的数量要大于一台主机上Primary的数量。

如果要定制镜像策略，可以在gpinitsystem时不配置Mirror，在初始化成功之后，通过gpaddmirrors命令来创建Mirror，通过对Mirror配置文件进行定制化编辑的方式实现任意的Mirror策略。编者的自动化命令实现了更复杂和优化的镜像策略，可以参考"Instance镜像"章节。

在执行gpinitsystem命令过程中，命令会自动检查所有主机的环境，初始化参数文件，确定可以继续初始化后，会提示确认是否继续，根据提示输入需要的信息后继续初始化安装操作。例如：

```
Continue with Greenplum creation? Yy/Nn
```

gpinitsystem命令在确认输入[Yy]后，会继续进行并行的集群初始化操作，在初始化成功之后，GP数据库集群就处于已启动状态，并会输出如下消息：

```
Greenplum Database instance successfully created
```

初始化异常排查

在初始化过程中，如果某个Instance创建或启动失败，都会导致初始化报错失败，初始化操作的日志信息会存储在gpadmin用户的gpAdminLogs目录下，日志文件以命令的名称和日期命名。例如：

```
gpinitsystem_20200705.log
```

通常，在初始化日志中会有相关的失败原因，比如操作系统参数没有正确的修改导致内存分配不足，或者SELinux未禁用，或者防火墙未关闭等，SELinux和防火墙的问题可能会表现为无法访问远程主机或数据库Instance。对于实例启动失败的情况，还可以查看具体Instance的日志，日志中会详细显示为什么实例启动失败。应该根据日志信息解决相关的问题，之后再重新初始化集群。

回退脚本

在gpinitsystem命令失败时，会给出一个回退脚本，位于gpadmin用户的gpAdminLogs目录下。格式为：

```
~/gpAdminLogs/backout_gpinitsystem_<user>_<timestamp>
```

通过运行该回退脚本，将会清除由gpinitsystem命令产生的目录，文件和日志等信息，关闭残余的postgres数据库进程，清理完成后，解决了失败的原因，就可以重新尝试初始化GP集群了。例如：

```
$ sh backout_gpinitsystem_gpadmin_20200705_141658
```

为 gpadmin 用户配置环境变量

为了使得gpadmin用户每次登陆操作系统之后，能够直接对数据库进行操作，需要为gpadmin用户配置好必要的环境变量。通常修改.bashrc配置文件，比如使用vi命令修改。例如：

```
$ vi ~/.bashrc
```

加入如下环境变量设置信息：

```
####GP_BEGIN
if [ -e /usr/local/greenplum-db ];then
. /usr/local/greenplum-db/greenplum_path.sh
fi
export MASTER_DATA_DIRECTORY=/data/default/gpseg-1
export PGPORT=5432
export LD_PRELOAD=/lib64/libz.so.1 ps
####GP_END
```

如果有必要,还可以指定缺省的登录用户和登录的数据库名称,甚至指定登录密码。例如：

```
export PGUSER=gpadmin
export PGDATABASE=postgres
export PGPASSWORD=gparray
```

第十三章：启动与停止GP数据库

在使用GP数据库时，常规的启动和停止数据库都是一个分布式操作。所有主机上的Master、Standby、Primary和Mirror，共同组成了一个分布式系统的整体，表现为一个统一的数据库系统，所以，启动，是所有实例都启动，停止，是所有实例都停止。

既然是分布式系统，启动过程自然也就不同于一般的数据库系统。细心的用户可能已经发现，整个启动过程大致可以分为5个阶段：

1. 启动Master，以Master Only的模式启动，其效果与gpstart -m一致。
2. 获取集群的配置信息。
3. 关闭Master，其效果与gpstop -m一致。
4. 启动所有的Primary和Mirror。
5. 正常启动Master。

这里需要解释一下，为什么会分为这么多阶段，因为这是一个分布式系统，整个系统的信息是存储在数据库中的，我们在安装配置GP集群的时候，会涉及到Master的主机名、目录和端口，Standby的主机名、目录和端口，所有Primary以及Mirror的主机名、目录和端口，这些信息统一保存在gp_segment_configuration系统表中。当然，这些信息不可能保存在Master的某个配置文件中(虽然表中存储数据的也是文件)，因为那样太容易被篡改了。

因此，启动命令需要根据MASTER_DATA_DIRECTORY环境变量或者-d参数来确定Master的工作目录，读取postgresql.conf配置文件，获取port参数，启动Master实例。gp_segment_configuration系统表其实是Master Only的系统表，Master启动之后，启动命令就可以通过Utility模式获取整个集群的配置信息，然后关闭Master(因为此时的Master的模式不能提供正常的连接和访问)，然后根据获取的集群配置信息来启动Primary和Mirror，一定要先启动Primary和Mirror，因为，只有数据是完整的，数据库才能提供正常的服务，Primary和Mirror能否正常启动，直接决定了集群启动的成败，如果有成对的Primary和Mirror启动失败，将会报出如下错误，且数据库启动整体失败：

```
Do not have enough valid segments to start the array.
```

这种情况，一般称为double fault、double down，或者双宕。解决这种问题并没有什么简单的方法，找到最后一个没有在gp_segment_configuration系统表中被标记为down的启动失败的Instance，根据报错日志解决相关的问题，之后再尝

试重新启动集群。这种情况不是gprecoverseg命令能处理的，因为数据库已经不可用了，所有依赖数据库可正常工作的命令都已经不能正常工作，gprecoverseg是根据健康的Primary或Mirror来恢复与其对应的Mirror或Primary，所以该命令无法处理双宕故障。

通过位于安装路径的bin目录下的gpstart和gpstop命令来启动和停止GP数据库集群，这两个命令和PostgreSQL的pg_ctl命令的功能类似，区别是，gpstart和gpstop提供了集群操作，对于单实例的操作，实际上仍然可以通过pg_ctl来完成，编者提醒，pg_ctl命令应该仅在必要时刻使用。

注意：尽量不要使用OS的kill命令来终止Postgres进程，而是要使用pg_cancel_backend函数或者pg_terminate_backend函数来完成。当然也不是完全不可以用，除非有把握确保不会导致数据库损坏。通过kill -9或者kill -11可能会导致数据库崩溃，且无法记录异常日志，以至于无法进行RCA(root cause analysis)。另外，kill -9或者kill -11即便没有导致数据库宕机，也会导致所有连接中断，这个副作用是必然会发生的。

启动 GP 数据库

在Master上，使用gpstart命令来启动一个已经存在的GP集群。GP数据库在初始化成功时已经处于启动状态，不需要执行gpstart命令，对已经启动的集群执行gpstart命令会收到如下报错：

```
Master instance process running
```

对于初始化成功之后的GP集群，已经处于停止状态的情况下，通过gpstart命令来启动集群，gpstart命令会完成整个集群的启动，Primary和Mirror的启动是并行的。gpstart命令应该在Master主机上，由gpadmin用户来执行：

```
$ gpstart
```

gpstart命令常见的参数有：

1. -a，不提示确认，如果没有指定该参数，将会有确认继续的提示信息：

```
Continue with Greenplum instance startup Yy|Nn (default=N):
```

2. -m，启动的只是Master实例，在运维的时候会经常用到。不过，对于Primary实例来说，也可以使用gpstart -m的方式来单独启动，比如在处理故障时，可以通过这种方式来测试某个Primary是否可以正常启动，虽然启动时会收到如下错误，但这并不影响成功启动。

4版本和5版本的错误信息为：

```
gpstart failed. (Reason='Database does not contain gp_fault_strategy entry')
exiting...
```

6版本的错误信息为：

```
FATAL - no master dbs defined!
gpstart failed. (Reason='Error: GpArray() - no master dbs defined')
exiting...
```

3. -R，限制模式启动数据库，在运维的时候会经常用到。限制模式，仅允许SUPERUSER连接数据库，在进行系统表VACUUM FULL操作时，限制模式将可以确保VACUUM FULL不受其他用户访问的影响，尽量命令执行的时间。

4. -B，同时启动的实例数量，缺省值为64，最大值为128，一般不需要修改此参数。

停止 GP 数据库

要停止一个GP数据库系统，使用gpstop命令来完成，只要Master是活着的，gpstop会尝试停止集群中的所有实例，包括Standby、Primary和Mirror，哪怕这些实例已经停止。缺省情况下，gpstop命令会一直等到所有连接都断开才停止数据库。

gpstop命令常见的参数有：

1. -a，不提示确认，如果没有指定该参数，将会有确认继续的提示信息：

```
Continue with Greenplum instance shutdown Yy|Nn (default=N):
```

2. -m，只关闭Master实例，在运维的时候会经常用到。与gpstart命令不同的是，不能使用gpstop命令来单独停止Primary和Mirror实例，应该使用pg_ctl stop的方式来停止。例如：

```
$ pg_ctl stop -D /data/primary/default/gpseg0
```

3. -B，同时停止的实例数量，缺省值为64，最大值为128，一般不需要修改此参数。

4. -u，在不停止数据库的情况下，使得pg_hba.conf配置文件的修改生效，使得postgresql.conf配置文件中的运行时参数生效。类似于PostgreSQL的pg_ctl reload命令，不同的是，gpstop -u是并行的，所有实例都会生效参数的修改。需要注意的是，对于postgresql.conf配置文件中的参数，那些启动参数不会立即生效，必须等到数据库重启时才能生效。

5. -M,停止数据库的模式,可选值有:smart、fast、immediate,实际上,-M smart、-M fast和-M immediate分别对应了三种缩写:-s、-f和-i,虽然help中没有说明,但是命令是这样工作的,详情可以参考gpstop的Python源码。缺省情况下,是smart模式,gpstop命令会一直等到所有连接都断开才停止数据库,一般无法满足这个条件。最常使用的是fast模式,该模式会中断并回滚正在执行的事务。immediate模式一般不建议使用,该模式有时可能会导致严重的数据损坏,需要手动恢复数据库。

6. --host,停止指定主机名上的所有实例。该参数主要用于将指定主机名的机器剥离集群。

7. -r,停止数据库并重新启动。该方式在重新启动数据库时不会在命令行输出日志信息,所以,一般不建议这样操作,建议停止数据库之后通过执行gpstart命令来启动数据库。

访问 Master Only 模式的 Master

通过gpstart -m启动的是Master Only模式,该模式下,启动的只有Master实例,不会启动任何Primary和Mirror,也不会启动Standby。通常是为了进行特定的维护操作。

注意: Master Only模式应该由专业服务人员进行操作。在该模式下,专业服务人员将可以只对Master实例进行访问,以便进行Catalog的修改等操作。修改Catalog的操作属于高风险操作,建议不要擅自尝试,否则后果自负。

使用gpstart的-m参数来启动Master Only模式:

```
$ gpstart -m
```

使用Utility模式连接Master实例。例如:

```
$ PGOPTIONS='-c gp_session_role=utility' psql postgres
```

在完成维护操作之后应该停止Master Only模式的Master实例。然后再正常启动GP数据库集群。例如:

```
$ gpstop -m
```

注意: Master Only模式修改Catalog可能会导致集群状态不一致,甚至导致数据库无法正常启动,这个操作应该由专业服务人员进行操作。通常Master Only模式修改Catalog会导致Master和Standby的不一致,可能需要重建Standby,所以,正常

启动数据库之后，需要注意Standby的启动信息以及同步状态。

中断客户端进程

GP数据库会为每个客户端连接启动一个后台进程，计算实例上也会有相应的后台进程。SUPERUSER可以取消查询或者中断客户端连接的后台进程。

通过pg_cancel_backend()函数来取消正在执行或者排队的SQL，通过pg_terminate_backend()函数来中断客户端连接。

pg_cancel_backend()函数有两种参数模式(不过并不是所有版本都提供msg参数选项，4版本没有msg参数，该参数从5版本开始提供)：

- pg_cancel_backend(pid int4)
- pg_cancel_backend(pid int4, msg text)

pg_terminate_backend()函数也有两种参数模式(不过并不是所有版本都提供msg参数选项，4版本没有msg参数，该参数从5版本开始提供)：

- pg_terminate_backend(pid int4)
- pg_terminate_backend(pid int4, msg text)

pg_cancel_backend()函数，是取消正在执行或者排队的SQL，但不会中断客户端的连接，客户端在SQL被取消之后，仍然可以继续执行其他SQL。pg_terminate_backend()函数是中断客户端连接，客户端将无法继续使用被中断的连接，要想继续访问数据库，必须重连建立数据库连接。

如果提供了msg参数，数据库在取消SQL或者中断客户端连接时，会将该信息发送给客户端。消息的长度限制是128个字节，超出的部分会被截断，如果有中文被截断导致半个中文，客户端将会收不到msg信息，而是收到如下的报错信息：

```
ERROR:  Message skipped due to incorrect encoding.
```

如果pg_cancel_backend()函数和pg_terminate_backend()函数执行成功了，将会返回true，否则会返回false，不过，返回true不等于说SQL就立即被取消了，或者说连接就立即被中断了，有时，对于一些复杂事务，可能需要执行多次才能达到取消和中断的目的。对于一些使用了外部资源的查询来说，还可能出现无法取消和中断的情况，此时可能需要找到使用外部资源的子进程，先杀死这些子进程。

要取消查询或中断连接，要先找到后台的进程号。可通过pg_stat_activity系统视图来查询，该视图在6版本有了较大的变化，在6版本中，进程号的字段为pid，在6版本之前，进程号的字段为procpid。例如，查看所有正在执行和排队的SQL的信息：

```
=# SELECT usename, pid, waiting, state, query, datname FROM pg_stat_activity;
```

查询输出的示例：

	usename name	pid integer	waiting boolean	state text	query text	datname name
1	gpadmin	1544	f	idle	set client encoding to 'UNICODE'	postgres
2	gpadmin	1499	f	active	SELECT usename, pid, waiting, sta	postgres
3	gpadmin	1542	f	idle	SELECT usename, pid, waiting, sta	postgres

可以根据查询的输出来确定需要取消或中断的后台进程号。比如，要中断查出来的空闲连接，并通知客户端：”中断空闲连接”，可以这样执行中断函数：

```
=# SELECT pg_terminate_backend(1542,'中断空闲连接');
```

```
FATAL:  terminating connection due to administrator command: "中断空闲连接"
server closed the connection unexpectedly
       This probably means the server terminated abnormally
       before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
```