# HashData

# 目录

# 最佳实践

最佳实践是指能持续产生比其它方法更好结果的方法或者技术，它来自于实战经验，并被证实了遵循这些方法可以获得可靠的预期结果。

本最佳实践旨在通过利用所有可能的知识和技术为正确使用 HashData 数据仓库提供有效参考。

本文不是在教您如何使用 HashData 数据仓库的功能，而是帮助您在设计、实现和使用 HashData 数据仓库时了解需要遵循哪些最佳实践。

本文目的不是要涵盖整个产品或者产品特性，而是概述 HashData 数据仓库实践中最重要的因素。

本文不涉及依赖于 HashData 数据仓库具体特性的边缘用例，后者需要精通数据库特性和您的环境，包括 SQL 访问、查询执行、并发、负载和其它因素。

通过掌握这些最佳实践知识，会增加 HashData 数据仓库集群在维护、支持、性能和可扩展性等方面的成功率。

无论是技术还是产品角度，HashData 数据仓库从企业级数据库 PostgreSQL 和开源 MPP 数据库 Greenplum Database 继承了很多功能特性，尤其是数据分析接口和使用习惯方面，HashData 数据仓库 一直保持跟 Greenplum Database 100% 的兼容。

这个最佳实践文档也是在《Greenplum Database 最佳实践》基础上，根据 HashData 数据仓库自身的特点做了相应调整而成稿的。

所以，这个文档里面提到的很多方法或者技术，可以直接应用到 Greenplum Database 上面。

本部分概述 HashData 数据仓库最佳实践所涉及的概念与要点。

# 最佳实践提纲

## 数据模型

- HashData 数据仓库是一个基于大规模并行处理（MPP）和无共享架构（shared nothing）的分布式分析型数据库。这种数据库的数据模式与高度规范化的事务性（OLTP）数据库显著不同。通过使用非规范化数据库模式，例如具有大事实表和小维度表的星型或者雪花模式，HashData 数据仓库在处理分析型业务（OLAP）时表现优异。
- 跨表关联（JOIN）时字段使用相同的数据类型。

## 堆存储和追加优化存储

- 若表和分区表需要进行迭代式的批处理或者频繁执行单个 UPDATE、DELETE 或 INSERT 操作，使用堆存储。
- 若表和分区表需要并发执行 UPDATE、DELETE 或 INSERT 操作，使用堆存储。
- 若表和分区表在数据初始加载后更新不频繁，且仅以批处理方式插入数据，则使用 AO 存储。
- 不要对 AO 表执行单个 INSERT、UPDATE 或 DELETE 操作。
- 不要对 AO 表执行并发批量 UPDATE 或 DELETE 操作，但可以并发执行批量 INSERT 操作。

## 行式存储和列式存储

- 若数据需要经常更新或者插入，则使用行存储。
- 若需要同时访问一个表的很多字段，则使用行存储。
- 对于通用或者混合型业务，建议使用行存储。
- 若查询访问的字段数目较少，或者仅在少量字段上进行聚合操作，则使用列存储。
- 若仅常常修改表的某一字段而不修改其他字段，则使用列存储。

## 压缩

- 对于大 AO 表和分区表使用压缩，以提高系统 I/O。
- 在字段级别配置压缩。
- 考虑压缩比和压缩性能之间的平衡。

## 分布

- 为所有表定义分布策略：要么定义分布键，要么使用随机分布，不要使用缺省分布方式。
- 优先选择可均匀分布数据的单个字段做分布键。
- 不要选择经常用于 WHERE 子句的字段做分布键。
- 不要使用日期或时间字段做分布键。
- 分布键和分区键不要使用同一字段。
- 对经常执行 JOIN 操作的大表，优先考虑使用关联字段做分布键，尽量做到本地关联，以提高性能。
- 数据初始加载后或者每次增量加载后，检查数据分布是否均匀。
- 尽可能避免数据倾斜。

## 内存管理

- 使用 statement_mem 控制节点数据库为单个查询分配的内存量。
- 使用资源队列设置队列允许的当前最大查询数（ACTIVE_STATEMENTS）和允许使用的内存大小（MEMORY_LIMIT）。

- 不要使用默认的资源队列，为所有用户都分配资源队列。
- 根据负载和时间段，设置和队列实际需求相匹配的优先级（PRIORITY）。
- 保证资源队列的内存配额不超 gp_vmem_protect_limit。
- 动态更新资源队列配置以适应日常工作需要。

## 分区

- 只为大表设置分区，不要为小表设置分区。
- 仅在根据查询条件可以实现分区裁剪时使用分区表。
- 建议优先使用范围 (Range) 分区，否则使用列表 (List) 分区。
- 根据查询特点合理设置分区。
- 不要使用相同的字段既做分区键又做分布键。
- 不要使用默认分区。
- 避免使用多级分区；尽量创建少量的分区，每个分区的数据更多些。
- 通过查询计划的 EXPLAIN 结果来验证查询对分区表执行的是选择性扫描（分区裁剪）。
- 对于列存储的表，不要创建过多的分区，否则会造成物理文件过多： *Physical files = Segments × Columns × Partitions* 。

## 索引

- 一般来说 HashData 数据仓库中索引不是必需的。
- 对于高基数的列式存储表，如果需要遍历且查询选择性较高，则创建单列索引。
- 频繁更新的列不要建立索引。
- 在加载大量数据之前删除索引，加载结束后再重新创建索引。
- 优先使用 B 树索引。
- 不要为需要频繁更新的字段创建位图索引。
- 不要为唯一性字段，基数非常高或者非常低的字段创建位图索引。
- 不要为事务性负载创建位图索引。
- 一般来说不要索引分区表。如果需要建立索引，则选择与分区键不同的字段。

## 资源队列

- 使用资源队列管理集群的负载。
- 为所有角色定义适当的资源队列。
- 使用 ACTIVE_STATEMENTS 参数限制队列成员可以并发运行的查询总数。
- 使用 MEMORY_LIMIT 参数限制队列中查询可以使用的内存总量。
- 不要设置所有队列为 MEDIUM，这样起不到管理负载的作用。
- 根据负载和时间段动态调整资源队列。

# ANALYZE

- 不要对整个数据库运行 ANALYZE，只对需要的表运行该命令。
- 建议数据加载后即刻运行 ANALYZE。
- 如果 INSERT、UPDATE 和 DELETE 等操作修改大量数据，建议运行 ANALYZE。
- 执行 CREATE INDEX 操作后建议运行 ANALYZE。
- 如果对大表 ANALYZE 耗时很久，则只对 JOIN 字段、WHERE、SORT、GROUP BY 或 HAVING 字句的字段运行 ANALYZE。

# VACCUM

- 批量 UPDATE 和 DELETE 操作后建议执行 VACUUM。
- 不建议使用 VACUUM FULL。建议使用 CTAS（CREATE TABLE…AS）操作，然后重命名表名，并删除原来的表。
- 对系统表定期运行 VACUUM，以避免系统表臃肿和在系统表上执行 VACUUM FULL 操作。
- 禁止杀死系统表的 VACUUM 任务。
- 不建议使用 VACUUM ANALYZE。

# 加载

- 使用 gpfdist 进行数据的加载和导出。
- 随着 Segment 数据库个数的增加，并行性增加。
- 尽量将数据均匀地分布到多个 ETL 节点上。
- 将非常大的数据文件切分成相同大小的块，并放在尽量多的文件系统上。
- 一个文件系统运行两个 gpfdist 实例。
- 在尽可能多的网络接口上运行 gpfdsit。
- 使用 gp_external_max_segs 控制访问每个 gpfdist 服务器的 Segment 数据库的个数。 建议 **gp_external_max_segs** 的值和 **gpfdist** 进程个数为偶数。
- 数据加载前删除索引，加载完后重建索引。
- 数据加载完成后运行 ANALYZE 操作。
- 数据加载过程中，设置 gp_autostats_mode 为 NONE，取消统计信息的自动收集。
- 若数据加载失败，使用 VACUUM 回收空间。

# gptransfer

- 为了更好的性能，建议使用 gptransfer 迁移数据到相同大小或者更大的集群。
- 避免使用 –full 或者 –schema-only 选项。建议使用其他方法拷贝数据库模式到目标数据库，然后迁移数据。
- 迁移数据前删除索引，迁移完成后重建索引。
- 使用 SQL COPY 命令迁移小表到目标数据库。
- 使用 gptransfer 批量迁移大表。
- 在正式迁移生产环境前测试运行 gptransfer。试验 –batch-size 和 –sub-batch-size 选项以获得最大平行度。如果需要，迭代运行多次 gptransfer 来确定每次要迁移的表的批次。
- 仅使用完全限定的表名。表名字中若含有点、空格、单引号和双引号，可能会导致问题。
- 如果使用 –validation 选项在迁移后验证数据，则需要同时使用 -x 选项，以在源表上加排它锁。
- 确保在目标数据库上创建了相应的角色、函数和资源队列。gptransfer -t 不会迁移这些对象。
- 从源数据库拷贝 postgres.conf 和 pg_hba.conf 到目标数据库集群。
- 使用 gppkg 在目标数据库上安装需要的扩展。

# 安全

- 妥善保护 gpadmin 账号，只有在必要的时候才能允许系统管理员访问它。
- 仅当执行系统维护任务（例如升级或扩容），管理员才能以 gpadmin 登录 HashData 数据仓库集群。
- 限制具有 SUPERUSER 角色属性的用户数。HashData 数据仓库中，身为超级用户的角色会跳过所有访问权限检查和资源队列限制。仅有系统管理员具有数据库超级用户权限。参考《HashData 数据仓库管理员指南》中的“修改角色属性”。
- 严禁数据库用户以 gpadmin 身份登录，严禁以 gpadmin 身份执行 ETL 或者生产任务。
- 为有登录需求的每个用户都分配一个不同的角色。
- 考虑为每个应用或者网络服务分配一个不同的角色。
- 使用用户组管理访问权限。
- 保护好 ROOT 的密码。
- 对于操作系统密码，强制使用强密码策略。
- 确保保护好操作系统的重要文件。

# 加密

- 加密和解密数据会影响性能，仅加密需要加密的数据。
- 在生产系统中实现任何加密解决方案之前都要做性能测试。
- HashData 数据仓库生产系统使用的服务器证书应由证书签名颁发机构（CA）签名，这样客户端可以验证服务器。如果所有客户端都是本地的，则可以使用本地 CA。
- 如果客户端与 HashData 数据仓库的连接会经过不安全的链路，则使用 SSL 加密。
- 加密和解密使用相同密钥的对称加密方式比非对称加密具有更好的性能，如果密钥可以安全共享，则建议使用对称加密方式。
- 使用 pgcrypto 包中的函数加密磁盘上的数据。数据的加密和解密都由数据库进程完成，为了避免传输明文数据，需要使用 SSL 加密客户端和数据库间的连接。
- 数据加载和导出时，使用 gpfdists 协议保护 ETL 数据安全。

# 高可用

- 使用 8 到 24 个磁盘的硬件 RAID 存储解决方案。
- 使用 RAID1、5 或 6，以使磁盘阵列可以容忍磁盘故障。
- 为磁盘阵列配备热备磁盘，以便在检测到磁盘故障时自动开始重建。
- 在重建时通过 RAID 卷镜像防止整个磁盘阵列故障和性能下降。
- 定期监控磁盘利用率，并在需要时增加额外的空间。
- 定期监控段数据库倾斜，以确保在所有段数据库上数据均匀分布，存储空间均匀消耗。
- 配置备用主服务器，当主服务器发生故障时由备用主服务器接管。
- 规划好当主服务器发生故障时如何切换客户端连接到新的主服务器实例，例如通过更新 DNS 中主服务器的地址。
- 建立监控系统，当主服务器发生故障时，可以通过系统监控应用或电子邮件发送通知。
- 分配主段数据库和其镜像到不同的主机上，以防止主机故障。
- 建立监控系统，当主段数据库发生故障时，可以通过系统监控应用或电子邮件发送通知。
- 使用 gprecoverseg 工具及时恢复故障段，并使系统返回最佳平衡状态。
- 在主服务器上配置并运行 gpsnmpd 以发送 SNMP 通知给网络监控器。
- 在 $Master_DATA_DIRECTORY/postgresql.conf 配置文件中设置邮件通知，以便检测到关键问题时, HashData 数据仓库系统可以通过电子邮件通知管理员。
- 考虑双集群配置，提供额外的冗余和查询处理能力。
- 除非 HashData 数据仓库数据仓库的数据很容易从数据源恢复，否则定期备份。
- 如果堆表相对较小，或者两次备份之间仅有少量 AO 或列存储分区有变化，则使用增量备份。
- 如果备份保存在集群的本地存储系统上，则备份结束后，将文件移到其他的安全存储系统上。

HashData 数据仓库是一个基于大规模并行处理（MPP）和无共享架构的数据仓库。这种分析型数据库的数据模式与高度规范化的事务性 OLTP 数据库显著不同。使用非规范化数据仓库模式，例如具有大事实表和小维度表的星型或者雪花模式，处理 MPP 分析型业务时，HashData 数据仓库表现优异。

# 数据类型

## 类型一致性

关联列使用相同的数据类型。如果不同表中的关联列数据类型不同，HashData 数据仓库必须动态的进行类型转换以进行比较。考虑到这一点，你可能需要增大数据类型的大小，以便关联操作更高效。

## 类型最小化

建议选择最高效的类型存储数据，这可以提高数据库的有效容量及查询执行性能。建议使用 TEXT 或者 VARCHAR 而不是 CHAR。不同的字符类型间没有明显的性能差别，但是 TEXT 或者 VARCHAR 可以降低空间使用量。建议使用满足需求的最小数值类型。如果 INT 或 SAMLLINT 够用，那么选择 BIGINT 会浪费空间。

# 内存管理

内存管理对 HashData 数据仓库集群性能有显著影响。默认设置可以满足大多数环境需求。不要修改默认设置，除非你理解系统的内存特性和使用情况。如果精心设计内存管理，大多数内存溢出问题可以避免。

下面是 HashData 数据仓库内存溢出的常见原因：

- 集群的系统内存不足
- 内存参数设置不当
- 段数据库 (Segment) 级别的数据倾斜
- 查询级别的计算倾斜

有时不仅可以通过增加系统内存解决问题，还可以通过正确的配置内存和设置恰当的资源队列管理负载，以避免很多内存溢出问题。

建议使用如下参数来配置操作系统和数据库的内存：

- vm.overcommit_memory

  > 这是 /etc/sysctl.conf 中设置的一个 Linux 内核参数。总是设置其值为 2。它控制操作系统使用什么方法确定分配给进程的内存总数。对于 HashData 数据仓库，唯一建议值是 2。

- vm.overcommit_ratio

  > 这是 /etc/sysctl.conf 中设置的一个 Linux 内核参数。它控制分配给应用程序进程的内存百分比。建议使用缺省值 50.

- 不要启用操作系统的大内存页

- gp_vmem_protect_limit

  > 使用 *gp_vmem_protect_limit* 设置段数据库(Segment)能为所有任务分配的最大内存。切勿设置此值超过系统物理内存。如果 *gp_vmem_protect_limit* 太大，可能造成系统内存不足，引起正常操作失败，进而造成段数据库故障。如果 gp_vmem_protect_limit 设置为较低的安全值，可以防止系统内存真正耗尽；打到内存上限的查询可能失败，但是避免了系统中断和 Segment 故障，这是所期望的行为。 *gp_vmem_protect_limit* 的计算公式为:

  ```
  ( SWAP + ( RAM * vm.overcommit_ratio )) * .9 /number_segments_per_server
  ```

- runaway_detector_activation_percent

  > HashData 数据仓库提供了失控查询终止（Runaway Query Termination）机制避免内存溢出。系统参数*runaway_detector_activation_percent*控制内存使用达到*gp_vmem_protect_limit*的多少百分比时会终止查询，默认值是 90%。如果某个Segment使用的内存超过了*gp_vmem_protect_limit*的90%（或者其他设置的值），HashData 数据仓库会根据内存使用情况终止那些消耗内存最多的 SQL 查询，直到低于期望的阈值。

- statement_mem

  > 使用*statement_mem*控制 Segment 数据库分配给单个查询的内存。如果需要更多内存完成操作，则会溢出到磁盘（溢出文件，spill files）。*statement_mem*的计算公式为:

  ```
  (vmprotect * .9) / max_expected_concurrent_queries
  ```

  > *statement_mem* 的默认值是128MB。例如使用这个默认值， 对于每个服务器安装 8 个 Segment 的集群来说，一条查询在每个服务器上需要 1GB 内存（8 Segments *128MB*）。对于需要更多内存才能执行的查询，可以设置回话级别的 statement_mem*。对于并发度比较低的集群，这个设置可以较好的管理查询内存使用量。并发度高的集群也可以使用资源队列对系统运行什么任务和怎么运行提供额外的控制。

- gp_workfile_limit_files_per_query

> *gp_workfile_limit_files_per_query* 限制一个查询可用的临时溢出文件数。当查询需要比分配给它的内存更多的内存时将创建溢出文件。当溢出文件超出限额时查询被终止。默认值是 0，表示溢出文件数目没有限制，可能会用光文件系统空间。

- gp_workfile_compress_algorithm

> 如果有大量溢出文件，则设置*gp_workfile_compress_algorithm*对溢出文件压缩。压缩溢出文件也有助于避免磁盘子系统 I/O 操作超载。

# 配置资源队列

HashData 数据仓库的资源队列提供了强大的机制来管理集群的负载。队列可以限制同时运行的查询的数量和内存使用量。当 HashData 数据仓库收到查询时，将其加入到对应的资源队列，队列确定是否接受该查询以及何时执行它。

- 不要使用默认的资源队列，为所有用户都分配资源队列。每个登录用户（角色）都关联到一个资源队列；用户提交的所有查询都由相关的资源队列处理。如果没有明确关联到某个队列，则使用默认的队列 *pg_default*。
- 避免使用 gpadmin 角色或其他超级用户角色运行查询 超级用户不受资源队列的限制，因为超级用户提交的查询始终运行，完全无视相关联的资源队列的限制。
- 使用资源队列参数 *ACTIVE_STATEMENTS* 限制某个队列的成员可以同时运行的查询的数量。
- 使用 *MEMORY_LIMIT* 参数控制队列中当前运行查询的可用内存总量。联合使用 *ACTIVE_STATEMENTS* 和 **MEMORY_LIMIT** 属性可以完全控制资源队列的活动。

队列工作机制如下：假设队列名字为 *sample_queue* ， *ACTIVE_STATEMENTS* 为 10， *MEMORY_LIMIT* 为 2000MB。这限制每个段数据库 (Segment) 的内存使用量约为 2GB。如果一个服务器配置 8 个 Segments，那么一个服务器上，*sample_queue* 需要 16GB 内存。如果 Segment 服务器有 64GB 内存，则该系统不能超过 4 个这种类型的队列，否则会出现内存溢出。（4队列 * 16GB/队列）。

> 注意 *STATEMENT_MEM* 参数可使得某个查询比队列里其他查询分配更多的内存，然而这也会降低队列中其他查询的可用内存。

- 资源队列优先级可用于控制工作负载以获得期望的效果。具有 *MAX* 优先级的队列会阻止其他较低优先级队列的运行，直到 *MAX* 队列处理完所有查询。
- 根据负载和一天中的时间段动态调整资源队列的优先级以满足业务需求。根据时间段和系统的使用情况，典型的环境会有动态调整队列优先级的操作流。可以通过脚本实现工作流并加入到 crontab 中。
- 使用 *gp_toolkit* 提供的视图查看资源队列使用情况，并了解队列如何工作。

本章介绍日常维护最佳实践以确保 HashData 数据仓库的高可用性和最佳性能。

# 监控

HashData 数据仓库带有一套系统监控工具。

*gp_toolkit* 模式包含可以查询系统表、日志和操作环境状态的视图，使用 SQL 命令可以访问这些视图。

*gp_stats_missing* 视图可以显示没有统计信息、需要运行 ANALYZE 的表。

关于 *gpstate* 和 *gpcheckperf* 的更多信息，请参考 《HashData 数据仓库工具指南》。关于 *gp_toolkit* 模式的更多信息，请参考《HashData 数据仓库参考指南》。

## gpstate

gpstate 工具显示了 HashData 数据仓库的系统状态，包括哪些段数据库 (Segments) 宕机，主服务器 (Master) 和 Segment 的配置信息（主机、数据目录等），系统使用的端口和 Segments 的镜像信息。

运行 gpstate -Q 列出 Master 系统表中标记为“宕机”的 Segments。

使用 gpstate -s 显示 HashData 数据仓库集群的详细状态信息。

## gpcheckperf

*gpcheckperf* 工具测试给定主机的基本硬件性能。其结果可以帮助识别硬件问题。它执行下面的检查：

- 磁盘 I/O 测试 - 使用操作系统的 dd 命令读写一个大文件，测试磁盘的 IO 性能。它以每秒多少兆包括读写速度。
- 内存带宽测试 - 使用 STREAM 基准程序测试可持续的内存带宽。
- 网络性能测试 - 使用 *gpnetbench* 网络基准程序（也可以用 *netperf* ）测试网络性能。测试有三种模式：并行成对测试（-r N ）,串行成对测试（-r n ），全矩阵测试（-r M ）。测试结果包括传输速率的最小值、最大值、平均数和中位数。

运行 *gpcheckperf* 时数据库必须停止。如果系统不停止，即使没有查询， *gpcheckperf* 的结果也可能不精确。

*gpcheckperf* 需要在待测试性能的主机间建立可信无密码SSH连接。它会调用 *gpssh* 和 *gpscp* ，所以这两个命令必须在系统路径 PATH 中。可以逐个指定待测试的主机（-h host1 -h host2 …）或者使用 -f hosts_file，其中 hosts_file 是包含待测试主机信息的文件。如果主机有多个子网，则为每个子网创建一个主机文件，以便可以测试每个子网。

默认情况下， *gpcheckperf* 运行磁盘 I/O 测试、内存测试和串行成对网络性能测试。对于磁盘测试，必须使用 -d 选项指定要测试的文件系统路径。下面的命令测试文件 subnet_1_hosts 中主机的磁盘和内存性能：

```
$ gpcheckperf -f subnet_1_hosts -d /data1 -d /data2 -r ds
```

-r 选项指定要运行的测试：磁盘 IO（d），内存带宽（s），网络并行成对测试（N），网络串行成对测试（n），网络全矩阵测试（M）。只能选择一种网络测试模式。更多信息，请参考《HashData 数据仓库参考指南》。

## 使用操作系统工具监控

下面的 Linux/Unix 工具可用于评估主机性能：

- iostat 监控段数据库 (Segments) 的磁盘活动
- top 显示操作系统进程的动态信息
- vmstate 显示内存使用情况的统计信息

可以使用 gpssh 在多个主机上运行这些命令。

## 最佳实践

- 实现《HashData 数据仓库管理员指南》中推荐的监控和维护任务。
- 安装前运行 *gpcheckperf* ，此后周期性运行 *gpcheckperf* ，并保存每次的结果，以用于比较系统随着时间推移的性能变

化。
- 使用一切可用的工具来更好地理解系统在不同负载下的行为。
- 检查任何异常事件并确定原因。
- 通过定期运行解释计划监控系统查询活动，以确保查询处于最佳运行状态。
- 检查查询计划，以确定是否按预期使用了索引和进行了分区裁剪。

### 额外信息

- 《HashData 数据仓库工具指南》中 *gpcheckperf*
- 《HashData 数据仓库管理员指南》中“监控和维护任务建议”
- Sustainable Memory Bandwidth in Current High Performance Computers. John D. McCalpin. Oct 12, 1995.
- 关于 netperf，可参考 www.netperf.org，需要在每个待测试的主机上安装 netperf。 参考 gpcheckperf 指南获的更多信息。

## 更新统计信息

良好查询性能的最重要前提是精确的表数据统计信息。使用 ANALYZE 语句更新统计信息后，优化器可以选取最优的查询计划。分析完表数据后，相关统计信息保存在系统表中。如果系统表存储的信息过时了，优化器可能生成低效的计划。

### 选择性统计

不带参数运行 ANALYZE 会更新数据库中所有表的统计信息。这可能非常耗时，不推荐这样做。当数据发生变化时，建议对变化的表进行 ANALYZE。

对大表执行 ANALYZE 可能较为耗时。如果对大表的所有列运行 ANALYZE 不可行，则使用 ANALYZE table(column, ...) 仅为某些字段生成统计信息。确保包括关联、WHERE 子句、SORT 子句、GROUP BY 子句、HAVING 子句中使用的字段。

对于分区表，可以只 ANALYZE 发生变化的分区，譬如只分析新加入的分区。注意可以 ANALYZE 分区表的父表或者最深子表。统计数据和用户数据一样，存储在最深子表中。中间层子表既不保存数据，也不保存统计信息，因而 ANALYZE 它们没有效果。可以从系统表 pg_partitions 中找到分区表的名字。

```
SELECT partitiontablename from pg_partitions WHERE tablename='parent_table;
```

### 提高统计数据质量

需要权衡生成统计信息所需时间和统计信息的质量（或者精度）。

为了在合理的时间内完成大表的分析，ANALYZE 对表内容随机取样，而不是分析每一行。调整配置参数 *default_statistics_target* 可以改变采样率。其取值范围为 1 到 1000；默认是 25。 默认 *default_statistics_target* 影响所有字段。较大的值会增加 ANALYZE 的时间，然而会提高优化器的估算质量。对于那些数据模式不规则的字段更是如此。可以在主服务器 (Master) 的会话中设置该值，但是需要重新加载。

配置参数 *gp_analyze_relative_error* 会影响为确定字段基数而收集的统计信息的采样率。例如 0.5 表示可以接受 50% 的误差。默认值是 0.25。使用 *gp_analyze_relative_error* 设置表基数估计的可接受的相对误差。如果统计数据不能产生较好的基数估计，则降低相对误差率（接受更少的错误）以采样更多的行。然而不建议该值低于 0.1，否则会大大延长 ANALYZE 的时间。

### 运行 **ANALYZE**

运行 ANALYZE 的时机包括：

- 加载数据后
- CREATE INDEX 操作后
- 影响大量数据的 INSERT、UPDATE 和 DELETE 操作后

ANALYZE 只需对表加读锁，所以可以与其他数据库操作并行执行，但是在数据加载和执行

INSERT/UPDATE/DELETE/CREATE INDEX 操作时不要运行 ANALYZE。

### 自动统计

配置参数 *gp_autostats_mode* 和 *gp_autostats_on_change_threshold* 确定何时触发自动分析操作。当自动统计数据收集被触发后，planner 会自动加入一个 ANALYZE 步骤。

*gp_autostats_mode* 默认设置是 on_no_stats ，如果表没有统计数据，则 CREATE TABLE AS SELECT, INSERT, COPY 操作会触发自动统计数据收集。

如果 *gp_autostats_mode* 是 on_change，则仅当更新的行数超过 *gp_autostats_on_change_threshold* 定义的阈值时才触发统计信息收集，其默认值是 2147483647。这种模式下，可以触发自动统计数据收集的操作有：CREATE TABLE AS SELECT, UPDATE, DELETE, INSERT 和 COPY。

设置 *gp_autostats_mode* 为 none 将禁用自动统计信息收集功能。

对分区表，如果从最顶层的父表插入数据不会触发统计信息收集。如果数据直接插入到叶子表（实际存储数据的表），则会触发统计信息收集。

## 管理数据库膨胀

HashData 数据仓库的堆表使用 PostgreSQL 的多版本并发控制（MVCC）的存储实现方式。删除和更新的行仅仅是逻辑删除，其实际数据仍然存储在表中，只是不可见。这些删除的行，也称为过期行，由空闲空间映射表（FSM, Free Space Map）记录。VACUUM 标记这些过期的行为空闲空间，并可以被后续插入操作重用。

如果某个表的 FSM 不足以容纳所有过期的行，VACUUM 命令无法回收溢出 FSM 的过期行空间。这些空间只能由 VACUUM FULL 回收，VACUUM FULL 会锁住整个表，逐行拷贝到文件头部，并截断（TRUNCATE）文件。对于大表，这一操作非常耗时。仅仅建议对小表执行这种操作。如果试图杀死 VACUUM FULL 进程，系统可能会被破坏。

> 注意：大型 UPDATE 和 DELETE 操作之后务必运行 VACUUM，避免运行 VACUUM FULL。

如果出现 FSM 溢出，需要回收空间，则建议使用 CREATE TABLE…AS SELECT 命令拷贝数据到新表，删除原来的表，然后重命名新表为原来的名字。

频繁更新的表会有少量过期行和空闲空间，空闲空间可被新加入的数据重用。但是当表变得非常大，而可见数据只占整体空间的一小部分，其余空间被过期行占用时，称之为膨胀（bloated）。膨胀表占用更多空间，需要更多 I/O，因而会降低查询效率。

膨胀会影响堆表、系统表和索引。

周期性运行 VACUUM 可以避免表增长的过大。如果表变得非常膨胀，必须使用 VACUUM FULL语句（或者其方法）精简该表。如果大表变得非常膨胀，则建议使用消除数据膨胀（后续章节）中介绍的方法消除膨胀的表。

> 警告：切勿运行 VACUUM FULL ; 或者对大表运行 VACUUM FULL。 运行 VACUUM 时，堆表的过期行被加入到共享的空闲空间映射表中。FSM 必须足够容纳过期行。如果不够大，则溢出 FSM 的空间不能被 VACUUM 回收。必须使用 VACUUM FULL 或者其他方法回收溢出空间。

定期性运行 VACUUM 可以避免 FSM 溢出。表越膨胀，FSM 就需要记录越多的行。对于非常大的具有很多对象的数据库，需要增大 FSM 以避免溢出。

配置参数 *max_fsm_pages* 设置在共享空闲空间映射表中被 FSM 跟踪的磁盘页最大数目。一页占用 6 个字节共享空间。默认值是 200,000。

配置参数 *max_fsm_relations* 设置在共享空间映射表中被 FSM 跟踪的表的最大数目。该值需要大于数据库中堆表、索引和系统表的总数。每个段数据库的每个表占用 60 个字节的共享内存。默认值是 1000。

更详细的信息，请参考《HashData 数据仓库参考指南》。

### 检测数据膨胀

ANALYZE 收集的统计信息可用于计算存储一个表所期望的磁盘页数。期望的页数和实际页数之间的差别是膨胀程度的一个

度量。gp_toolkit 模式的 gp_bloat_diag 视图通过比较期望页数和实际页数的比例识别膨胀的表。使用这个视图前，确保数据库的统计信息是最新的，然后运行下面的 SQL：

```
gpadmin=# SELECT * FROM gp_toolkit.gp_bloat_diag;
 bdirelid | bdinspname | bdirelname | bdirelpages | bdiexppages |
 bdidiag
----------+------------+------------+-------------+-------------
+------------
    21488 |     public |         t1 |          97 |           1 | significant amount
of bloat suspected
(1 row)
```

结果中包括中度或者严重膨胀的表。实际页数与期望页数之比位于 4 和 10 之间是中度膨胀，大于 10 为严重膨胀。

视图 gp_toolkit.gp_bloat_expected_pages 列出每个数据库对象实际使用的页数和期望使用的磁盘页数。

```
gpadmin=# SELECT * FROM gp_toolkit.gp_bloat_expected_pages LIMIT 5;
 btdrelid | btdrelpages | btdexppages
----------+-------------+-------------
    10789 |           1 |           1
    10794 |           1 |           1
    10799 |           1 |           1
     5004 |           1 |           1
     7175 |           1 |           1
(5 rows)
```

btdrelid 是表的对象 ID。btdrelpages 字段表示表适用的实际页数；bdtexppages 字段表示期望的页数。注意，这些数据基于统计信息，确保表发生变化后运行 ANALYZE。

## 消除数据膨胀表

VACUUM 命令将过期行加入到空闲空间映射表中以便以后重用。如果对频繁更新的表定期运行 VACUUM，过期行占用的空间可以被及时的回收并重用，避免表变得非常大。同样重要的是在 FSM 溢出前运行 VACUUM。对更新异常频繁的表，建议至少每天运行一次 VACUUM 以防止表变得膨胀。

> 警告：如果表严重膨胀，建议运行 VACUUM 前运行 ANALYZE。因为 ANALYZE 使用块级别抽样，如果一个表中无效/有效行的块的比例很高，则 ANALYZE 会设置系统表 pg_class 的 reltuples 字段为不精确的值或者 0，这会导致查询优化器不能生成最优查询。VACUUM 命令设置的数值更精确，如果在 ANALYZE 之后运行可以修正不精确的行数估计。

如果一个表膨胀严重，运行 VACUUM 命令是不够的。对于小表，可以通过 VACUUM FULL 回收溢出 FSM 的空间，降低表的大小。然而 VACUUM FULL 操作需要 ACCESS EXCLUSIVE 锁，可能需要非常久或者不可预测的时间才能完成。注意，消除大表膨胀的每种方法都是资源密集型操作，只有在极端情况下才使用。

第一种方法是创建大表拷贝，删掉原表，然后重命名拷贝。这种方法使用 CREATE TABLE AS SELECT 创建新表，例如：

```
gpadmin=# CREATE TABLE mytable_tmp AS SELECT * FROM mytable;
gpadmin=# DROP TABLE mytable;
gpadmin=# ALTER TABLE mytabe_tmp RENAME TO mytable;
```

第二种消除膨胀的方法是重分布。步骤为：

1. 记录表的分布键
2. 修改表的分布策略为随机分布

   ```
   ALTER TABLE mytable SET WITH (REORGANIZE=false)
   DISTRIBUTED randomly;
   ```

   此命令修改表的分布策略，但是不会移动数据。这个命令会瞬间完成。
3. 改回原来的分布策略

```
ALTER TABLE mytable SET WITH (REORGANIZE=true)
DISTRIBUTED BY (<original distribution columns>);
```

此命令会重新分布数据。因为和原来的分布策略相同，所以仅会在同一个段数据库上重写数据，并去掉过期行。

## 消除系统表膨胀

HashData 数据仓库系统表也是堆表，因而也会随着时间推移而变得膨胀。随着数据库对象的创建、修改和删除，系统表中会留下过期行。

使用 gpload 加载数据会造成膨胀，因为它会创建并删除外部表。（建议使用 gpfdist 加载数据）。

## 系统表膨胀

会拉长表扫描所需的时间，例如生成解释计划时。系统表需要频繁扫描，如果它们变得膨胀，那么系统整体性能会下降。

建议每晚 (至少每周) 对系统表运行 VACUUM。

下面的脚本对系统表运行 VACUUM ANALYZE。

```
#!/bin/bash
DBNAME="<database_name>"
VCOMMAND="VACUUM ANALYZE"psql -tc "select '$VCOMMAND' || ' pg_catalog.' || relname || ';'FROM pg_class a, pg_namespace b \
where a.relnamespace=b.oid andb.nspname='pg_catalog' and a.relkind='r'"
$DBNAME | psql -a $DBNAME
```

如果系统表变得异常膨胀，则必须执行一次集中地系统表维护操作。对系统表不能使用 CREATE TABLE AS SELECT 和上面介绍的重分布方法消除膨胀，而只能在计划的停机期间，运行 VACUUM FULL。在这个期间，停止系统上的所有系统表操作，VACUUM FULL 会对系统表使用排它锁。定期运行 VACUUM 可以防止这一代价高昂的操作。

## 消除索引表膨胀

VACUUM 命令仅恢复数据表的空间，要恢复索引表的空间，使用 REINDEX 重建索引。 使用 REINDEX table_name 重建一个表的所有索引；使用 REINDEX index_name 重建某个索引。

## 消除 **AO** 表膨胀

AO 表的处理方式和堆表完全不同。尽管 AO 表可以更新和删除，然而 AO 表不是为此而优化的，建议对 AO 表避免使用更新和删除。如果 AO 表用于一次加载/多次读取的业务，那么 AO 表的 VACUUM 操作几乎会立即返回。

如果确实需要对 AO 表执行 UPDATE 或者 DELETE，则过期行会记录在辅助的位图表中，而不是像堆表那样使用空闲空间映射表。使用 VACUUM 回收空间。对含有过期行的 AO 表运行 VACUUM 会通过重写来精简整张表。如果过期行的百分比低于配置参数 gp_appendonly_compaction_threshold , 则不会执行任何操作，默认值是 10（10%）。每个段数据库 (Segment) 上都会检查该值，所以有可能某些 Segment 上执行空间回收操作，而另一些 Segment 不执行任何操作。可以通过设置 gp_appendonly_compaction 参数为 no 禁止 AO 表空间回收。

# 监控日志文件

了解系统日志文件的位置和内容，并定期的监控这些文件。

下标列出了 HashData 各种日志文件的位置。文件路径中，date 是格式为 YYYYMMDD 的日期， instance 是当前实例的名字，n 是 Segment 号。

| 路径 | 描述 |
| --- | --- |
| /var/gpadmin/gpadminlogs/* | 很多不同类型的日志文件 |
| /var/gpadmin/gpadminlogs/gpstart_date.log | 启动日志 |
| /var/gpadmin/gpadminlogs/gpstop_date.log | 停止服务器日志 |
| /var/gpadmin/gpadminlogs/gpsegstart.py_idb*gpadmin_date.log | Segment 启动日志 |
| /var/gpadmin/gpadminlogs/gpsegstop.py_idb*gpadmin_date.log | Segment 停止日志 |
| /var/gpdb/instance/dataMaster/gpseg-1/pg_log/startup.log | 实例启动日志 |
| /var/gpdb/instance/dataMaster/gpseg-1/gpperfmon/logs/gpmon.*.log | Gpperfmon 日志 |
| /var/gpdb/instance/datamirror/gpseg{n}/pg_log/.*csv | 镜像 Segment 日志 |
| /var/gpdb/instance/dataprimary/gpseg{n}/pg_log/.*csv | 主 Segment 日志 |
| /var/log/messages | Linux 系统日志 |

首先使用 gplogfilter -t(–trouble) 从 Master 日志中搜索 ERROR:, FATAL:, PANIC: 开头的消息。WARNING 开头的消息也可能提供有用的信息。

若需搜索段数据库（Segment）日志，从主服务器（Master）上使用 gpssh 连接到 Segment 上，然后使用 gplogfilter 工具搜索消息。可以通过 Segment_id 识别是哪个 Segment 的日志。

配置参数 *log_rotation_age* 控制什么时候会创建新的日志文件。默认情况下，每天创建一个新的日志文件。

# 数据加载

HashData 数据仓库支持多种数据加载方式，每种都有其适用的场景。

## INSERT 语句直接插入字段数据

带有字段值的 INSERT 语句插入单行数据。数据通过主服务器（Master）分发到某个段数据库（Segment）。这是最慢的方法，不适合加载大量数据。

## COPY 语句

PostgreSQL 的 COPY 语句从外部文件拷贝数据到数据库表中。它可以插入多行数据，比 INSERT 语句效率高，但是所有行数据仍然会通过 Master。所有数据由一个命令完成拷贝，不能并行化。

COPY 命令的数据输入可以是文件或者标准输入。例如：

```
COPY table FROM '/data/mydata.csv' WITH CSV HEADER;
```

COPY 适合加载少量数据，例如数千行的维度表数据或者一次性数据加载。

使用脚本加载少于 10k 行数据时可以使用 COPY。

COPY 是单个命令，所以不需要禁用自动提交（autocommit）。

可以并发运行多个 COPY 命令以提高效率。

## 外部表

HashData 数据仓库外部表可以访问数据库之外的数据。可以使用 SELECT 访问外部数据，常用于抽取、加载、转换（ELT）模式中。ELT 是 ETL 模式的一个变种，可以充分利用 HashData 数据仓库的快速并行数据加载能力。

使用 ETL 时，数据从数据源抽取，在数据库之外使用诸如 Informatica 或者 Datastage 的外部工具进行转换，然后加载到数据库中。

使用 ELT 时，HashData 数据仓库外部表提供对外部数据的直接访问，包括文件（例如文本文件、CSV 或者 XML 文件）、Web 服务器、Hadoop 文件系统、可执行操作系统程序或者下节介绍的 gpfdist 文件服务器。外部表支持选择、排序和关联等 SQL 操作，所以数据可以同时进行加载和转换；或者加载到某个表后在执行转换操作，并最终插入到目标表中。

外部表定义使用 CREATE EXTERNAL TABLE 语句，该语句的 LOCATION 子句定义了数据源，FORMAT 子句定义了数据的格式以便系统可以解析该数据。外部文件数据使用 file://协议，且必须位于 Segment 主机上可以被 HashData 数据仓库超级用户访问的位置。可以有多个数据文件，LOCATION 子句中文件的个数就是并发读取外部数据的 Segments 的个数。

## 外部表和 gpfdist

加载大量事实表的最快方法是使用外部表与 gpfdist。gpfdist 是一个基于 HTTP 协议的、可以为HashData 数据仓库的段数据库（Segments）并行提供数据的文件服务器。单个 gpfdist 实例速率可达 200MB/秒，多个 gpfdist 进程可以并行运行，每个提供部分加载数据。当使用

```
INSERT INTO <table> SELECT *FROM <external_table>
```

语句加载数据时，这个 INSERT 语句由 Master 解析，并分发到各个主 Segments 上。每个 Segment 连接到 gpfdist 服务器，并行获取数据、解析数据、验证数据、计算分布键的哈希值，并基于哈希值分布到存储该数据的目标 Segment 上。默认情况下，每个 gpfdist 实例可以接受至多来自64个Segment的连接。通过使用多个 gpfdist 服务器和大量 Segments，可以实现非常高的加载速度。

使用 gpfdist 时，最多可以有 *gp_external_max_segs* 个 Segment 并行访问外部数据。优化 gpfdist 性能时，随着 Segment

个数的增加，最大化并行度。均匀分布数据到尽可能的 ETL 节点。切分大文件为多个相同的部分，并分布到尽可能多的文件系统下。

每个文件系统运行两个 gpfdist 实例。加载数据时，在 Segment上 gpfdist 往往是 CPU 密集型任务。但是譬如有八个机架的 Segment 节点，那么 Segments 上会有大量 CPU 资源，可以驱动更多的 gpfdist 进程。在尽可能多的网卡上运行 gpfdist。清楚了解绑定的 NICs 个数，并启动足够多的 gpfdist 进程充分使用网络资源。

均匀地为加载任务分配资源很重要。加载速度取决于最慢的节点，加载文件布局的倾斜会造成该资源成为瓶颈。

配置参数 *gp_external_max_segs* 控制连接单个 gpfdist 的 Segments 的个数。默认值是 64。确保 gp_external_max_segs 和 gpfdist 进程个数是偶数因子，也就是说 *gp_external_max_segs* 值是 gpfdist 进程个数的倍数。例如如果有 12 个 Segments、4 个进程，优化器轮询 Segment 连接如下：

```
Segment 1  - gpfdist 1
Segment 2  - gpfdist 2
Segment 3  - gpfdist 3
Segment 4  - gpfdist 4
Segment 5  - gpfdist 1
Segment 6  - gpfdist 2
Segment 7  - gpfdist 3
Segment 8  - gpfdist 4
Segment 9  - gpfdist 1
Segment 10 - gpfdist 2
Segment 11 - gpfdist 3
Segment 12 - gpfdist 4
```

加载数据到现有表之前删除索引，加载完成后重建索引。在已有数据上创建索引比边加载边更新索引快。

数据加载后对表运行 ANALYZE。加载过程中通过设置 *gp_autostats_mode* 为 NONE 禁用统计信息自动收集。

对有大量分区的列表频繁执行少量数据加载会严重影响系统性能，因为每次访问的物理文件很多。

## gpload

gpload 是一个数据加载工具，是 HashData 数据仓库外部表并行数据加载特性的一个接口。

小心使用 gpload，因为它会创建和删除外部表而造成系统表膨胀。建议使用性能最好的 gpfdist。

gpload 使用 YAML 格式的控制文件来定义数据加载规范，执行下面操作：

- 启动 gpfdist 进程；
- 基于定义的外部数据创建临时外部表；
- 执行 INSERT、UPDATE 或者 MERGE 操作，加载数据到数据库中的目标表；
- 删除临时外部表；
- 清理 gpfdist 进程。

加载操作处于单个事务之中。

## 最佳实践

- 加载数据到现有表前删除索引，加载完成后重建索引。创建新索引比边加载边更新索引快。
- 加载过程中，设置 *gp_autostats_mode* 为 NONE，禁用统计信息自动收集。
- 外部表不适合频繁访问或者 ad-hoc 访问。
- 外部表没有统计数据。可以使用下面的语句为外部表设置大概估计的行数和磁盘页数：

  ```
  UPDATE pg_class SET reltuples=400000, relpages=400   WHERE     relname='myexttable';
  ```

- 使用 gpfdist 时，为 ETL 服务器上的每个 NIC 运行一个 gpfdist 实例以最大化利用网络带宽。均匀分布数据到多个 gpfdist 实例上。
- 使用 gpload 时，运行尽可能多的 gpload。充分利用 CPU、内存和网络资源以提高从 ETL 服务器加载数据到 HashData 的速度。

- 使用 LOG ERRORS INTO 子句保存错误行。错误行 – 例如，缺少字段值或者多了额外值，或者不正确的数据类型 – 保存到错误表中，加载继续执行。Segment REJECT LIMIT 子句设置命令中止前允许的错误行的数目或者百分比。
- 如果加载报错，对目标表运行 VACUUM 释放空间。
- 数据加载完成后，对堆表包括系统表运行 VACUUM，对所有表运行 ANALYZE。对 AO 表不必运行 VACUUM。如果表是分区表，则可以仅对数据加载影响的分区执行 VACUUM 和 ANALYZE。这样可以清除失败的加载占用的空间、删除或者更新的行占用的空间，并更新统计数据。
- 加载大量数据后重新检查数据倾斜。使用下面查询检查倾斜状况：

```
SELECT gp_segment_id, count(*)
FROM schema.table GROUP BY gp_Segment_id ORDER BY 2;
```

- 默认情况，gpfdist 可以处理的最大行为 32K。如果行大于 32K，则需要使用 gpfdist 的 -m 选项增大最大行长度。如果使用 gpload，配置控制文件中的 MAX_LINE_LENGTH 参数。

## 额外信息

关于 gpfdist 和 gpload 的更多信息，请参考《HashData 数据仓库参考指南》。

# 查询计划

HashData 数据仓库采用基于成本的优化器来评估执行查询的不同策略，并选择成本最低的方法。和其他关系数据库系统的优化器相似，在计算不同执行计划的成本时，HashData 数据仓库的优化器会考虑诸如关联表的行数、是否有索引、字段数据的基数等因素。还会考虑到数据的位置，尽可能在段数据库 (Segment) 上完成任务，降低在不同 Segments 间传输的数据量。

在一个查询运行速度比预期慢时，可以查看优选器生成的查询计划以及执行每一步的代价。这有助于确定哪一步最耗资源，进而可以修改查询或者模式，以便生成更好的计划。查看查询计划使用 EXPLAIN 语句。

优化器基于每个表的统计信息生成计划，所以准确的统计信息对生成最有计划至关重要。关于更新统计信息，请参看本文档的 ”使用 ANALYZE 更新统计信息” 一节。

## 生成查询计划

EXPLAIN 和 EXPLAIN ANALYZE 语句有助于改进查询性能。EXPLAIN 显示一个查询的执行计划以及估算的代价，但是不执行该查询。EXPLAIN ANALYZE 除了显示执行计划外还会执行查询。EXPLAIN ANALYZE 会丢弃 SELECT 语句的输出，其他操作会被执行（例如 INSERT，UPDATE 和 DELETE）。若需对 DML 语句使用 EXPLAIN ANALYZE 而不影响数据，可置 EXPLAIN ANALYZE 于事务中（BEGIN; EXPLAIN ANALYZE …; ROLLBACK;）。

EXPLAIN ANALYZE 除了执行查询、显示查询计划外，还显示如下信息：

- 执行查询的总时间（以毫秒为单位）；
- 查询节点需要的工作进程个数；
- 每个操作中处理最多行的 Segment 处理的最大行数以及 Segment 的序号；
- 内存使用量；
- 从处理最多行的 Segment 上获得第一行花费的时间（单位是毫秒）及从该Segment获得所有行花费的时间。

## 理解查询计划

解释计划详细描述了 HashData 执行查询的步骤。查询计划是颗节点树，从下向上读数据，每个节点将它的执行结果数据传递给其上的节点。每个节点表示查询计划的一个步骤，每个节点都有一行信息描述了该步骤执行的操作 – 例如扫描、关联、聚合或者排序操作等，此外还有显示执行该操作的具体方法。例如，扫描操作可能是顺序扫描或者索引扫描，关联操作可能是哈希关联或者嵌套循环关联。

下面是一个简单查询的解释计划，这个查询的结果应为每个 Segment 上 contributions 表的行数。

```
gpacmin=# EXPLAIN SELECT gp_Segment_id, count(*)
                  FROM contributions
                  GROUP BY gp_Segment_id;
                                                                        QUERY PLAN
-----------------------------------------------------------------------------------
Gather Motion 2:1 (slice2; Segments: 2) (cost=0.00..4.44 rows=4 width=16)
    -> HashAggregate (cost=0.00..3.38 rows=4 width=16)
        Group By: contributions.gp_Segment_id
        -> Redistribute Motion 2:2 (slice1; Segments: 2) (cost=0.00..2.12 rows=4 width=8)
            Hash Key: contributions.gp_Segment_id
            -> Sequence (cost=0.00..1.09 rows=4 width=8)
                -> Result (cost=10.00..100.00 rows=50 width=4)
                    -> Function Scan on gp_partition_expansion (cost=10.00..100.00 rows=50 width=4)
                -> Dynamic Table Scan on contributions (partIndex: 0) (cost=0.00..0.03 rows=4 width=8)
Settings: optimizer=on(10 rows)
```

此计划有 7 个节点 - Dynamic Table Scan, Function Scan, Result, Sequence, Redistribute Motion, HashAggregate 及最后的 Gather Motion。每个节点含有三个估计代价：代价（顺序读页数），行数和行的长度。

代价本身包含 2 个估计值。一部分是启动代价估计，即得到第一行的代价，第二部分是总代价估计，即处理完所有行的代价。代价为 1.0 意味着顺序读一个磁盘页。

行数是查询节点输出行数的估计。考虑到 WHERE 子句条件的选择性，可能比实际处理或者扫描的行数小。总代价假设处理

所有行，而有些时候可能不需要（例如 LIMIT 子句）。

长度是查询节点输出的所有字段的总长度，单位是字节。

节点的估计代价包括所有子节点的代价，所以计划的最顶层节点（通常是 Gather Motion）包含执行计划的总代价。优化器试图降低的也正是这个数字。

扫描操作符扫描数据库表以找到期望的数据行。不同的存储类型有不同的扫描操作：

- 堆表顺序扫描 - 扫描表的所有行
- AO 扫描 - 扫描面向行的AO表
- AO 列扫描 - 扫描面向列的AO表
- 索引扫描 - 遍历B树索引，从表中获取期望的行
- Bitmap AO 行扫描 - 从索引中获得 AO 表行的指针，并根据磁盘位置排序
- 动态扫描 - 使用分区选择函数选择待扫描的分区。Function Scan 节点包含分区选择函数的名字：

> 1. gp_partition_expansion - 选择表的所有分区，不会裁剪分区
> 2. gp_partition_selection - 根据等价表达式选择分区
> 3. gp_partition_inversion - 根据范围表达式选择分区

Function Scan 节点将动态选择的分区传递个 Result 节点，进而传递给 Sequence 节点。

关联操作符：

- 哈希关联 - 用关联字段做哈希键，对小表建立哈希表。然后扫描大表，计算大表每行关联字段的哈希键，从哈希表中查找哈希值相同的行。通常哈希关联是速度最快的关联方式。查询计划中的 Hash Cond 是关联字段。
- 嵌套循环 - 遍历大数据集，对其每一行，扫描小数据集，并找到匹配的行。嵌套循环关联需要广播一个表的数据，以便另一个表的数据可以和该表的每一行进行比较。对于小表或者使用索引的表性能良好。也用于笛卡尔关联和范围关联。对大表使用嵌套关联性能不佳。如果查询节点使用嵌套循环关联操作符，则检查 SQL ，确保结果是期望的。设置配置参数 *enable_nestloop* 为 OFF （默认）以优先使用哈希关联。
- 合并关联 - 对两个数据集排序，然后合并。合并关联对已经排序的数据性能很好，但是较少使用。如要使用合并关联，设置 *enable_mergejoin* 为 ON。

某些查询计划节点是移动操作符 (Motion)。移动操作符负责在段数据库（Segment）间传输行数据。这种节点会标识执行移动操作的方法：

> - 广播：每个 Segment 发送其表数据给所有其他 Segments，这样每个 Segment 都有该表的完整本地拷贝。广播移动操作比充分发移动操作符效率低，所以优化器仅仅对小表使用广播操作，广播大表性能欠佳。
> - 重分发：每个 Segment 对数据关联字段计算哈希值，并发送到对应的 Segments。
> - 收集：所有 Segments 的结果数据发送给单个节点，这通常是大多数查询计划的最后一步。

其他操作符有：

- 物化（Material） - 优化器物化子查询，避免多次使用时重复计算。
- InitPlan - 仅仅需要执行一次的子查询，且对外围查询没有依赖。
- 排序（Sort） - 对数据集排序，多用于为聚合或者合并关联准备数据。
- 分组（Group By）- 根据一个或者多个字段对数据集分组。
- 分组/哈希聚合（Aggregation）- 使用哈希对数据集进行聚合计算。
- 追加（Append）- 合并数据集，例如当扫描分区表的多个分区时。
- 过滤器 (Filter) - 根据 WHERE 子句条件选择匹配的数据行。
- 限制（Limit）- 限制返回的行数。

## 优化查询

本节介绍某些情况下可以改善系统性能的数据库特性和编程实践。

分析查询计划时首先需找估计代价很高的操作。对比估算的行数及代价与操作实际需要处理的行数，以判断是否合理。

如果有分区表，判断分区裁剪是否有效。分区裁剪需要查询条件（ WHERE 子句）必须和分区条件一样。此外，WHERE 子句不能含有显式值，也不能包含子查询。

检查查询计划树的执行顺序，检查行数估计值。希望先处理小表，或者哈希关联的结果集，然后处理大表。理想情况是最后处理最大的表，以降低沿着查询树向上传递直到最顶层节点的行数。如果执行计划顺序不是最优的，检查数据库统计信息是否最新的。运行 ANALYZE 通常会解决这个问题，并生成最优计划。

注意计算倾斜。当执行诸如哈希聚合和哈希关联等操作符时，若不同 Segments 执行代价分布不均，则发生计算倾斜。由于某些 Segment 比其他 Segment 使用更多的 CPU 和内存，性能下降明显。原因可能是关联、排序、聚合字段基数低或者分布不均匀。通过 EXPLAIN ANALYZE 输出可以检测计算倾斜。每个节点都含有 Segment 处理的最多行数和所有 Segment 的平均行数。如果最大行数比均值大很多，那么至少有一个 Segment 需要处理更多的工作，因而有计算倾斜的可能性。

注意执行排序或者聚合操作的查询节点。聚合操作隐含排序操作。如果排序和聚合操作需要处理大量数据，则存在优化查询性能的机会。当需要对大量数据行进行排序和聚合时，优先使用 HashAggregate 操作符。通常情况下，不良 SQL 会造成优化器选择使用排序操作符。如果重写查询，大多数的排序操作可以被 HashAggregate 替代。设置 *enable_groupagg* 参数以优先使用 HashAggregate 而非排序聚合操作。

若查询计划显示对大量数据集使用了广播移动操作符，需要尝试避免使用广播操作符。一种方法是使用 gp_segments_for_planner 配置参数增加移动数据的估计代价。该变量告诉优化器在计算移动代价时使用多少个 segments。默认值是 0，意味着使用实际 Segment 个数。增大这个数字，移动的代价会跟着增大，优化器会优先使用重分发移动操作符。例如设置 gp_segments_for_planner=100000 告诉优化器有 100000 个 Segments。相反为了优先使用广播移动操作符，为该值设置一个小数字，例如 2。

## 分组扩展

HashData 数据仓库的 GROUP BY 扩展可以执行某些常用的计算，且比应用程序或者存储过程效率高。

- GROUP BY ROLLUP(col1, col2, col3)
- GROUP BY CUBE(col1, col2, col3)
- GROUP BY GROUPING SETS((col1, col2), (col1, col3))

ROLLUP 对分组字段（或者表达式）从最详细级别到最顶级别计算聚合计数。ROLLUP 的参数是一个有序分组字段列表，它计算从右向左各个级别的聚合。例如 ROLLUP(c1, c2, c3) 会为下列分组条件计算聚集：

- (c1, c2, c3)
- (c1, c2)
- (c1)
- ()

CUBE 为分组字段的所有组合计算聚合。例如 CUBE(c1, c2, c3) 会计算一下聚合：

- (c1, c2, c3)
- (c1, c2)
- (c2, c3)
- (c1, c3)
- (c1)
- (c2)
- (c3)
- ()

GROUPING SETS 指定对那些字段计算聚合，它可以比 ROLLUP 和 CUBE 更精确地控制分区条件。

更多细节请参见《HashData 数据仓库参考指南》。

## 窗口函数

窗口函数可以实现在结果集的分组子集上的聚合或者排名函数，例如 sum(population) over (partition by city)。窗口函数功能强大，性能优异。因为它在数据库内部进行计算，避免了数据传输。

- 窗口函数 row_number() 计算一行在分组子集中的行号，例如 row_number() over (order by id)。

- 如果查询计划显示某个表被扫描多次，那么通过窗口函数可能可以降低扫描次数。
- 窗口函数通常可以避免使用自关联。

# 数据库导入

HashData支持各种工具从其它数据库导入数据。

- IBM DataStage：商业解决方案。 https://www.informatica.com/
- Infomatica PowerCenter: 商业解决方案。 https://www.ibm.com/us-en/marketplace/datastage
- Pentaho Data Integration（Kettle）：开源解决方案。 https://github.com/pentaho/pentaho-kettle
- DataX (HashData Release)：开源解决方案。 https://github.com/HashDataInc/DataX

## DataX(HashData Release)

- DataX(HashData Release) 和Alibaba DataX 有何不同？

  DataX 是阿里巴巴集团内被广泛使用的离线数据同步工具/平台，实现包括
  MySQL、Oracle、HDFS、Hive、OceanBase、HBase、OTS、ODPS 等各种异构数据源之间高效的数据同步功能。DataX(HashData Release)在开源的 DataX 基础上，针对 GPDB Writer 专门做了性能优化，可以达到 10X 写入速度。

- 导入 HashData 该选择哪种 Writer？

  GPDB Writer，详细参考：https://github.com/HashDataInc/DataX/blob/master/gpdbwriter/doc/gpdbwriter.md

# Kafka导入

通过 Kafka Comsumer 组件读取数据，然后以 Copy 模式批量写入 HashData.

关于 Kafka Comsumer 如何读取数据，参考 Kafka 官方文档：http://kafka.apache.org/10/documentation/

## Java

Copy 模式写入示例如下

```
import java.io.FileReader;
import java.sql.Connection;
import java.sql.DriverManager;

import org.postgresql.copy.CopyManager;
import org.postgresql.core.BaseConnection;

public class PgSqlJdbcCopyStreamsExample {

  public static void main(String[] args) throws Exception {

      if(args.length!=4) {
          System.out.println("Please specify database URL, user, password and file on the command line.");
          System.out.println("Like this: jdbc:postgresql://localhost:5432/test test password file");
      } else {

          System.err.println("Loading driver");
          Class.forName("org.postgresql.Driver");

          System.err.println("Connecting to " + args[0]);
          Connection con = DriverManager.getConnection(args[0],args[1],args[2]);

          System.err.println("Copying text data rows from stdin");

          CopyManager copyManager = new CopyManager((BaseConnection) con);

          FileReader fileReader = new FileReader(args[3]);
          copyManager.copyIn("COPY t FROM STDIN", fileReader );

          System.err.println("Done.");
        }
   }
}
```

关于 CopyManager 详细参考：https://jdbc.postgresql.org/documentation/publicapi/org/postgresql/copy/CopyManager.html

## GoLang

Copy 模式写入示例如下:

```
txn, err := db.Begin()
if err != nil {
        log.Fatal(err)
}

stmt, err := txn.Prepare(pq.CopyIn("users", "name", "age"))
if err != nil {
        log.Fatal(err)
}

for _, user := range users {
      _, err = stmt.Exec(user.Name, int64(user.Age))
        if err != nil {
              log.Fatal(err)
        }
}

_, err = stmt.Exec()
if err != nil {
        log.Fatal(err)
}

err = stmt.Close()
if err != nil {
        log.Fatal(err)
}

err = txn.Commit()
if err != nil {
      log.Fatal(err)
}
```

关于 pq 包的使用，详细参考：https://godoc.org/github.com/lib/pq#hdr-Bulk_imports