

# Greenplum 企业应用实战

何勇 陈晓峰 著

---

Enterprise Application with Greenplum

---

- 阿里巴巴资深技术工程师撰写，完全展现阿里巴巴的Greenplum企业实战经验
- 系统介绍Greenplum的功能特性、使用方法、高级应用；详细讲解Greenplum的系统架构、运维管理、性能优化和各种技巧；包含大量企业级应用案例和实操指导

资源由 [www.eimhe.com](http://www.eimhe.com) 美河学习在线收集分享



机械工业出版社  
China Machine Press

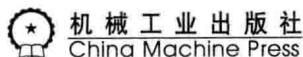
# Greenplum 企业应用实战

---

Enterprise Application with Greenplum

---

何勇 陈晓峰 著



资源由 [www.eimhe.com](http://www.eimhe.com) 美河学习在线收集分享

## 图书在版编目 (CIP) 数据

Greenplum 企业应用实战 / 何勇, 陈晓峰著. —北京: 机械工业出版社, 2014.10  
(数据库技术丛书)

ISBN 978-7-111-48100-3

I. G… II. ①何… ②陈… III. 关系数据库系统 IV. TP311.138

中国版本图书馆 CIP 数据核字 (2014) 第 226297 号

# Greenplum 企业应用实战

---

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 姜 影

责任校对: 董纪丽

印 刷: 北京市荣盛彩色印刷有限公司

版 次: 2014 年 10 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 21.75

书 号: ISBN 978-7-111-48100-3

定 价: 69.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

## 为什么写作本书

阿里巴巴是国内最早使用 Greenplum 作为数据仓库计算中心的公司。从 2009 年到 2012 年 Greenplum 都是阿里巴巴 B2B 最重要的数据计算中心，它替换掉了之前的 Oracle RAC，有非常多的优点。

- Greenplum 的性能在数据量为 TB 级别时表现非常优秀，单机性能相比 Hadoop 要快好几倍。
- Greenplum 是基于 PostgreSQL 的一个完善的数据库，在功能和语法上都要比 Hadoop 上的 SQL 引擎 Hive 好用很多，对于普通用户来说更加容易上手。
- Greenplum 有着完善的工具，相比 Hive，整个体系都比较完善，不需要像 Hive 一样花太多的时间和精力进行改造，非常适合作为一些大型的数据仓库解决方案。
- Greenplum 能够方便地与 Hadoop 进行结合，可直接把数据写在 Hadoop 上，还可以直接在数据库上写 MapReduce 任务，并且配置简单。

从 2010 年毕业加入阿里巴巴 B2B 的数据仓库起，我就开始接触 Greenplum 数据库，并有幸维护了一年多的 Greenplum 数据库，积累了很多数据库的相关知识。Greenplum 在国内的应用相对比较少，尤其是网上资料相当匮乏。在使用 Greenplum 的过程中，阿里巴巴遇到了很多困难，也积累了很多宝贵经验。

由于学习资料的匮乏，我和何勇有了将阿里巴巴使用 Greenplum 的一些经验技巧汇聚成书的想法，这样既总结和沉淀了自身知识，同时也可以给国内使用 Greenplum 的同行们提供一点帮助。

## 本书组织结构

本书从实战角度出发，结合了大量实践案例（附有详细的代码），由浅入深介绍了

资源由 [www.eimhe.com](http://www.eimhe.com) 美河学习在线收集分享

Greenplum。本书由 15 章组成，主要分为 3 篇。

### 上篇（第 1 ~ 3 章）——基础篇

基础篇目的是帮助读者快速了解 Greenplum，从实战的角度介绍一些入门必备的基础知识。从如何安装部署 Greenplum 开始，一步步引导读者搭建自己的 Greenplum 数据库，然后介绍基本的语法及相关操作。本篇最后通过分析两个具体的数据仓库 ETL 的案例，加强读者对 Greenplum 功能特性的了解，提高实践能力。

### 中篇（第 4 ~ 7 章）——进阶篇

进阶篇重点介绍 Greenplum 的数据字典、执行计划、架构以及一些高级特性。

数据字典是 Greenplum 对元数据信息的组织方式，执行计划是数据库执行 SQL 的灵魂，高级特性则是 Greenplum 的优势所在。本篇结合了大量案例对以上内容进行了深入分析。通过对这些内容的学习，可以深入理解数据库的工作原理，是进阶的必经之路，可以让读者游刃有余地应对各种 Greenplum 的日常操作。

### 下篇（第 8 ~ 15 章）——管理篇

管理篇主要介绍一些与数据库管理员相关的知识，包括线上部署、性能优化、权限控制、监控、容灾 / 扩容方案、常用脚本以及常见问题等。这些更偏向于后台管理，是 DBA 必修的课程。

## 本书面向的读者

阅读本书需要读者对关系型数据库有基本了解，最好也了解一些 Linux 的基本操作。本书面向的读者主要有：

- Greenplum 数据库管理员
- 使用 Greenplum 的 ETL 开发工程师
- 数据库爱好者、分布式系统爱好者
- 数据分析师、商业智能分析师

## 如何阅读本书

对于刚接触 Greenplum 的读者来说，建议从第 1 章读起，书中有很多结合实践的例子，帮助读者加深对 Greenplum 的理解。

如果对 Greenplum 已经比较熟悉，可以从第 4 章开始读起，应重点关注第 5 章讲解分布

式执行计划。了解执行计划是熟悉数据库工作原理的一种有效方法，分布式执行计划与普通执行计划还是有较大差别的。

如果只是想通过本书了解 Greenplum 数据库与普通关系型数据库的差异，可以先阅读第 7 章，了解 Greenplum 架构，然后再选取自己喜欢的章节进行阅读。

本书偏向于实战，理论方面的知识相对较少，感兴趣的读者可以结合其他介绍数据库的资料进行阅读，如“PostgreSQL8.2 官方文档”、《数据库系统概念》、《数据库系统实现》。

虽然 Greenplum 并不是开源的，但是 Greenplum 是基于 PostgreSQL 开发的，而 PostgreSQL 是开源的。还有，Greenplum 自带了很多维护脚本，大部分采用 Shell 和 Python 编写。因此，对于想深入理解 Greenplum 的读者来说，建议结合 PostgreSQL 源码及 Greenplum 安装目录下的一些脚本代码进行阅读。通过阅读这些源码，可以加深对 Greenplum 底层的了解。

本书大部分例子都是基于 Greenplum 4.1 版本进行编写的，在写作的过程，Greenplum 已经升级到 Greenplum 4.3，因此在书稿快完成时，笔者对全书进行了审核。由于 Greenplum 4.3 与 Greenplum 4.1 大部分的内容都是相同的，因此，笔者在书稿中必要的地方采用修改或备注的方式加入了一些 Greenplum 4.3 的新特性，尽量保证了本书的时效性。

## 勘误和支持

本书第 1、3、7、8、14 章由何勇撰写，其余章节由陈晓峰撰写，由于是第一次写作，在写作上肯定会有许多的不足，加上笔者能力有限，难免会有遗漏或错误。如果读者发现书中有关于 Greenplum 的问题，可以发邮件到 greenplum\_book@163.com，同时欢迎大家通过邮件一起交流关于 Greenplum 的问题。

## 致谢

感谢姜迅、吴伟两位老大，我在阿里巴巴 B2B 数据平台工作时他们给予了我很多帮助，是他们给予了我充足的时间来学习和锻炼；感谢孔令西、孙伟光、张耀耀、江小辉、曾春秋等各位同事对我工作的支持和帮助；感谢一起维护 Greenplum 数据库的何勇、唐成、王海、任振中，他们带给我一段难忘的共同奋斗的经历。特别感谢女友李梅花在写书过程中的支持和体谅，她作为一个数据分析师，从用户的角度，在本书写作过程中给予了很多宝贵的建议。没有这些支持，我们不可能完成本书的写作。

最后，感谢杨福川、姜影在本书出版过程中提供的帮助，尤其是姜影，她在本书审阅过程中非常认真和仔细，指出了书稿中很多的问题。

陈晓峰

# 目 录 *Contents*

前言

## 上篇 基 础 篇

<b>第 1 章 Greenplum 简介</b>	2
1.1 Greenplum 的起源和发展历程	2
1.2 OLTP 与 OLAP	3
1.3 PostgreSQL 与 Greenplum 的关系	3
1.3.1 PostgreSQL	3
1.3.2 Greenplum	5
1.4 Greenplum 特性及应用场景	6
1.4.1 Greenplum 特性	6
1.4.2 Greenplum 应用场景	7
1.5 小结	8
<b>第 2 章 Greenplum 快速入门</b>	9
2.1 软件安装及数据库初始化	9
2.1.1 Greenplum 架构	9
2.1.2 环境搭建	11
2.1.3 Greenplum 安装	13
2.1.4 创建数据库	20
2.1.5 数据库启动与关闭	20
2.2 安装 Greenplum 的常见问题	22

2.2.1 /etc/hosts 配置错误 .....	22
2.2.2 MASTER_DATA_DIRECTORY 设置错误 .....	24
2.3 畅游 Greenplum.....	25
2.3.1 如何访问 Greenplum .....	25
2.3.2 数据库整体概况 .....	27
2.3.3 基本语法介绍 .....	28
2.3.4 常用数据类型 .....	35
2.3.5 常用函数 .....	37
2.3.6 分析函数 .....	43
2.3.7 分区表 .....	46
2.3.8 外部表 .....	49
2.3.9 COPY 命令.....	51
2.4 小结.....	52
<b>第 3 章 Greenplum 实战 .....</b>	<b>53</b>
3.1 历史拉链表 .....	53
3.1.1 应用场景描述 .....	53
3.1.2 原理及步骤 .....	54
3.1.3 表结构 .....	55
3.1.4 Demo 数据准备.....	57
3.1.5 数据加载 .....	58
3.1.6 数据刷新 .....	61
3.1.7 分区裁剪 .....	64
3.1.8 数据导出 .....	64
3.2 日志分析.....	65
3.2.1 应用场景描述 .....	65
3.2.2 数据 Demo.....	65
3.2.3 日志分析实战 .....	66
3.3 数据分布.....	68
3.3.1 数据分散情况查看 .....	69
3.3.2 数据加载速度影响 .....	69
3.3.3 数据查询速度影响 .....	72

3.4	数据压缩.....	73
3.4.1	数据加载速度影响 .....	73
3.4.2	数据查询速度影响 .....	74
3.5	索引.....	75
3.6	小结.....	75

## 中篇 进 阶 篇

第 4 章	数据字典详解.....	78
4.1	oid 无处不在 .....	78
4.2	数据库集群信息 .....	80
4.2.1	Gp_configuration 和 gp_segment_configuration.....	80
4.2.2	Gp_id .....	82
4.2.3	Gp_configuration_history .....	84
4.2.4	pg_filespace_entry .....	84
4.2.5	集群配置信息表转化 .....	84
4.3	常用数据字典.....	85
4.3.1	pg_class.....	85
4.3.2	pg_attribute .....	88
4.3.3	gp_distribution_policy .....	89
4.3.4	pg_statistic 和 pg_stats .....	90
4.4	分区表信息 .....	90
4.4.1	如何实现分区表 .....	91
4.4.2	pg_partition .....	91
4.4.3	pg_partition_rule .....	92
4.4.4	pg_partitions 视图及其优化 .....	93
4.5	自定义类型以及类型转换.....	94
4.6	主、备节点同步的相关数据字典 .....	95
4.7	数据字典应用示例.....	96
4.7.1	获取表的字段信息 .....	96
4.7.2	获取表的分布键 .....	96
4.7.3	获取一个视图的定义 .....	97

4.7.4	查询 comment (备注信息).....	98
4.7.5	获取数据库建表语句 .....	99
4.7.6	查询表上的视图 .....	103
4.7.7	查询表的数据文件创建时间 .....	104
4.7.8	分区表总大小 .....	106
4.7.9	如何分析数据字典变化 .....	108
4.7.10	获取数据库锁信息 .....	111
4.8	Gp_toolkit 介绍 .....	112
4.9	小结.....	114
	<b>第 5 章 执行计划详解.....</b>	<b>115</b>
5.1	执行计划入门 .....	115
5.1.1	什么是执行计划 .....	115
5.1.2	查看执行计划 .....	116
5.2	分布式执行计划概述.....	116
5.2.1	架构 .....	116
5.2.2	重分布与广播 .....	117
5.2.3	Greenplum Master 的工作 .....	119
5.3	Greenplum 执行计划中的术语.....	120
5.3.1	数据扫描方式 .....	120
5.3.2	分布式执行 .....	121
5.3.3	两种聚合方式 .....	122
5.3.4	关联 .....	123
5.3.5	SQL 消耗.....	126
5.3.6	其他术语 .....	126
5.4	数据库统计信息收集.....	128
5.4.1	Analyze 分析 .....	128
5.4.2	固定执行计划 .....	129
5.5	控制执行计划的参数介绍.....	130
5.6	规划器开销的计算方法 .....	131
5.7	各种执行计划原理分析 .....	133
5.7.1	详解关联的广播与重分布 .....	133

5.7.2 HashAggregate 与 GroupAggregate.....	137
5.7.3 Nestloop Join 、Hash Join 与 Merge Join .....	141
5.7.4 分析函数：开窗函数和 grouping sets.....	142
5.8 案例.....	144
5.8.1 关联键强制类型转换，导致重分布 .....	144
5.8.2 统计信息过期 .....	145
5.8.3 执行计划出错 .....	145
5.8.4 分布键选择不恰当 .....	147
5.8.5 计算 distinct.....	148
5.8.6 union 与 union all .....	150
5.8.7 子查询 not in .....	152
5.8.8 聚合函数太多导致内存不足 .....	154
5.9 小结.....	155
<b>第 6 章 Greenplum 高级应用 .....</b>	<b>156</b>
6.1 Appendonly 表与压缩表 .....	157
6.1.1 应用场景及语法介绍 .....	157
6.1.2 压缩表的性能差异 .....	157
6.1.3 Appendonly 表特性.....	158
6.1.4 相关数据字典 .....	164
6.2 列存储 .....	165
6.2.1 应用场景 .....	165
6.2.2 数据文件存储特性 .....	166
6.2.3 如何使用列存储 .....	166
6.2.4 性能比较 .....	166
6.3 外部表高级应用 .....	168
6.3.1 外部表实现原理 .....	168
6.3.2 可写外部表 .....	171
6.3.3 HDFS 外部表 .....	173
6.3.4 可执行外部表 .....	177
6.4 自定义函数——各个编程接口 .....	179
6.4.1 pl/pgsql.....	180

6.4.2 C 语言接口 .....	182
6.4.3 plpython .....	185
6.5 Greenplum MapReduce .....	187
6.6 小结 .....	193
<b>第 7 章 Greenplum 架构介绍 .....</b>	<b>195</b>
7.1 并行和分布式计算 .....	195
7.2 并行数据库 .....	197
7.3 Greenplum 架构分析 .....	198
7.4 冗余与故障切换 .....	199
7.5 数据分布及负载均衡 .....	200
7.6 跨库关联 .....	202
7.7 分布式事务 .....	203
7.8 其他大数据分析方案 .....	205
7.9 小结 .....	208

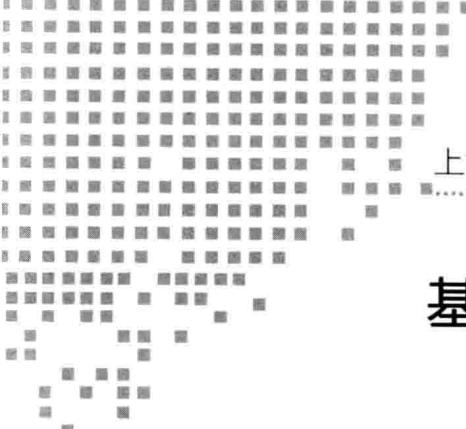
## 下篇 管理篇

<b>第 8 章 Greenplum 线上环境部署 .....</b>	<b>210</b>
8.1 服务器硬件选型 .....	210
8.1.1 CPU .....	211
8.1.2 内存 .....	211
8.1.3 磁盘及硬盘接口 .....	211
8.1.4 网络 .....	213
8.2 服务器系统参数调整 .....	213
8.2.1 Solaris 参数修改 .....	214
8.2.2 Linux 参数修改 .....	216
8.2.3 系统参数及性能验证 .....	217
8.3 计算节点分配技巧 .....	221
8.4 数据库参数介绍 .....	221
8.5 数据库集群基准测试 .....	225
8.6 小结 .....	227

<b>第 9 章 数据库管理.....</b>	228
9.1 用户及权限管理 .....	228
9.1.1 Greenplum 数据库逻辑结构 .....	228
9.1.2 Grant 语法 .....	229
9.2 登录权限控制.....	231
9.3 资源队列及并发控制.....	232
9.4 Greenplum 锁机制.....	236
9.5 数据目录结构.....	238
9.6 数据文件存储分布.....	240
9.7 表空间管理.....	241
9.8 小结.....	244
<b>第 10 章 数据库监控及调优 .....</b>	245
10.1 Linux 监控工具介绍 .....	245
10.1.1 监控磁盘 .....	245
10.1.2 监控网络 .....	246
10.1.3 监控 CPU.....	247
10.1.4 监控内存 .....	247
10.2 安装 Performance Monitor.....	248
10.3 监控 Segment 是否正常 .....	252
10.4 VACUUM 系统表.....	253
10.5 数据倾斜排查.....	255
10.6 查看子节点的 SQL 运行状态 .....	258
10.7 自动加分区 .....	261
10.8 自动赋权 .....	266
10.9 清理过期数据 .....	266
10.10 小结 .....	267
<b>第 11 章 解读 Greenplum 维护脚本 .....</b>	268
11.1 添加 Greenplum Contrib 模块.....	268
11.2 启动和关闭脚本 gpstart 和 gpstop .....	270
11.3 初始化系统脚本 gpinitsystem .....	272

11.4 集群操作脚本 gpssh 和 gpscp.....	274
11.5 数据库状态检查脚本 gpstate.....	275
11.6 数据库升级脚本 gpmigrate .....	276
11.7 参数修改脚本 gpconfig .....	281
11.8 数据库一致性检查脚本 gpcheckcat.....	282
11.9 小结.....	284
<b>第 12 章 备份及恢复策略 .....</b>	<b>286</b>
12.1 Greenplum 3.x.....	286
12.2 Greenplum 4.x.....	287
12.3 gp_dump 和 pg_dump .....	290
12.4 Greenplum Master 备份策略.....	294
12.4.1 增加 Standby Master .....	295
12.4.2 重新同步 Standby Master .....	296
12.4.3 启用 Standby Master .....	296
12.5 小结 .....	297
<b>第 13 章 数据库扩容 .....</b>	<b>299</b>
13.1 迁移计算节点.....	299
13.1.1 两种备份方案 .....	300
13.1.2 数据迁移实战 .....	301
13.2 增加计算节点 .....	306
13.3 小结 .....	311
<b>第 14 章 基于 Greenplum 的海量数据实时分析服务平台 .....</b>	<b>312</b>
14.1 需求概述.....	312
14.2 典型方案.....	313
14.2.1 NoSQL .....	313
14.2.2 分布式数据库 / 集群 .....	314
14.2.3 分表分库 .....	315
14.2.4 方案优劣分析 .....	316
14.3 基于 Greenplum 的混合架构 .....	316

14.3.1 架构分析 .....	317
14.3.2 实施要点 .....	317
14.4 小结 .....	318
<b>第 15 章 使用 Greenplum 的常见报错及小技巧</b> .....	<b>319</b>
15.1 分析常见报错 .....	319
15.1.1 找不到类型 705 对应的操作符 .....	319
15.1.2 SQL 占用的资源超过了资源队列限制 .....	321
15.1.3 自定义函数不能在 Segment 上执行 .....	321
15.1.4 子查询没有别名 .....	322
15.1.5 字段名有歧义 .....	322
15.1.6 字段重名 .....	323
15.1.7 gpfdist 错误：无法读取文件 .....	323
15.1.8 事务被中止 .....	324
15.1.9 网络异常错误 .....	324
15.1.10 无法删除表 .....	324
15.1.11 内存不足 .....	325
15.1.12 文件名在 pg_class 中已存在 .....	325
15.1.13 不能对分布键执行 Update .....	325
15.1.14 网络错误 .....	326
15.1.15 无法找到数据文件 .....	326
15.2 常见问题及解决办法 .....	326
15.3 常用的一些小技巧 .....	329
15.3.1 显示 SQL 执行的时间 .....	330
15.3.2 获取某个 schema 下所有的表或视图 .....	330
15.3.3 查找分区最多的表 .....	330
15.3.4 连接 Segment 节点 .....	331
15.3.5 psql 默认密码登录 .....	331
15.3.6 查看数据库启动时间 .....	331
15.3.7 查看在 psql 中 \d 到底查询了哪些数据字典 .....	331
15.4 小结 .....	332



上篇      *Part 1*

## 基础篇

- 第1章 Greenplum简介
  - 第2章 Greenplum快速入门
  - 第3章 Greenplum实战
- \*\*\*\*\*

## 第1章

# Greenplum 简介

本章先介绍 Greenplum 的产生背景、特性及应用场景、与 PostgreSQL 关系，以及发展历程。

## 1.1 Greenplum 的起源和发展历程

短短十多年，互联网在中国经历了从门户网站、搜索、即时通信、游戏娱乐、垂直细分……到电子商务、Web 2.0，再到社会化网络、移动互联网的一系列进化和变革。无论是互联网还是移动互联网，都是由海量的数据构成。对海量数据分析的需求开始突破传统边界，不再局限于电信、移动、金融、保险、制造等传统企业，涌现出大批将海量、庞杂的数据转化为知识，提供业务经营决策支持的企业。针对数据密集型计算中的海量数据处理这一问题，研究者开始考虑如何利用大规模集群系统所具有的可伸缩性和容错性的优势，实现高效的数据管理功能。比较典型的解决方案有 Teradata、Greenplum、Hadoop Hive、Oracle Exadata、IBM Netezza 等。

Greenplum 是一家总部位于美国加利福尼亚州，为全球大型企业用户提供新型企业级数据仓库（EDW）、企业级数据云（EDC）和商务智能（BI）提供解决方案和咨询服务的公司。选择 Greenplum 的产品的国际大客户有：纳斯达克、纽约证券交易所、Skype、FOX、T-Mobile 等，在中国，中信实业银行、东方航空公司、阿里巴巴、华泰保险、中国远洋（Cosco）、李宁公司等大型企业用户也选择了 Greenplum 的产品。Greenplum 的发展历程简要如下。

- 2003：Greenplum 由 Scott Yara 和 Luke Lonergan 成立。
- 2005：Greenplum 数据库第一个版本发布。

- 2006：与 Sun 公司合作，成为其合伙人。
- 2008：Greenplum MapReduce 发布，同年 12 月份进入中国市场，一年多后，Greenplum 正式宣布在中国独立运营。
- 2010：Greenplum 被 EMC 收购，并被整合到 EMC 的云计算战略中。
- 2011—2012：Greenplum 社区版发布，Greenplum Chorus 发布并开源。
- 2014：Greenplum 4.3 发布。

## 1.2 OLTP 与 OLAP

数据库系统一般分为两种类型，一种是面向前台应用的，应用比较简单，但是重吞吐和高并发的 OLTP 类型；一种是重计算的，对大数据集进行统计分析的 OLAP 类型。Greenplum 属于后者，下面简单介绍下这两种数据库系统的特点。

OLTP（On-Line Transaction Processing，联机事务处理）系统也称为生产系统，它是事件驱动的、面向应用的，比如电子商务网站的交易系统就是一个典型的 OLTP 系统。OLTP 的基本特点是：

- 数据在系统中产生；
- 基于交易的处理系统（Transaction-Based）；
- 每次交易牵涉的数据量很小；
- 对响应时间要求非常高；
- 用户数量非常庞大，主要是操作人员；
- 数据库的各种操作主要基于索引进行。

OLAP（On-Line Analytical Processing，联机分析处理）是基于数据仓库的信息分析处理过程，是数据仓库的用户接口部分。OLAP 系统是跨部门的、面向主题的，其基本特点是：

- 本身不产生数据，其基础数据来源于生产系统中的操作数据（OperationalData）；
- 基于查询的分析系统；
- 复杂查询经常使用多表联结、全表扫描等，牵涉的数据量往往十分庞大；
- 响应时间与具体查询有很大关系；
- 用户数量相对较小，其用户主要是业务人员与管理人员；
- 由于业务问题不固定，数据库的各种操作不能完全基于索引进行。

## 1.3 PostgreSQL 与 Greenplum 的关系

### 1.3.1 PostgreSQL

PostgreSQL 是一种非常先进的对象 – 关系型数据库管理系统（ORDBMS），是目前功能

最强大，特性最丰富和技术最先进的自由软件数据库系统之一，其某些特性甚至连商业数据库都不具备。这个起源于伯克利（BSD）的数据库研究计划目前已经衍生成一项国际开发项目，并且有非常广泛的用户。

PostgreSQL 的特点可以说是数不胜数，称其为最先进的开源软件数据库当之无愧，支持绝大部分的主流数据库特性，主要体现在如下几方面。

#### (1) 函数 / 存储过程

PostgreSQL 对非常丰富的过程类语言提供支持，可以编写自定义函数 / 存储过程。

- ❑ 内置的 plpgsql，一种类似 Oracle 的 PLsql 的语言。
- ❑ 支持的脚本语言有：PL/Lua、PL/OLECODE、PL/Perl、PL/HP、PL/Python、PL/Ruby、PL/sh、PL/Tcl 和 PL/Scheme。
- ❑ 编译语言有 C、C++ 和 JAVA。
- ❑ 统计语言 PL/R。

#### (2) 索引

PostgreSQL 支持用户定义的索引访问方法，并且内置了 B-tree、哈希和 GiST 索引。

PostgreSQL 中的索引有下面几个特点。

- ❑ 可以从后向前扫描。
- ❑ 可以创建表达式索引。
- ❑ 部分索引。

#### (3) 触发器

触发器是由 SQL 查询的动作触发的事件。比如，一个 INSERT 查询可能激活一个检查输入值是否有效的触发器。大多数触发器都只对 INSERT 或者 UPDATE 查询有效。

PostgreSQL 完全支持触发器，可以附着在表上，但是不能在视图上。不过视图可以有规则。多个触发器是按照字母顺序触发的。我们还可以用其他过程语言书写触发器函数，不仅仅 PL/PgSQL。

#### (4) 并发管理 (MVCC)

PostgreSQL 的并发管理使用的是种叫做“MVCC”(多版本并发机制)的机制，这种机制实际上就是现在在众多所谓的编程语言中极其火爆的“Lock Free”，其本质是通过类似科幻世界的时空穿梭的原理，给予每个用户一个自己的“时空”，然后通过原子的“时空”控制来控制时间基线，并以此控制并发更改的可见区域，从而实现近乎无锁的并发，而同时还能在很大程度上保证数据库的 ACID 特性。

#### (5) 规则 (RULE)

规则允许我们对由一个查询生成的查询树进行改写。

#### (6) 数据类型

PostgreSQL 支持非常广泛的数据类型，包括：

- ❑ 任意精度的数值类型；

- 无限长度的文本类型；
- 几何原语；
- IPv4 和 IPv6 类型；
- CIDR 块和 MAC 地址；
- 数组。

用户还可以创建自己的类型，并且可以利用 GiST 框架把这些类型做成完全可索引的，比如来自 PostGIS 的地理信息系统（GIS）的数据类型。

#### (7) 用户定义对象

因为 PostgreSQL 使用一种基于系统表的可扩展的结构设计，所以 PostgreSQL 内部的几乎所有对象都可以由用户定义，这些对象包括：

- 索引；
- 操作符（内部操作符可以被覆盖）；
- 聚集函数；
- 域；
- 类型转换；
- 编码转换。

#### (8) 继承

PostgreSQL 的表是可以相互继承的。一个表可以有父表，父表的结构变化会导致子表的结构变化，而对子表的插入和数据更新等也会反映到父表中。

#### (9) 其他特性与扩展

PostgreSQL 还支持大量其他的特性，比如：

- 二进制和文本大对象存储；
- 在线备份；
- TOAST（The Oversized-Attribute Storage Technique）用于透明地在独立的地方保存大的数据库属性，当数据超过一定大小的时候，会自动进行压缩以节省空间；
- 正则表达式。

此外 PostgreSQL 还有大量的附加模块和扩展版本，比如，多种不同的主从 / 主主复制方案：

- Slony-I；
- pgcluster；
- Mammoth replicator；
- Bucardo。

### 1.3.2 Greenplum

简单地说，Greenplum 就是一个与 Oracle、DB2、PostgreSQL 一样面向对象的关系型数

据库。我们通过标准的 SQL 可以对 Greenplum 中的数据进行访问存取。

本质上讲，Greenplum 是一个关系型数据库集群，它实际上是由数个独立的数据库服务组合成的逻辑数据库。与 Oracle RAC 的 Shared-Everything 架构不同，Greenplum 采用 Shared-Nothing 架构，整个集群由很多个数据节点（Segment Host）和控制节点（Master Host）组成，其中每个数据节点上可以运行多个数据库。简单来说，Shared-Nothing 是一个分布式的架构，每个节点相对独立。在典型的 Shared-Nothing 中，每一个节点上所有的资源（CPU，内存，磁盘）都是独立的，每个节点都只有全部数据的一部分，也只能使用本节点的资源。

基于对 Shared-Nothing 分布式架构模式的分析，Greenplum 高效处理 I/O 数据吞吐和并发计算的过程就很好理解了。在 Greenplum 中，需要存储的数据在进入数据库时，将先进行数据分布的处理工作，将一个表中的数据平均分布到每个节点上，并为每个表指定一个分发列（distribute Column），之后便根据 Hash 来分布数据。基于 Shared-Nothing 的原则，Greenplum 这样处理可以充分发挥每个节点处 I/O 的处理能力。在这一过程中，控制节点（Master Host）将不再承担计算任务，而只负责必要的逻辑控制和客户端交互。I/O 瓶颈的解决为并行计算能力的提升创造了良好的环境，所有节点服务器组成一个强大的计算平台，实现快速的海量并行运算。Greenplum 在数据仓库、商业智能的应用上，尤其是在海量数据的处理方面性能极其优异。

Greenplum 是面向数据仓库应用的关系型数据库，它是基于目前流行的 PostgreSQL 开发的，跟 PostgreSQL 的兼容性非常好，大部分的 PostgreSQL 客户端工具及 PostgreSQL 应用都能运行在 Greenplum 平台上。

## 1.4 Greenplum 特性及应用场景

### 1.4.1 Greenplum 特性

#### （1）支持海量数据存储和处理

当今是个数据迅速增长的时代，数据量从过去的 MB 到 GB，再到 TB 增长到现在的 PB 级规模，传统的 OLTP 数据库在 TB 级别以上的数据管理中已经捉襟见肘。Greenplum 使用 MPP 架构，同时使用多台机器并行计算，极大地提高了对海量数据的处理能力。采取 MPP 架构的数据库系统才能对海量数据进行管理。

#### （2）高性价比

Greenplum 数据库可以搭建在业界各种开放式硬件平台上，在硬件选型上有很强的自由性。

相比其他封闭式数据仓库专用系统及 Hadoop 分析平台，Greenplum 在每 TB 数据量上的投资是前者的 1/5 甚至更低。

Greenplum licence 相比 Oracle RAC、Teradata 等，价格低廉。

Greenplum 易于维护，可以节省大量的维护成本。

#### (3) 支持 Just In Time BI

Greenplum 通过准实时、实时的数据加载方式，实现数据仓库的实时更新，进而实现动态数据仓库（ADW）。基于动态数据仓库，业务用户能对当前业务数据进行 BI 实时分析（Just In Time BI），能够让企业敏锐感知市场的变化，加快决策支持反应速度。

#### (4) 系统易用性

Greenplum 是基于 PostgreSQL 开发的，语法与 PostgreSQL 几乎一样，PostgreSQL 的工具基本上都能够再 Greenplum 中使用，比如 pgadmin 等。Greenplum 使用通用的 PostgreSQL 连接包即可与数据库连接，支持绝大部分开发语言。Greenplum 的易用性具体表现如下。

- 支持主流的 SQL 语法，使用起来十分方便，学习成本低。
- 扩展性好，支持多语言的自定义函数和自定义类型等。
- 提供了大量的维护工具，使用维护起来很方便。
- 在 Internet 上有着丰富的 PostgreSQL 资源供用户参考。

#### (5) 支持线性扩展

Greenplum 采用 MPP 并行处理架构。在 MPP 架构中增加节点就可以线性提高系统的存储容量和处理能力。Greenplum 在扩展节点时操作简单，在很短时间内就能完成数据的重新分布。Greenplum 线性扩展支持为数据分析系统将来的拓展提供了技术上的保障，使用户可根据实施需要进行容量和性能的扩展。

#### (6) 较好的并发支持及高可用性支持

Greenplum 是高可用的系统，在已有案例中最多使用了 96 台机器的集群 MPP 环境。除了硬件级的 Raid 技术外，Greenplum 还提供数据库层 Mirror 机制保护，也就是将每个节点的数据在另外的节点中同步镜像，单个节点的错误不影响整个系统的使用。对于主节点，Greenplum 提供 Master/Stand by 机制进行主节点容错，当主节点发生错误时，可以切换到 Stand by 节点继续服务。

#### (7) 支持 MapReduce

MapReduce 已经被谷歌和雅虎等互联网领先企业证明是一种大规模数据分析技术，Greenplum 将这种能力提供给企业。

#### (8) 数据库内部压缩

面对海量数据，压缩可以节省很大的空间，而且在对大数据的分析时，压缩也可能减少对磁盘的访问。Greenplum 支持对数据库表进行压缩处理，从而提升数据库的性能。

### 1.4.2 Greenplum 应用场景

Greenplum 数据引擎是为新一代数据仓库和大规模分析处理而建立的软件解决方案，其最大的特点是不需要高端的硬件支持仍然可以支撑大规模的高性能数据仓库和商业智能查询。在数据仓库、商业智能的应用上，尤其在海量数据的处理方面 Greenplum 表现出极其优

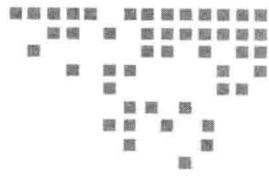
异的性能。

传统数据库侧重交易处理，关注的是多用户的同时的双向操作，在保障即时性的要求下，系统通过内存来处理数据的分配、读写等操作，存在 IO 瓶颈。而分析型数据库是以实时多维分析技术作为基础，对数据进行多角度的模拟和归纳，从而得出数据中所包含的信息和知识。Greenplum 虽然是关系型数据库产品，但是它具有查询速度快、数据装载速度快、批量 DML 处理快的主要特点，而且性能可以随着硬件的添加呈线性增加，拥有非常好的可扩展性。因此，Greenplum 主要适用于面向分析的应用，比如构建企业级 ODS/EDW、数据集市等。

在国内，笔者所在公司——阿里巴巴（中国）网络技术有限公司，从 2008 年开始引入 Greenplum，将原有的 Oracle RAC 迁移到 Greenplum 上，作为数据仓库的计算中心，其中一个应用就是通过分析用户的网络点击日志进行产品的关联分析。支付宝在 2008 年也引入了 Greenplum 数据库作为数据中心。国内还有很多银行也引入了 Greenplum 作为基础的数据平台，如北京银行、深发展银行、中信银行信用卡中心等。在 TB 级的数据仓库的 OLAP 应用中 Greenplum 在易用性和性能方面有着很大的优势。

## 1.5 小结

本章非常简短地介绍了 Greenplum 产生的背景，并分别对比 OLTP 与 OLAP、PostgreSQL 和 Greenplum，另外简要介绍了 Greenplum 的特性、应用场景及发展历程，相信通过这一章能让读者在总体上对 Greenplum 有所认识。



## 第2章

# Greenplum 快速入门

*Chapter 2*

本章将介绍如何快速安装部署 Greenplum，以及 Greenplum 的一些常用命令及工具。“工欲善其事，必先利其器”，因此我们先从如何安装 Greenplum 开始介绍，然后介绍一些简单的工具，以及 Greenplum 的语法及特性。

为了让读者更加快速地入门，避免涉及太多底层的东西。本章不会涉及硬件选型、操作系统参数讲解、机器性能测试等高级内容，这些会在“第 8 章 Greenplum 线上环境部署”中介绍。

## 2.1 软件安装及数据库初始化

下面先介绍如何搭建一个完整的 Greenplum 环境。在搭建环境之前，我们必须对 Greenplum 的架构有一定的了解，并且准备好安装部署的机器，机器硬件、操作系统的安装配置读者可自行完成。

### 2.1.1 Greenplum 架构

在安装数据库的时候，我们先要对 Greenplum 架构有一定的了解，这样可以对数据库的安装和使用起到一个指导性的作用。同时在搭建 Greenplum 环境的过程中，可以加深对 Greenplum 架构的理解。Greenplum 总体架构图如图 2-1 所示。

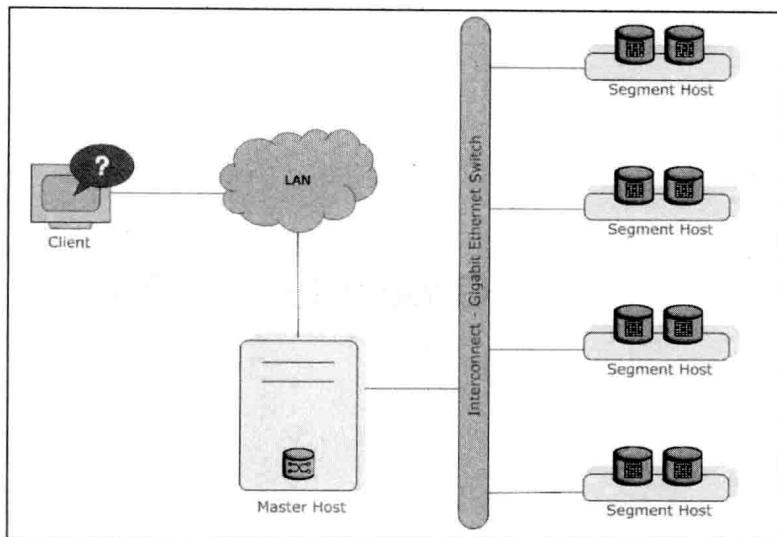


图 2-1 Greenplum 总体架构图

下面介绍每个部件的主要功能，如表 2-1 所示。

表 2-1 Master 主机与 Segment 主机对比

Master 主机负责	Segment 主机负责
1) 建立与客户端的会话连接和管理	1) 业务数据的存储和存取
2) SQL 的解析并形成分布式的执行计划	2) 执行由 Master 分发的 SQL 语句
3) 将生成好的执行计划分发到每个 Segment 上执行	3) 对于 Master 来说，每个 Segment 都是对等的，负责对应数据的存储和计算
4) 收集 Segment 的执行结果	4) 每一台机器上可以配置一到多个 Segment
5) Master 不存储业务数据，只存储数据字典	5) 由于每个 Segment 都是对等的，建议采用相同的机器配置
6) Master 主机可以一主一备，分布在两台机器上	6) Segment 分 primary 和 mirror 两种，一般交错地存放在子节点上
7) 为了提高性能，Master 最好单独占用一台机器	

通过图 2-2 可以看出 Master 与 Segment 的关系。

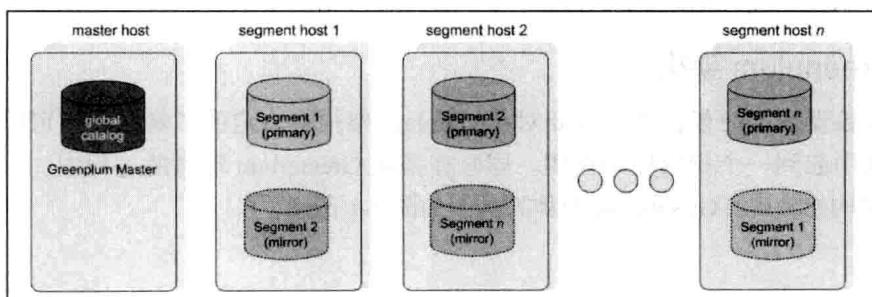


图 2-2 Master 与 Segment 的关系

Master 和 Segment 其实都是一个单独的 PostgreSQL 数据库。每一个都有自己单独的一套元数据字典，在这里，Master 节点一般也叫主节点，Segment 也叫做数据节点。

Segment 节点与 Master 节点的通信，通过千兆（或万兆）网卡组成的内部连接（InterConnect），在同一台数据节点机器上可以放多个 Segment，不同的 Segment 节点会被赋予不同的端口，同时，Segment 之间也不断地进行着交互。为了实现高可用，每个 Segment 都有对应的备节点（Mirror Segment），分别存在于不同的机器上。

Client 一般只能与 Master 节点进行交互，Client 将 SQL 发给 Master，然后 Master 对 SQL 进行分析后，再将其分配给所有的 Segment 进行操作，并且将汇总结果返回给客户端。

## 2.1.2 环境搭建

### 1. Greenplum 集群介绍

在这里的 Greenplum 集群中，有 4 台机器，IP 分别是：

- 10.20.151.7；
- 10.20.151.8；
- 10.20.151.9；
- 10.20.151.10。

机器对应的 Master 和 Segment 如下分配：10.20.151.7 作为 Master 节点，10.20.151.8 ~ 10 作为 Segment 节点，每个机器上配置两个 Primary Segment 和两个 Mirror Segment；同时 10.20.151.10 作为 Master Standby 节点。

通过图 2-3 的架构图可以清晰地知道我们所搭建的集群的概况。

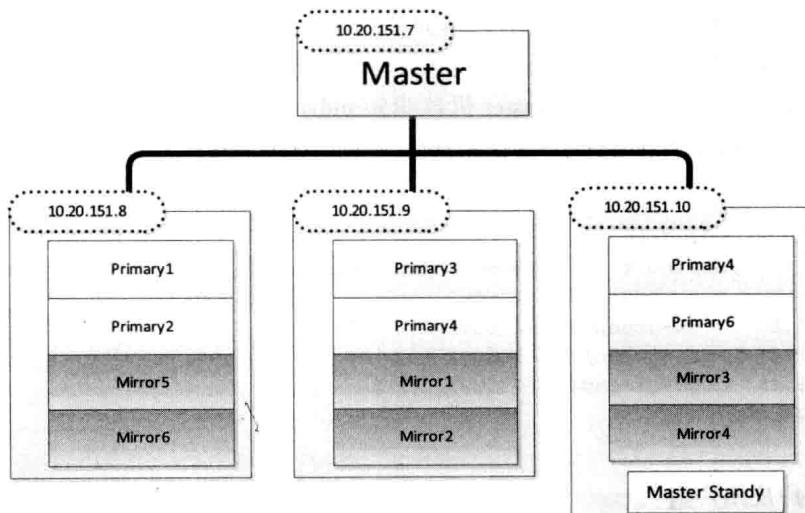


图 2-3 安装测试环境的 Greenplum 架构

## 2. 安装 Linux

Greenplum 没有 Windows 版本，只能安装在类 UNIX 的操作系统上，例如：

- SUSE Linux SLES 10.2 or higher;
- CentOS 5.0 or higher;
- RedHat Enterprise Linux 5.0 or higher;
- Solaris x86 v10 update 7。

如果读者没有现成的 Linux 机器，可以在虚拟机（如 VMWare）上安装，用户可自行安装 VMWare 及对应的 Linux 操作系统，网上相应的资料很多，这里就不再阐述。

## 3. 数据库存储

对于数据库来说，在性能上磁盘 IO 很容易成为瓶颈，由于数据库的特性，每一个 SQL 基本都是对全表数据进行分析，每次处理的数据量非常大，数据基本上都是没有缓存的（数据字典除外），极度消耗 IO 资源（全表扫描主要都是顺序 IO），所以 Greenplum 对存储的要求比较高。在文件系统的选择上，在 Linux 下建议使用 XFS，在 Solaris 下建议使用 ZFS，对于 Raid 根据需求选择硬 Raid 或软 Raid，如果需要更大的空间，建议使用 Raid 5，如果对性能有更高的要求，可以选择 Raid 1+0。相关内容请参考第 8 章关于 Greenplum 线上环境部署的介绍。

如果只是想试用 Greenplum，数据量比较小，那么怎么存储无所谓，可以用任意的目录作为数据库数据目录。

## 4. 网络 (hosts)

在确定机器配置的时候，要保证所有机器的网络都是通的，并且每台机器的防火墙都是关闭的，避免存在网络不通的问题。

在配置 /etc/hosts 时，习惯将 Master 机器叫做 mdw，将 Segment 机器叫做 sdw，配置好后，使用 ping 命令确定所有 hostname 都是通的。

```
#cat /etc/hosts
#BEGIN_GROUP_CUSTOMER
127.0.0.1      localhost  localhost.localdomain
10.20.151.7    dw-greenplum-1 mdw
10.20.151.8    dw-greenplum-2 sdw1
10.20.151.9    dw-greenplum-3 sdw2
10.20.151.10   dw-greenplum-4 sdw3
#End_Group_Customer
```

## 5. 创建用户及用户组

创建 gpadmin 用户及用户组，将其作为安装 Greenplum 的操作系统用户。

将原有用户删除：

```
#groupdel gpadmin
#userdel gpadmin
```

创建新的用户和用户组：

```
#groupadd -g 530 gpadmin
#useradd -g 530 -u 530 -m -d /home/gpadmin -s /bin/bash gpadmin
```

对文件夹进行赋权，为新用户创建密码：

```
#chown -R gpadmin:gpadmin /home/gpadmin/
#passwd gpadmin
Changing password for user gpadmin.
New UNIX password:
Retype new UNIX password:
```

## 2.1.3 Greenplum 安装

### 1. 安装数据库软件

读者可以在 Greenplum 官网上下载最新版本的 Greenplum，下载地址是：<https://network.gopivotal.com/products/pivotal-gpdb>，如图 2-4 所示。

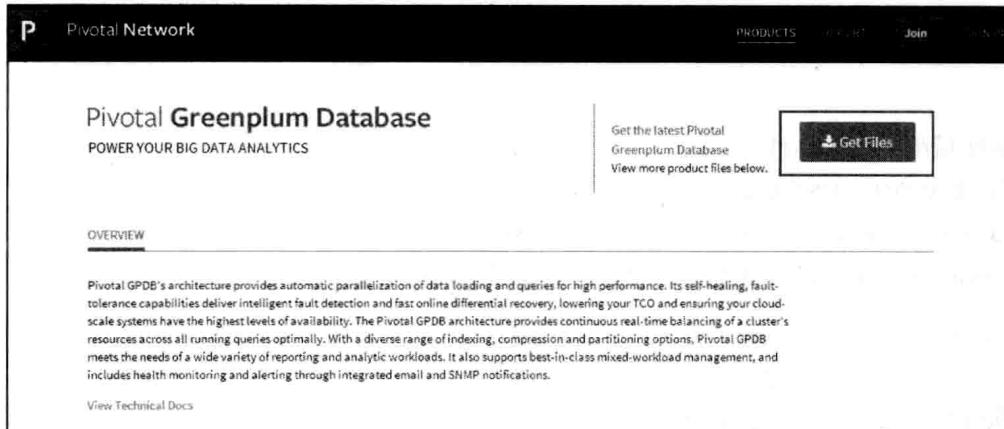


图 2-4 官方网站 Greenplum 下载页面

选择相应的操作系统版本，比如 RedHat 操作系统选择“RHEL”，下载时需要在网站上注册一个账号，如图 2-5 所示。

读者可以选择自己需要的版本下载，这里以正式版本 Greenplum 4.1.1.1 为例，介绍如何安装 Greenplum。

AVAILABLE FILES						
<input type="text"/> RHEL	RHEL					
Status	Name	Version	Release Date	Size	Update Type	
NEW	Greenplum Database 4.3 - Clients for RHEL4	4.3.0.0	Apr 01, 2014	7.5 MB	Major release	
NEW	Greenplum Database 4.3 - PL/Perl Extension for RHEL6	5.12.4_pv1.2	Apr 01, 2014	140 KB	Major release	
NEW	Greenplum Database 4.3 - PL/Perl Extension for RHEL5	5.12.4_pv1.2	Apr 01, 2014	160 KB	Major release	
NEW	Greenplum Database 4.3 - PL/Java Extension for RHEL	1.4.0_pv1.1	Apr 01, 2014	31 MB	Major release	
NEW	Greenplum Database 4.3 - PL/R Extension for RHEL	8.3.0.12_pv1.0	Apr 01, 2014	29 MB	Major release	
NEW	Greenplum Database 4.3 - PostGIS Extension for RHEL	2.0.3_pv2.0	Apr 01, 2014	18 MB	Major release	
NEW	Greenplum Database 4.3 - Pgcrypto Extension for RHEL	1.1_pv1.1	Apr 01, 2014	180 KB	Major release	
NEW	Greenplum Database 4.3 - Partner Connector for RHEL	1.2	Apr 01, 2014	120 KB	Major release	
✓	Greenplum Database 4.3.0.0 Server for RHEL 5 & 6	4.3.0.0	Apr 01, 2014	120 MB	Major release	

11 - 19 of 19 (Filtered from 55 total entries) [«Previous](#) [Next»](#)

Most recent download: Greenplum Database 4.3.0.0 Server for RHEL 5 & 6-4.3.0.0 on Apr 22, 2014 | [View all download history](#)

图 2-5 Greenplum 版本选择



注意 Greenplum 最新版本是 Greenplum 4.3，但是安装方式都是一样的。

首先准备好安装文件：

```
greenplum-db-4.1.1.1-build-1-RHEL5-x86_64.zip
```

执行 unzip 命令解压安装文件：

```
unzip greenplum-db-4.1.1.1-build-1-RHEL5-x86_64.zip
```

解压后生成两个文件：

□ README\_INSTALL；

□ greenplum-db-4.1.1.1-build-1-RHEL5-x86\_64.bin。

为 Greenplum 软件创建安装目录，并且赋给 gpadmin 用户权限：

```
mkdir /opt/greenplum
chown -R gpadmin:gpadmin /opt/greenplum
```

执行以下命令开始安装软件：

```
./greenplum-db-4.1.1.1-build-1-RHEL5-x86_64.bin
```

屏幕上会出现 License 的一些信息，按“空格”键使信息显示完全，如图 2-6 所示。

确认 License 之后，接着出现如图 2-7 所示的信息。

输入“yes”后会提示安装目录，我们选择安装在 /opt/greenplum/greenplum-db-4.1.1.1 下面，如图 2-8 所示。

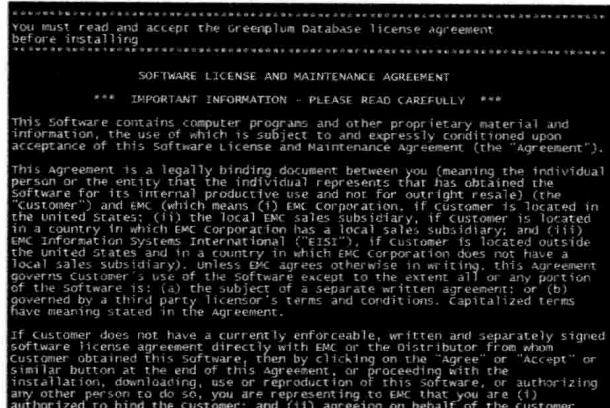


图 2-6 安装 Greenplum 的 License 信息



图 2-7 是否接受 License

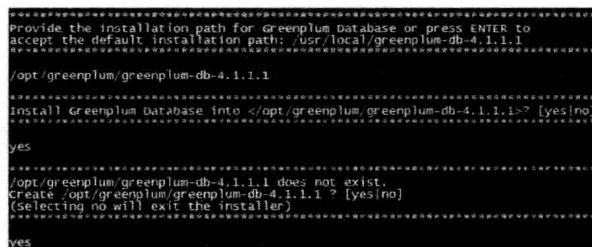
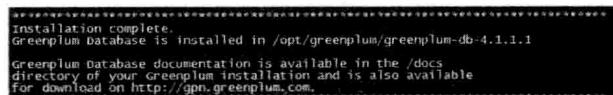
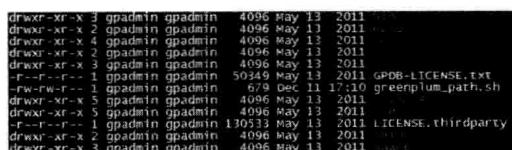


图 2-8 选择安装目录

完成以上步骤后，软件开始自动安装。最后显示软件安装成功，安装目录如图 2-9b 所示。



a) 安装成功的提示信息



b) 软件目录的结构

图 2-9 安装后的 Greenplum 软件目录

Greenplum 的环境变量已经在 greenplum\_path.sh 中设置了，这里需要应用一下这个环境变量配置：

```
source /opt/greenplum/greenplum-db/greenplum_path.sh
```

## 2. 配置 hostlist

配置 hostlist 文件，将所有的服务器名记录在里面。

```
[gpadmin@dw-greenplum-1 conf]$ cat hostlist
mdw
sdw1
sdw2
sdw3
```

seg\_hosts 只保存 segment 节点的 hostname。

```
[gpadmin@dw-greenplum-1 conf]$ cat seg_hosts
sdw1
sdw2
sdw3
```

## 3. 使用 gpssh-exkeys 打通所有服务器

使用 gpssh-exkeys 将所有机器的通道打开，这样就不用输入密码使登录在每台机器之间跳转了，代码如下：

```
[gpadmin@dw-greenplum-1 conf]$ gpssh-exkeys -f hostlist
[STEP 1 of 5] create local ID and authorize on local host
[STEP 2 of 5] keyscan all hosts and update known_hosts file
[STEP 3 of 5] authorize current user on remote hosts
... send to sdw1
 ***
 *** Enter password for sdw1:
... send to sdw2
... send to sdw3
[STEP 4 of 5] determine common authentication file content
[STEP 5 of 5] copy authentication files to all remote hosts
... finished key exchange with sdw1
... finished key exchange with sdw2
... finished key exchange with sdw3
[INFO] completed successfully
```

在打通所有机器通道之后，我们就可以使用 gpssh 命令对所有机器进行批量操作了。

```
[gpadmin@dw-greenplum-1 conf]$ gpssh -f hostlist
=> pwd
[sdw3] /home/gpadmin
```

```
[ mdw] /home/gpadmin
[ sdw1] /home/gpadmin
[ sdw2] /home/gpadmin
```

#### 4. 将软件分发到每一台机器上

接下来将安装后的文件打包：

```
tar -cf gp4.1.tar greenplum-db-4.1.1.1/
```

然后利用 **gpscp** 命令将这个文件复制到每一台机器上：

```
gpscp -f /home/gpadmin/conf/hostlist gp4.1.tar =:/opt/greenplum
```

使用 **gpssh** 命令批量解压文件包：

```
=> cd /opt/greenplum
[sdw3]
[ mdw]
[sdw1]
[sdw2]
=> tar -xf gp4.1.tar
[sdw3]
[ mdw]
[sdw1]
[sdw2]
```

建立软连接，如图 2-10 所示。

```
[sdw3] total 153340
[sdw3] -rw-rw-r-- 1 gpadmin gpadmin 157061120 Dec 11 17:24 gp4.1.tar
[sdw3] lrwxrwxrwx 1 gpadmin gpadmin      20 Dec 11 17:26 greenplum-db -> greenplum-db-4.1.1.1
[sdw3] drwxrwxr-x 11 gpadmin gpadmin    4096 Dec 11 17:26 greenplum-db-4.1.1.1
[ mdw] total 153340
[ mdw] -rw-rw-r-- 1 gpadmin gpadmin 157061120 Dec 11 17:24 gp4.1.tar
[ mdw] lrwxrwxrwx 1 gpadmin gpadmin      22 Dec 11 17:10 greenplum-db -> ./greenplum-db-4.1.1.1
[ mdw] drwxrwxr-x 11 gpadmin gpadmin    4096 Dec 11 17:26 greenplum-db-4.1.1.1
[sdw1] total 153340
[sdw1] -rw-rw-r-- 1 gpadmin gpadmin 157061120 Dec 11 17:24 gp4.1.tar
[sdw1] lrwxrwxrwx 1 gpadmin gpadmin      20 Dec 11 17:26 greenplum-db -> greenplum-db-4.1.1.1
[sdw1] drwxrwxr-x 11 gpadmin gpadmin    4096 Dec 11 17:26 greenplum-db-4.1.1.1
[sdw2] total 153340
[sdw2] -rw-rw-r-- 1 gpadmin gpadmin 157061120 Dec 11 17:24 gp4.1.tar
[sdw2] lrwxrwxrwx 1 gpadmin gpadmin      20 Dec 11 17:26 greenplum-db -> greenplum-db-4.1.1.1
[sdw2] drwxrwxr-x 11 gpadmin gpadmin    4096 Dec 11 17:26 greenplum-db-4.1.1.1
```

图 2-10 建立 Greenplum 安装路径的软连接

下面创建数据库数据目录。

MASTER 目录：

```
=> mkdir -p /home/gpadmin/gpdata/gpmaster
```

Primary 节点目录：

```
=> mkdir -p /home/gpadmin/gpdata/gpdatap1
=> mkdir -p /home/gpadmin/gpdata/gpdatap2
```

## □ Mirror 节点目录:

```
=> mkdir -p /home/gpadmin/gpdata/gpdatam1
=> mkdir -p /home/gpadmin/gpdata/gpdatam2
```

Gpmaster 目录保存 Master 的数据，每个机器上的 gpdatap1、gpdatap2 分别对应这个机器上的两个主数据节点目录，同样的，gpdatam1、gpdatam2 对应备数据节点目录。

## 5. 配置 `~/.bash_profile`

要对系统的环境变量进行配置，需要修改 `~/.bash_profile`，添加以下内容：

```
source /opt/greenplum/greenplum-db/greenplum_path.sh
export MASTER_DATA_DIRECTORY=/home/gpadmin/gpdata/gpmaster/gpseg-1
export PGPORT=2345
export PGDATABASE=testDB
```

其中 `greenplum_path.sh` 保存了运行 Greenplum 的一些环境变量设置，包括 `GPHOME`、`PYTHONHOME` 等设置。

## 6. 初始化 Greenplum 的配置文件

配置文件的模板可以在 `$GPHOME/docs/cli_help/gpconfigs/` 目录下找到。`gpinitsystem_config` 文件是初始化 Greenplum 的模板，在这个模板中，Mirror Segment 的配置都被注释掉了，模板中基本初始化数据库的参数都是有的。

下面是初始化的配置文件 `initgp_config`。

```
# 数据库的代号
ARRAY_NAME="Greenplum"
MACHINE_LIST_FILE=/home/gpadmin/conf/seg_hosts
#Segment 的名称前缀
SEG_PREFIX=gpseg
#Primary Segment 起始的端口号
PORT_BASE=33000
# 指定 Primary Segment 的数据目录
declare -a DATA_DIRECTORY=(/home/gpadmin/gpdata/gpdatap1 /home/gpadmin/gpdata/gpdatap2)
#Master 所在机器的 Hostname
MASTER_HOSTNAME=mdw
# 指定 Master 的数据目录
MASTER_DIRECTORY=/home/gpadmin/gpdata/gpmaster
#Master 的端口
MASTER_PORT=2345
# 指定 Bash 的版本
TRUSTED_SHELL=/usr/bin/ssh
# 字符集 ENCODING=UNICODE
#Mirror Segment 起始的端口号
MIRROR_PORT_BASE=43000
#Primary Segment 主备同步的起始端口号
```

```

REPLICATION_PORT_BASE=34000
#Mirror Segment 主备同步的起始端口号
MIRROR_REPLICATION_PORT_BASE=44000
#Mirror Segment 的数据目录
declare -a MIRROR_DATA_DIRECTORY=(/home/gpadmin/gpdata/gpdatam1 /home/gpadmin/
gpdata/gpdatam2)

```

## 7. 初始化数据库

使用 gpinit system 脚本来初始化数据库，命令如下：

```
gpinit system -c initgp_config -s sdw3
```

根据脚本出现的提示操作即可，如图 2-11 所示。

```

gpadmin:[INFO]:-GP_LIBRARY_PATH IS           = /opt/greenplum/greenplum-db/.lib
gpadmin:[WARN]:-ulimit check
gpadmin:[INFO]:-Array host connect type
gpadmin:[INFO]:-Primary IP address [1]      = 10.20.151.7
gpadmin:[INFO]:-Standby Master
gpadmin:[INFO]:-Primary segment #
gpadmin:[INFO]:-standby IP address          = 10.20.151.10
gpadmin:[INFO]:-Total Database segments      = 6
gpadmin:[INFO]:-Trusted shell
gpadmin:[INFO]:-Segment hosts
gpadmin:[INFO]:-Mirror port base            = 43000
gpadmin:[INFO]:-Replication port base       = 34000
gpadmin:[INFO]:-Mirror replication port base = 44000
gpadmin:[INFO]:-Mirror segment #             = 2
gpadmin:[INFO]:-Mirroring config            = ON
gpadmin:[INFO]:-Mirroring type              = Group
gpadmin:[INFO]:-----
gpadmin:[INFO]:-Greenplum Primary Segment Configuration
gpadmin:[INFO]:-----
gpadmin:[INFO]:-sdw1   /home/gpadmin/gpdata/gpdatapl/gpseg0  33000  2    0    34000
gpadmin:[INFO]:-sdw1   /home/gpadmin/gpdata/gpdatapl/gpseg1  33001  3    1    34001
gpadmin:[INFO]:-sdw2   /home/gpadmin/gpdata/gpdatapl/gpseg2  33000  4    2    34000
gpadmin:[INFO]:-sdw2   /home/gpadmin/gpdata/gpdatapl/gpseg3  33001  5    3    34001
gpadmin:[INFO]:-sdw3   /home/gpadmin/gpdata/gpdatapl/gpseg4  33000  6    4    34000
gpadmin:[INFO]:-sdw3   /home/gpadmin/gpdata/gpdatapl/gpseg5  33001  7    5    34001
gpadmin:[INFO]:-----
gpadmin:[INFO]:-Greenplum Mirror Segment Configuration
gpadmin:[INFO]:-----
gpadmin:[INFO]:-sdw2   /home/gpadmin/gpdata/gpdatam1/gpseg0  43000  8    0    44000
gpadmin:[INFO]:-sdw2   /home/gpadmin/gpdata/gpdatam1/gpseg1  43001  9    1    44001
gpadmin:[INFO]:-sdw3   /home/gpadmin/gpdata/gpdatam2/gpseg2  43000  10   2    44000
gpadmin:[INFO]:-sdw3   /home/gpadmin/gpdata/gpdatam2/gpseg3  43001  11   3    44001
gpadmin:[INFO]:-sdw4   /home/gpadmin/gpdata/gpdatam1/gpseg4  43000  12   4    44000
gpadmin:[INFO]:-sdw4   /home/gpadmin/gpdata/gpdatam2/gpseg5  43001  13   5    44001

```

a) 数据库初始化过程中提示的 Segment 信息

```

-[INFO]:-No db instance process, entering recovery startup mode
-[INFO]:-Commencing parallel primary and mirror segment instance startup, please wait...
-[INFO]:-Process results...
-[INFO]:-----[INFO]:- Successful segment starts                               = 12
-[INFO]:- Failed segment starts                                         = 0
-[INFO]:- Skipped segment starts (segments are marked down in configuration) = 0
-[INFO]:-----[INFO]:-Successfully started 12 of 12 segment instances
-[INFO]:-----[INFO]:-starting Master instance inc-dw-hadoop-151-7.hst.bjc.kfc.alidc.net directory /home/gpadmin/g
-[INFO]:-Command pg_ctl reports Master inc-dw-hadoop-151-7.hst.bjc.kfc.alidc.net instance active
-[INFO]:-Database successfully started with no errors reported

```

b) 数据库初始化成功

图 2-11 初始话数据库

这样，数据库就初始化成功了，尝试登录 Greenplum 默认的数据库 postgres：

```
[gpadmin@dw-greenplum-1 ~]$ psql -d postgres
psql (8.2.15)
Type "help" for help.
```

```
postgres=#
```

## 2.1.4 创建数据库

现在我们开始创建测试数据库：

```
createdb testDB -E utf-8
```

没有设置 PGDATABASE 这个环境变量时，使用 psql 进行登录，默认的数据库是与操作系统用户名一致的，这时候会报错：

```
[gpadmin@dw-greenplum-1 ~]$ psql
psql: FATAL: database "gpadmin" does not exist
```

然后设置 (export) 环境变量 PGDATABASE=testDB，这样就默认登录 testDB 数据库：

```
[gpadmin@dw-greenplum-1 ~]$ export PGDATABASE=testDB
[gpadmin@dw-greenplum-1 ~]$ psql
psql (8.2.15)
Type "help" for help.
```

```
testDB=#
```

查询数据库版本并创建一张简单的表：

```
testDB=# select version();
version
-----
 PostgreSQL 8.2.15 (Greenplum Database 4.1.1.1 build 1) on x86_64-unknown-
linux-gnu, compiled by GCC gcc (GCC) 4.4.2 compiled on May 12 2011 18:07:08
(1 row)

testDB=# create table test01(id int primary key,name varchar(128));
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "test01_pkey"
for table "test01"
CREATE TABLE
```

## 2.1.5 数据库启动与关闭

### 1. 启动数据库

Greenplum 提供一系列的脚本来管理数据库，其中 gpstart 就是启动数据库的脚本，我们可以用 gpstart --help 来查看帮助，例如：

```
[gpadmin@inc-dw-hadoop-151-7 ~]$ gpstart --help
COMMAND NAME: gpstart
Starts a Greenplum Database system.

*****
SYNOPSIS
```

```
*****
gpstart [-d <master_data_directory>] [-B <parallel_processes>]
        [-R] [-m] [-y] [-a] [-t <timeout_seconds>]
        [-l logfile_directory] [-v | -q]
gpstart -? | -h | --help
gpstart --version
```

一般用 gpstart -a 直接启动数据库，不用输入“yes”，如图 2-12 所示。

```
[INFO]:-Starting gpstart with args: -a
[INFO]:-Gathering information and validating the environment...
[INFO]:-Greenplum Catalog Version: pgc@192.168.1.10:5432 (greenplum database) 4.1.1.1 build 1'
[INFO]:-Obtaining Greenplum Catalog configuration: 201101130
[INFO]:-Starting Master instance in admin mode
[INFO]:-Obtaining Greenplum Master catalog information
[INFO]:-Obtaining Segment details from master...
[INFO]:-Master Started...
[INFO]:-Shutting down master
[INFO]:-Starting standby master
[INFO]:-Checking if standby master is running on host: sdw3 in directory: /home/gpadmin/gpdata/gpmaster/gps
[INFO]:-No db instance process, entering recovery startup mode
[INFO]:-Commencing parallel primary and mirror segment instance startup, please wait...
[INFO]:-Process results...
[INFO]:-- Successful segment starts = 12
[INFO]:-- Failed segment starts = 0
[INFO]:-- Skipped segment starts (segments are marked down in configuration) = 0
[INFO]:-
[INFO]:-Successfully started 12 of 12 segment instances
[INFO]:-Starting Master instance inc-dw-hadoop-151-7.hst.bjc.kfc.alidc.net directory /home/gpadmin/gpdata/gpmaster/gps
[INFO]:-Command pg_ctl reports Master inc-dw-hadoop-151-7.hst.bjc.kfc.alidc.net instance active
[INFO]:-database successfully started with no errors reported
```

图 2-12 启动数据库

## 2. 关闭数据库

关闭数据库的脚本是 gpstop，其使用方法如下：

```
COMMAND NAME: gpstop
Stops or restarts a Greenplum Database system.
*****
SYNOPSIS
*****
gpstop [-d <master_data_directory>] [-B <parallel_processes>]
        [-M smart | fast | immediate] [-t <timeout_seconds>]
        [-r] [-y] [-a] [-l <logfile_directory>] [-v | -q]
gpstop -m [-d <master_data_directory>] [-y] [-l <logfile_directory>]
        [-v | -q]
gpstop -u [-d <master_data_directory>] [-l <logfile_directory>]
        [-v | -q]
gpstop --version
gpstop -? | -h | --help
```

与 gpstart一样，在关闭数据库时，一般使用 gpstop -a，这样就不用输入“yes”了，如图 2-13 所示。

```
[INFO] : -starting gpstop with args: -a
[INFO] : -Gathering information and validating the environment...
[INFO] : -obtaining Greenplum Master catalog information
[INFO] : -obtaining Segment details from master...
[INFO] : -Greenplum version: 'postgres (Greenplum Database) 4.1.1.1 build 1'
[INFO] : -There are 0 connections to the database
[INFO] : -Commencing Master instance shutdown with mode='smart'
[INFO] : -Master host=inc-dw-hadoop-151-8 instance=kfc-allidc.net
[INFO] : -Commencing Master instance shutdown with mode='smart'
[INFO] : -Master segment instance directory=/home/gpadmin/gpdata/gpmaster/gpseg-1
[INFO] : -Stopping gpssyncmaster on standby host sdw3 mode=fast
[INFO] : -Successfully shutdown sync process on sdw3
[INFO] : -Commencing parallel primary segment instance shutdown, please wait...
[INFO] : -Commencing parallel mirror segment instance shutdown, please wait...
[INFO] : -----
[INFO] : Segments stopped successfully = 12
[INFO] : Segments with errors during stop = 0
[INFO] : -----
[INFO] : Successfully shutdown 12 of 12 segment instances
[INFO] : Database successfully shutdown with no errors reported
```

图 2-13 关闭数据库

## 2.2 安装 Greenplum 的常见问题

安装 Greenplum 最常见的错误就是环境变量设置错误，网卡配置错误，或者是每个 Segment 的通道或网络没有打通。

如果子节点的操作系统环境不一样，也可能会导致各种各样的错误。所以在搭建环境的时候，要求每一台机器的配置基本一样，方便以后管理与维护，避免一些奇怪的问题。下面将介绍几个常见的报错及处理方法。

### 2.2.1 /etc/hosts 配置错误

现在来看一个奇怪的报错 SQL，查询一张普通表时报如下错误，但是查询数据字典又不报错：

```
testDB=# select * from test002;
WARNING: Greenplum Database detected segment failure(s), system is reconnected
WARNING: Greenplum Database detected segment failure(s), system is reconnected
ERROR: No primary gang allocated (cdbgang.c:1635)
testDB=# select count(1) from pg_class;
 count
-----
 633
(1 row)
```

报这个错误是因为 Master 连接不到 Segment。如果原先是一个正常的系统，突然报错了，就要想想是否修改了什么导致的。在这个例子中，是因为修改了 /etc/hosts，不小心将：

```
10.20.151.8      inc-dw-hadoop-151-8 sdw1
```

改成了：

```
10.20.151.18      inc-dw-hadoop-151-8 sdw1
```

Master 连接 Segment 的时候连接不上，就报了这个错误，但是 pg\_class 是每一个节点都

有的，而 Master 上的数据只需要在 Master 上查询，不用连接 Segment，所以没有报错，只需将配置修改回来即可。

有些时候，也可以利用这个方法来判断一个操作是否需要与 Master 交互，比如生成执行计划（关于执行计划的详细内容可阅读第 5 章）是否只在 Master 上执行，与 Segment 有没有交互：

```
testDB=# explain select * from test002;
WARNING: Greenplum Database detected segment failure(s), system is reconnected
WARNING: Greenplum Database detected segment failure(s), system is reconnected
ERROR: No primary gang allocated (cdbgang.c:1635)
testDB=# explain select count(1) from pg_class;
          QUERY PLAN
-----
Aggregate (cost=39.20..39.21 rows=1 width=0)
 -> Seq Scan on pg_class (cost=0.00..20.48 rows=7488 width=0)
(2 rows)
```

这样就可以看出，生成分布式执行计划也是需要与 Segment 进行交互的。那么为什么下面生成 pg\_class 的执行计划不会报错呢？这是因为表 test002 是业务数据，数据分布在 Segment 上，生成的执行计划是分布式的，而 pg\_class 是保存表元数据信息的数据字典，数据保存在 Master 上，生成的执行计划是单机的，详见第 5 章执行计划。



**注意** 在 Greenplum 4.3 中，这个报错是：ERROR: Unexpected internal error (cdbgang.c:1622)。报错的内容与原因都是一样的，只是在显示上 Greenplum 4.1 与 Greenplum 4.3 有所区别。

在 Greenplum 3.3.x 版本中，如果修改了 Master 的 /etc/hosts 配置，将：

```
127.0.0.1 localhost localhost.localdomain
```

错写成：

```
127.0.0.1 localhost localhost.localdomain mdw
```

那么还会有如下的奇怪现象发生。

在创建表的时候 Master 没有报错，说明 Master 与 Segment 是可以通信的，但是查询这张表的时候就会报如下的错误：

```
testDB=# create table aaa(like bbb);
NOTICE: Table doesn't have 'distributed by' clause, defaulting to distribution
columns from LIKE table
CREATE TABLE
testDB =# select * from bbbb;
ERROR: Interconnect timeout: Unable to complete setup of all connections
within time limit.
DETAIL: Completed 2 of 6 incoming and 0 of 0 outgoing connections.
gp_interconnect_setup_timeout = 20 seconds.
```

错误提示已经超时了，建议将超时时间设置大一点，但是修改相关参数还是没有用。其实是因为 Segment 在连接到 Master 的时候使用的 IP 是 127.0.0.1，指定到了本地机器，导致 Segment 连接不到 Master，故报出这个错误。将 Master 连接到 Segment 就没有问题，因此将对应的 /etc/hosts 修改回来就可以了。



**提示** 将 /etc/hosts 改回来之后，进行查询后还是报同样的错，这是因为当前连接还没有生效，要退出当前的 session，重新连接 Greenplum 才行。

Greenplum 内部会有很多的网络通信交互，因此建议将操作系统的防火墙关闭，避免各种奇怪的问题，其中防火墙就会引起本节介绍的在 Greenplum 3.3.x 版本中遇到的问题。

## 2.2.2 MASTER\_DATA\_DIRECTORY 设置错误

没有设置 MASTER\_DATA\_DIRECTORY，会报这样的错误：

```
[gpadmin@dw-greenplum-1 ~]$ gpstop
20120110:11:16:52:gpstop:dw-greenplum-1:gpadmin1-[INFO]:-Starting gpstop with
args: ''
20120110:11:16:52:gpstop:dw-greenplum-1:gpadmin1-[INFO]:-Gathering information
and validating the environment...
20120110:11:16:52:gpstop:dw-greenplum-1:gpadmin1-[CRITICAL]:-gpstop failed.
(Reason='Environment Variable MASTER_DATA_DIRECTORY not set!') exiting...
```

这样会导致 MASTER\_DATA\_DIRECTORY 参数的目录设置错误：

```
[gpadmin@dw-greenplum-1 ~]$ echo $MASTER_DATA_DIRECTORY
/home/gpadmin/gpdata/gpmaster/gpseg-1
[gpadmin@dw-greenplum-1 ~]$ export MASTER_DATA_DIRECTORY=/home/gpadmin/gpdata/
gpmaster/gpseg1
[gpadmin@dw-greenplum-1 ~]$ gpstop
20120110:11:19:43:gpstop:dw-greenplum-1:gpadmin-[INFO]:-Starting gpstop with args:
20120110:11:19:43:gpstop:dw-greenplum-1:gpadmin-[INFO]:-Gathering information
and validating the environment...
20120110:11:19:43:gpstop:dw-greenplum-1:gpadmin-[CRITICAL]:-gpstop failed.
(Reason='[Errno 2] No such file or directory: \'/home/gpadmin/gpdata/gpmaster/
gpseg1/postgresql.conf\'') exiting...
```

在初始化数据库的时候要多注意环境变量的设置，如果环境变量设置不当，很容易造成数据库初始化错误。

## 2.3 畅游 Greenplum

本节只介绍一些常用的命令，重点是 Greenplum 特有的一些命令，而对于一般数据库都具备的特性及 SQL 标准语法，本节提到的比较少，因此要求读者在阅读本节具备一定的 SQL 基础。

### 2.3.1 如何访问 Greenplum

#### 1. psql

`psql` 是 Greenplum/PostgreSQL 默认的客户端，前面初始化数据库的时候已经使用过了，下面介绍一些详细的用法。

```
[gpadmin@dw-greenplum-1 ~]$ psql --help
This is psql 8.2.15, the PostgreSQL interactive terminal (Greenplum version).
Usage:
  psql [OPTION]... [DBNAME [USERNAME]]

General options:
  -c, --command=COMMAND      run only single command (SQL or internal) and exit
  -d, --dbname=DBNAME        database name to connect to (default: "testDB")
...
  ...
  ...

Connection options:
  -h, --host=HOSTNAME        database server host or socket directory (default:
"local socket")
  -p, --port=PORT             database server port (default: "2345")
  -U, --username=USERNAME    database user name (default: "gpadmin")
```

我们可以在其他机器上使用 `psql` 连接到数据库中，例如：

```
admin@test1:/home/admin>psql -h 10.20.151.7 -p 2345 -d testDB -U gpadmin
psql: FATAL:  no pg_hba.conf entry for host "10.20.151.1", user "gpadmin",
database "testDB", SSL off
```

之所以报错，是因为 Greenplum 有权限控制，并不是所有的机器都可以连接到数据库上。关于如何配置权限控制，会在“第 9 章数据库管理”的前两节详细介绍。如果有其他计算机要登录 Greenplum，先为数据库用户 `gpadmin` 创建一个密码，然后在 `pg_hba.conf` 文件中增加客户端机器的权限配置，这样就可以成功登录了。

```
testDB=# alter role gpadmin with password 'gpadmin';
ALTER ROLE
```

接着在 `$MASTER_DATA_DIRECTORY/pg_hba.conf` 文件中增加：

```
host testDB gpadmin 10.20.151.1/32 md5
```

之后通过 gpstop -u 命令使配置生效，如图 2-14 所示。

```
[INFO]:-Starting gpstop with args: -u
[INFO]:-Gathering information and validating the environment...
[INFO]:-Obtaining Greenplum Master catalog information
[INFO]:-Obtaining segment details from master...
[INFO]:-Greenplum Version: 'postgres (Greenplum Database) 4.1.1.1 build 1'
[INFO]:-Signalling all postmasters processes to reload ]
```

图 2-14 使用 gpstop -u 重新加载配置

这样我们就可以在其他机器上登录数据库了。

```
admin@test1:/home/admin>psql -h 10.20.151.7 -p 2345 -d testDB -U gpadmin -W
Password for user gpadmin:
psql (8.2.13, server 8.2.15)
Type "help" for help.
testDB=#
```

## 2. pgAdmin

当然，除了采用 psql 的登录方法之外，还可以利用图形界面的 GUI，就是 pgAdmin 这款软件。我们可以从 <http://www.pgadmin.org/download/> 这个网址上下载 pgAdmin。

这里以 Windows 为例，介绍下 pgAdmin 软件的安装及使用。

首先，下载 pgadmin3-1.14.1.zip 后将其解压并进行安装。

然后，启动 pgAdmin，自动弹出“新增服务器登记”界面，如图 2-15 所示。



图 2-15 PgAdmin 新增配置界面

接下来单击“确认”按钮创建连接，登录到 Greenplum 数据库中（需要在 pg\_hba.conf 中将本机的 IP 加入认证）。



图 2-16 PgAdmin 连接成功后的界面

单击图 2-16 中的 ，弹出如图 2-17 所示的 pgAdmin SQL 编辑器界面，我们就可以在其中编写 SQL 查询语句了。

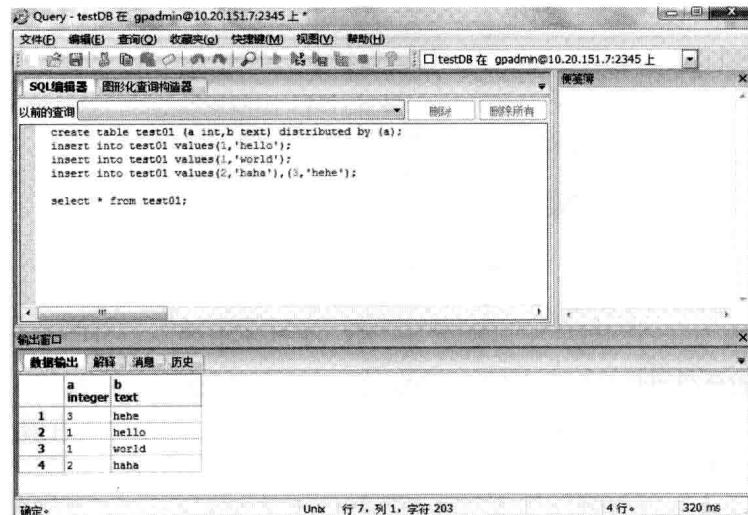


图 2-17 pgAdmin SQL 编辑器界面

### 2.3.2 数据库整体概况

本小节稍微介绍一下 Greenplum 的数据分布，以方便读者理解 Greenplum 的一些操作。

## 1. Greenplum 基于 PostgreSQL 开发

Greenplum 是基于开源数据库软件 PostgreSQL 8.2 开发的，其大部分语法和数据字典都与 PostgreSQL 一样，很多工具使用的规范也基本跟 PostgreSQL 一样。因此，在学习的过程中，可以参考网上的一些 PostgreSQL 的资料。强烈建议参考 PostgreSQL 8.2 的官方中文文档，其中大部分内容都适用于 Greenplum。

## 2. Greenplum 的数据分布

可以说 Greenplum 将 PostgreSQL 改造成一个分布式数据库。其中，Segment 节点都是一个单独的 PostgreSQL 数据库，Master 本身也是一个 PostgreSQL 数据库。

Master 本身不储存数据，所有数据拆分保存到每一个节点上。在指定分布键的时候，数据按照分布键的 Hash 值来分布数据，称为哈希分布。还有一种分布不用指定分布键，数据随机分布到每一个节点上，称作随机分布（也叫平均分布）。

以一张 Student 表为例，它在 Greenplum 中的数据分布如图 2-18 所示。

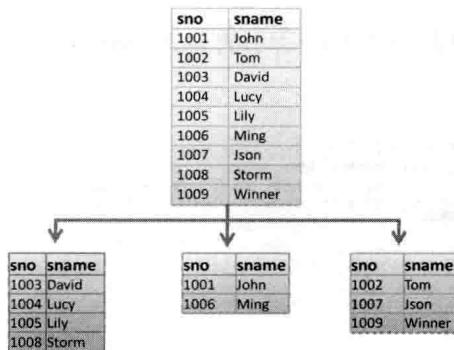


图 2-18 Student 表在 Greenplum 中的数据分布

### 2.3.3 基本语法介绍

Greenplum 是一个很全面的数据库，提供了很多的功能。如果将所有语法都讲一遍，篇幅会很大，所以这里只将一些基本的语法简单罗列一下。

#### 1. 获取语法介绍

可以使用 \h 查看 Greenplum 支持的所有语法，如图 2-19 所示。

在 psql 中使用 \h command 可以获取具体命令的语法：

```
testDB=# \h create view
Command: CREATE VIEW
```

Description: define a new view

Syntax:

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW name [ ( column_name [, ...] ) ]
      AS query
```



图 2-19 Greenplum 支持的 SQL 语法

## 2. CREATE TABLE

Greenplum 中的建表语句跟普通数据库的建表语句区别不大，仅有几个地方与其他数据库不同。

- ❑ 在 Greenplum 中建表时需要指定表的分布键。
  - ❑ 如果表需要用某个字段分区，可以通过 `partition by` 将表建成分区表。
  - ❑ 可以使用 `like` 操作创建与 `like` 的表一样结构的表，功能类似 `create table t1 as select * from t2 limit 0`。
  - ❑ 可以使用 `inherits` 实现表的继承，具体的实现可以参考 PostgreSQL 文档。

在 Greenplum 中，建表语句的语法大致如下：

```
[, ... ] ] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter=value [, ... ] ) ]
[ ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP} ]
[ TABLESPACE tablespace ]
[ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
[ PARTITION BY partition_type (column)
    [ SUBPARTITION BY partition_type (column) ]
        [ SUBPARTITION TEMPLATE ( template_spec ) ]
    [...]
( partition_spec )
    | [ SUBPARTITION BY partition_type (column) ]
    [...]
( partition_spec
    [ ( subpartition_spec
        [(...)]
    ) ]
)
```

由于 Greenplum 是一个分布式数据库，数据肯定是分布在每一个节点上的。在 Greenplum 中有两种数据分布策略：

1) Hash 分布。指定一个或多个分布键，计算 hash 值，并且通过 hash 值路由到特定的 Segment 节点上，语法为 `Distributed by ( .. )`。如果不指定分布键，默认将第一个字段作为分布键。

2) 随机分布,也叫平均分布。数据随机分散在每一个节点中,这样无论数据是什么内容,都可以平均分布在每个节点上,但是在执行SQL的过程中,关联等操作都需要将数据重新分布,性能较差。语法为在表字段定义的后面加上Distributed randomly。

下面两个建表语句的执行结果一样，都是以 id 作为分布键：

```
testDB=# create table test001(id int ,name varchar(128));
NOTICE:  Table doesn't have 'DISTRIBUTED BY' clause -- Using column named 'id'
as the Greenplum Database data distribution key for this table.
HINT:   The 'DISTRIBUTED BY' clause determines the distribution of data. Make
sure column(s) chosen are the optimal data distribution key to minimize skew.
CREATE TABLE
```

```
testDB=# create table test002(id int ,name varchar(128)) distributed by(id);  
CREATE TABLE
```

在下面的建表语句中指定了多个分布键：

```
testDB=# create table test003(id int ,name varchar(128)) distributed by(id,name);
CREATE TABLE
```

在下面的建表语句中采用了随机分布：

```
testDB=# create table test004(id int ,name varchar(128)) distributed randomly;
CREATE TABLE
```

采用随机分布策略的表默认将主键或唯一键作为分布键，因为每一个 Segment 都是一个单一的数据库，单个的数据库可以保证唯一性，多个数据库节点就无法保证全局的跨库唯一性，故只能按照唯一键分布，同一个值的数据都在一个节点上，以此来保证唯一性。

```
testDB=# create table test005(id int primary key,name varchar(128));
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "test005_pkey"
for table "test005"
CREATE TABLE
testDB=# create table test006(id int unique,name varchar(128));
NOTICE: CREATE TABLE / UNIQUE will create implicit index "test006_id_key" for
table "test006"
CREATE TABLE
```

如果指定的分布键与主键不一样，那么分布键会被更改为主键：

```
testDB=# create table test007(id int unique,name varchar(128)) distributed
by(id,name);
NOTICE: updating distribution policy to match new unique index
NOTICE: CREATE TABLE / UNIQUE will create implicit index "test007_id_key" for
table "test007"
CREATE TABLE
testDB=# \d test007
      Table "public.test007"
 Column |          Type          | Modifiers
-----+-----+-----+
 id    | integer           |
 name  | character varying(128) |
Indexes:
 "test007_id_key" UNIQUE, btree (id)
Distributed by: (id)
```

在创建表的时候，如果要建一张表结构一模一样的表，可以利用 create table like 命令：

```
testDB=# create table test001_like (like test001);
NOTICE: Table doesn't have 'distributed by' clause, defaulting to distribution
columns from LIKE table
CREATE TABLE
```

使用 like 创建的表，只是表结构会与原表一模一样，表的一些特殊属性并不会一样，例如压缩、只增（appendonly）等属性。如果不指定分布键，则默认分布键与原表一样。

### 3. SELECT

Greenplum 中的 SELECT 操作基本上与普通关系型数据库中的 SELECT 操作没有太大的区别。要了解 SQL 的执行计划，对于 Greenplum 而言，重点要了解分布式执行计划，这个具体会在“第 5 章执行计划详解”中详细介绍。

下面介绍 SELECT 语句的语法。SELECT 语句的基本语法跟其他数据库类似，也有自己的一些特性，例如分页采用 offset 加 limit 操作：

```
Command:      SELECT
Description:  retrieve rows from a table or view
Syntax:
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
        * | expression [ [AS] output_name ] [, ...]
        [ FROM from_item [, ...] ]
        [ WHERE condition ]
        [ GROUP BY grouping_element [, ...] ]
        [ HAVING condition [, ...] ]
        [ WINDOW window_name AS (window_specification) ]
        [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
        [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
        [ LIMIT { count | ALL } ]
        [ OFFSET start ]
        [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]
```

一个简单的示例如下：

```
testDB=# select id,name from test001 order by id;
 id | name
----+-----
 100 | jack
 101 | david
 102 | tom
 103 | lily
(4 rows)
```

SELECT 可以不用指定 From 子句，如执行函数：

```
testDB=# select greatest(1,2);
greatest
-----
 2
(1 row)
```

进行一些简单的科学计算等：

```
testDB=# select 2^3+3+9*(8+1);
?column?
-----
 92
(1 row)
```

需要注意的是，Greenplum 的数据切分放在所有的 Segment 上。当从一个表查询数据的时候，Master 的数据展现顺序是以 Master 先接收到的数据的顺序，每个 Segment 的数据到达 Master 的顺序是随机的，不是固定的，所以执行 SELECT 的结果的顺序是随机的，即使表中数据一点变化都没有。这一点跟其他数据库是不一样的，下面是两次执行 SELECT 的结果，

从中明显可以看出数据的顺序是不一样的。如果要求多次查询的结果一样，必须显式地加一个 `order by` 语句，强制输出的结果排序。

下面是连续两次对同一个表执行查询操作的结果（中间数据没有发生过任何变更），可以看出，如果不加 `order by` 子句，在查询的结果中，数据的顺序是不能够保证的。

```
testDB=# select * from test001;      testDB=# select * from test001;
      id | name                  id | name
-----+-----  -----
    101 | david                101 | david
   103 | lily                 100 | jack
   100 | jack                 102 | tom
   102 | tom                  103 | lily
(4 rows)                         (4 rows)
```

#### 4. `create table as` 与 `select into`

`create table as` 与 `select into` 有一样的功能，都可以使表根据直接执行 `SELECT` 的结果创建出一个新的表，这个在临时分析数据的时候十分方便。例如，在创建一个表的时候如果默认不指定分布键，那么 Greenplum 根据执行 `SELECT` 得到的结果集来选择，不用再次重分布数据的字段作为表的分布键：

```
testDB=# create table test2 as select * from test1;
NOTICE:  Table doesn't have 'DISTRIBUTED BY' clause -- Using column(s) named
'id' as the Greenplum Database data distribution key for this table.
HINT:  The 'DISTRIBUTED BY' clause determines the distribution of data. Make
sure column(s) chosen are the optimal data distribution key to minimize skew.
SELECT 10000
```

当然，也可以手工加入 `distributed` 关键字，指定分布键，这样数据就会根据指定分布键再建表：

```
testDB=# create table test3 as select * from test1 distributed by(id);
SELECT 10000
```

`select into` 的语法比 `create table as` 更简单，虽然功能一样，但是执行 `select into` 不能指定分布键，只能使用默认的分布键，例如：

```
testDB=# select * into test4 from test1;
NOTICE:  Table doesn't have 'DISTRIBUTED BY' clause -- Using column(s) named
'id' as the Greenplum Database data distribution key for this table.
HINT:  The 'DISTRIBUTED BY' clause determines the distribution of data. Make
sure column(s) chosen are the optimal data distribution key to minimize skew.
SELECT 10000
```

## 5. explain

explain 用于查询一个表的执行计划，它在 SQL 优化的时候经常要用到，详细的执行计划解释请参考“第 5 章执行计划详解”。

下面代码演示了简单的执行计划的查看方法：

```
testDB=# explain select * from test1 x,test2 y where x.id=y.id;
          QUERY PLAN
-----
 Gather Motion 6:1  (slice1; segments: 6)  (cost=243.00..511.00 rows=1667
 width=30)
 -> Hash Join  (cost=243.00..511.00 rows=1667 width=30)
     Hash Cond: x.id = y.id
     -> Seq Scan on test1 x  (cost=0.00..118.00 rows=1667 width=15)
     -> Hash  (cost=118.00..118.00 rows=1667 width=15)
           -> Seq Scan on test2 y  (cost=0.00..118.00 rows=1667 width=15)
(6 rows)
```

可以看出，上面代码查看的是一个简单的关联生成的执行计划，该执行计划是一个层次关系，先从最右边开始查看。

第一步，数据库先顺序扫描 test2 表，扫描大概有 118 单位的消耗，有 1667 行数据，平均长度为 15 字节。其中，1667 行数据是一个估计值，是一个 Segment 的数据量，如果数据分布均匀，大概是总数据量除以 Segment 的个数。由于这个 Greenplum 集群有 6 个 Segment 节点，因此可以推断 test2 表大概有 1 万行数据。

第二步，扫描出 test2 表，并且计算 hash 值，将其保存在内存中。

第三步，顺序扫描 test1 表。

第四步，在扫描 test1 表的过程中，与 test2 表进行 hash 后的结果关联（hash join），关联的条件是两表的 id 字段相同。

第五步，将数据汇总到 Master 上。Master 将数据结果进行汇总并展现。

## 6. insert、update 和 delete

这 3 个操作是数据库最基本的操作，基本语法就不介绍了，只讲几点数据切片带来的问题。

1 ) insert：在执行 insert 语句的时候，要留意分布键不要为空，否则分布键默认会变成 null，数据都被保存在一个节点上，造成数据分布不均。

insert 可以批量操作，语法如下：

```
testDB=# insert into test001 values(100,'tom'),(101,'lily'),(102,'jack');
INSERT 0 3
```

2 ) update：不能批量对分布键执行 update，因为对分布键执行 update 需要将数据重分布，而 Greenplum 暂时不支持这个功能。

```

testDB=# \d test001
      Table "public.test001"
      Column | Type      | Modifiers
-----+-----+-----+
      id   | integer   |
     name  | text      |
Indexes:
  "test001_idx" btree (id)
Distributed by: (id)

testDB=# update test001 set id=5 where id=6;
ERROR:  Cannot parallelize an UPDATE statement that updates the distribution
columns

```

3) delete：在 Greenplum 3.x 的版本中，如果 Delete 操作涉及子查询，并且子查询的结果还会涉及数据重分布，这样的删除语句会报错，如下所示（在 Greenplum 4.x 中支持该操作）：

```

testDB=# delete from test001 where name in (select name from test002);
ERROR:  Cannot parallelize that DELETE yet
DETAIL:  Passage of data from one segment to another is not yet supported
during DELETE operations.
HINT:  The WHERE condition must specify equality between corresponding
DISTRIBUTED BY columns of the target table and all joined tables.

```

如果对整张表执行 Delete 会比较慢，建议使用 TRUNCATE。

## 7. TRUNCATE

与在 Oracle 中一样，执行 TRUNCATE 直接删除表的物理文件，然后创建新的数据文件。TRUNCATE 操作比 Delete 操作在性能上有非常大的提升，当前如果有 SQL 正在操作这张表，那么 TRUNCATE 操作会被锁住，直到表上面的所有锁被释放。

```

testDB=# truncate test001;
TRUNCATE TABLE

```

### 2.3.4 常用数据类型

Greenplum 的数据类型基本跟 PostgreSQL 的一样，类型十分丰富，下面介绍几种最常见的数据类型。对于 Greenplum 支持的其他数据类型，读者可以参考 PostgreSQL 文档。

#### 1. 数值类型

Greenplum 支持的数值类型数据如表 2-2 所示。

表 2-2 Greenplum 支持的数据类型——数值类型

类型名称	存储空间	描述	范围
smallint	2 字节	小范围整数	-32 768 ~ +32 767
integer	4 字节	常用的整数	-2 147 483 648 ~ +2 147 483 647
bigint	8 字节	大范围的整数	-9 223 372 036 854 775 808 ~ 9 223 372 036 854 775 807
decimal	变长	用户声明精度, 精确	无限制
numeric	变长	用户声明精度, 精确	无限制
real	4 字节	变精度, 不精确	6 位十进制数字精度
double precision	8 字节	变精度, 不精确	15 位十进制数字精度
serial	4 字节	自增整数	1 ~ 2 147 483 647
bigserial	8 字节	大范围的自增整数	1 ~ 9 223 372 036 854 775 807

## 2. 字符类型

Greenplum 支持的字符类型数据如表 2-3 所示。

表 2-3 Greenplum 支持的数据类型——字符类型

类型名称	描述
character varying(n), varchar(n)	变长, 有长度限制
character(n), char(n)	定长, 不足补空白
text	变长, 无长度限制

## 3. 时间类型

Greenplum 支持的时间类型数据如表 2-4 所示。

表 2-4 Greenplum 支持的数据类型——时间类型

类型名称	存储空间	描述	最低值	最高值	时间精度
timestamp [ (p) ] [ without time zone ]	8 字节	日期和时间	4713 BC	5874897 AD	1 毫秒
timestamp [ (p) ] with time zone	8 字节	日期和时间, 带时区	4713 BC	5874897 AD	1 毫秒
interval [ (p) ]	12 字节	时间间隔	-178 000 000 年	178 000 000 年	1 毫秒
date	4 字节	只用于表示日期	4713 BC	5 874 897 AD	1 天
time [ (p) ] [ without time zone ]	8 字节	只用于表示一日内时间	0:00:00	24:00:00	1 毫秒
time [ (p) ] with time zone	12 字节	只用于表示一日内时间, 带时区	00:00:00+1459	24:00:00-1459	1 毫秒

## 2.3.5 常用函数

### 1. 字符串函数

Greenplum 中的字符串函数如表 2-5 所示。

表 2-5 Greenplum 中的常用函数——字符串函数

函 数	返回类型	描 述	例 子	结 果
string    string	text	字符串连接	'Post'    'greSQL'	PostgreSQL
length(string)	int	string 中字符的数目	length('jose')	4
position(substring in string)	int	指定的子字符串的位置	position('om' in 'Thomas')	3
substring(string [from int] [for int])	text	抽取子字符串	substring('Thomas' from 2 for 3)	hom
trim([leading   trailing   both] [characters] from string)	text	从字符串 string 的开头/结尾/两边删除只包含 characters 中字符（默认是一个空白）的最长的字符串	trim(both 'x' from 'xTomxx')	Tom
lower(string)	text	把字符串转化为小写	lower('TOM')	tom
upper(string)	text	把字符串转化为大写	upper('tom')	TOM
overlay(string placing string from int [for int])	text	替换子字符串	overlay('Txxxxas' placing 'hom' from 2 for 4)	Thomas
replace(string text, from text, to text)	text	把字符串 string 中出现的所有子字符串 from 替换成子字符串 to	replace('abcdefabcdef', 'cd', 'XX')	abXXefabXXef
split_part(string text, delimiter text, field int)	text	根据 delimiter 分隔 string 返回生成的第 field 个子字符串 (1 为基)	split_part('abc defghi', ' ', 2)	def

下面是一个字符串拼接的示例：

```
testDB=# select 'green'||'plum' as dbname;
      dbname
-----
      greenplum
(1 row)
```

下面以 | 为分隔符，将字符串分割：

```
testDB=# select split_part(col,'|',1)
testDB-#           ,split_part(col,'|',2)
testDB-#   from (values ('hello|wolrd!'), ('greenplum|database'))
testDB-#           t(col) ;
      split_part | split_part
-----+-----
```

```
hello      | wolrd!
greenplum | database
(2 rows)
```

其中 Values 是 Greenplum 特有的语法，在这里可以将其看成一张表，表中有两行数据，表名为 t，字段名为 col，Values 的用法如下：

```
testDB=# values ('hello|wolrd!'), ('greenplum|database');
          column1
-----
hello|wolrd!
greenplum|database
(2 rows)
```

获取字符串的第 2 个字符之后的 3 个字符：

```
testDB=# select substr('hello world!',2,3);
          substr
-----
ell
(1 row)
```

获取子串在字符串中的位置：

```
testDB=# select position('world' in 'hello world!');
          position
-----
7
(1 row)
```

## 2. 时间函数

本节内容可参考 PostgreSQL8.2 文档“9.9. 时间 / 日期函数和操作符”。Greenplum 支持的时间函数如表 2-6 所示。

表 2-6 Greenplum 中的常用函数——时间函数

函 数	返 回 类 型	描 述	例 子	结 果
age(timestamp, timestamp)	interval	减去参数后的“符号化”结果	age(timestamp '2001-04-10', timestamp '1957-06-13')	43 years 9 mons 27 days
age(timestamp)	interval	从 current_date 减去参数后的结果	age(timestamp '1957-06-13')	43 years 8 mons 3 days
current_date	date	当前的日期		
current_time	time with time zone	当日时间		
current_timestamp	timestamp with time zone	当前事务开始时的时间戳		

(续)

函 数	返 回 类 型	描 述	例 子	结 果
<code>date_part(text, timestamp)</code>	double precision	获取子域(等效于 extract)	<code>date_part('hour', timestamp '2001-02-16 20:38:40')</code>	20
<code>date_trunc(text, timestamp)</code>	timestamp	截断成指定的精度	<code>date_trunc('hour', timestamp '2001-02-16 20:38:40')</code>	2001/2/16 20:00
<code>extract(field from timestamp)</code>	double precision	获取子域	<code>extract(hour from timestamp '2001-02-16 20:38:40')</code>	20
<code>now()</code>	timestamp with time zone	当前事务开始时的时间戳		

**时间加减：**

```
testDB=# select '2011-10-01 10:0:0'::timestamp + interval '10 days 2 hours 10 seconds';
?column?
-----
2011-10-11 12:00:10
(1 row)
```

Interval 是一种表示时间间隔的一种数据类型。利用 interval 这种数据类型可以实现时间的加减，两个时间的时间差就是一个 Interval 类型。

**获取当前时间：**

```
testDB=# select now(), current_date, current_time , current_timestamp;
now           |   date    |   timetz   |      now
-----+-----+-----+-----+
2012-01-15 16:04:52.15361+08 | 2012-01-15 | 16:04:52.15361+08 | 2012-01-15
16:04:52.15361+08
(1 row)
```

**获取当月的第一天：**

```
aligputf8=# select date_trunc('months',now())::date;
date_trunc
-----
2012-01-01
(1 row)
aligputf8=# select date_trunc('months',now())::date;
date_trunc
-----
2012-01-01
(1 row)
```

获取当前时间距离 2011-10-10 10:10:10 过了多少秒:

```
testDB=# SELECT EXTRACT(EPOCH FROM now() - '2011-10-10 10:10:10');
          date_part
-----
 8403301.725044
(1 row)
```

除了这些函数以外, Greenplum 还支持 SQL 的 OVERLAPS 操作符:

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

这个表达式在两个时间域 (用它们的终点定义) 重叠的时候生成真值。终点可以用一对日期、时间或时间戳来声明, 或者在后面跟一个时间间隔的日期、时间或时间戳。

```
testDB=# SELECT (DATE '2011-02-16', DATE '2011-12-21') OVERLAPS
testDB-#           (DATE '2011-10-30', DATE '2012-1-15');
      overlaps
-----
 t
(1 row)
testDB=# SELECT (DATE '2011-02-16', INTERVAL '100 days') OVERLAPS
testDB-#           (DATE '2011-10-30', DATE '2012-10-30');
      overlaps
-----
 f
(1 row)
```

### 3. 数值计算函数

Greenplum 中的数值计算函数如表 2-7 所示。

表 2-7 Greenplum 中的常用函数——数值计算函数

函 数	返 回 类 型	描 述	例 子	结 果
abs(x)	(与 x 相同)	绝对值	abs(-17.4)	17.4
ceil(dp 或 numeric)	(与输入相同)	不小于参数的最小的整数	ceil(-42.8)	-42
ceiling(dp 或 numeric)	(与输入相同)	不小于参数的最小整数 (ceil 的别名)	ceiling(-95.3)	-95
exp(dp 或 numeric)	(与输入相同)	自然指数	exp(1.0)	2.718 281 828
ln(dp 或 numeric)	(与输入相同)	自然对数	ln(2.0)	0.693 147 181
log(dp 或 numeric)	(与输入相同)	以 10 为底的对数	log(100.0)	2
log(b numeric, x numeric)	numeric	以 b 为底数的对数	log(2.0, 64.0)	6
mod(y, x)	(与参数类型相同)	y/x 的余数 (模)	mod(9,4)	1
pi()	dp	"π" 常量	pi()	3.141 592 654
power(a numeric, b numeric)	numeric	a 的 b 次幂	power(9.0, 3.0)	729

(续)

函数	返回类型	描述	例子	结果
radians(dp)	dp	把角度转为弧度	radians(45.0)	0.785 398 163
random()	dp	0.0 ~ 1.0 之间的随机数	random()	
floor(dp 或 numeric)	(与输入相同)	不大于参数的最大整数	floor(-42.8)	-43
round(v numeric, s int)	numeric	圆整为 s 位小数	round(42.4382, 2)	42.44
sign(dp 或 numeric)	(与输入相同)	参数的符号 (-1, 0, +1)	sign(-8.4)	-1
sqrt(dp 或 numeric)	(与输入相同)	平方根	sqrt(2.0)	1.414 213 562
cbrt(dp)	dp	立方根	cbrt(27.0)	3
trunc(v numeric, s int)	numeric	截断为 s 位小数	trunc(42.4382, 2)	42.43

## 4. 其他常用函数

### (1) 序列号生成函数——generate\_series

该函数生成多行数据，从一个数字 (start) 到另外一个数字 (end) 按照一定的间隔，默认是 1，生成一个结果集，具体的使用方法如下：

```
testDB=# select * from generate_series(6,10);
generate_series
-----
6
7
8
9
10
(5 rows)
```

我们可以很方便地使用这个函数来创建一些测试表的数据：

```
testDB=# create table test_gen as select generate_series(1,10000) as id,'hello'::text as name distributed by (id);
SELECT 10000
```

我们还可以用 generate\_series 来做一些数值计算，比如，计算 1 ~ 2000 之间所有的奇数之和：

```
testDB=# select sum(num) from generate_series(1,2000,2) num;
sum
-----
1000000
(1 row)
```

### (2) 字符串列转行函数——string\_agg

有时候我们需要将一个列的字符串按照某个分割符将其拼接起来：

```
testDB=# select * from test_string;
```

```

id | str
---+-----
1 | hello
1 | world
2 | greenplum
2 | database
2 | system
(5 rows)

```

要按照 id 字段将字符串拼接起来，可以像下面这样使用 `string_agg` 来实现。

```

testDB=# select id,string_agg(str,'|') from test_string group by id;
id | string_agg
---+-----
2 | greenplum|database|system
1 | hello|world
(2 rows)

```

我们还可以先按照某一个字段做排序，再做拼接：

```

testDB=# select id,string_agg(str,'|' order by str) from test_string group by id;
id | string_agg
---+-----
2 | database|greenplum|system
1 | hello|world
(2 rows)

```



这个函数是 Greenplum 4.0 之后加入的新功能，要在 Greenplum 3.x 中实现同样的功能，需要自己自定义一个聚合函数，第 6 章将介绍如何编写自定义函数。

### (3) 字符串行转列——`regexp_split_to_table`

上面介绍进行字符串拼接的函数，那么反过来怎么把拼接好的字符串重新拆分，变成多个字段呢？

我们可以使用 `regexp_split_to_table` 函数来实现这个功能。这次对上面例子的结果进行操作。

```

testDB=# select * from test_string2;
id | str
---+-----
1 | hello|world
2 | database|greenplum|system
(2 rows)

testDB=# select id,regexp_split_to_table(str,E'\\|') str from test_string2;
id | str
---+-----
1 | hello

```

```

1 | world
2 | database
2 | greenplum
2 | system
(5 rows)

```

由于|字符被转义了，因此想正确表达，必须对其进行转义。

#### (4) hash 函数——md5, hashbpchar

Greenplum 中内置了很多 hash 函数，下面以其中两个为例进行介绍，其他的大同小异，只是按照不同的数据类型来执行 hash 函数。

md5 的 hash 算法的精确度是 128 位，返回值是一个字符串。

```

testDB=# select md5('helloworld');
          md5
-----
fc5e038d38a57032085441e7fe7010b0
(1 row)

```

Hashbpchar 的精确度是 32 位的，返回值是一个 integer 类型。

```

testDB=# select hashbpchar('helloworld');
      hashbpchar
-----
252807993
(1 row)

```

## 2.3.6 分析函数

学习过 Oracle 的读者应该知道，Oracle 中的分析函数功能十分强大，Greenplum 中也内置了这些函数，对于数据仓库应用来说，在对大量数据进行分析的时候，这些函数可以为实现一些复杂逻辑节省很多的时间。

### 1. 开窗函数

使用数据库进行数据分析的时候经常用聚合函数来做一些不同维度的统计分析，但是聚合函数统计出的是汇总后的结果，没有明细数据，如果既要统计结果又要明细数据，那么用普通的 SQL 将会很复杂。如果利用开窗函数，这一切就会变得很简单。简言之，聚合函数返回各个分组的结果，开窗函数则为每一行返回结果。下面将结合 PostgreSQL8.4 英文文档中的例子来简单介绍下开窗函数的使用。

原表的数据为：

```

testDB=# select * from empsalary order by depname;
   depname  |  empno  |  salary
-----+-----+-----+
  develop  |      9 |    4500

```

```

develop | 11 | 5200
develop | 7 | 4200
develop | 10 | 5200
develop | 8 | 6000
personnel | 5 | 3500
personnel | 2 | 3900
sales | 3 | 4800
sales | 1 | 5000
sales | 4 | 4800
(10 rows)

```

每个部门的人，在部门内工资的排名：

```

testDB=# SELECT depname, empno, salary
testDB-#      ,rank() OVER (PARTITION BY depname ORDER BY salary DESC)
testDB-#      ,row_number() OVER (PARTITION BY depname ORDER BY salary DESC)
testDB-# FROM empsalary;
   depname | empno | salary | rank | row_number
-----+-----+-----+-----+
personnel | 2 | 3900 | 1 | 1
personnel | 5 | 3500 | 2 | 2
develop | 8 | 6000 | 1 | 1
develop | 11 | 5200 | 2 | 2
develop | 10 | 5200 | 2 | 3
develop | 9 | 4500 | 4 | 4
develop | 7 | 4200 | 5 | 5
sales | 1 | 5000 | 1 | 1
sales | 3 | 4800 | 2 | 2
sales | 4 | 4800 | 2 | 3
(10 rows)

```

执行 rank 函数，同样工资的排名会是一样，执行 row\_number 函数，则同一个数值，都会有先后顺序的，比如 develop 部门的 empno 为 10、11 的两个员工的工资都是 5200 元，使用 rank，则排名都是 2，如果使用 row\_number，则 10 的排名是 2,11 的排名是 3。

在执行 sum、count 和 avg 这一类的聚集函数时，加不加 order by 是不相同的。不加 order by 所有的结果都是一样的，都是根据 partition by 的字段将所有的值聚合，加了 order by 则是根据排序的字段递增的。例如下面这个 SQL：

```

testDB=# SELECT *
testDB-#      ,sum(salary) OVER () sum1
testDB-#      ,sum(salary) OVER (ORDER BY salary) sum2
testDB-#      ,sum(salary) OVER (partition by depname) sum3
testDB-#      ,sum(salary) OVER (partition by depname order by salary) sum4
testDB-# FROM empsalary;
   depname | empno | salary | sum1 | sum2 | sum3 | sum4
-----+-----+-----+-----+-----+-----+
personnel | 5 | 3500 | 47100 | 3500 | 7400 | 3500
personnel | 2 | 3900 | 47100 | 7400 | 7400 | 7400
develop | 7 | 4200 | 47100 | 11600 | 25100 | 4200

```

```

develop | 9 | 4500 | 47100 | 16100 | 25100 | 8700
sales  | 3 | 4800 | 47100 | 25700 | 14600 | 9600
sales  | 4 | 4800 | 47100 | 25700 | 14600 | 9600
sales  | 1 | 5000 | 47100 | 30700 | 14600 | 14600
develop | 10 | 5200 | 47100 | 41100 | 25100 | 19100
develop | 11 | 5200 | 47100 | 41100 | 25100 | 19100
develop | 8 | 6000 | 47100 | 47100 | 25100 | 25100
(10 rows)

```

字段 sum1 是所有雇员工资总和的大小，sum2 则是全局根据工资排序后的递增结果，sum3 是每个部门内雇员工资的总和，sum4 是部门内按照工资排序后的递增结果。

## 2. grouping sets

如果需要对几个字段的组合进行 group by，就需要用到 Grouping Sets 的功能了。为了方便读者理解，下面介绍一些等价的定义，如表 2-8 所示。

表 2-8 grouping sets 的等价定义

group sets 语法	等价的普通 SQL 的语法
SELECT C1, C2, SUM(C3) FROM T GROUP BY GROUPING SETS ((C1), (C2))	SELECT C1, NULL as C2, SUM(C3) FROM T GROUP BY T UNION ALL SELECT NULL as C1, C2, SUM(C3) FROM T GROUP BY year
GROUP BY GROUPING SETS ((C1, C2, ..., Cn))	GROUP BY C1, C2, ..., Cn
GROUP BY ROLLUP (C1, C2, ..., Cn-1, Cn)	GROUP BY GROUPING SETS ((C1, C2, ..., Cn-1, Cn) ,(C1, C2, ..., Cn-1) ... ,(C1, C2) ,(C1) ,()
GROUP BY CUBE (C1, C2, C3)	GROUP BY GROUPING SETS ((C1, C2, C3) ,(C1, C2) ,(C1, C3) ,(C2, C3) ,(C1) ,(C2) ,(C3) ,()

假设我们要统计大部门的人数，也要统计大部门中每个小组的人数，如下：

```

testDB=# select depname, depname2, count(1)
testDB-#   from empsalary2

```

```

testDB-# group by grouping sets(depname, (depname, depname2))
testDB-# order by depname2;
  depname | depname2 | count
-----+-----+-----
  develop | dev1      |    2
  develop | dev2      |    3
personnel | perl      |    1
personnel | per2      |    1
  sales   | sal1      |    2
  sales   | sal2      |    1
  develop |          |    5
  sales   |          |    3
personnel |          |    2
(9 rows)

```

### 2.3.7 分区表

Greenplum 支持分区表，具体的实现原理可查看“第 4 章数据字典详解”。

在创建表时，关于 `partition` 的语法如下：

```

[ PARTITION BY partition_type (column)
  [ SUBPARTITION BY partition_type (column) ]
  [ SUBPARTITION TEMPLATE ( template_spec ) ]
  [...]
  ( partition_spec )
  | [ SUBPARTITION BY partition_type (column) ]
  [...]
  ( partition_spec
    [ ( subpartition_spec
        [...]
      ) ]
and partition_element is:

  DEFAULT PARTITION name
  | [PARTITION name] VALUES (list_value [,...] )
  | [PARTITION name]
    START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
    [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
    [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
  | [PARTITION name]
    END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
    [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
  [ WITH ( partition_storage_parameter=value [, ... ] ) ]
  [ TABLESPACE tablespace ]

```

按照时间分区，创建 20111230 ~ 20120104 号的分区：

```

create table public.test_partition_range(
  id          numeric

```

```

, name           character varying(32)
, dw_end_date   date
)Distributed by (id)
PARTITION BY range(dw_end_date)
(
    PARTITION p20111230 START ('2011-12-30'::date) END ('2011-12-31'::date),
    PARTITION p20111231 START ('2011-12-31'::date) END ('2012-01-01'::date),
    PARTITION p20120101 START ('2012-01-01'::date) END ('2012-01-02'::date),
    PARTITION p20120102 START ('2012-01-02'::date) END ('2012-01-03'::date),
    PARTITION p20120103 START ('2012-01-03'::date) END ('2012-01-04'::date),
    PARTITION p20120104 START ('2012-01-04'::date) END ('2012-01-05'::date)
);

```

使用 Every, 创建 20111201 ~ 20111231 的分区:

```

create table public.test_partition_every(
    id             numeric
    ,name          character varying(32)
    ,dw_end_date   date
)Distributed by (id)
PARTITION BY range(dw_end_date)
(
    PARTITION p201112 START ('2011-12-1'::date) END ('2011-12-31'::date)
every ('1 days'::interval)
);

```

创建 list 分区:

```

create table public.test_partition_list(
    member_id      numeric
    ,city          character varying(32)
)Distributed by (member_id)
PARTITION BY list(city)
(
    partition guangzhou values('guangzhou'),
    partition hangzhou values('hangzhou'),
    partition shanghai values('shanghai'),
    partition beijing values('beijing'),
    DEFAULT PARTITION other_city
);

```

下面是 alter table 对分区表特有一些操作的语法:

```

where partition_action is one of:
    ALTER DEFAULT PARTITION
    DROP DEFAULT PARTITION [IF EXISTS]
    DROP PARTITION [IF EXISTS] { partition_name |
        FOR (RANK(number)) | FOR (value) } [CASCADE]
    TRUNCATE DEFAULT PARTITION
    TRUNCATE PARTITION { partition_name | FOR (RANK(number)) |

```

```

        FOR (value) }
RENAME DEFAULT PARTITION TO new_partition_name
RENAME PARTITION { partition_name | FOR (RANK(number)) |
        FOR (value) } TO new_partition_name
ADD DEFAULT PARTITION name [ ( subpartition_spec ) ]
ADD PARTITION [name] partition_element
[ ( subpartition_spec ) ]
EXCHANGE PARTITION { partition_name | FOR (RANK(number)) |
        FOR (value) } WITH TABLE table_name
[ WITH | WITHOUT VALIDATION ]
EXCHANGE DEFAULT PARTITION WITH TABLE table_name
[ WITH | WITHOUT VALIDATION ]
SET SUBPARTITION TEMPLATE (subpartition_spec)
SPLIT DEFAULT PARTITION
{ AT (list_value)
| START([datatype] range_value) [INCLUSIVE | EXCLUSIVE]
END([datatype] range_value) [INCLUSIVE | EXCLUSIVE] }
[ INTO ( PARTITION new_partition_name,
        PARTITION default_partition_name ) ]
SPLIT PARTITION { partition_name | FOR (RANK(number)) |
        FOR (value) } AT (value)
[ INTO (PARTITION partition_name, PARTITION partition_name) ]

```

接下来介绍对分区表进行的一些常用操作。

### (1) 新增分区

```

testDB=# alter table public. test_partition_every add partition p20120105_6
START ('2012-01-05'::date) END ('2012-01-07'::date);
NOTICE:  CREATE TABLE will create partition "test_partition_1_1_prt_
p20120105_6" for table "test_partition_1"
ALTER TABLE

```

### (2) drop/truncate 分区

删除 p20120104 分区：

```
alter table public. test_partition_every drop partition p20120105_6;
```

truncate p20120103 分区：

```
alter table public. test_partition_every truncate partition p20120105_6;
```

### (3) 拆分分区

```
alter table public. test_partition_every split partition p20120105_6
at('2012-01-06'::date) into (PARTITION p20120105,PARTITION p20120106);
```

### (4) 交换分区

```
alter table public. test_partition_every exchange partition p20120102 with
table public.test_one_partition;
```

### 2.3.8 外部表

Greenplum 在数据加载上有一个明显的优势，就是支持数据并发加载，gpfldist 就是并发加载的工具，在数据库中对应的就是外部表。

gpfldist 的实现架构图如图 2-20 所示。

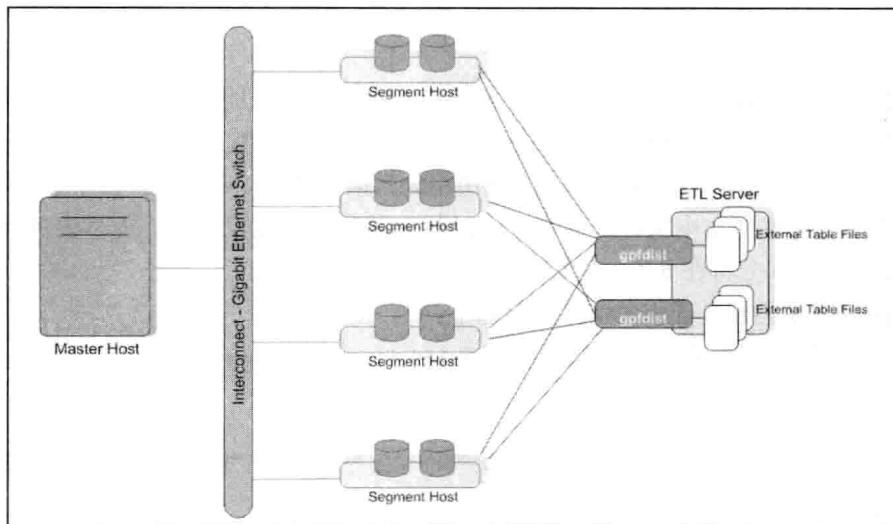


图 2-20 Greenplum 外部表 (gpfldist) 的实现架构图

外部表，顾名思义就是一张表的数据是指向数据库之外的数据文件的。在 Greenplum 中，我们可以对一个外部表执行正常的 DML 操作，当读取数据的时候，数据库就从数据文件中加载数据。外部表支持在 Segment 上并发地高速从 gpfldist 导入数据，由于是直接从 Segment 上导入数据，所以效率非常的高。

创建外部表语法：

```

CREATE [READABLE] EXTERNAL TABLE table_name
  ( column_name data_type [, ...] | LIKE other_table )
  LOCATION ('file://seghost[:port]/path/file' [, ...])
  | ('gpfldist://filehost[:port]/file_pattern' [, ...])
  | ('gphdfs://hdfs_host[:port]/path/file')
  FORMAT 'TEXT'
    [( [HEADER]
      [DELIMITER [AS] 'delimiter' | 'OFF']
      [NULL [AS] 'null string']
      [ESCAPE [AS] 'escape' | 'OFF']
      [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
      [FILL MISSING FIELDS] )
    | 'CSV'
    [( [HEADER]
      [QUOTE [AS] 'quote']

```

```
[DELIMITER [AS] 'delimiter']
[NULL [AS] 'null string']
[FORCE NOT NULL column [, ...]]
[ESCAPE [AS] 'escape']
[NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
[FILL MISSING FIELDS] )
[ ENCODING 'encoding' ]
[ [LOG ERRORS INTO error_table] SEGMENT REJECT LIMIT count
[ROWS | PERCENT] ]
```

外部表需要指定 gpfdist 的 IP 和端口，还要有详细的目录地址，其中文件名支持通配符匹配。可以编写多个 gpfdist 的地址，但是不能超过总的 Segment 数，否则会报错。在创建外部表的时候可以指定分隔符、err 表、指定允许出错的数据条数，以及源文件的字符编码等信息。

外部表还支持本地文本文件的导入，不过效率较低，不建议使用。外部表还支持 HDFS 的文件操作，这一部分将在 6.3.3 节介绍。

启动 gpfdist 及创建外部表的实际步骤如下：

- 1) 首先在文件服务器（这里假设是 10.20.151.11）上启动 gpfdist 的服务，指定文件目录及端口。

```
nohup $GPHOME/bin/gpfdist -d /home/admin -p 8888 > /tmp/gpfdist.log 2>&1 &
```

启动 gpfdist 后，在 log 中可以看到：

```
Serving HTTP on port 8888, directory /home/admin
```

说明程序已经成功启动了，端口是 8888，这个服务只需要启动一次以后就不用启动了。  
nohup 保证程序在 Server 端执行，当前会话关闭后，程序仍然正常运行。



**提示** nohup 是 UNIX/Linux 中的一个命令，普通进程通过 & 符号放到后台运行，如果启动该程序的控制台退出，则该进程随即终止。nohup 命令启动程序，则在控制台退出后，进程仍然继续运行，起到守护进程的作用。

- 2) 准备好需要加载的数据文件，将其放在 10.20.151.11 机器的 /home/admin/ 目录或该目录的子目录下，在 Greenplum 中创建对应的外部表：

```
create external table public.test001_ext(
id integer,
name varchar(128)
)
Location (
'gpfdist://10.20.151.11:8888/gpextdata/test001.txt'
)
Format 'TEXT' (delimiter as E'|' null as '' escape 'OFF')
Encoding 'GB18030' Log errors into public.test001_err segment reject limit 10 rows;
```

### 3 ) 外部表查询及数据加载:

```
testDB=# select * from public.test001_ext;
 id | name
----+-----
 100 | jack
 102 | tom
 103 | lily
 101 | david
(4 rows)

testDB=# insert into test001 select * from test001_ext;
INSERT 0 4
Time: 345.514 ms
```

### 4 ) 如果加载报错, 报错的数据会被插入到 err 表中, 并显示报错的详细信息:

```
testDB=# select * from public.test001_err;
-[ RECORD 1 ]-----
cmdtime | 2012-01-11 23:42:35.242877+08
relname | test001_ext
filename | gpfdist://10.20.151.11:8888/gpextdata/test001.txt [/home/admin/
gpextdata/test001.txt]
linenum | 5
byt enum |
errmsg | extra data after last expected column
rawdata | sdfdsf|sdfdsfdsf|sfd
rawbytes |
```

## 2.3.9 COPY 命令

使用 COPY 命令可以实现将文件导出和导入, 只不过要通过 Master, 效率没有外部表高, 但是在数据量比较小的情况下, COPY 命令比外部表要方便很多。

使用 COPY 命令的语法如下:

```
Command:      COPY
Description:  copy data between a file and a table
Syntax:
COPY table [(column [, ...])] FROM {'file' | STDIN}
  [ [WITH]
    [OIDS]
    [HEADER]
    [DELIMITER [ AS ] 'delimiter']
    [NULL [ AS ] 'null string']
    [ESCAPE [ AS ] 'escape' | 'OFF']
    [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
    [CSV [QUOTE [ AS ] 'quote']
     [FORCE NOT NULL column [, ...]]
    [FILL MISSING FIELDS]
```

```

[ [LOG ERRORS INTO error_table] [KEEP]
  SEGMENT REJECT LIMIT count [ROWS | PERCENT] ]

COPY {table [(column [, ...])] | (query)} TO {'file' | STDOUT}
  [ [WITH]
    [OIDS]
    [HEADER]
    [DELIMITER [ AS ] 'delimiter']
    [NULL [ AS ] 'null string']
    [ESCAPE [ AS ] 'escape' | 'OFF']
    [CSV [QUOTE [ AS ] 'quote']
     [FORCE QUOTE column [, ...]] ]

```

Greenplum 4.x 中引入了可写外部表，在导出数据的时候可以用可写外部表并发导出，性能很好，但是在 Greenplum 3.x 版本中，导出数据只能通过 COPY 命令实现，数据在 Master 上汇总导出。

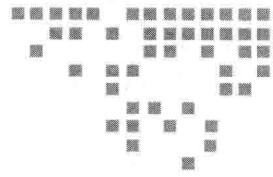
如果需要将数据远程导出到其他机器上，可以使用 copy to stdout，远程执行 psql 连接到数据库上，然后通过管道将数据重定向成文件。

## 2.4 小结

本章主要介绍了 Greenplum 的安装和部署，以及一些简单的基本的数据库操作及使用注意点，力求使读者可以快速了解 Greenplum 的特性，迅速上手，后续的章节会详细介绍 Greenplum 的一些高级特性以及管理和优化。

与普通的 PostgreSQL 数据库的最大不同就是，Greenplum 是分布式数据库，所有的数据都切分在 Segment 上，在使用过程中要时刻注意这一点。

Greenplum 的函数与 PostgreSQL 的函数大部分是一样的，在 PostgreSQL 的文档中对函数有很多的解释和说明，强烈建议读者对照 PostgreSQL 的文档来学习 Greenplum，这样必将事半功倍。



第3章

*Chapter 3*

## Greenplum 实战

从本章开始我们结合实际需求，阐述一下日常项目开发中如何结合 Greenplum 的特性进行高效的开发，展现出 Greenplum 在海量数据分析中的优势。

本章将介绍两个完整的例子：数据仓库拉链记历史和网页浏览日志分析。在这两个例子中，会结合 Greenplum 的一些特性加以描述，之后会介绍使用 Greenplum 中要注意的一些特性，以及这些特性对性能的影响。

### 3.1 历史拉链表

数据仓库是一个面向主题的、集成的、相对稳定的、反映历史变化的数据集合，用于支持管理决策。由于需要反映历史变化，数据仓库中的数据通常包含历史信息，系统记录了企业从过去某一时点（如开始应用数据仓库的时点）到目前的各个阶段的信息，通过这些信息，可以对企业的发展历程和未来趋势做出定量分析和预测。

历史拉链表是一种数据模型，主要是针对数据仓库设计中表存储数据的方式而定义的。顾名思义，所谓历史拉链表，就是记录一个事物从开始一直到当前状态的所有变化的信息。拉链表可以避免按每一天存储所有记录造成的海量存储问题，同时也是处理缓慢变化数据的一种常见方式。

#### 3.1.1 应用场景描述

现假设有如下场景：一个企业拥有 5000 万会员信息，每天有 20 万会员资料变更，我们需要在 Greenplum 中记录会员表的历史变化以备数据挖掘分析等使用，即每天都要保留一个

快照供查询，反映历史数据的情况。在此场景中，需要反映 5000 万会员的历史变化，如果保留快照，存储两年就需要  $2 \times 365 \times 5000W$  条数据存储空间，数据量为 365 亿，如果存储更长时间，则无法估计需要的存储空间。而用拉链算法存储，每日只向历史表中添加新增和变化的数据量，每日不过 20 万条，存储 4 年也只需要 3 亿存储空间。

接下来我们将概要讲述整个分析实施过程。

### 3.1.2 原理及步骤

在拉链表中，每一条数据都有一个生效日期（dw\_beg\_date）和失效日期（dw\_end\_date）。假设在一个用户表中，在 2011 年 12 月 1 日新增了两个用户，如表 3-1 所示，则这两条记录的生效时间为当天，由于到 2011 年 12 月 1 日为止，这两条记录还没有被修改过，所以失效时间为无穷大，这里设置为数据库中的最大值（3000-12-31）。

表 3-1 2011-12-01 新增用户

用户名	电话号码	生效时间	失效时间
10001	13500000001	2011-12-01	3000-12-31
10002	13500000002	2011-12-01	3000-12-31

第二天（2011-12-02），用户 10001 被删除了，用户 10002 的电话号码被修改成 13600000002。为了保留历史状态，用户 10001 的失效时间被修改为 2011-12-02，用户 10002 则变成两条记录，如表 3-2 所示。

表 3-2 2011-12-2 用户表

用户名	电话号码	生效时间	失效时间
10001	13500000001	2011-12-01	2011-12-02
10002	13500000002	2011-12-01	2011-12-02
10002	13600000002	2011-12-02	3000-12-31

第三天（2011-12-03），又新增了用户 10003，则用户表数据如表 3-3 所示。

表 3-3 2011-12-3 用户表

用户名	电话号码	生效时间	失效时间
10001	13500000001	2011-12-01	2011-12-02
10002	13500000002	2011-12-01	2011-12-02
10002	13600000002	2011-12-02	3000-12-31
10003	13500000003	2011-12-03	3000-12-31

如果要查询最新的数据，那么只要查询失效时间为 3000-12-31 的数据即可，如果要查询 12 月 1 号的历史数据，则筛选生效时间  $\leq 2011-12-01$  并且失效时间  $> 2011-12-01$  的数据即可。如果查询的是 12 月 2 号的数据，那么筛选条件则是生效时间  $\leq 2011-12-02$  并且失效时

间 >2011-12-02。读者可对表 3-3 的数据进行筛选，以检验结果是否正确。

在 Greenplum 中，则可以利用分区表按照 dw\_end\_date 保存时间，这样在查询的时候可以利用 Greenplum 的分区裁剪，从而减少 IO 消耗。下面通过图 3-1 讲解拉链表刷新的步骤，连线代表数据流向，线上的编号就是步骤编号。

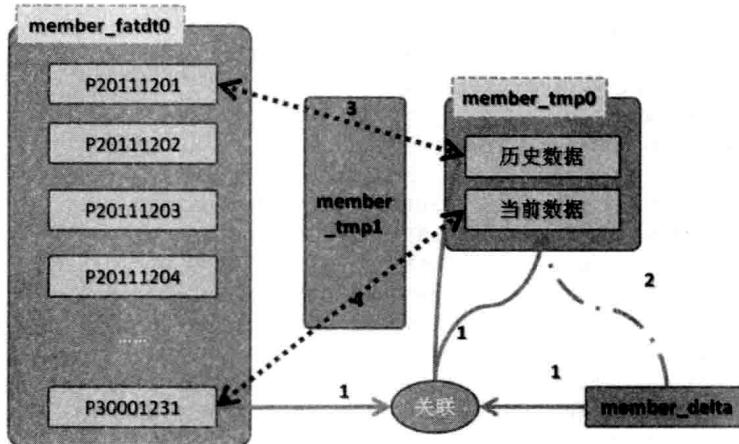


图 3-1 拉链表刷新过程

首先介绍每个表的用途。

- **member\_fatdt0**: 表示 member 的事实表，其中 P30001231 保存的是最新数据，每个分区保留的都是历史已失效的数据。
- **member\_delta**: 当天的数据库变更数据，action 字段表示该数据为新增 (I)、更新 (U)、删除 (D)。
- **member\_tmpt0**: 刷新过程中的临时表，这个表有两个分区，分别记录历史数据，即当天失效数据，另一个分区记录的是当前数据。
- **member\_tmpt1**: 同样是刷新过程中的临时表，主要是在交换分区的时候使用。

刷新过程简单来说，就是将前一天的全量数据（分区 P30001231）与当天的增量数据进行关联，并对不同的变更类型（action）进行相应的处理，最终生成最新数据，以及当天发生变更的历史数据。后续数据刷新实战中会介绍具体的步骤，下面先从表结构开始介绍。

### 3.1.3 表结构

#### 1. 拉链表（member\_fatdt0）结构

member\_fatdt0 使用 member\_id 作为分布键，使数据尽量打散在每个机器上（参考 3.3 节数据分布），通过 with (appendonly=true,compresslevel=5) 指定该表为压缩表，可以减少 IO 操作（参考 3.4 节数据压缩），将 dw\_end\_date 作为分区字段。建表语句如下：

```

Create table public.member_fatdt0
(
    Member_id      varchar(64)      -- 会员 ID
    ,phoneno       varchar(20)       -- 电话号码
    ,dw_beg_date   date            -- 生效日期
    ,dw_end_date   date            -- 失效日期
    ,dtype          char(1)          -- 类型 (历史数据、当前数据)
    ,dw_status     char(1)          -- 数据操作类型 (I,D,U)
    ,dw_ins_date   date            -- 数据仓库插入日期
) with(appendonly=true,compresslevel=5)
distributed by(member_id)
PARTITION BY RANGE (dw_end_date)
(
    PARTITION p20111201 START (date '2011-12-01') INCLUSIVE ,
    PARTITION p20111202 START (date '2011-12-02') INCLUSIVE ,
    PARTITION p20111203 START (date '2011-12-03') INCLUSIVE ,
    PARTITION p20111204 START (date '2011-12-04') INCLUSIVE ,
    PARTITION p20111205 START (date '2011-12-05') INCLUSIVE ,
    PARTITION p20111206 START (date '2011-12-06') INCLUSIVE ,
    PARTITION p20111207 START (date '2011-12-07') INCLUSIVE ,
    PARTITION p30001231 START (date '3000-12-31') INCLUSIVE
END (date '3001-01-01') EXCLUSIVE
);

```

## 2. 增量表 (member\_delta) 结构

建表语句如下：

```

Create table public.member_delta
(
    Member_id      varchar(64)      -- 会员 ID
    ,phoneno       varchar(20)       -- 电话号码
    ,action         char(1)          -- 类型 (新增, 删除, 更新)
    ,dw_ins_date   date            -- 数据仓库插入日期
) with(appendonly=true,compresslevel=5)
distributed by(member_id);

```

## 3. 临时表 0 (member\_tmp0) 结构

dtype 为分区字段，H 表示历史数据，C 表示当前数据，建表语句如下：

```

Create table public.member_tmp0
(
    Member_id      varchar(64)      -- 会员 ID
    ,phoneno       varchar(20)       -- 电话号码
    ,dw_beg_date   date            -- 生效日期
    ,dw_end_date   date            -- 失效日期
    ,dtype          char(1)          -- 类型 (历史数据、当前数据)
    ,dw_status     char(1)          -- 数据操作类型 (I,D,U)
)

```

```

, dw_ins_date date          -- 数据仓库插入日期
) with(appendonly=true,compresslevel=5)
distributed by(member_id)
PARTITION BY LIST (dtype)
( PARTITION PHIS VALUES ('H'),
PARTITION PCUR VALUES ('C'),
DEFAULT PARTITION other );

```

#### 4. 临时表1 (member\_tmp1) 结构

表结构与 member\_tmp1、member\_fatdt0 一模一样，建表语句如下：

```

Create table public.member_tmp1
(
    Member_id      varchar(64)      -- 会员 ID
    ,phoneno       varchar(20)       -- 电话号码
    ,dw_beg_date   date            -- 生效日期
    ,dw_end_date   date            -- 失效日期
    ,dtype          char(1)          -- 类型 (历史数据、当前数据)
    ,dw_status     char(1)          -- 数据操作类型 (I, D, U)
    ,dw_ins_date   date            -- 数据仓库插入日期
) with(appendonly=true,compresslevel=5)
distributed by(member_id);

```

#### 3.1.4 Demo 数据准备

在这里为了清晰展示整个逻辑，仅以少量 demo 数据为例。

##### (1) 增量表数据

12月2号增量数据，新增、删除、更新各有一条记录，如表 3-4 所示。

表 3-4 12月2号增量数据

Member_id	phoneno	action	dw_ins_date
mem006	13100000006	I	2011-12-03
mem002	13100000002	D	2011-12-03
mem003	13800000003	U	2011-12-03

12月3号增量数据，新增、删除、更新各有一条记录，如表 3-5 所示。

表 3-5 12月3号增量数据

member_id	phoneno	action	dw_ins_date
mem007	13100000007	I	2011-12-04
mem004	13100000004	D	2011-12-04
mem005	13800000005	U	2011-12-04

## (2) 历史表初始数据

初始数据为 12 月 1 号，失效日期为 3000 年 12 月 31 号，如表 3-6 所示。

表 3-6 历史表初始数据

member_id	phoneno	dw_beg_date	dw_end_date	dtype	dw_status	dw_ins_date
mem001	13100000001	2011-12-01	3000/12/31	C	I	2011-12-02
mem002	13100000002	2011-12-01	3000/12/31	C	I	2011-12-02
mem003	13100000003	2011-12-01	3000/12/31	C	I	2011-12-02
mem004	13100000004	2011-12-01	3000/12/31	C	I	2011-12-02
mem005	13100000005	2011-12-01	3000/12/31	C	I	2011-12-02

### 3.1.5 数据加载

Greenplum 数据加载主要包括标准 SQL 的 insert、copy、外部表、gupload、web external table 几种方式，通过这个例子，将这种方式一起来向读者介绍一下。

#### 1. insert

这种数据加载方式效率最差，只适合加载极少量数据。这里向 12 月 2 号会员增量表中插入数据：

```
insert into public.member_delta values('mem006','13100000006','I',date'2011-12-03');
insert into public.member_delta values('mem002','13100000002','D',date'2011-12-03');
insert into public.member_delta values('mem003','13800000003','U',date'2011-12-03');
```

#### 2. copy

copy 这种数据加载方式源于 PostgreSQL，较 SQL 的 insert 方式效率大大提升，但是数据仍然需通过 Master 节点，无法实现并行高效数据加载。这里向会员历史表加载 12 月 1 号开始的初始数据。

我们将数据以逗号分隔，存放在 member\_his\_init.dat 文件中，内容如下：

```
mem001,13100000001,2011-12-01,3000-12-31,C,I,2011-12-01
mem002,13100000002,2011-12-01,3000-12-31,C,I,2011-12-01
mem003,13100000003,2011-12-01,3000-12-31,C,I,2011-12-01
mem004,13100000004,2011-12-01,3000-12-31,C,I,2011-12-01
mem005,13100000005,2011-12-01,3000-12-31,C,I,2011-12-01
```

copy 命令如下，指定分隔符还有数据文件。

```
testDB=# copy public.member_fatdt0_1_prt_p30001231 from '/home/gpadmin/member_his_init.dat' with delimiter ',';
COPY 5
```

### 3. 外部表

外部表在 2.3.8 节中已经简单介绍过了，首先，启动 gpfdist 服务：

```
$nohup gpfdist -d /home/gpadmin/data/ -p 8888 -l /home/gpadmin/data/gpfdist.log &
```

其次，创建外部表：

```
drop external table if exists public.member_ext;
create external table public.member_ext
(
    Member_id      varchar(64)
    ,phoneno       varchar(20)
    ,action         char(1)
    ,dw_ins_date   date
)
location ('gpfdist://localhost:8888/member_delta.dat')
format 'text'
(delimiter ',' null as '' escape 'off')
encoding 'gb18030'
log errors into member_err segment reject limit 2 rows;
```

最后，执行数据装载：

```
testDB=# Insert into public.member_delta select * from public.member_ext;
INSERT 0 3
```

### 4. gupload

gupload 是对外部表的一层封装，详细可参考用户手册，这里直接介绍使用语法，首先，编写 gupload 控制文件 member.yml，代码如下：

```
---
VERSION: 1.0.0.1
DATABASE: testDB
USER: gpadmin
HOST: localhost
PORT: 5432
GUPLOAD:
    INPUT:
        - SOURCE:
            LOCAL_HOSTNAME:
                - mdw
            PORT: 8081
        FILE:
```

```

        - /home/gpadmin/data/member_delta.dat
- COLUMNS:
    - Member_id: varchar(64)
    - phoneno: varchar(20)
    - action: char(1)
    - dw_ins_date: date
- FORMAT: text
- DELIMITER: ','
- ERROR_LIMIT: 2
- ERROR_TABLE: public.member_err
OUTPUT:
- TABLE: public.member_delta
- MODE: INSERT
SQL:
- BEFORE: "truncate table public.member_delta"
- AFTER: "analyze public.member_delta"

```

其次，执行数据加载：

```

$gupload -f member.yml
2012-01-08 14:30:01|INFO|gupload session started 2012-01-08 14:30:01
2012-01-08 14:30:01|INFO|started gpfdist -p 8081 -P 8082 -f "/home/gpadmin/
data/member_delta.dat" -t 30
2012-01-08 14:30:09|INFO|running time: 7.85 seconds
2012-01-08 14:30:09|INFO|rows Inserted = 3
2012-01-08 14:30:09|INFO|rows Updated = 0
2012-01-08 14:30:09|INFO|data formatting errors = 0
2012-01-08 14:30:09|INFO|gupload succeeded

```

最后，验证数据：

```

testDB=# select * from public.member_delta;
member_id | phoneno | action | dw_ins_date
-----+-----+-----+-----
mem006   | 13100000006 | I      | 2011-12-02
mem002   | 13100000002 | D      | 2011-12-02
mem003   | 13100000003 | U      | 2011-12-02
(3 rows)

```

## 5. 可执行外部表

可执行外部表会在 6.3.4 节中介绍，其中基于操作系统命令读取数据文件的方式如下，用法跟普通外部表类似，不过不用启动 gpfdist 服务，下面的外部表只在 Master 上执行：

```

drop external web table if exists public.member_ext;
create external web table public.member_ext
(
    Member_id      varchar(64)
    ,phoneno       varchar(20)
    ,action        char(1)

```

```

,dw_ins_date date
)
EXECUTE 'cat /home/gpadmin/data/member_delta.dat' ON master
format 'text'
(delimiter ',' null as '' escape 'off')
encoding 'gb18030'
;

```

清空 member\_delta 表并插入数据：

```

testDB=# truncate table public.member_delta ;
TRUNCATE TABLE
testDB=# Insert into public.member_delta select * from public.member_ext;
INSERT 0 3

```

### 3.1.6 数据刷新

#### 1. 拉链表刷新

Step1：对事实表中最新数据（分区 P30001231）与 member\_delta 表中的更新、删除数据进行左外连接，关联上则说明该数据已发生过变更，需要将该数据的失效时间更新为当天，并插入到 member\_tmp0 表中的历史数据分区中，关联不上则说明没有发生过变更，需要将该数据插入到 member\_tmp0 表的当前数据分区中。Greenplum 会根据 dtype 的数据自动选择对应的分区。

初始全量数据为 2011-12-01 号，在 12 月 3 号刷新 12 月 2 号增量数据，代码如下：

```

truncate table public.member_tmp0;
-- 清理临时表
INSERT INTO public.member_tmp0
(
  Member_id
 ,phoneno
 ,dw_beg_date
 ,dw_end_date
 ,dtype
 ,dw_status
 ,dw_ins_date
 )
SELECT    a.Member_id
          ,a.phoneno
          ,a.dw_beg_date
          ,CASE WHEN b.Member_id IS NULL THEN a.dw_end_date
          ELSE date'2011-12-02'
          END AS dw_end_date
          ,CASE WHEN b.Member_id IS NULL THEN 'C'
          ELSE 'H'
          END AS dtype

```

```

,CASE WHEN b.Member_id IS NULL THEN a.dw_status
      ELSE b.action
      END AS dw_status
      ,date'2011-12-03'
FROM   public.member_fatdt0          a
left join public.member_delta      b
ON     a.Member_id=b.Member_id
AND    b.action IN('D','U')
WHERE   a.dw_beg_date<=cast('2011-12-02' as date)-1
AND    a.dw_end_date>cast('2011-12-02' as date)-1;

```

Step2：将 member\_delta 的新增、更新数据插入到 member\_tmp0 表的当前数据分区中。

```

INSERT INTO public.member_tmp0
(
Member_id
,phoneno
,dw_beg_date
,dw_end_date
,dtype
,dw_status
,dw_ins_date
)
SELECT  Member_id
,phoneno
,cast('2011-12-02' as date)
,cast('3000-12-31' as date)
,'C'
,action
,cast('2011-12-03' as date)
FROM   public.member_delta
WHERE  action IN('I','U');

```

Step3：将 member\_fatdt0 表中的对应分区（P20121201）与 member\_tmp0 表的历史数据分区交换。

```

Truncate table public.member_tmpl;
alter table public.member_tmp0 exchange partition for('H') with table public.
member_tmpl;
alter table public.member_fatdt0 exchange partition for('2011-12-02') with
table public.member_tmpl;

```

Step4：将 member\_fatdt0 表中对应的当前数据分区（p30001231）与 member\_tmp0 表的当前数据分区交换。

```

alter table public.member_tmp0 exchange partition for('C') with table public.
member_tmpl;
alter table public.member_fatdt0 exchange partition for('3000-12-31') with
table public.member_tmpl;

```

至此，拉链表数据刷新完成，数据验证如图 3-2 所示。

aliputf8=# select * from public.member_fatdt0_1_prt_p20001231 order by member_id;						
member_id	phoneno	dw_beg_date	dw_end_date	dtype	dw_status	dw_ins_date
mem001	13100000001	2011-12-01	3000-12-31	C	I	2011-12-03
mem003	13800000003	2011-12-02	3000-12-31	C	U	2011-12-03
mem004	13100000004	2011-12-01	3000-12-31	C	I	2011-12-03
mem005	13100000005	2011-12-01	3000-12-31	C	I	2011-12-03
mem006	13100000006	2011-12-02	3000-12-31	C	I	2011-12-03

aliputf8=# select * from public.member_fatdt0_1_prt_p20111202 order by member_id;						
member_id	phoneno	dw_beg_date	dw_end_date	dtype	dw_status	dw_ins_date
mem02	13100000002	2011-12-01	2011-12-02	H	D	2011-12-03
mem03	13100000003	2011-12-01	2011-12-02	H	U	2011-12-03

a) 最新数据分区

b) 历史分区数据

图 3-2 拉链表刷新后最终数据

同样，更新对应的日期，可以刷新 3 号的增量数据。

## 2. 历史数据查询

基于拉链表，我们可以回溯到历史上任意一天的数据状态。

(1) 12月1号数据，如图 3-3 所示。

aliputf8=# select * from public.member_fatdt0 where dw_beg_date<=date'2011-12-01' and dw_end_date>date'2011-12-01' order by member_id;						
member_id	phoneno	dw_beg_date	dw_end_date	dtype	dw_status	dw_ins_date
mem001	13100000001	2011-12-01	3000-12-31	C	I	2011-12-04
mem002	13100000002	2011-12-01	2011-12-02	H	D	2011-12-03
mem003	13800000003	2011-12-01	2011-12-02	H	U	2011-12-03
mem004	13100000004	2011-12-01	2011-12-03	H	D	2011-12-04
mem005	13100000005	2011-12-01	2011-12-03	H	U	2011-12-04

图 3-3 12月1号数据

(2) 12月2号数据，如图 3-4 所示。

aliputf8=# select * from public.member_fatdt0 where dw_beg_date<=date'2011-12-02' and dw_end_date>date'2011-12-02' order by member_id;						
member_id	phoneno	dw_beg_date	dw_end_date	dtype	dw_status	dw_ins_date
mem01	13100000001	2011-12-01	3000-12-31	C	I	2011-12-04
mem03	13800000003	2011-12-02	3000-12-31	C	U	2011-12-04
mem04	13100000004	2011-12-01	2011-12-03	H	D	2011-12-04
mem05	13100000005	2011-12-01	2011-12-03	H	U	2011-12-04
mem06	13100000006	2011-12-02	3000-12-31	C	I	2011-12-04

图 3-4 12月2号数据

(3) 12月3号数据，如图 3-5 所示。

aliputf8=# select * from public.member_fatdt0 where dw_beg_date<=date'2011-12-03' and dw_end_date>date'2011-12-03' order by member_id;						
member_id	phoneno	dw_beg_date	dw_end_date	dtype	dw_status	dw_ins_date
mem001	13100000001	2011-12-01	3000-12-31	C	I	2011-12-04
mem03	13800000003	2011-12-02	3000-12-31	C	U	2011-12-04
mem04	13100000004	2011-12-03	3000-12-31	C	U	2011-12-04
mem05	13100000005	2011-12-02	3000-12-31	C	I	2011-12-04
mem07	13100000007	2011-12-03	3000-12-31	C	I	2011-12-04

图 3-5 12月3号数据

### 3.1.7 分区裁剪

下面通过查看执行计划（第 5 章讲详细介绍执行计划）来介绍 Greenplum 的分区表的功能。

全表扫描的执行计划如下：

```
testDB=# explain select * from public.member_fatdt0;
          QUERY PLAN
-----
 Gather Motion 6:1  (slice1; segments: 6)  (cost=0.00..108.40 rows=1708
 width=36)
   -> Append  (cost=0.00..108.40 rows=1708 width=36)
     -> Append-only Scan on member_fatdt0_1_prt_p20111201 member_fatdt0
      (cost=0.00..0.00 rows=1 width=232)
     -> Append-only Scan on member_fatdt0_1_prt_p20111202 member_fatdt0
      (cost=0.00..0.00 rows=1 width=232)
     -> Append-only Scan on member_fatdt0_1_prt_p20111203 member_fatdt0
      (cost=0.00..0.00 rows=1 width=232)
     ...
     -> Append-only Scan on member_fatdt0_1_prt_p30001231 member_fatdt0
      (cost=0.00..108.40 rows=1707 width=3
5)
      (10 rows)
```

通过执行计划可以看出，Greenplum 扫描了所有的分区。当加入筛选条件 dw\_end\_date='3000-12-31' 时，执行计划如下：

```
testDB=# explain select * from public.member_fatdt0 where dw_end_date='3000-12-31';
          QUERY PLAN
-----
 Gather Motion 6:1  (slice1; segments: 6)  (cost=0.00..134.00 rows=1707
 width=35)
   -> Append  (cost=0.00..134.00 rows=1707 width=35)
     -> Append-only Scan on member_fatdt0_1_prt_p30001231 member_fatdt0
      (cost=0.00..134.00 rows=1707 width=35)
           Filter: dw_end_date = '3000-12-31'::date
      (4 rows)
```

这时，分区裁剪发生了作用，只扫描了 P30001231 这个分区。

### 3.1.8 数据导出

Greenplum 在处理大数据量数据导出时常用的方式主要有并行导出（可写外部表）和非并行导出（COPY），copy 命令比较简单，就不细说了。下面我们分别简单介绍下可写外部表数据导出方式，通过 gpfdist 可写外部表将数据导出至文件服务器。

1) 创建可写外部表：

```
testDB=# CREATE WRITABLE EXTERNAL TABLE member_tmp1_unload
```

```

testDB-# ( LIKE member_tmp1 )
testDB-# LOCATION ('gpfldist://localhost:8080/member_tmp1.dat')
testDB-# FORMAT 'TEXT' (DELIMITER ',')
testDB-# DISTRIBUTED BY (member_id);
CREATE EXTERNAL TABLE

```

WRITABLE 关键字表示该外部表是可写外部表；Like 语句表示创建的外部表的表结构与 member\_tmp1 表结构一样；LOCATION 指定 gpfldist 的机器名跟端口，还有保存的文件名；FORMAT 为导出文件格式定义。

2) 执行数据导出：

```

testDB=# insert into member_tmp1_unload select * from member_tmp1;
INSERT 0 5

```

跟普通 insert 语句一样，只需要将数据插入外部表即可。

3) 验证生成的文件：

```

$less member_tmp1.dat
mem004,1310000004,2011-12-01,3000-12-31,C,I,2011-12-03
mem006,1310000006,2011-12-02,3000-12-31,C,I,2011-12-03
mem001,1310000001,2011-12-01,3000-12-31,C,I,2011-12-03
mem005,1310000005,2011-12-01,3000-12-31,C,I,2011-12-03
mem003,1380000003,2011-12-02,3000-12-31,C,U,2011-12-03
$ 

```

## 3.2 日志分析

日志分析是网站分析的基础，通过对网站浏览的日志进行分析，可以为网站优化提供数据支持，了解用户群以及用户浏览特性，对改进网站体验，提升流量有非常重要的意义。

下面将通过 Greenplum 实现一个简单的网站浏览日志的分析。

### 3.2.1 应用场景描述

- 分析全网站每分钟的 PV、UV，并导出到 Excel 中，画出折线图。
- 解析 URL，获取 URL 中的参数列表。
- 通过 URL 取得 member\_id，然后统计当天浏览次数的用户分布，如浏览次数在 1 ~ 5、6 ~ 10、11 ~ 50、51 ~ 100 以及 100 次以上的这五个区间段分别有多少个用户。

### 3.2.2 数据 Demo

为了简单起见，笔者对数据进行了一些预处理，只保留了几个字段，建表语句及字段描述如下：

```

DROP TABLE IF EXISTS log_path;
CREATE table log_path(
    log_time timestamp(0)          -- 浏览时间
    ,cookie_id varchar(256)        -- 浏览的 cookie_id
    ,url      varchar(1024)        -- 浏览页面的 url
    ,ip       varchar(64)          -- 用户 ip
    ,refer_url varchar(1024)       -- 来源的 url, 这里只保留域名
)distributed by(cookie_id);

```

Demo 数据如下：

```

testDB=# select * from log_path limit 1;
-[ RECORD 1 ]-----
log_time | 2012-07-14 23:44:58
cookie_id | 119.187.6.228.1337696430725.8
url       | /china.alibaba.com/ims/chat_card_60.htm?member_
id=scutshuxue&cssName=default
ip        | 119.178.198.222
refer_url | www2.im.alisoft.com

```

### 3.2.3 日志分析实战

#### 1. PV、UV 分布

cookie\_id 可以视为唯一的用户标识，故 UV 可视为去重后的 cookie\_id 数。SQL 如下：

```

SELECT  TO_CHAR(log_time,'yyyy-mm-dd HH24:mi:00')
        ,COUNT(1)  pv
        ,COUNT(DISTINCT cookie_id) uv
  FROM log_path
 GROUP BY 1
 ORDER BY 1;

```

这里只是较少的样例数据，结果如下：

```

testDB=# select * from log_pv_uv_result;
log_time      |  pv  |  uv
-----+-----+-----
2012-07-14 23:01:00 | 4758 | 1699
2012-07-14 23:45:00 |  552 |  257
2012-07-14 23:03:00 | 1656 |  712
2012-07-14 23:34:00 | 5554 | 1878
2012-07-14 23:04:00 | 3504 | 1325
2012-07-14 23:00:00 |    12 |     6
2012-07-14 23:44:00 | 4498 | 1540
2012-07-14 23:33:00 |     4 |     2
(8 rows)

```

将数据导出成 csv 格式，在 Excel 中展现，copy 命令的语法如下：

```
testDB=# copy log_pv_uv_result to '/tmp/log_pv_uv.csv' csv;
COPY 8
```

在 Excel 中打开并画图，结果如图 3-6 所示。

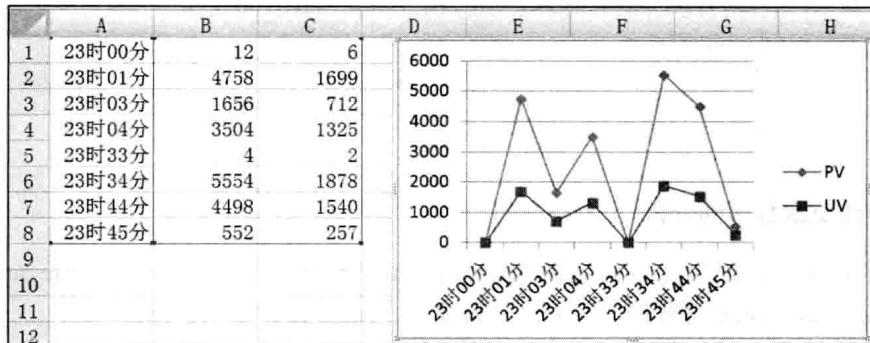


图 3-6 在 Excel 中展示 PV/UV 折线图

## 2. 解析 URL 参数

解析 URL，是指通过 substring 对 URL 进行正则表达式匹配，将域名取出，例如对于下面这个 URL：

```
http://page.china.alibaba.com/others/feedbackfromalitalk.html
```

正则表达式 `\w+://(\w.+)+` 可以将域名匹配出来。

同样的，可以将参数后面关键字（member\_id 或 memberId）的值获取出来，作为字段 member\_id。

`split_part` 函数可以将字符串按照某个字符串分割，然后获取其中一个子串。

`regexp_split_to_array` 函数可以将字符串按照某个字符串分割，然后转换为数组变量。

```
DROP TABLE IF EXISTS log_path_tmp1;
CREATE TABLE log_path_tmp1 AS
SELECT
    log_time
    ,cookie_id
    ,substring(url,E'\w+://(\w.+)+') AS host
    ,split_part(url,'?',1) AS url
    ,substring(url,E'member[_]?[i|I]d=(\w+)') AS member_id
    ,regexp_split_to_array(split_part(url,'?',2), '&') AS paras
    ,ip
    ,refer_url
FROM log_path
DISTRIBUTED BY (cookie_id);
```

数据 Demo 的样例数据解析后结果如下：

```
testDB=# select * from log_path_tmpl where member_id='scutshuxue' limit 1;
-[ RECORD 1 ]-----
log_time | 2012-07-14 23:44:58
cookie_id | 119.187.6.228.1337696430725.8
host      | china.alibaba.com
url       | http://china.alibaba.com/ims/chat_card_60.htm
member_id | scutshuxue
paras     | {member_id=scutshuxue,cssName=default}
ip        | 119.178.198.222
refer_url | www2.im.alisoft.com
```

### 3. 用户浏览次数区间分析

要计算浏览次数的分布，首先按照 cookie\_id 做聚合，计算出每个 cookie\_id 的浏览次数，之后再用 case when 对数据进行分区，再聚合，SQL 如下：

```
SELECT CASE WHEN cnt>100 THEN '100+'
            WHEN cnt>50 THEN '51-100'
            WHEN cnt>10 THEN '11-50'
            WHEN cnt>5 THEN '6-10'
            ELSE '<=5' END tag
    ,COUNT(1) AS NUMBER
FROM (
    SELECT cookie_id,COUNT(1) cnt
    FROM log_path_tmpl
    GROUP BY 1
) t
GROUP BY 1;
```

结果如下：

tag	number
6-10	440
11-50	126
<=5	6501

(3 rows)

## 3.3 数据分布

由于 Greenplum 是分布式的架构，为了充分体现分布式架构的优势，我们有必要了解数据是如何分散在各个数据节点上的，有必要了解数据倾斜对数据加载、数据分析、数据导出的影响。

### 3.3.1 数据分散情况查看

我们来简单做个测试，首先，利用 `generate_series` 和 `repeat` 函数生成一些测试数据，代码如下：

```
create table test_distribute_1
as
select a as id
    ,round(random()) as flag
    ,repeat('a',1024) as value
from generate_series(1,5000000)a;
```

500 万数据分散在 6 个数据节点，利用下面这个 SQL 可以查询数据的分布情况（SQL 在后面 5.8.4 中将会介绍）：

```
testDB=# select gp_segment_id,count(*)
testDB-#   from test_distribute_1
testDB-# group by 1;
gp_segment_id | count
-----+-----
1  | 833396
4  | 833297
5  | 833294
3  | 833309
2  | 833359
0  | 833345
(6 rows)
```



**注意** 上述 SQL 中的 `group by 1`，其中 1 代表 `select` 后面的第一个字段，即 `gp_segment_id`。

### 3.3.2 数据加载速度影响

接下来将通过实验来测试在分布键不同的情况下数据加载的速度。

#### (1) 数据倾斜状态下的数据加载

1) 测试数据准备，将测试数据导出：

```
testDB=# copy test_distribute_1 to '/home/gpadmin/data/test_distribute.dat'
with delimiter '|';
COPY 5000000
```

2) 建立测试表，以 `flag` 字段为分布键：

```
testDB=# create table test_distribute_2 as select * from test_distribute_1
limit 0 distributed by(flag);
SELECT 0
```

### 3) 执行数据导入:

```
$ time psql -h localhost -d testDB -c "copy test_distribute_2 from stdin with
delimiter '|'" < /home/gpadmin/data/test_distribute.dat

real    14m1.381s
user    0m24.080s
sys     0m26.387s
```

### 4) 由于分布键 flag 取值只有 0 和 1, 因此数据只能分散到两个数据节点, 如下:

```
testDB=# select gp_segment_id,count(*) from table_distribute_4 group by 1;
gp_segment_id | count
-----+-----
            3 | 2498434
            2 | 2501566
(2 rows)
Time: 50751.740 ms
```

### 5) 由于数据分布在 2 和 3 节点, 对应 Primary Segment 在 dell3、Mirror 节点 dell4 上, 可通过以下 SQL 查询 gp\_segment\_configuration 获得:

```
testDB=# select dbid,content,role,port,hostname from gp_segment_configuration
testDB-# where content in(2,3) order by role;
dbid | content | role | port | hostname
-----+-----+-----+-----+
 11 |      3 | m   | 60001 | dell14
 10 |      2 | m   | 60000 | dell14
  5 |      3 | p   | 50001 | dell3
  4 |      2 | p   | 50000 | dell3
(4 rows)
```

在执行数据导入期间, Greenplum Performance Monitor 页面可监控到: 仅有 dell3 和 dell4 两台服务器有磁盘和 CPU 消耗, 如图 3-7 所示。

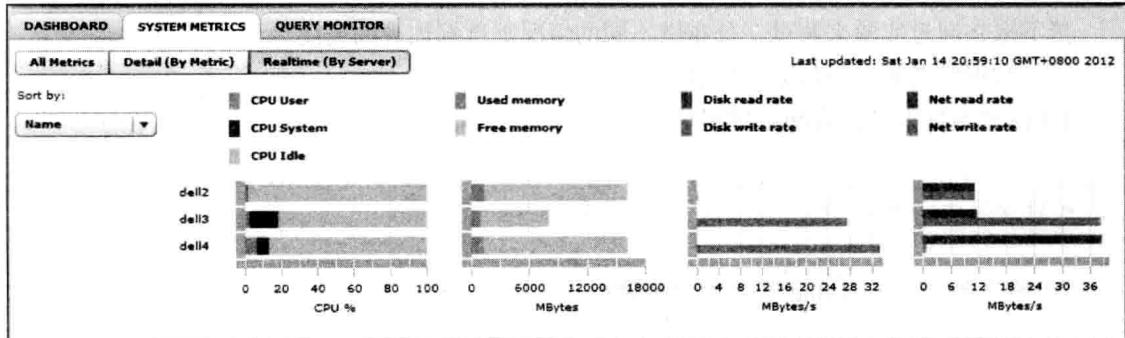


图 3-7 在数据分布不均情况下, 数据导入时服务器的磁盘和 CPU 消耗

 注意 Greenplum Performance Monitor 的安装部署，将在 10.2.1——GPmonitor 介绍中介绍。

## (2) 数据分布均匀状态下的数据加载

### 1) 建立测试表，以 id 字段为分布键：

```
testDB=# create table test_distribute_3 as select * from test_distribute_1
limit 0 distributed by(id);
SELECT 0
```

### 2) 执行数据导入：

```
$ time psql -h localhost -d testDB -c "copy test_distribute_3 from stdin with
delimiter '|' < /home/gpadmin/data/test_distribute.dat
```

```
real    9m40.607s
user    0m24.364s
sys     0m24.491s
```

### 3) 由于分布键 id 取值顺序分布，因此数据可均匀分散至所有数据节点，如下：

```
testDB=# select gp_segment_id,count(*) from test_distribute_3 group by 1;
gp_segment_id | count
-----+-----
      1 | 833396
      4 | 833297
      5 | 833294
      3 | 833309
      2 | 833359
      0 | 833345
(6 rows)

Time: 17999.875 ms
```

在执行数据导入期间，Greenplum Performance Monitor 页面可监控到：3台服务器的所有节点都有磁盘和CPU消耗，可见，在数据均匀的情况下，可以利用更多的机器进行工作，性能也比较高，如图 3-8 所示。

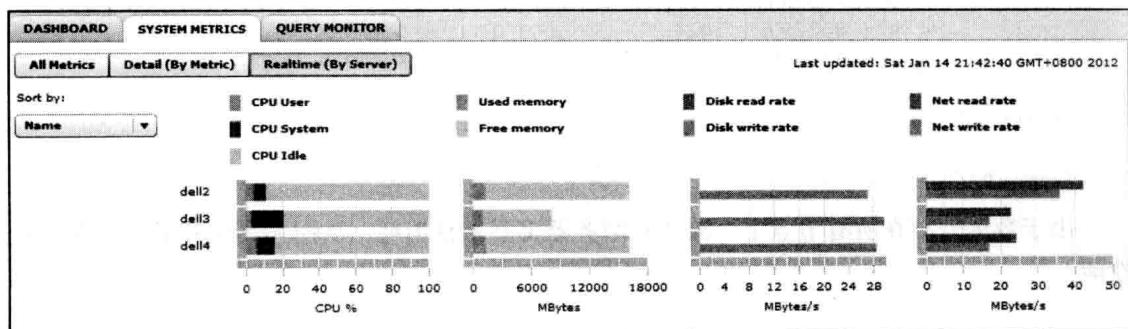


图 3-8 在数据分布均匀情况下，数据导入时服务器的磁盘和 CPU 消耗

### 3.3.3 数据查询速度影响

#### (1) 数据倾斜状态下的数据查询

```
testDB=# select gp_segment_id,count(*),max(length(value)) from test_distribute_2 group by 1;
gp_segment_id | count | max
-----+-----+-----
 3 | 2498434 | 1024
 2 | 2501566 | 1024
(2 rows)

Time: 79840.885 ms
```

由于数据分布在 2 和 3 节点上，即对应 dell3 和相应的 Mirror 节点 dell4 上，但是数据查询只需要 Primary 节点，故只有 dell3 节点有磁盘消耗，如图 3-9 所示。

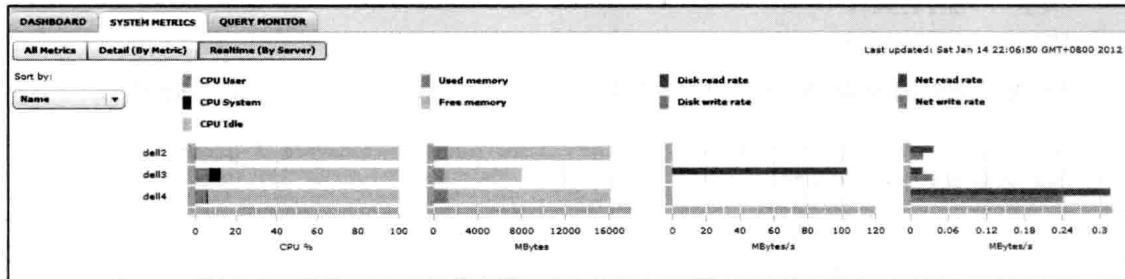


图 3-9 在数据分布不均情况下，查询数据时服务器的磁盘和 CPU 消耗

#### (2) 数据分布均匀状态下的数据查询

```
testDB=# select gp_segment_id,count(*),max(length(value)) from test_distribute_3 group by 1;
gp_segment_id | count | max
-----+-----+-----
 1 | 833396 | 1024
 3 | 833309 | 1024
 5 | 833294 | 1024
 4 | 833297 | 1024
 2 | 833359 | 1024
 0 | 833345 | 1024
(6 rows)

Time: 6976.840 ms
```

由于数据分布在所有节点上，故所有服务器都有磁盘消耗，从而大大提升了数据查询的性能。

## 3.4 数据压缩

### 3.4.1 数据加载速度影响

基于 table\_distribute\_4 表创建一个普通的表。从 Greenplum Performance Monitor 页面可看到，在 dell3 和 dell4 上有大量磁盘写操作，如图 3-10 所示。

建表语句如下：

```
testDB=# create table test_compress_1 as select * from test_distribute_1
distributed by(flag);
SELECT 5000000
```

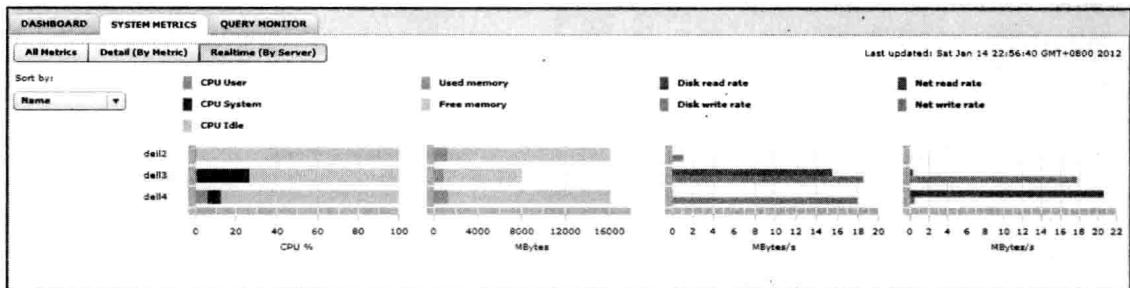


图 3-10 在无压缩情况下，数据加载时的服务器磁盘和 CPU 消耗

基于 table\_distribute\_4 表创建一个压缩表。由于数据压缩比很大，从 Greenplum Performance Monitor 页面可看到，在 dell3 和 dell4 上基本没有磁盘写操作，只有读操作，如图 3-11 所示。建表语句如下：

```
testDB=# create table test_compress_2 with(appendonly=true, compresslevel=5) as
select * from test_distribute_1 distributed by(flag);
SELECT 5000000
```

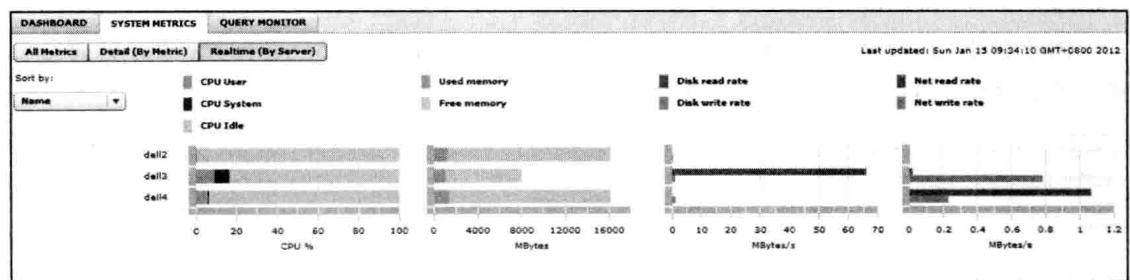


图 3-11 在压缩情况下，数据加载时的服务器磁盘和 CPU 消耗

### 3.4.2 数据查询速度影响

#### (1) 普通表的数据查询

```
testDB=# select gp_segment_id,count(*),max(length(value)) from test_compress_1
group by 1;
gp_segment_id | count | max
-----+-----+-----
3 | 2498434 | 1024
2 | 2501566 | 1024
(2 rows)
Time: 65589.914 ms
```

磁盘消耗较大，如图 3-12 所示。

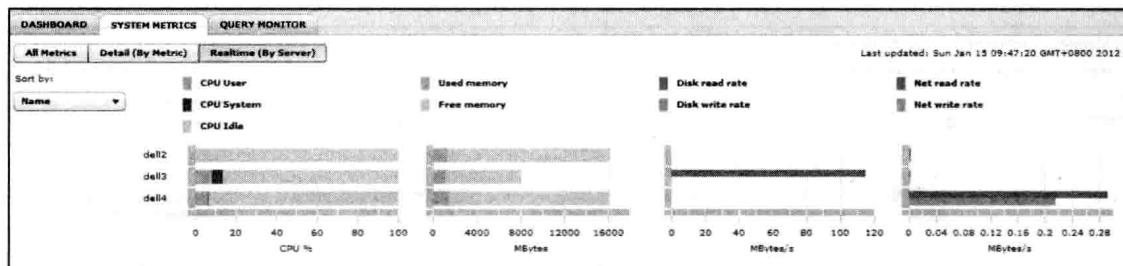


图 3-12 在无压缩情况下，查询表时服务器的磁盘和 CPU 消耗

#### (2) 压缩表的数据查询

```
testDB=# select gp_segment_id,count(*),max(length(value)) from test_compress_2
group by 1;
gp_segment_id | count | max
-----+-----+-----
3 | 2498434 | 1024
2 | 2501566 | 1024
(2 rows)
Time: 23004.626 ms
```

由于数据经过压缩，占用存储空间很小，从 Greenplum Performance Monitor 页面可看到，几乎没有磁盘读操作，如图 3-13 所示。

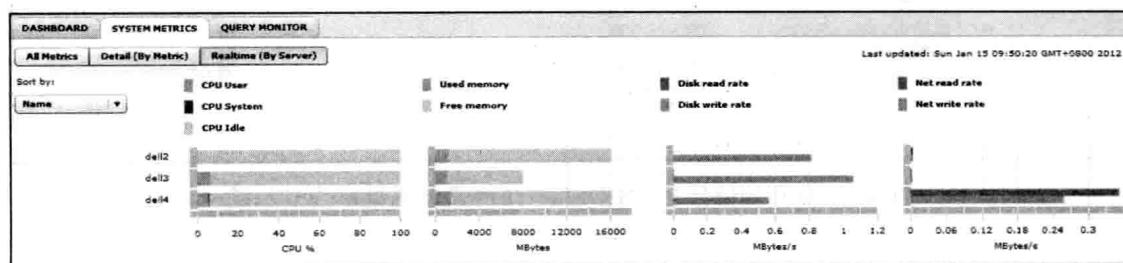


图 3-13 在压缩情况下，查询表时服务器的磁盘和 CPU 消耗

## 3.5 索引

Greenplum 支持 B-tree、bitmap、函数索引等，在这里我们简单介绍一下 B-tree 索引：

```
testDB=# create table test_index_1 as select * from test_distribute_1;
SELECT 5000000

testDB=# select id,flag from test_index_1 where id=100;
 id | flag
----+---
 100 |    0
(1 row)
Time: 2606.125 ms
```

接下来我们在 `flag` 字段上创建 bitmap 索引：

```
testDB=# CREATE INDEX test_index_1_idx ON test_index_1 (id);
CREATE INDEX
Time: 34997.881 ms
```

再次查看执行计划，采用了索引扫描，如下所示。

```
testDB=# explain select id,flag from test_index_1 where id=100;
          QUERY PLAN
-----
Gather Motion 1:1  (slice1; segments: 1)  (cost=0.00..200.84 rows=1 width=12)
  ->  Index Scan using test_index_1_idx on test_index_1  (cost=0.00..200.84
      rows=1 width=12)
          Index Cond: id = 100
(3 rows)
```

建好索引后，再次执行上面的查询语句，有索引的情况下，用了 23 毫秒，相比未创建索引时 2606 毫秒，有了质的提升。

另外，表关联字段上的索引和 appen-only 压缩表上的索引都能带来较大的性能提升，虽然在数据库应用中，索引的应用场景不多，但是读者仍然可以结合实际的场景来运用索引。

## 3.6 小结

本章简单介绍了基于 Greenplum 数据库实现数据库数据模型刷新的过程，包括典型的需求场景分析、物理模型定义、数据加载、数据刷新、数据访问、数据导出等。另外也讲解了 Greenplum 典型的特性，比如数据分布策略、数据压缩、统计信息、表分区、列存储、索引等。



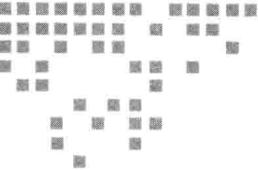


中篇

*Part 2*

## 进阶篇

- 第4章 数据字典詳解
  - 第5章 执行计划詳解
  - 第6章 Greenplum 高级应用
  - 第7章 Greenplum 架构介绍
- \*\*\*\*\*



## 第4章 数据字典详解

Greenplum 是基于 PostgreSQL 开发的，所以其大部分数据字典是一样的。这些数据字段会根据其特性做一些修改，并不是完全一样的，如 pg\_class、pg\_attribute 等。

Greenplum 也有自己的一些数据字典，这些数据字典一般是以 gp\_ 开头的。在本章中，我们还将介绍一些 gp\_toolkit 的工具箱。Greenplum 是一个分布式数据库，每个节点都是一个数据库，除了一些集群配置信息的数据字典外，其他所有数据字典 master 和 segment 都有，而且这些数据字典的信息与 master 的大部分是一致的。

### 4.1 oid 无处不在

跟 PostgreSQL 一样，大部分的数据字典都是以 oid 关联的，oid 是一种特殊的数据类型。在 PG/GP 中，oid 都是递增的，每一个表空间、表、索引、数据文件名、函数、约束等都对应有一个唯一标识的 oid。oid 是全局递增的，可以把 oid 想象成一个递增的序列（SEQUENCE）。

通过下面的语句可以找到数据字典中带隐藏字段 oid 的所有表，这些表的 oid 增加都是共享一个序列的。当然，还有其他的表也是共享这些序列的，比如 pg\_class 的 relfilenode。

```
testDB=# select attrelid::regclass,attnname
testDB-#   from pg_attribute a,pg_class b
testDB-#   where a.attrelid=b.oid
testDB-#     and b.relnamespace=11
testDB-#     and atttypid=26
testDB-#     and b.relstorage='h'
testDB-#     and attnname='oid'
```

```

testDB=#      and b.relname not like '%index';
              attrelid      | attname
-----+-----
pg_authid          | oid
pg_user_mapping    | oid
pg_type            | oid
pg_proc             | oid
pg_class            | oid
pg_attrdef          | oid
pg_constraint        | oid
pg_operator          | oid
pg_opclass           | oid
pg_am               | oid
pg_language          | oid
pg_rewrite           | oid
pg_trigger            | oid
pg_cast               | oid
pg_namespace          | oid
pg_conversion         | oid
pg_tablespace         | oid
pg_resqueue           | oid
pg_resourcetype       | oid
pg_resqueuecapability | oid
pg_partition           | oid
pg_partition_rule      | oid
pg_filespace          | oid
pg_foreign_data_wrapper | oid
pg_foreign_server      | oid
pg_database            | oid
(26 rows)

```

由于数据库中存在大量的 oid, oid 是一个 32 位的数字, 这个数字没有什么特殊的含义, 使我们理解数据字典很困难。下面介绍几个数据类型, 如表 4-1 所示。将 oid 转换成这些数据类型, 可以简化很多的操作。

表 4-1 将 oid 转换成名称的几种类型

名 字	引 用	描 述
regproc	pg_proc	函数名
regprocedure	pg_proc	带参数类型的函数
regoper	pg_operator	操作符名
regoperator	pg_operator	带参数类型的操作符
regclass	pg_class	关系名

最常用的是 regclass, 它关联数据字典的 oid, 使用方法如下:

```

testDB=# select 1259::regclass;
regclass

```

```
-----
pg_class
(1 row)
```

1259 是 pg\_class 对应的 oid，这个是默认的，每一个 PostgreSQL 都是如此。

```
testDB=# select oid,relname from pg_class where oid='pg_class'::regclass;
oid | relname
-----+-----
1259 | pg_class
(1 row)
```

这样就可以通过 regclass 寻找一个表的信息，就不用去关联 pg\_class 和 pg\_namespace（记录 schema 信息）了，比较方便。

同样的，其他几个类型也是一样的用法，如 regproc (regprocedure) 是与 pg\_proc (保存普通函数的命令) 关联的。regoper (regoperator) 是与 pg\_operator (操作符) 的 oid 关联的。

```
testDB=# select oid::regoper,oid::regoperator,oid,oprname from pg_operator limit 1;
oid      |      oid      | oid | oprname
-----+-----+-----+
pg_catalog.= | =(integer,bigint) | 15 | =
(1 row)

testDB=# select oid::regoper,oid::regoperator,oid,oprname from pg_operator limit 1;
oid      |      oid      | oid | oprname
-----+-----+-----+
pg_catalog.= | =(integer,bigint) | 15 | =
(1 row)
```

下面我们先介绍一下 Greenplum 保存集群配置信息的数据字典。后面的例子将会给出如何使用 regclass，并专门讲解一下 oid 是如何增长的，以及什么时候会用到 oid。

## 4.2 数据库集群信息

Greenplum 的集群配置信息在 master 上面，这些配置信息对集群管理非常重要，通过这些配置信息可以了解整个集群的状况，可以得知是否有节点失败，通过修改这些配置可以实现集群的扩容等操作。

### 4.2.1 Gp\_configuration 和 gp\_segment\_configuration

在 Greenplum 3.x 版本中，集群的配置信息记录在 gp\_configuration 中。表 gp\_configuration 中的字段含义如表 4-2 所示。

表 4-2 gp\_configuration 表结构

列名	类型	字段说明
content	smallint	数据库节点的标识 ID，在 Segment 的主、备节点中，它们的 content 值是相同的。Master 节点的 content 值是 -1，数据节点的 content 值是从 0 ~ N (N 是集群中 primary 节点的数量)
definedprimary	boolean	该节点是否被定义为主节点
dbid	smallint	唯一标识 Segment 的 ID，这个 ID 的一个 Master 节点为 1，然后主节点按照 content 递增，之后是备节点按照 content 递增，最后是 Master Standby
isprimary	boolean	该节点现在是否被定义成主节点，一般备节点为 false，如果有主节点失败，则该备节点就被设置成 true，变成主节点
valid	boolean	标识该节点是否失效
hostname	name	子节点所在机器的 hostname 或 ip
port	integer	子节点的端口
datadir	text	子节点的数据目录

在 Greenplum 4.x 版本中，由于引入了文件空间 (filespace) 的概念，一个节点的数据目录可以是多个，因此将 gp\_configuration 拆分成两个表，gp\_segment\_configuration (如表 4-3 所示) 和 pg\_filespace\_entry。Greenplum4.x 引入了基于文件的数据同步策略，所以也相应地增加了几个数据字典来体现这一个特性。

表 4-3 gp\_segment\_configuration 表结构

列名	类型	字段说明
dbid	smallint	唯一标识 Segment 的 ID，这个 ID 的一个 Master 节点为 1，然后主节点按照 content 递增，之后备节点按照 content 递增，最后是 Master Standby
content	smallint	数据库节点的标识 ID，在 Segment 的主、备节点中，它们的 content 值是相同的。主节点的 content 值是 -1，数据节点的 content 值是从 0 ~ N (N 是集群中 primary 节点的数量)
role	char	该节点现在的角色 (primary 或 mirror)
preferred_role	char	该节点被定义的角色 (primary 或 mirror)
mode	char	主备同步的状态，有三个值，s (synchronized) 代表已同步，r (resyncing) 代表正在重新同步，c (change logging) 代表不同步
status	char	判断节点状态的值，有两个值，u (up) 标识该节点正常运行，d (down) 标识该节点失败
port	integer	子节点的端口
hostname	text	子节点所在机器的 hostname
address	text	子节点所在机器的 IP
replication_port	integer	主节点和备节点同步的端口
san_mounts	int2vector	一个数组对应 gp_san_configuration 的 oid，这个字段是只在使用共享存储的时候使用的，如 SAN 共享存储

这两张表是在 pg\_global 表空间下面的，是全局的，同一个集群中所有数据库共用的信息。

## 4.2.2 Gp\_id

在 Greenplum 3.x 中，每一个子节点都有 gp\_id 表，这个表记录该节点在集群中的配置信息。

表 4-4 gp\_id 表结构

列 名	类 型	字段说明
gpname	name	Greenplum 数据库的名称
numsegments	smallint	集群中数据节点的数量
dbid	smallint	该节点的 dbid
content	smallint	该节点的 content ID

在 Greenplum 4.x 中，这个 gp\_id 表已经废弃掉，所有子节点的 gp\_id 数据都是一模一样的：

```
testDB=# select * from gp_id;
gpname | numsegments | dbid | content
-----+-----+-----+-----
Greenplum | -1 | -1 | -1
(1 row)
```

因为这些信息都是在启动子节点的时候通过 pg\_ctl 的启动参数传进来的，这些信息在 Greenplum 4.x 中都是以数据库参数的形式存在的，所以可以建一个视图，获取跟 Greenplum 3.x 中 gp\_id 表一模一样的信息。

```
create view v_gp_id
as
select 'Greenplum'::name as gpname
      ,current_setting('gp_num_contents_in_cluster') as numsegments
      ,current_setting('gp_dbid')
      ,current_setting('gp_contentid') as content;
```

其实 gp\_id 表除了其本身所要表达的数据之外，还有一个特殊的功能，那就是这个表在每一个数据节点中都只有一条数据，这对获取子节点的数据十分有用。

比如想获取集群中正在运行的数据节点的 hostname 信息，就可以使用如下的方法。

Greenplum 中没有获取 hostname 的函数，我们可以通过 python 来创建一个函数（关于自定义函数如何创建，可参第 6 章关于 Greenplum 高级应用的介绍）：

```
testDB=# CREATE or replace FUNCTION public.hostname()
testDB-#     RETURNS text
testDB-# AS $$
testDB$#     import socket
testDB$#     return socket.gethostname()
testDB$# $$ LANGUAGE plpythonu;
CREATE FUNCTION
```

```
testDB=# select hostname() ;
      hostname
-----
inc-dw-151-11.hst.bjc.kfc.alidc.net
(1 row)
```

在使用 plpythonu 函数时，要先创建语言：

```
testDB=# create language plpythonu;
CREATE LANGUAGE
```

要获取子节点的信息，还需要借助一个函数——`gp_dist_random`。对于 `pg_catalog` 中的数据字典表，我们都是在主节点上查询的，不会查询子节点上的数据字典，如果想在主节点上查询子节点的数据字典，可以利用 `gp_dist_random` 函数。看下面的例子我们就清楚 `gp_dist_random` 函数是怎么使用的。

```
testDB=# select gp_segment_id,count(1) from gp_dist_random('pg_class') group by
1 order by 1;
   gp_segment_id | count
-----+-----
       0 |     387
       1 |     387
       2 |     387
       3 |     387
       4 |     387
       5 |     387
(6 rows)
```

这样我们就可以获取每个数据字典的大小。

同样的，要查看子节点的 `hostname`，就必须从每个子节点都取出一条数据。刚好，`gp_id` 在每个子节点上都只有一条数据，所以，我们可以使用下面的语句进行查询。

```
testDB=# select hostname() from gp_dist_random('gp_id');
      hostname
-----
inc-dw-151-12.hst.bjc.kfc.alidc.net
inc-dw-151-12.hst.bjc.kfc.alidc.net
inc-dw-151-13.hst.bjc.kfc.alidc.net
inc-dw-151-11.hst.bjc.kfc.alidc.net
inc-dw-151-11.hst.bjc.kfc.alidc.net
inc-dw-151-13.hst.bjc.kfc.alidc.net
(6 rows)
```

在 Greenplum 自带的一些检测数据字典的脚本中，经常使用 `gp_dist_random` 函数来实现子节点数据字典的分析，比如 `gpcheckcat` 脚本，其升级脚本（`gpmigrator`）就调用这个函数来检测数据字典。10.6 节介绍查看子节点的 SQL 运行状态的工具时还会用到 `gp_id` 和 `gp_dist_random`。

### 4.2.3 Gp\_configuration\_history

当数据节点失败的时候，GP MASTER 通过心跳检测机制检测出 Segment 失败，就会触发主、备数据节点切换的动作，每一个动作都会记录在 gp\_configuration\_history 表中。gp\_configuration\_history 表结构如表 4-5 所示。

表 4-5 gp\_configuration\_history 表结构

列名	类型	字段说明
time	timestamp	发生切换的时间
dbid	smallint	发现主、备切换的数据节点的 dbid
desc	text	发生切换的原因描述

当数据库发生切换的时候，我们可以通过 gp\_configuration\_history 表来了解数据库切换的原因，以及发生切换的时间。

### 4.2.4 pg\_filespace\_entry

在 Greenplum 4.x 中，引入了文件空间（filespace）的概念，一个数据库的数据节点可以有多个数据目录，所以数据目录的字段信息从 gp\_configuration 中抽离出来，保存在 pg\_filespace\_entry 表中。pg\_filespace\_entry 表结构如表 4-6 所示。

表 4-6 pg\_filespace\_entry 表结构

列名	类型	字段说明
fsefsoid	oid	文件空间（filespace）的 oid
fsedbld	smallint	节点的 dbid
fselocation	text	数据文件的目录

### 4.2.5 集群配置信息表转化

当在实际应用中，Greenplum 3.x 跟 Greenplum 4.x 的集群同时存在，或者 Greenplum 3.x 升级到 Greenplum 4.x 的时候，一些外部程序如果需要获取 GP 集群的配置信息，就必须对 Greenplum 3.x 跟 Greenplum 4.x 分别进行识别处理，使其对应不同的数据字典。为了维护代码的统一性和升级方便，不用修改代码，可以统一定义一个视图，将 Greenplum 4.x 的集群配置的数据字典转换成 Greenplum 3.x 的模式，免得在代码中判断很多的逻辑。

Greenplum 3.x 的视图定义：

```
create view v_gp_configuration as select * from gp_configuration;
```

Greenplum 4.x 的视图定义：

```
create view v_gp_configuration as
select content,
```

```

case when preferred_role='p' then 't'::boolean else 'f'::boolean end as definedprimary,
a.dbid,
case when role='p' then 't'::boolean else 'f'::boolean end as isprimary,
case when status='u' then 't'::boolean else 'f'::boolean end as valid,
hostname,
port,
fselocation as datadir
from gp_segment_configuration a,pg_filespace_entry b,pg_filespace c
where a.dbid=b.fsedbid and b.fsefsoid=c.oid and c.fsname='pg_system';

```

## 4.3 常用数据字典

下面介绍几个 PostgreSQL 和 Greenplum 常用的数据字典，这几个数据字典都可以在 PostgreSQL 的文档中找到相应的解释。这里再强调一遍，这些数据字典在数据库里面非常重要。

### 4.3.1 pg\_class

`pg_class` 可以说是数据字典最重要的一个表了，它保存着所有表、视图、序列、索引的原数据信息，每一个 DDL/DML 操作都必须跟这个表发生联系，表 4-7 就是其表结构。

表 4-7 `pg_class` 表结构

字段	类型	引用	字段说明
oid	oid		<code>pg_class</code> 中的唯一标识
relname	name		表、索引、视图等的名称
relnamespace	oid	<code>pg_namespace.oid</code>	包含这个关系的名称空间（模式）的 oid
reltype	oid	<code>pg_type.oid</code>	对应这个表的行类型的数据类型（索引为零，它们没有 <code>pg_type</code> 记录）
relowner	oid	<code>pg_authid.oid</code>	关系所有者
relam	oid	<code>pg_am.oid</code>	如果行是索引，那么就是所用的访问模式（B-tree、hash 等）
relfilenode	oid		这个关系在磁盘上的文件的名称，如果没有则为 0
reltablespace	oid	<code>pg_tablespace.oid</code>	这个关系存储所在的表空间。如果为零，则意味着使用该数据库的默认表空间。如果关系在磁盘上没有文件，则这个字段没有什么意义
relpages	int4		此表在磁盘中以页（大小为 <code>BLCKSZ</code> ）的大小，在 Greenplum 一页默认为 32KB。这个字段只是规划器使用的一个近似值， <code>VACUUM</code> 、 <code>ANALYZE</code> 和几个 DDL 命令会触发这个字段的更新，比如 <code>CREATE INDEX</code>
reltuples	float4		表中行的数目。只是规划器使用的一个估计值， <code>VACUUM</code> 、 <code>ANALYZE</code> 和几个 DDL 会触发这个字段的更新，比如 <code>CREATE INDEX</code>
reltoastrelid	oid	<code>pg_class.oid</code>	与此表关联的 TOAST 表的 oid，如果没有则为 0。TOAST 表在一个从属表里“离线”存储大字段

(续)

字段	类 型	引 用	字段说明
reltoastidxit	oid	pg_class.oid	对应 TOAST 表的索引的 oid，如果不是 TOAST 表，则为 0
relasegidxit	oid	pg_class.oid	这个字段在 Greenplum 3.4 之后的版本中废弃了，记录此表关联的 appendonly 表的 oid，如果非 appendonly 表，则为 0
relasegrelid	oid	pg_class.oid	这个字段在 Greenplum 3.4 之后的版本中废弃了，记录 appendonly 表索引的 oid，如果非 appendonly 表，则为 0
relhasindex	bool		如果它是一个表且至少有（或者最近有过）一个索引，则为真。它是由 CREATE INDEX 设置的，不过 DROP INDEX 不会立即将它清除。如果 VACUUM 出现一个表没有索引，那么它将清理 relhasindex
relisshared	bool		如果该表在整个集群中由所有数据库共享，则为真。只有某些系统表（比如 pg_database）是共享的
relkind	char		r = 普通表或者 appendonly 表，i = 索引，s = 序列，v = 视图，c = 复合类型，t = TOAST 表，o = 内存 appendonly 文件，u = 未列入目录的临时表
relstorage	char		表中的物理存储类别。a=appendonly 表，h=堆表，v=虚拟的表（如视图等），x=外部表
relnatts	int2		关系中用户字段数目（除了系统字段以外）。在 pg_attribute 中肯定有相同数目对应行。见 pg_attribute.attnum
relchecks	int2		表中的检查约束的数目；参阅 pg_constraint 表
reltriggers	int2		表中的触发器的数目；参阅 pg_trigger 表
relukeys	int2		未使用（不是唯一值的数目）
relfkeys	int2		未使用（不是表中外键的数目）
relrefs	int2		未使用
relhasoids	bool		如果关系中每行都生成一个 oid，则为真
relhaskey	bool		如果这个表有一个（或者曾经有一个）主键，则为真
relhasrules	bool		如果表有规则，就为真；参阅 pg_rewrite 表
relhassubclass	bool		如果表有（或者曾经有）任何继承的子表，则为真
relfrozenxit	xid		该表中所有在这个之前的事务 ID 已经被一个固定的('frozen') 事务 ID 替换。这用于跟踪该表是否需要为防止事务 ID 重叠或允许收缩 pg_clog 而进行清理。如果该关系不是表，则为零（InvalidTransactionId）
relacl	aclitem[]		访问权限
reloptions	text[]		访问方法特定的选项，使用 'keyword=value' 格式的字符串

建在这个表上的索引：

Indexes:

```
"pg_class_oid_index" UNIQUE, btree (oid)
"pg_class_relname_nsp_index" UNIQUE, btree (relname, relnamespace)
```

了解这些基础数据字典的索引结构，对优化数据字典查询速度有很大的帮助，例如下文

中介绍了优化 pg\_partitions 就充分利用了索引，大大提升了查询速度。

权限控制对于一个完善的数据表是必不可少的，对于表、视图来说，pg\_class 中有一个字段 relacl 用于保存了权限信息，如下：

```
testDB=# select relacl from pg_class where relname='cxfaf3';
          relacl
-----
{gpadmin=arwdxt/gpadmin,role_aquery=arwdxt/gpadmin}
(1 row)
```

具体解释如下：

```
=xxxx -- 赋予 PUBLIC 的权限
uname=xxxx -- 赋予一个用户的权限
group gname=xxxx -- 赋予一个组的权限
r -- SELECT ("读")
      w -- UPDATE ("写")
      a -- INSERT ("追加")
      d -- DELETE
      x -- REFERENCES
      t -- TRIGGER
      X -- EXECUTE
      U -- USAGE
      C -- CREATE
      c -- CONNECT
      T -- TEMPORARY
arwdxt -- ALL PRIVILEGES (用于表)
* -- 前面权限的授权选项
```

查 relacl 这个字段有点不方便，不过利用数据库中的很多函数可以方便一些，如下：

```
testDB=# \df *privilege*
              List of functions
 Schema |       Name        | Result data type | Argument data types
-----+-----+-----+-----+-----+
 pg_catalog | has_database_privilege | boolean           | name, oid, text
 pg_catalog | has_database_privilege | boolean           | name, text, text
 pg_catalog | has_database_privilege | boolean           | oid, oid, text
 pg_catalog | has_database_privilege | boolean           | oid, text
 pg_catalog | has_database_privilege | boolean           | oid, text, text
 pg_catalog | has_database_privilege | boolean           | text, text
 pg_catalog | has_function_privilege | boolean           | name, oid, text
 pg_catalog | has_function_privilege | boolean           | name, text, text
 pg_catalog | has_function_privilege | boolean           | oid, oid, text
...
...
```

示例：查询 role\_aquery 用户是否具有访问 public.cxfaf3 表的 select 权限。结果为 't' 表示有这个权限，结果为 'f' 表示没有这个权限。

```
testDB=# select has_table_privilege('role_aquery','public.cxfaf3','select');
```

```

has_table_privilege
-----
t
(1 row)
testDB=# select has_table_privilege('role_dhw','public.cxfaf3','select');
has_table_privilege
-----
f
(1 row)

```

### 4.3.2 pg\_attribute

前面介绍了记录表的 pg\_class，现在介绍记录字段内容的 pg\_attribute，该表的表结构及说明如表 4-8 所示。

表 4-8 pg\_attribute 表结构

字 段	类 型	引 用	字段说明
attrelid	oid	pg_class.oid	此字段所属的表
attname	name		字段名称
atttypid	oid	pg_type.oid	这个字段的数据类型
attstattarget	int4		控制 ANALYZE 为这个字段积累的统计细节的级别。零值表示不收集统计信息，负数表示使用系统默认的统计对象，正数值的确切信息是和数据类型相关的。对于标量数据类型，attstattarget 既是要收集的“最常用数值”的目标数目，也是要创建的柱状图的目标数量
attlen	int2		是本字段类型的 pg_type.typlen 的副本
attnum	int2		字段数目。普通字段是从 1 开始计数的。系统字段（比如 oid）有（任意）正数个
attndims	int4		如果该字段是数组，那么是维数，否则是 0。目前，一个数组的维数并未强制，因此任何非零值都表示“这是一个数组”
attcacheoff	int4		在磁盘上的时候总是 -1，但是如果加载内存的行描述器，记录的是行中字段的偏移量
atttypmod	int4		记录创建新表时支持的类型特定的数据（比如一个 varchar 字段的最大长度）。它传递给类型相关的输入和长度转换函数作第三个参数。对于那些不需要 atttypmod 的类型其值通常为 -1
attbyval	bool		这个字段类型的 pg_type.typbyval 的副本
attstorage	char		这个字段类型的 pg_type.typstorage 的副本。对于可压缩的数据类型（TOAST），这个字段可以在字段创建之后改变，以便于控制存储策略
attalign	char		这个字段类型的 pg_type.typalign 的副本
attnotnull	bool		这代表一个非空约束。可以改变这个字段以打开或关闭这个约束
atthasdef	bool		这个字段有一个默认值，此时它对应 pg_attrdef 表中实际定义此值的记录
attisdropped	bool		这个字段已经被删除了，不再有效。一个已经删除的字段物理上仍然存在表中，但是会被分析器忽略，因此不能再通过 SQL 访问

(续)

字段	类 型	引 用	字段说明
attislocal	bool		这个字段是局部定义在关系中的。注意一个字段可以同时是局部定义和继承的
attinhcount	int4		这个字段所拥有的直接祖先的个数。如果一个字段的祖先个数非零，那么它就不能被删除或重命名

查看 pg\_attribute，我们会发现，同一个表在 pg\_attribute 中的记录数会比实际表的字段数多，这是因为表中有很多的隐藏字段，这些隐藏字段如表 4-9 所示。

表 4-9 每个表的隐藏字段描述

编号(attnum)	字段名	字段说明
-8	gp_segment_id	标记数据是保存在哪个 Segment 上。从主节点上查询这个字段，字段内容即为该 Segment 的 content ID
-7	tableoid	包含本行的表的 oid。这个字段对那些从继承层次中选取的查询特别有用，因为如果没有它的话，我们就很难说明一行来自哪个独立的表
-6	cmax	删除事务内部的命令标识符
-5	xmax	删除事务的标识(事务 ID)。在一个可见行版本中，这个字段有可能是非零，这通常意味着删除事务还没有提交，或者是一个删除的企图被回滚掉了
-4	cmin	插入事务内部的命令标识(从零开始)
-3	xmin	插入该行版本的事务标识(事务 ID)。注意：在这个环境中，一个行版本是一行的一个状态；一行的每次更新都为同一个逻辑行创建一个新的行版本
-2	oid	行对象标识符(对象 ID)。这个字段只有在创建表的时候使用了 WITH OIDS 或配置参数 default_with_oids 的值为真时出现。这个字段的类型是 oid(和字段同名)。这个字段对于普通表来说一般是没有的
-1	ctid	一个行版本在它所处的表内的物理位置。注意，尽管 ctid 可以用于非常快速地定位行版本，但每次 VACUUM FULL 之后，一个行的 ctid 都会被更新或移动。因此 ctid 不能作为长期的行标识符。应该使用 oid，或者最好是用户定义的序列号，来标识一个逻辑行

### 4.3.3 gp\_distribution\_policy

表的分布键保存在 gp\_distribution\_policy 表中，其表结构如表 4-10 所示。

表 4-10 gp\_distribution\_policy 表结构

字段名	类 型	引 用	字段说明
localoid	oid	pg_class.oid	表的 oid
ocaloid	smallint[]	pg_attribute.attnum	保存分布键对应的 attnum 的一个数组

#### 4.3.4 pg\_statistic 和 pg\_stats

数据库中表的统计信息保存在 pg\_statistic 中，表中的记录是由 ANALYZE 创建的，并且随后被查询规划器使用。注意所有统计信息天生都是近似的数值。

这里提到的表上面有一个视图 pg\_stats，可以方便我们查看 pg\_statistic 的内容。这个视图的数据比 pg\_statistic 好理解，其结构如表 4-11 所示。

表 4-11 pg\_stats 视图结构

字 段	类 型	引 用	字段说明
schemaname	name	pg_namespace.nspname	包含此表的模式名称
tablename	name	pg_class.relname	表的名称
attname	name	pg_attribute.attname	这一行描述的字段的名称
null_frac	real		记录中字段为空的百分比
avg_width	integer		记录以字节记的平均宽度
n_distinct	real		字段里唯一的非 NULL 数据值的数目。该字段的值为一个大于零的数值表示独立数值的实际数目。该字段值为一个小于零的数值表示表中行数的分数的负数（比如，一个字段的数值平均出现概率为 1/2，那么可以表示为 stadistinct = -0.5）。该字段值为零值表示独立数值的数目未知
most_common_vals	anyarray		一个字段中最常用数值的列表。如果看上去没有数值比其他数值更常见，则为 NULL
most_common_freqs	real[]		一个最常用数值的频率列表，也就是说，每个出现的次数除以行数。如果 most_common_vals 是 NULL，则这个字段为 NULL
histogram_bounds	anyarray		一个数值的列表，它把字段的数值分成几组大致相同的热门组。如果字段数据类型没有 < 操作符或者 most_common_vals 列表代表了整个分布性，则这个字段为 NULL
correlation	real		统计与字段值的物理行序和逻辑行序有关。它的范围从 -1 到 +1。在数值接近 -1 或者 +1 的时候，在字段上的索引扫描将被认为比它接近零的时候开销更少，因为减少了对磁盘的随机访问。如果字段数据类型没有 < 操作符，那么这个字段为 NULL

#### 4.4 分区表信息

Greenplum 支持对表进行分区。本节将讲述在 Greenplum 中如何实现分区表，以及与分区表相关的数据字典。

#### 4.4.1 如何实现分区表

分区的意思是把逻辑上的一个大表分割成物理上的几块。Greenplum 中分区表的实现基本上是与 PostgreSQL 中实现的原理一样，都是通过表继承、规则、约束来实现的。PostgreSQL 的分区表实现可以查看 PostgreSQL8.2.3 中文档的 5.9 节。

下面是 PostgreSQL 中分区表的建立步骤：

1) 创建“主表”，所有分区都从它继承。

这个表中没有数据，不要在这个表上定义任何检查约束，除非希望约束同样也适用于所有分区。同样，在主表上定义任何索引或唯一约束也没有意义。

2) 创建几个“子表”，每个都从主表上继承。通常，这些表不会增加任何字段。

我们将把子表称为分区，尽管它们就是普通的 PostgreSQL 表。

3) 为分区表增加约束，定义每个分区允许的键值。

典型的例子是：

```
CHECK ( x = 1 )
CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ) )
CHECK ( outletID >= 100 AND outletID < 200 )
```

确保这些约束在不同的分区中不会有重叠的键值。一个常见的错误是设置下面这样的范围：

```
CHECK ( outletID BETWEEN 100 AND 200 )
CHECK ( outletID BETWEEN 200 AND 300 )
```

这样做是错误的，因为它没说清楚键值 200 属于哪个范围，应该明确指定 200 所属的区间。

注意在范围和列表分区的语法方面没有什么区别，这些术语只是用于描述的。

4) 对于每个分区，在关键字字段上创建一个索引，以及其他想创建的索引。关键字字段索引并非必需的，但是在大多数情况下它是很有帮助的。如果希望关键字值是唯一的，那么应该总是为每个分区创建一个唯一或主键约束。

5) 另外，定义一个规则或触发器，把对主表的修改重定向到合适的分区表。

对于 Greenplum 来说，分区表的实现原理与上面介绍的一样，只不过 Greenplum 对分区表进行了一个更好的封装，使用户使用起来更加方便，可以通过 `create table` 直接建立分区表，可以使用 `alter table` 等对分区表进行操作。

跟 PostgreSQL 一样，分区表中的子表也是通过对父表的继承得来的，这些继承关系是放在 `pg_inherits` 这个数据字典中的。

#### 4.4.2 pg\_partition

一个表是否是分区表保存在 `pg_partition` 中，如果一个表是分区表（不包括子分区），则

对应有一条记录在这个数据字典中。pg\_partition 表结构如表 4-12 所示。

表 4-12 pg\_partition 表结构

字段名	类 型	引 用	字段说明
oid	oid		唯一标识分区表的 oid
parrelid	oid	pg_class.oid	表对应在 pg_class 中的 oid
parkind	char		分区类型, R=Range 模式, L=List 模式
parlevel	smallint		分区的层级, 0 表示分区表的父表, 1 表示第一层分区, 2 表示第二层分区
paristemplate	boolean		该分区是否为子分区模板或者是确定层级的分区
parnatts	smallint	pg_attribute.oid	分区键的个数
paratts	int2vector		分区键中对应 pg_attribute 中的 attnum 的列表
parclass	oidvector	pg_opclass.oid	跟 pg_opclass 关联, 定义分区键的操作方法

该表有 3 个索引：

```
Indexes:
"pg_partition_oid_index" UNIQUE, btree (oid)
"pg_partition_parrelid_index" btree (parrelid)
"pg_partition_parrelid_parlevel_istemplate_index" btree (parrelid, parlevel,
paristemplate)
```

如果想查询一个表是否是分区表，只要将 pg\_partition 与 pg\_class 关联，然后执行 count 即可，如果这个表中有数据，则为分区表，否则不是分区表。

```
testDB=# select count(*) from pg_partition where parrelid='public.
cxa3'::regclass;
count
-----
1
(1 row)
```

#### 4.4.3 pg\_partition\_rule

分区表的分区规则保存在 pg\_partition\_rule 中。在这个表中，我们可以找到一个分区表对应的子表有哪些及分区规则等信息，如表 4-13 所示。

表 4-13 pg\_partition\_rule 表结构

字段名	类 型	引 用	字段说明
oid	oid		唯一标识的 oid
paroid	oid	pg_partition.oid	与 pg_partition 关联，对应其 oid
parchildrelid	oid	pg_class.oid	分区表子表对应 pg_class 的 oid
parparentrule	oid	pg_partition_rule.paroid	父表对应的分区的 paroid

(续)

字段名	类型	引 用	字段说明
parname	name		子分区名
parisdefault	boolean		该分区是否为默认分区
parruleord	smallint		对于 range 分区，对应的分区序号
parrangestartincl	boolean		对于 range 分区，该分区是否包括起始值
parrangeendincl	boolean		对于 range 分区，该分区是否包括结束值
parrangestart	text		起始值
parrangeend	text		结束值
parrangeevery	text		对于 range 分区，每一个分区的间隔值
parlistvalues	text		对于 list 分区，该分区对应的数值
parreloptions	text[]		描述特定分区的存储特性

#### 4.4.4 pg\_partitions 视图及其优化

在 Greenplum 中，定义了一个 pg\_partitions 的视图，方便对分区表进行查看，这个视图的定义非常复杂，考虑了多重分区的情况，对很多数据字典做了连接和内连接。因此，当数据字典很大的时候，查询这个视图的效率极差，不能够很好地利用索引，查询采用的是全表扫描。下面定义这个函数来代替 pg\_partitions 视图，充分利用索引，查询是通过 regclass 减少表的关联，大大提高查询的效率。

```
create view public.v_pg_partitions
as
SELECT pp.parrelid tableoid,pr1.parchildrelid,pr1.parname as partitionname,
CASE
    WHEN pp.parkind = 'h'::"char" THEN 'hash'::text
    WHEN pp.parkind = 'r'::"char" THEN 'range'::text
    WHEN pp.parkind = 'l'::"char" THEN 'list'::text
    ELSE NULL::text
END AS partitiontype,
case when pg_get_expr(pr1.parrangeend, pr1.parchildrelid) = '' then
pg_get_expr(pr1.parlistvalues, pr1.parchildrelid) else pg_get_expr(pr1.
parrangeend, pr1.parchildrelid) end as HIGH_VALUE ,
    pg_get_partition_rule_def(pr1.oid, true) AS partitionboundary,
    pr1.parruleord AS partitionposition
FROM pg_partition pp, pg_partition_rule pr1
WHERE pp.paristemplate = false AND pr1.paroid = pp.oid ;
```

在查询时通过 ::regclass 来查询表信息，从而避免关联 pg\_class。

```
SELECT * from public.v_pg_partitions
where tableoid= 'schemaname.tablename'::regclass
order by partitionposition;
```

## 4.5 自定义类型以及类型转换

在 Greenplum 中，我们经常使用 cast 函数或 ::type 进行类型转换，究竟哪两种类型之间是可以转换的，哪两种类型之间不能转换，转换的规则是什么，这些都在 pg\_cast 中定义了。pg\_cast 表结构如表 4-14 所示。

表 4-14 pg\_cast 表结构

字段	类 型	引 用	字段说明
castsource	oid	pg_type.oid	源数据类型的 oid
casttarget	oid	pg_type.oid	目标数据类型的 oid
castfunc	oid	pg_proc.oid	用于执行这个转换的函数的 oid。如果该数据类型是二进制兼容的，那么这个字段为零（也就是说，不需要运行时的操作来执行转换）
castcontext	char		标识这个转换可以在什么环境中调用。e 表示只能进行明确的转换（使用 CAST 或 :: 语法）。a 表示在赋值给目标字段的时候隐含调用，也可以明确调用。i 表示在表达式中隐含调用，当然也包括其他情况

要想知道 text 类型到 date 类型的转换是用了哪个函数，可以这么查：

```
testDB=# select castfunc::regprocedure from pg_cast where castsource='text'
::regtype and casttarget='date'::regtype;
castfunc
-----
date(text)
(1 row)

testDB=# select '20110302'::date;
      date
-----
2011-03-02
(1 row)

testDB=# select date('20110302');
      date
-----
2011-03-02
(1 row)
```

可以看出，cast('20110302' as date) 和 '20110302'::date 其实都调用了 date('20110302') 函数进行类型转换。

是否可以自定义类型转换呢？答案是肯定的。

比如，Greenplum 默认的类型转换中，是没有 regclass 类型到 text 类型的转换的：

```
testDB=# select 1259::regclass::text;
ERROR:  cannot cast type regclass to text
LINE 1: select 1259::regclass::text;
```

先创建一个类型转换函数：

```
CREATE or replace FUNCTION regclass2text(a regclass)
    RETURNS text
AS $$

    return a;
$$ LANGUAGE plpythonu;
```

然后定义一个 cast 类型转换规则：

```
testDB=# create cast(regclass as text) with function regclass2text(a regclass);
CREATE CAST
```

这样就定义好了一个类型转换，效果如下：

```
testDB=# select 1259::regclass::text;
      text
-----
 pg_class
(1 row)

testDB=# select cast(1259::regclass as text);
      text
-----
 pg_class
(1 row)
```

## 4.6 主、备节点同步的相关数据字典

在 Greenplum 4.x 版本之后，数据库主、备节点之间的同步通过基于数据文件的物理备份来实现，这与 Greenplum 3.x 的逻辑备份有很大的区别。

在 Greenplum 4.x 中，分别由表 4-15 中的 5 张数据字典表来保存基于数据文件的备份信息。这些数据字典都是用于在主节点与备节点间基于文件备份的同步信息。

表 4-15 主、备节点同步的相关数据字典表

表 名	描 述
gp_persistent_database_node	数据库的同步信息
gp_persistent_filespace_node	文件空间的同步信息
gp_persistent_relation_node	文件的同步信息
gp_persistent_tablespace_node	表空间的同步信息
gp_relation_node	表、视图、索引等的同步信息

在这几张表中，数据量最大、最重要的表应该是 gp\_persistent\_relation\_node 和 gp\_relation\_node。这些数据字典在每一个节点中都有，如果主节点和备节点处于完全同步的状态，则主节点和备节点对应的这几张数据字典表的内容应该是一模一样的。

## 4.7 数据字典应用示例

前面介绍了很多数据字典的一些基础知识，下面我们通过几个示例来演示一些好用的函数，它们通过数据字典来获取我们需要的信息。

### 4.7.1 获取表的字段信息

表名放在 pg\_class 中，schema 名放在 pg\_namespace 中，字段信息放在 pg\_attribute 中。一般关联这 3 张表：

```
SELECT a.attname,pg_catalog.format_type(a.atttypid, a.atttypmod) AS data_type
  FROM pg_catalog.pg_attribute a,
       (
           SELECT c.oid
             FROM pg_catalog.pg_class c
            LEFT JOIN pg_catalog.pg_namespace n
               ON n.oid = c.relnamespace
              WHERE c.relname = 'pg_class'
                AND n.nspname = 'pg_catalog'
        ) b
 WHERE a.attrelid = b.oid
   AND a.attnum > 0
   AND NOT a.attisdropped ORDER BY a.attnum;
```

使用 regclass 就会简化很多：

```
SELECT a.attname,pg_catalog.format_type(a.atttypid, a.atttypmod) AS data_type
  FROM pg_catalog.pg_attribute a
 WHERE a.attrelid = 'pg_catalog.pg_class'::regclass
   AND a.attnum > 0
   AND NOT a.attisdropped ORDER BY a.attnum;
```

其实 regclass 就是一个类型，oid 或 text 到 regclass 有一个类型转换，与多表关联不一样。在多数据字典表关联的情况下，如果表不存在，会返回空记录，不会报错，如果采用了 regclass，则会报错，所以在不确定表是否存在的情况下，慎用 regclass。

### 4.7.2 获取表的分布键

gp\_distribution\_policy 是记录分布键信息的数据字典，localoid 与 pg\_class 的 oid 关联。attrnums 是一个数组，记录字段的 attnum，与 pg\_attribute 中的 attnum 关联。

```
testDB=# create table cxfaf2 ( a int ,b int ,c int ,d int ) distributed by (c,a);
testDB=# select * from gp_distribution_policy where localoid='cxfaf2'::regclass;
 localoid | attrnums
-----+-----
 334868 | {3,1}
(1 row)
```

这样就可以关联 pg\_attribute 来获取分布键了：

```
select a.attrnums[i.i],b.attname,a.localoid::regclass
  from gp_distribution_policy a,
       (select generate_series(1,10)) i(i),
       pg_attribute b
 where a.attrnums[i.i] is not null
   and a.localoid=b.attrelid
   and a.attrnums[i.i]=b.attnum
   and a.localoid='public.cxfa2'::regclass
  order by i.i;
```

结果如下：

attrnums	attname	localoid
3	c	cxfa2
1	a	cxfa2

(2 rows)

### 4.7.3 获取一个视图的定义

在数据库中，有一个函数（pg\_get\_viewdef），可以直接获取视图的定义，函数的使用方法如下：

```
testDB=# \df pg_get_viewdef
          List of functions
 Schema |      Name      | Result data type | Argument data types
-----+-----+-----+-----+
 pg_catalog | pg_get_viewdef | text | oid
 pg_catalog | pg_get_viewdef | text | oid, boolean
 pg_catalog | pg_get_viewdef | text | text
 pg_catalog | pg_get_viewdef | text | text, boolean
(4 rows)
```

使用这个系统函数可以获取视图的定义，可以传入表的 oid 或表名，第二个参数表示是否格式化输出，默认不格式化输出。

```
testDB=# create table cxfa( a int) distributed by (a);
CREATE TABLE
testDB=# create view v_cxfa as select * from cxfa;
CREATE VIEW
testDB=# select pg_get_viewdef('v_cxfa',true);
pg_get_viewdef
-----
SELECT cxfa.a
      FROM cxfa;
(1 row)
```

其实这个函数是获取数据字典 pg\_rewrite (存储为表和视图定义的重写规则), 将规则重新还原出 SQL 语句展现给我们。可以通过下面语句去查询数据库保存的重写规则, 图 4-1 是一个简单视图的规则定义。

```
select ev.action from pg_rewrite where ev.class='v' cxfa':::reaclass;
```

```
((?QUERY-commandtype 1 &querysource 0 &scanning true &utilization > 1 &resolution 0 &intel  
++ &hashassoc false &hashdrivemode False &rtablename False &alias alias &aliasalias  
name "OLD" &colnames => ) &ref &alias :aliasname "OLD" &colnames ("a") &createkind 0 &relid 262812  
&inh false &infroncl false &requiredPerms 0 &checkUser 0 &forceRandom false &pseudocols  
(>) &createkind 0 &relid 262812 &inh false &infroncl 0 &requiredPerms 0 &checkUser 0 &colnames  
("a") &createkind 0 &relid 262812 &inh false &infroncl 0 &requiredPerms 0 &checkUser 0 &forceRandom  
false &pseudocols (>) &RTT alias &ref &alias :aliasname exfa &colnames ("a") &rtt  
relid 82026 &inh true &infroncl true &requiredPerms 2 &checkUser 0 &forceRandom f  
&createkind 0 &relid 82026 &inh true &infroncl 0 &requiredPerms 0 &checkUser 0 &forceRandom  
true &list [&TARGETSYSTEM &exp (&VAR &varobj :varattinfo 1 &partype 23 &varitype 1 &varlevel  
enroll 0 &varattinfo 1) &resno 1 &resname &ressourcegroup 0 &resrigtbl 262806 &resrigcol 1 &  
resjunktbl 0 &resjunkcol 0 &groupLabel > &havingval > &window &lae > &distinct  
&sortOrder 0 &sortCol 0 &sortOffset 0 &sortCount 0 &sortMark 0 &sortOperations <> &sortRelatio  
ns 0 > &result_partitions <> &resultMessages <> &returningList >> )  
(1 row)
```

图 4-1 视图定义在数据库中的规则定义

与 pg\_get\_viewdef 类似的函数还有很多，如图 4-2 所示。这些函数的原理都类似，将数据字典的重写规则翻译为 SQL 语句。

List of functions				
Schema	Name	Result data type	Argument data types	
pg_catalog	pg_get_constraintdef	text	oid	
pg_catalog	pg_set_constraintdef	text	oid, boolean	
pg_catalog	pg_set_indexdef	text	oid	
pg_catalog	pg_set_indexdef	text	oid, integer, boolean	
pg_catalog	pg_get_partition_def	text	oid	
pg_catalog	pg_set_partition_def	text	oid, boolean	
pg_catalog	pg_get_partition_rule_def	text	oid	
pg_catalog	pg_set_partition_rule_def	text	oid, boolean	
pg_catalog	pg_set_indexeddef	text	oid	
pg_catalog	pg_set_indexdef	text	oid, boolean	
pg_catalog	pg_set_irrefdef	text	oid	
pg_catalog	pg_get_viewdef	text	oid	
pg_catalog	pg_set_viewdef	text	oid, boolean	
pg_catalog	pg_get_viewdef	text	text	
pg_catalog	pg_set_viewdef	text	text, boolean	

图 4-2 获取规则的定义函数



触发器在 Greenplum 里面是不支持的。

#### 4.7.4 查询 comment (备注信息)

comment 信息是放在表 pg\_description 中的。pg\_description 表结构如表 4-16 所示。

表 4-16 pg\_description 表结构

字段名	类 型	引 用	字段说明
objoid	oid	任意 oid 属性	这条描述所描述对象的 oid
classoid	oid	pg_class.oid	这个对象出现的系统表的 oid
objsubid	int4		对于一个表字段的注释, 它是字段号 (objoid 和 classoid 指向表自身)。 对于其他对象类型, 它是零
description	text		作为该对象的描述的任意文本

查询表上的 comment 信息：

```
testDB=# select COALESCE(description,'') as comment from pg_description where
objoid='cxf'a'::regclass and objsubid=0;
      comment
-----
 a table created by scutshuxue.chenxf
(1 row)
```

查询表中字段的 comment 信息：

```
testDB=# select b.attname as columnname, COALESCE(a.description,'') as comment
testDB-#   from pg_catalog.pg_description a,pg_catalog.pg_attribute b
testDB-#   where objoid='cxf'a'::regclass
testDB-#     and a.objoid=b.attrelid
testDB-#     and a.objsubid=b.attnum;
      columnname |      comment
-----+-----
 a          | column a of table cxf
(1 row)
```

#### 4.7.5 获取数据库建表语句

前面介绍了各个数据字典，通过对这些数据字典的了解，我们可以清楚地知道表的元数据都分别放在哪些表中，就可以利用这些元数据来实现一些特殊的功能，比如下面介绍的获取建表语句，这个函数在日常使用和将表迁移到另外一个数据库中是非常有用的。

`get_create_sql` 是用于获取表和视图的 DDL 语句，不支持外部表，建表语句包括以下内容：

- 1) 字段信息。
- 2) 索引信息。
- 3) 分区信息（主要考虑到性能，目前只支持单一分区键，一层分区）。
- 4) comment 信息。
- 5) distributed key。
- 6) 是否压缩、列存储、appendonly。
- 7) 只有一个参数，tablename 为 schemaname.tablename，输出为一个 text 文本。

下面是这个函数的代码：

```
CREATE or replace FUNCTION public.get_create_sql(tablename text)
RETURNS text
AS $$

try:
    table_name = tablename.lower().split('.')[1]
    table_schema = tablename.lower().split('.')[0]
except (IndexError):
```

```

        return 'Please input "tableschema.table_name" !'
# 获取表的 oid
    get_table_oid = " select oid,reloptions,relkind from pg_class where
oid='%s'::regclass"%(tablename)
    try:
        rv_oid = plpy.execute(get_table_oid, 5)
        if not rv_oid:
            return 'Did not find any relation named "' + tablename + '".'
    except (Error):
        return 'Did not find any relation named "' + tablename + '".'

    table_oid = rv_oid[0]['oid']
    rv_reloptions = rv_oid[0]['reloptions']
    rv_relkind=rv_oid[0]['relkind']
    create_sql="";
    table_kind='table';
# 如果该表不是一般表或视图，则报错
    if rv_relkind!='r' and rv_relkind!='v':
        plpy.error('%s is not table or view'%(tablename));
    elif rv_relkind=='v':
        get_view_def="select pg_get_viewdef(%s,'t') as viewdef;"%(table_oid)
        rv_viewdef=plpy.execute(get_view_def);
        create_sql = 'create view %s as \n'%(tablename)
        create_sql += rv_viewdef[0]['viewdef']+ '\n';
        table_kind='view'
    else:
#get column name and column type - 获取字段名、字段类型和默认值
        get_columns = "SELECT a.attname, pg_catalog.format_type(a.atttypid,
a.atttypmod), \
                           (SELECT substring(pg_catalog.pg_get_expr(d.
adbin, d.adrelid) for 128) \
                           FROM pg_catalog.pg_attrdef d WHERE
d.adrelid = a.attrelid AND d.adnum = a.attnum AND a.attisdropped) as default,\n
                           a.attnotnull as isnull \
                           FROM pg_catalog.pg_attribute a \
                           WHERE a.attrelid = %s AND a.attnum > 0 AND NOT
a.attisdropped \
                           ORDER BY a.attnum;" %(table_oid);
        rv_columns = plpy.execute(get_columns)

#get distributed key -- 获取分布键
        get_table_distribution1 = "SELECT attrnums FROM pg_catalog.gp_
distribution_policy t WHERE localoid = '" + table_oid + "' "
        rv_distribution1 = plpy.execute(get_table_distribution1, 500)
        rv_distribution2 = ''
        if rv_distribution1 and rv_distribution1[0]['attrnums']:
            get_table_distribution2 = "SELECT attname FROM pg_attribute
WHERE attrelid = '" + table_oid + "' AND attnum in (" + str(rv_distribution1[0]
['attrnums']).strip('{}').strip(')').strip('[').strip(']') + ")"
            rv_distribution2 = plpy.execute(get_table_distribution2, 500)

```

```

#get index define
create_sql = 'create table %s(\n'%(tablename)
get_index = "select pg_get_indexdef(indexrelid) AS indexdef from pg_
index where indrelid=%s"%(table_oid);
rv_index = plpy.execute(get_index);

#get partition info ——获取分区信息
get_parinfo1 = "select attname as columnname from pg_attribute
where attnum = (select paratts[0] from pg_partition where parrelid=%s) and
attrelid=%s;%(table_oid,table_oid);
get_parinfo2 ="""
SELECT pp.parrelid,pr1.parchildrelid,
CASE
    WHEN pp.parkind = 'h'::"char" THEN 'hash'::text
    WHEN pp.parkind = 'r'::"char" THEN 'range'::text
    WHEN pp.parkind = 'l'::"char" THEN 'list'::text
    ELSE NULL::text
END AS partitionotype,
pg_get_partition_rule_def(pr1.oid, true) AS partitionboundary
FROM pg_partition pp, pg_partition_rule pr1
WHERE pp.paristemplate = false AND pp.parrelid = %s AND pr1.paroid
= pp.oid
order by pr1.parname;
"""%(table_oid);
v_par_parent = plpy.execute(get_parinfo1);
v_par_info = plpy.execute(get_parinfo2);
max_column_len = 10
max_type_len = 4
max_modifiers_len = 4
max_default_len=4
for i in rv_columns:
    if i['attname']:
        if max_column_len < i['attname'].__len__(): max_
column_len = i['attname'].__len__()
        if i['format_type']:
            if max_type_len < i['format_type'].__len__(): max_type_
len = i['format_type'].__len__()
            if i['default']:
                if max_type_len < i['default'].__len__(): max_default_
len = i['default'].__len__()
first = True
# 拼接字段内容
for i in rv_columns:
    if first==True:
        split_char=' ';
        first=False
    else :
        split_char=',';
if i['attname']:

```

```

        create_sql += " " + split_char + i['attnname'].ljust(max_
column_len + 6) + ''
        else:
            create_sql += "" + split_char + ' '.ljust(max_column_len + 6)
        if i['format_type']:
            create_sql += ' ' + i['format_type'].ljust(max_type_len + 2)
        else:
            create_sql += ' ' + ' '.ljust(max_type_len + 2)
        if i['isnull'] and i['isnull']:
            create_sql += ' ' + ' not null '.ljust(8)
        if i['default']:
            create_sql += ' default ' + i['default'].ljust(max_default_
len + 6)
        create_sql += "\n"
create_sql += ")"
# 拼接 with 语句的内容
if rv_reloptions:

    create_sql+="with("+str(rv_reloptions).strip('{}').strip(')').
strip('[').strip(']') +"")\n"
    if rv_distribution2:
        create_sql += 'Distributed by ('
        for i in rv_distribution2:
            create_sql +=      i['attnname'] + ','
        create_sql = create_sql.strip(',') + ')'
    elif rv_distribution1:
        create_sql += 'Distributed randomly\n'
    if v_par_parent:
        partitiontype = v_par_info[0]['partitiontype'];
        create_sql+='\nPARTITION BY ' + partitiontype + "("+v_par_
parent[0]['columnname']+")\n(\n";
        for i in v_par_info:
            create_sql+="      "+i['partitionboundary']+','\n';
        create_sql = create_sql.strip(',\n');
        create_sql+="\n)"
    create_sql += ";\n\n"
    # 拼接索引信息
    for i in rv_index:
        create_sql += i['indexdef']+';\n'

#get comment, 获取 comment 信息
get_table_comment="select 'comment on %s %s is '''|| COALESCE
(description,'')||''' as comment from pg_description where objoid=%s and
objsubid=0;%(table_kind,tablename,table_oid)
get_column_comment="select 'comment on column %s.'||b.attnname ||' is '''|| COALESCE(a.description,'') ||''' ' as comment from pg_catalog.pg_description
a,pg_catalog.pg_attribute b where objoid=%s and a.objoid=b.attrelid and
a.objsubid=b.attnum;"%(tablename,table_oid)
rv_table_comment=plpy.execute(get_table_comment);
rv_column_comment=plpy.execute(get_column_comment);

```

```

for i in rv_table_comment:
    create_sql += i['comment']+';\n'
for i in rv_column_comment:
    create_sql += i['comment']+';\n'

return create_sql;
$$ LANGUAGE plpythonu;

```

图 4-3 是一个 get\_create\_sql 的例子。

```

testDB=# \select * from pg_class where conname = 'get_create_sql';
      name      |      type      |      owner      |
 public.hello_partition |  table |      test      |
      id      |      character varying(32) |
      name      |      character varying(32) |
      dw_end_date      |      date      |
      )with(appendonly=true,compresslevel=5)
      distributed by (id)
      PARTITION BY range(dw_end_date)
      (
      PARTITION p20101230 START ('2010-12-30'::date) END ('2010-12-31'::date) WITH (appendonly=true, compresslevel=5),
      PARTITION p20101231 START ('2010-12-31'::date) END ('2011-01-01'::date) WITH (appendonly=true, compresslevel=5),
      PARTITION p20110101 START ('2011-01-01'::date) END ('2011-01-02'::date) WITH (appendonly=true, compresslevel=5),
      PARTITION p20110102 START ('2011-01-02'::date) END ('2011-01-03'::date) WITH (appendonly=true, compresslevel=5),
      PARTITION p20110103 START ('2011-01-03'::date) END ('2011-01-04'::date) WITH (appendonly=true, compresslevel=5),
      PARTITION p20110104 START ('2011-01-04'::date) END ('2011-01-05'::date) WITH (appendonly=true, compresslevel=5);
      );
      comment on table public.hello_partition is 'test partition';
      comment on column public.hello_partition.id is 'id' ;
      (1 row)

```

图 4-3 获取表定义的例子

#### 4.7.6 查询表上的视图

在 Greenplum 中，要获取一个表上依赖的视图很麻烦，因为视图定义在 pg\_rewrite\_rule 中，存放成一种规则，而且上面没有索引，所以获取比较麻烦。下面介绍通过 pg\_depend 表来获取表上依赖视图的方法（一个视图是定义在表上的，这个视图肯定是依赖于这个表的，所以在 pg\_depend 中有响应的信息）。

创建一个通过 oid 来获取模式名和表名的函数：

```

CREATE or replace FUNCTION public.tabname_oid(a oid)
RETURNS text
AS $$
rv = plpy.execute("select b.nspname||'.'||a.relname as tablename from pg_class
a,pg_namespace b where a.relnamespace=b.oid and a.oid=%s"%(a))
if rv:
    return rv[0]['tablename']
else:
    return 'unkown.unfound'
$$ LANGUAGE plpythonu;

```

然后建立视图：

```

CREATE VIEW public.views_on_tables as
SELECT tabname_oid(c.ev_class) AS viewname, tabname_oid(pc.oid) AS tablename
FROM pg_depend a, pg_depend b, pg_class pc, pg_rewrite c
WHERE a.refclassid = 1259::oid AND b.deptype = 'i'::"char" AND a.classid =

```

```
2618::oid AND a.objid = b.objid AND a.classid = b.classid AND a.refclassid =
b.refclassid AND a.refobjid <> b.refobjid AND pc.oid = a.refobjid AND c.oid = b.objid
GROUP BY c.ev_class, pc.oid;
```

下面是如何使用这个视图的例子：

```
testDB=# create table public.test1(id int primary key,values text);
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "test1_pkey" for
table "test1"
CREATE TABLE
testDB=# create view public.v_test1 as select * from test1;
CREATE VIEW
testDB=# select * from views_on_tables where tablename like 'public.test1';
    viewname      | tablename
-----+-----
 public.v_test1 | public.test1
(1 row)
```

#### 4.7.7 查询表的数据文件创建时间

在 Greenplum 中，创建一个表的时间都没有记录在数据字典中，要获取这个表的操作时间，只能对所有的日志文件进行分区，提取出所有操作的语句及时间，然后获取语句中的表名，非常麻烦。

下面介绍一个比较取巧的方法来获取一个表的修改时间。在 Greenplum 中，每一个表都对应文件系统上的几个文件，这样我们就可以通过数据文件的创建时间和修改时间估计这个表的创建时间。

通过一些自定义函数，利用数据字典获取数据文件对应的数据目录和文件名，就可以在数据库中获取到文件的时间，从而可以定义一个视图来方便查询，相当于自定义一个元数据视图。

##### (1) 如何在数据库中获取一个文件的信息

Greenplum 自带了一个函数 pg\_stat\_file。通过它可以获取文件的信息。下面介绍获取 data\_directory 目录下 pg\_hba.conf 文件的信息。

```
testDB=# \x
Expanded display is on.
testDB=# select * from pg_stat_file('pg_hba.conf');
-[ RECORD 1 ]+-----
size      | 4074
access    | 2012-02-11 19:45:57+08
modification | 2012-02-02 10:38:35+08
change    | 2012-02-02 10:38:35+08
creation   |
isdir     | f
```

但是这个函数只有用在数据库 data\_directory 目录（用 show data\_directory）下才可显示。

这在 Greenplum 3.x 版本中已经可以完成了，但是 Greenplum 4.x 引入了 `filespace` 的概念（具体参见“第 9 章数据库管理”），所以在 `filespace` 下面的文件不能用 `pg_stat_file` 来查看。因此下面重新利用 `plpythonu` 编写一个函数来获取文件信息，同时捕获文件不存在等异常，代码如下：

```

create type stat_file as(
    size bigint
    ,access timestamp(0)
    ,modification timestamp(0)
    ,change timestamp(0)
);

CREATE or replace FUNCTION public.get_file_stat(filename text)
RETURNS stat_file
AS $$

import os,time
size = None
access =None
modification =None
change =None
try:
    a=os.stat(filename);
    size = int(a.st_size)
    #plpy.info(a)
    access = time.strftime("%Y-%m-%d %H:%M:%S",time.localtime(a.st_atime))
    modification = time.strftime("%Y-%m-%d %H:%M:%S",time.localtime(a.st_mtime))
    change = time.strftime("%Y-%m-%d %H:%M:%S",time.localtime(a.st_ctime))
except Exception,e:
    pass
    #plpy.info(e);
#plpy.info([size,access,modification,change])
return [size,access,modification,change]
$$ LANGUAGE plpythonu;

```



**注意** Greenplum 3.x 的比较简单，这里就不介绍了，下面就介绍 Greenplum 4.x 的。

## (2) 利用数据字典拼接出文件目录和文件名

- 文件名：`pg_class` 的 `relfilenode` 字段。
- 表空间：`pg_class` 的 `reltablespace` 字段，有 `pg_default`, `pg_global` 要单独考虑。
- 表空间对应的 `filespace`: `pg_tablespace`。
- `Filespace` 目录: `pg_filespace_entry`。
- `database` 的 `oid`: 查询 `pg_database` 和当前数据库名。

视图的定义如下：

```

create view public.v_table_modify_time
as
select tab_oid,schemaname,tablename,(filestat).access,(filestat).
modification,(filestat).change
from
(select
    a.oid tab_oid,
    e.nspname as schemaname,
    a.relname as tablename,
    get_file_stat(fselocation||'/'|
        case when reltablespace=1664 then 'global'
        when reltablespace=0 then 'base'||'/'||d.oid
        else reltablespace||'/'||d.oid
    end
    ||'/'||relfilename) as filestat
from pg_class a,
     pg_tablespace b,
     pg_filespace_entry c,
     pg_namespace e,
     pg_database d
where d.datname=current_database()
and (case when a.reltypespace = 0 then 1663 else a.reltypespace end)=b.oid
and b.spcfsoid=c.fsefsoid
and e.oid = a.relnamespace
and c.fsdbid=1
and a.relstorage in ('a','h')
and a.relkind = 'r'
)t;

```

视图使用效果如图 4-4 所示。

tab_oid	schemaname	tablename	access	modification	change
23863	public	test001	2012-02-11 21:24:12	2012-02-11 21:24:12	2012-02-11 21:24:12

图 4-4 获取建表时间的例子

#### 4.7.8 分区表总大小

在 Greenplum 中，没有一个函数可以直接统计分区表的大小，这使得我们在检查分区表大小的时候要将所有子分区的大小累计起来，然后再做一个聚合，统计出整张表的大小。

下面提供一个函数将检查分区大小的操作封装起来。

```

-- 创建一个 type，定义后续函数的返回类型
create type public.table_size as
  (tablename text
   ,subparname text
   ,tablesize bigint
   ,prettysize text);

```

```

-- 创建一个将 oid 转换成 text 类型的函数
CREATE or replace FUNCTION regclass2text(a regclass)
RETURNS text
AS $$ 
    return a;
$$ LANGUAGE plpythonu;

CREATE or replace FUNCTION public.get_table_size(tablename text)
RETURNS setof table_size
AS $$ 
try:
    table_name = tablename.lower().split('.')[1]
    table_schema = tablename.lower().split('.')[0]
except (IndexError):
    return 'Please input "tableschema.tablename" !'

#get table oid——获取表的 oid
    get_table_oid = " select oid,reloptions from pg_class where
oid='%s'::regclass"%(tablename)
    try:
        rv_oid = plpy.execute(get_table_oid, 5)
        if not rv_oid:
            return 'Did not find any relation named "' + tablename + '".'
    except (Error):
        return 'Did not find any relation named "' + tablename + '".'

    table_oid = rv_oid[0]['oid']

#check if table is partition table - 判断该表是否是分区表
    check_par_table="select count(*) from pg_partition where
parrelid=%s"%(table_oid);
#get the sub partitions of the table - 获取子分区信息
    tablesize_sql1=""""
        select regclass2text(tablename) tablename,parname subparname,size
tablesize,pg_size_pretty(size) prettysize from (select pp.parrelid::regclass
tablename,pr1.parname,pg_relation_size(pr1.parchildrelid) size,parruleord FROM
pg_partition pp, pg_partition_rule pr1
        WHERE pp.paristemplate = false AND pp.parrelid = %s AND pr1.paroid =
pp.oid )t order by parruleord;
    """%(table_oid)
    tablesize_sql2=""""
        select '%s' tablename,null as subparname,pg_relation_size(%s)
tablesize,pg_size_pretty(pg_relation_size(%s)) prettysize;
    """%(tablename,table_oid,table_oid)

    rv=plpy.execute(check_par_table);

    if rv[0]['count']==1:
        a1=plpy.execute(tablesize_sql1);

```

```

result_rv=[];
total_size=0;
for i in al:
    total_size = total_size + int(i['tablesize']);

total_size"""
        select '%s' as tablename,'###ALL###' as subparname,%d as
tablesize,pg_size_pretty(%d::bigint) prettysize;
        """%(tablename,total_size,total_size)
a2=plpy.execute(total_size);
result_rv.append(a2[0])

for i in al:
    result_rv.append(i);
return result_rv;
else :
    result_rv=plpy.execute(tablesize_sql2)
    return result_rv;
$$ LANGUAGE plpythonu;

```

效果如图 4-5 所示。

tablename	subparname	tablesize	prettysize
etl.hello_partition	###ALL###	263680	258 kB
etl.hello_partition	p20101230	39792	39 kB
etl.hello_partition	p20101231	43832	43 kB
etl.hello_partition	p20110101	45008	44 kB
etl.hello_partition	p20110102	45024	44 kB
etl.hello_partition	p20110103	45008	44 kB
etl.hello_partition	p20110104	45016	44 kB

(7 rows)

图 4-5 获取表大小的例子

#### 4.7.9 如何分析数据字典变化

对于 Greenplum 的数据字典来说，总的数据字典也只有 60 来张表，如果对于每一个 DDL 命令，我们能够知道有哪些数据字典发生了变化，这样对于我们深入了解数据库底层逻辑有很大的帮助，了解了这些，对数据库优化跟原数据的应用有更深入的帮助。

下面介绍一种方法来观察每一个 SQL 对应数据字典的变化，原理是对所有数据字典在 DDL 操作之前跟之后都做一个快照（记录每个表的最大事务 ID 等信息），然后比较两个快照时间的数据发生了哪些变化，从而分析出数据字典的变化。

首先我们需要创建一张表和两个函数。

□ catalog\_result 表：保存 snapshot 结果的表，每一个表有一个 ID。

□ catalog\_snap 函数：对当前数据字典的最大 ctid，最小 xmin（事务 id）以及记录数，做一个快照（snapshot），返回快照 ID。

□ diff\_catalog 函数：比较两个 snapshot 之间的数据差异，找出变化的数据字典。

Greenplum 中记录事务 ID 的数据类型 xid 不能进行比较，故使用 UDF 将其转换成

`integer` 类型，方便比较：

```
CREATE or replace FUNCTION public.xcid2int(id xid)
RETURNS integer
AS $$

    return id;
$$ LANGUAGE plpythonu;
```

表结构：

```
create table public.catalog_cnt(
    cnt          integer
    ,cctid       tid
    ,cxmin       integer
    ,catatable   character varying(248)
    ,count        integer
)Distributed by (catatable);
```

创建 `snapshot` 的函数：

```
CREATE or replace FUNCTION public.catalog_snap()
RETURNS integer
AS $$

    rv=plpy.execute("select max(cnt) cnt from catalog_cnt");
    cntnumber = '1'
    if rv[0]['cnt']:
        #plpy.info(cntnumber);
        cntnumber = str(rv[0]['cnt']+1);
        sql=""""
            select 'insert into catalog_cnt select %s cnt,max(ctid),max(xcid2in
t(xmin)),'||a.relname ||''' as catatable,count(*) from'||b.nspname||'.'||a.
relename  as n
            from pg_class a,pg_namespace b
            where b.nspname='pg_catalog'
            and a.relnamespace=b.oid
            and relkind in ('r')
            order by b.nspname,relname;
        """%(cntnumber)
        #plpy.info(sql)
        rv=plpy.execute(sql);
        for i in rv:
            plpy.execute(i['n']);
        return int(cntnumber)
$$ LANGUAGE plpythonu;
```

比较两个 `snapshot` 之间差异的函数：

```
create type public.catalog_result as (
    catatable      character varying(248)
    ,pre_cctid     tid
    ,pre_cxmin     integer
```

```

,pre_count           integer
,cur_cctid          tid
,cur_cxmin          integer
,cur_count           integer
);

CREATE or replace FUNCTION public.diff_catalog(before int,after int)
RETURNS setof catalog_result
AS $$
sql"""
select x.catatable,x.cctid as pre_cctid,x.cxmin as pre_cxmin,x.count as pre_count
,y.cctid as cur_cctid,y.cxmin as
cur_cxmin,y.count as cur_count
from
(select * from catalog_cnt where cnt=%d)x,
(select * from catalog_cnt where cnt=%d)y
where x.catatable = y.catatable
and (x.cctid<>y.cctid
or x.cxmin<>y.cxmin
or x.count<>y.count)
"""%(before,after)
return plpy.execute(sql);
$$ LANGUAGE plpythonu;

```

使用方式如下：

```

testDB=# select catalog_snap();
catalog_snap
-----
9
(1 row)

testDB=# create table test_snap(id int,name text) distributed by(id);
CREATE TABLE
testDB=# select catalog_snap();
catalog_snap
-----
10
(1 row)

```

最后使用 diff\_catalog 来查看哪些数据字典发生了变化，如图 4-6 所示。

catatable	pre_cctid	pre_cxmin	pre_count	cur_cctid	cur_cxmin	cur_count
gp_distribution_policy	(0,159)	2945	92	(0,160)	2994	93
gp_relation_node	(1,134)	2945	466	(1,137)	2994	469
gp_persistent_relation_node	(3,154)	2	898	(3,157)	2	901
pg_stat_last_operation	(2,69)	2945	261	(2,70)	2994	262
pg_attribute	(26,196)	2946	466	(26,197)	2994	466
pg_class	(4,18)	2968	634	(6,152)	2994	617
pg_depend	(13,171)	2969	6611	(13,177)	2994	6617
pg_index	(0,276)	2945	206	(0,277)	2994	207
pg_type	(3,76)	2966	527	(3,78)	2994	529

图 4-6 获取数据字典变化的例子

以下是使用此方法观察数据字典的注意事项：

- 要在比较干净的测试环境中进行实验，这样分析数据字典时比较快速。
- 这个方法只是主节点上数据字典的变化，要观察 Segment 的变化，可以直接登录到 Segment 节点进行同样的操作。
- 在同一时间，只能有一个 DDL 操作，避免会话进行操作而造成干扰。
- 测试环境应当尽量对数据字典进行 vacuum 操作，避免 ctid 发生变化。
- 这个方法只能观察到数据字典发生 insert 和 update 操作，但是这对常见的 DDL 操作已经足够了。如果要观察到所有的操作，则需要将整个数据字典都进行保存，执行操作后将变化后的数据字典与变化前的进行差异比较得出结果。

#### 4.7.10 获取数据库锁信息

视图 pg\_locks 保存了数据库的锁信息，但是这个视图很不方便。要查询一个表被哪个进程锁住了，就需要将 pg\_class、pg\_locks、pg\_stat\_activity 关联起来，如下：

```
SELECT pgl.locktype AS lorlocktype, pgl.database AS lordatabase, pgc.relname
AS lorrelname, pgl.relation AS lorrelation, pgl.transaction AS lortransaction,
pgl.pid AS lorpid, pgl.mode AS lormode, pgl.granted AS lorgranted, pgsa.
current_query AS lorcurrentquery
FROM pg_locks pgl
JOIN pg_class pgc ON pgl.relation = pgc.oid
JOIN pg_stat_activity pgsa ON pgl.pid = pgsa.procpid
ORDER BY pgc.relname;
```



**注意** 上面这个 SQL 就是 gp\_toolkit.gp\_locks\_on\_relation 的定义，在 Greenplum 3.x 中可以自己创建。

Pg\_stat\_activity 可以获取当前正在运行的 SQL，具体的结构描述可以参考 PostgreSQL 的官方文档，比较简单。

下面再介绍两个函数，杀掉当前的进程，参数都是这个 SQL 的进程号 (pid)。

- pg\_cancel\_backend：取消一个正在执行的 SQL。
- pg\_terminate\_backend：终止一个正在执行的 SQL。

pg\_terminate\_backend 比 pg\_cancel\_backend 的强度大，一般要杀掉 SQL 进程，可以先用 pg\_cancel\_backend 杀掉 SQL 进程，如果杀不掉，再用 pg\_terminate\_backend 将 SQL 进程杀掉。

例如，可以利用这个函数，将锁住表 test001 的进程杀掉，如图 4-7 所示。

```
testDB=# SELECT pg_terminate_backend(lorpid) FROM gp_toolkit.gp_locks_on_relation where lorrelname='test001';
x
(1 row)
```

图 4-7 将锁住某个表的进程杀掉

## 4.8 Gp\_toolkit 介绍

Greenplum 4.x 之后的版本提供了 gp\_toolkit 的工具箱，方便用户对数据字典进行分析和管理，这个工具箱都是基于数据字典建立的视图（所有的视图都在 gp\_toolkit 的 schema 下面，如果经常使用这个工具箱，建议将这个 schema 加入到 search\_path 中），表 4-17 描述了这些视图的功能。

表 4-17 gp\_toolkit 工具箱视图描述

视图名	描述
_gp_fullname	显示每一个 relation 的 oid、模式名和表名
_gp_is_append_only	对应一个 oid 是否是 appendonly 表
_gp_number_of_segments	集群中有多少个 segment
_gp_user_data_tables	用户自定义表信息，在 _gp_user_tables 中，每个分区表就算一张表
_gp_user_data_tables_readable	当前用户有 select 权限的表信息
_gp_user_namespaces	用户自定义的 schema
_gp_user_tables	用户自定义的表信息，不包括系统 schema 下的表
gp_bloat_diag	获取数据库中哪些表需要进行 vacuum full
gp_bloat_expected_pages	表 oid 对应的理想的表大小和实际的表大小，不统计 appendonly 表，主要是分析表对应的数据文件的空洞
gp_locks_on_relation	查询 relation 上的锁信息
gp_locks_on_resqueue	查询资源队列中的信息
_gp_log_master_ext	可执行外部表，查询主节点上的 log 信息
_gp_log_segment_ext	可执行外部表，查询 segment 节点上的 log 信息
gp_log_command_timings	查询每个 session 执行的命令的时间
gp_log_database	查询当前数据库的 log 信息，包括主节点和 segment 节点
gp_log_master_concise	从 _gp_log_master_ext 选择几个字段，简要的 log 描述
gp_log_system	将主节点和 segment 节点的 log 信息汇总在一起
gp_param_settings_seg_value_diffs	查找出所有 segment 节点的配置参数不一样的地方
gp_resq_activity	资源队列相关
gp_resq_activity_by_queue	
gp_resq_priority_backend	
gp_resq_priority_statement	
gp_resq_role	
gp_resqueue_status	

(续)

视图名	描述
gp_size_of_all_table_indexes	
gp_size_of_database	
gp_size_of_index	
gp_size_of_partition_and_indexes_disk	这几张视图分别从数据库各个维度(表、索引、schema等)统计数据的大小。统计数据库大小的原理基本都是通过pg_relation_size等函数获取每个表或索引的大小，然后再汇总起来，从视图执行效率上看，效率比较低。
gp_size_of_schema_disk	
gp_size_of_table_and_indexes_disk	
gp_size_of_table_and_indexes_licensing	
gp_size_of_table_disk	
gp_size_of_table_uncompressed	
gp_stats_missing	查询当前数据库中，没有统计信息的表(精确到字段)
gp_table_indexes	获取当前用户有权限访问的表的索引信息名



注意 如果 gp\_toolkit 没有安装，可以用下面这个命令进行安装：

```
psql -f $GPHOME/share/postgresql/gp_toolkit.sql
```

下面我们选一个视图(gp\_bloat\_expected\_pages)来详细讲解下，以加深对Greenplum数据字典的认识。

这个视图是对堆表(heap)理想大小与实际大小的比较。首先看一下这个视图的定义：

```
testDB=# \d gp_bloat_expected_pages
View "gp_toolkit.gp_bloat_expected_pages"
 Column | Type | Modifiers
-----+-----+-----
 btdrelid | oid | 
 btdrelpages | integer | 
 btddexpages | numeric | 
View definition:
SELECT subq.btdrelid, subq.btdrelpages,
CASE
    WHEN subq.btddexpages < subq.numsegments::numeric THEN subq.
    numsegments::numeric
    ELSE subq.btddexpages
END AS btddexpages
FROM ( SELECT pgc.oid AS btdrelid, pgc.relpages AS btdrelpages, ceil((pgc.
    reltuples * (25::double precision + btwcols.width))::numeric / current_
    setting('block_size'::text)::numeric) AS btddexpages, ( SELECT __gp_number_of_
    segments.numsegments
        FROM __gp_number_of_segments) AS numsegments
    FROM ( SELECT pgc.oid, pgc.reltuples, pgc.relpages
        FROM pg_class pgc
        WHERE NOT (EXISTS ( SELECT __gp_is_append_only.iaooid
```

```

        FROM __gp_is_append_only
        WHERE __gp_is_append_only.iaoid = pgc.oid AND __gp_
is_append_only.iaotype = true))) pgc
    LEFT JOIN ( SELECT pgs.starelid, sum(pgs.stawidth::double precision *
(1.0::double precision - pgs.stanullfrac)) AS width
        FROM pg_statistic pgs
    GROUP BY pgs.starelid) btwcols ON pgc.oid = btwcols.starelid
    WHERE btwcols.starelid IS NOT NULL) subq;

```

其中, pg\_class 中的 relpages 字段作为实际大小, 理想大小通过 pg\_statistic 加上 pg\_class 两个合起来算, 算法如下:

$$\text{page 数} = \frac{(25 + \sum (\text{字段长度} * (1 - \text{空值率}))) * \text{表的总行数}}{\text{块大小}}$$

从这个视图的定义我们可以看出表大小的估算方法, 也可以看到这个视图的局限性:

- 表大小都是估算值, 因为统计信息只是估计值。
- 依赖于统计信息收集的时间, 要更加准确, 需要重新分析、收集最新的信息。

知道了这个方法, 我们可以简单验证这个估算方法的准确性, 如图 4-8 所示。

这些表的数据都是递增的, 即没有发生过 update 和 delete 的, 所以数据文件中应该是没有空洞的。可以看出, 估计值还是有挺大的差异的, 所以这个表得出的只能是大概的估计值, 对于不同的表, 偏差会比较大。

建议有兴趣了解数据字典的读者认真看看这些视图的定义, 这对深入了解数据字典有很大的帮助。

	表名 regclass	统计大小 integer	计算大小 numeric	实际大小 numeric(10,1)	差异百分比 text
1	test001	39111	33410	39111.0	17.1%
2	test01	30019	27891	30019.0	7.6%

图 4-8 表大小的估算

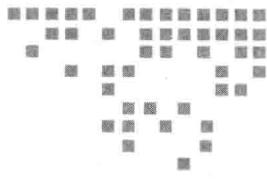
## 4.9 小结

本章介绍了常用数据字典的表结构, 介绍了各个数据字典之间的关系, 还通过几个例子加深读者对数据字典的认识。

深入了解数据字段对于数据库元数据信息的获取有很大的帮助, 我们可以通过这些信息开发一些工具, 方便优化数据库, 比如:

- 查询建表时间有助于删除一些临时表;
- 数据交换工具可以通过数据字段获取表的字段、分布键等信息;
- 当数据字典出问题时能够及时修复;
- 获取建表语句, 方便数据迁移及表重建等。

本章还介绍了几种研究数据字典的方法, 有兴趣的读者可以利用这些方法深入学习数据字典, 了解数据库的内部结构, 加深对 Greenplum 的了解。



第 5 章

Chapter 5

## 执行计划详解

本章开始部分将讲解在数据库中分析 SQL 性能问题最重要的手段——查看执行计划。Greenplum 是基于 PostgreSQL 开发的，其执行计划大多是跟 PostgreSQL 一样的，但是由于 Greenplum 是分布式并行数据库，在 SQL 执行上有很多 MPP 的痕迹，因此在理解 Greenplum 的执行计划时，一定要将其分布式框架熟读在心，从而能够通过调整执行计划给 SQL 带来很大的性能提升。

### 5.1 执行计划入门

#### 5.1.1 什么是执行计划

执行计划就是数据库运行 SQL 的步骤，相当于算法。读懂 Greenplum 的执行计划，对理解 SQL 的正确性及性能有很大的帮助。执行计划是数据库使用者了解数据库内部结构的一个重要途径。

举一个简单的例子，一个人要去旅行，从地点 A 到地点 B，怎么去呢？坐车还是搭飞机？如何从 A 到 B，这个就是一个执行计划。这个旅行要考虑的东西有：

- A 点到 B 点的距离。
- 飞机、汽车、火车的时刻表。
- 整个行程的费用。
- 消耗的时间。

前两点可以理解为数据库的统计信息，后两点可以理解为整个 SQL 的消耗。我们需要

寻找一个最好的行程以使 SQL 的消耗达到最小。多种消耗之间我们需要取一个权重，把消耗重新定义为一个单位，就像在数据库中评价消耗有 CPU、磁盘、重分布网络开销等。可以将距离想象成表的大小，将时刻表想象成表中每个字段的统计信息（唯一性，值的分布）等。

如果从 A 点到 B 点，没有直达的汽车或者飞机，那么就要选择先到中间点 C 点或者 D 点，通过计算消耗最小值得出最佳的方案，这就像要将数据库中三张表关联，到底是先关联哪两张表才能达到最佳性能一样。

前面在第 2 章的时候已经简单介绍了执行计划应该如何阅读，这一章我们将详细介绍 Greenplum 的执行计划，会用比较多的篇幅讲解分布式数据库的 SQL 是如何执行的，建议读者仔细阅读本章。

### 5.1.2 查看执行计划

跟 PostgreSQL 一样，Greenplum 通过 explain 命令来查看执行计划。具体的语法如下：

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

各个参数的含义如下：

- ❑ ANALYZE：执行命令并显示实际运行时间。
- ❑ VERBOSE：显示规划树完整的内部表现形式，而不仅是一个摘要。通常，这个选项只是在特殊的调试过程中有用。VERBOSE 输出是否打印工整的，具体取决于配置参数 explain.pretty\_print 的值。
- ❑ statement：查询执行计划的 SQL 语句，可以是任何 SELECT、INSERT、UPDATE、DELETE、VALUES、EXECUTE、DECLARE 语句。

例子：

```
testDB=# explain select * from test1;
          QUERY PLAN
-----
Gather Motion 6:1  (slice1)  (cost=0.00..0.00 rows=1 width=150)
  -> Seq Scan on test1  (cost=0.00..0.00 rows=1 width=150)
(2 rows)
```

## 5.2 分布式执行计划概述

### 5.2.1 架构

数据库架构本身决定分布式执行计划框架，在理解 Greenplum 执行计划的时候，要将其 SharedNothing 的架构牢记于脑中。这里笔者再重复一下 Greenplum 的架构图，以便读者能够很好地理解 Greenplum 的执行计划。

Greenplum 的详细架构会在第 7 章中具体介绍。这里先通过图 5-1 简单了解下。

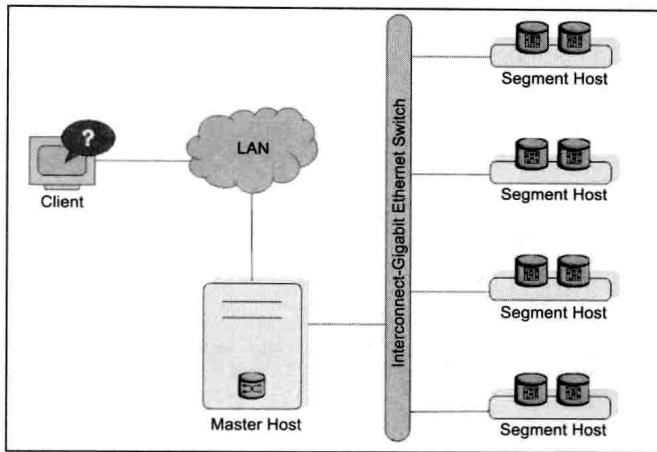


图 5-1 Greenplum 架构

图 5-1 很好地说明了 ShareNothing 的特点：

- 底层的数据完全不共享。
- 每个 Segment 只有一部分数据。
- 每一个节点都通过网络连接在一起。

## 5.2.2 重分布与广播

关联数据在不同节点上，对于普通关系型数据库来说，是无法进行连接的。关联的数据需要通过网络流入到一个节点中进行计算，这样就需要发生数据迁移。数据迁移有广播和重分布两种。

图 5-2 所示很好地展示了 Greenplum 中重分布数据的实现。

在图 5-2 中，两个 Segment 分别进行计算，但由于其中一张表的关联键与分布键不一致，需要关联的数据不在同一个节点上，所以在 SLICE1 上需要将其中一个表进行重分布，可理解为在每个节点之间互相交换数据。

关于广播与重分布，Greenplum 有一个很重要的概念：Slice（切片）。每一个广播或重分布会产生一个切片，每一个切片在每个数据节点上都会对应发起一个进程来处理该 Slice 负责的数据，上一层负责该 Slice 的进程会读取下级 Slice 广播或重分布的数据，然后进行相应的计算。



**注意** 由于在每个 Segment 上每一个 Slice 都会发起一个进程来处理，所以在 SQL 中要严格控制切片的个数，如果重分布或者广播太多，应适当将 SQL 拆分，避免由于进程太多给数据库或者是机器带来太多的负担。进程太多也比较容易导致 SQL 失败。

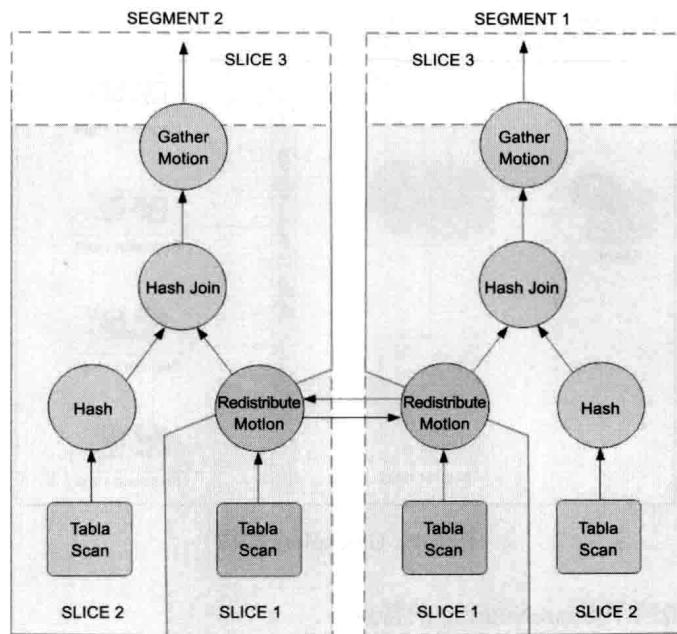


图 5-2 数据重分布逻辑架构

Slice 之间如何交互可以从图 5-3 中看出。

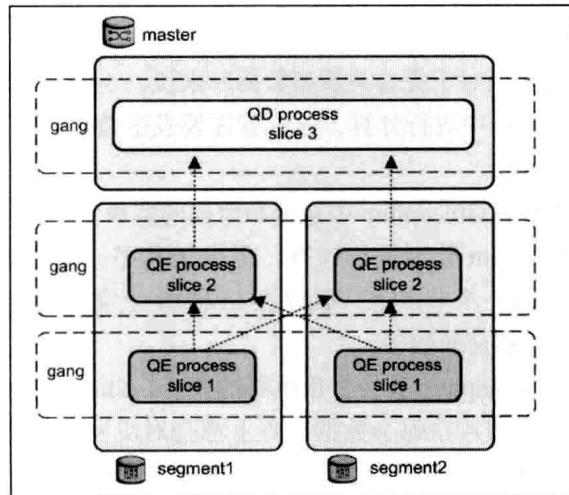


图 5-3 数据在 Slice 之间的传输

下面通过一个实际的数据形象地介绍数据在 Segment 中的切分。比方说，对一个成绩表来说，分布键是学号（sno），我们现在要按照成绩（score）来执行 Group By，那么就需要将数据按照 score 字段进行重分布，重分布前会对每个 Segment 的数据进行局部汇总，重分布

后，同一个 score 的数据都在同一个 Segment 上，再进行一次汇总即可，数据的具体情况如图 5-4 所示。

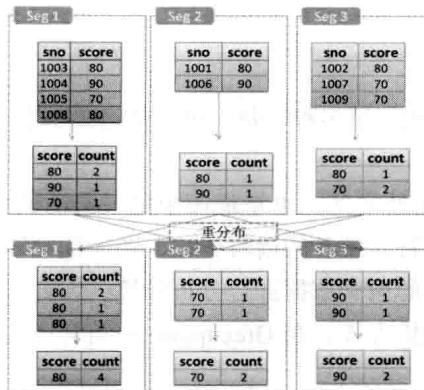


图 5-4 执行 Group By 导致的数据重分布

### 5.2.3 Greenplum Master 的工作

Master 在 SQL 的执行过程中承担着很多重要的工作，主要如下：

- 执行计划解析及分发。
- 将子节点的数据汇集在一起。
- 将所有 Segment 的有序数据进行归并操作（归并排序）。
- 聚合函数在 Master 上进行最后的计算。
- 需要有唯一的序列的功能（如开窗函数不带 partition by 子句）。

举个简单的例子，在计算学生的平均分数时，在每个节点上先计算好 sum 和 count 值，然后再由 Master 汇总，再次进行少量计算，算出平均值，如图 5-5 所示。

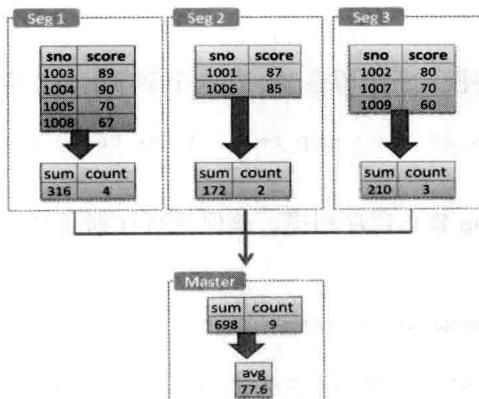


图 5-5 Master 在聚合计算的作用

## 5.3 Greenplum 执行计划中的术语

本节将介绍在查看执行计划的时候出现的术语的含义，以方便读者更好地阅读执行计划。

### 5.3.1 数据扫描方式

Greenplum 扫描数据的方式有很多种，每一种扫描方式都有其特点，下面将逐一介绍。

#### (1) Seq Scan: 顺序扫描

顺序扫描在数据库中，是最常见，也是最简单的一种方式，就是将一个数据文件从头到尾读取一次，这种方式非常符合磁盘的读写特性，顺序读写，吞吐很高。对于分析性的语句，顺序扫描基本上是对全表的所有数据进行分析计算，因此这一种方式非常有效。在数据仓库中，绝大部分都是这种扫描方式，在 Greenplum 中结合压缩表一起使用，可以减少磁盘 IO 的损耗。

#### (2) Index Scan: 索引扫描

索引扫描是通过索引来定位数据的，一般对数据进行特定的筛选，筛选后的数据量比较小（对于整个表而言）。使用索引进行筛选，必须事先在筛选的字段上建立索引，查询时先通过索引文件定位到实际数据在数据文件中的位置，再返回数据。

对于磁盘而言，索引扫描都是随机 IO，对于查询小数据量而言，速度很快。

#### (3) Bitmap Heap Scan: 位图堆表扫描

当索引定位到的数据在整表中占比较大的时候，通过索引定位到的数据会使用位图的方式对索引字段进行位图堆表扫描，以确定结果数据的准确。对于数据仓库应用而言，很少用这种扫描方式。

下面通过一个简单的例子来说明什么时候会发生 Bitmap Heap Scan。

创建 pg\_class\_tmp 并编造一些测试数据：

```
testDB=# create table pg_class_tmp as select * from pg_class distributed by(relname);
SELECT 1324
```

Pg\_class 中的 relkind 字段，重复值很多，在这个字段上建立索引：

```
testDB=# create index pg_class_tmp_relkind_idx on pg_class_tmp(relkind);
CREATE INDEX
```

通过参数 enable\_seqscan 禁止顺序扫描，确保执行计划通过 pg\_class\_tmp\_relkind\_idx 来查询数据：

```
testDB=# set enable_seqscan = off;
SET
testDB=# explain select * from pg_class_tmp where relkind='c';
          QUERY PLAN
```

---

```

Gather Motion 6:1  (slice1; segments: 6)  (cost=100.33..271.16 rows=2
width=234)
-> Bitmap Heap Scan on pg_class_tmp  (cost=100.33..271.16 rows=2 width=234)
    Recheck Cond: relkind = 'c'::"char"
        -> Bitmap Index Scan on pg_class_tmp_relkind_idx  (cost=0.00..100.33
rows=2 width=0)
            Index Cond: relkind = 'c'::"char"
    Settings: enable_seqscan=off
(6 rows)

```

由于 relkind 的重复值很多，因此数据库采用位图访问索引的方式来查询数据。

#### (4) Tid Scan: 通过隐藏字段 ctid 扫描

ctid 是 PostgreSQL 中标记数据位置的字段，通过这个字段来查找数据，速度非常快，类似于 Oracle 的 rowid。Greenplum 是一个分布式数据库，每一个子节点都是一个 PostgreSQL 数据库，每一个子节点都单独维护自己的一套 ctid 字段。

因此，如果在 Greenplum 中通过 ctid 来找数据，会有如下的提示：

```

Select * from test1 where ctid='(1,1)';
NOTICE:  SELECT uses system-defined column "test1.ctid" without the necessary
companion column "test1gp_segment_id"
HINT:  To uniquely identify a row within a distributed table, use the "gp_
segment_id" column together with the "ctid" column.

```

就是说，如果想确定到具体一行数据，还必须通过制定另外一个隐藏字段（gp\_segment\_id）来确定取哪一个数据库的 ctid 值。

```
Select * from test1 where ctid='(1,1)' and gp_segment_id=1;
```

#### (5) Subquery Scan '\*SELECT\*': 子查询扫描

只要 SQL 中有子查询，需要对子查询的结果做顺序扫描，就会进行子查询扫描。

#### (6) Function Scan: 函数扫描

数据库中有一些函数的返回值是一个结果集，当数据库从这个结果集中取出数据的时候，就会用到这个 Function Scan，顺序获取函数返回的结果集（这是函数扫描方式，不属于表扫描方式），如：

```
explain select * from generate_series(1,10);
```

### 5.3.2 分布式执行

#### (1) Gather Motion (N:1)

聚合操作，在 Master 上将子节点所有的数据聚合起来。一般的聚合规则是：哪一个子节点的数据先返回到 Master 上就将该节点的数据先放在 MASTER 上。

#### (2) Broadcast Motion (N:N)

广播，将每个 Segment 上某一个表的数据全部发送给所有 Segment。这样每一个

Segment 都相当于有一份全量数据，广播基本只会出现在两边关联的时候，相关内容再选择广播或者重分布，5.7 节中有详细的介绍。

### (3) Redistribute Motion (N:N)

当需要做跨库关联或者聚合的时候，当数据不能满足广播的条件，或者广播的消耗过大时，Greenplum 就会选择重分布数据，即数据按照新的分布键（关联键）重新打散到每个 Segment 上，重分布一般在以下三种情况下会发生：

- 关联：将每个 Segment 的数据根据关联键重新计算 hash 值，并根据 Greenplum 的路由算法路由到目标子节点中，使关联时属于同一个关联键的数据都在同一个 Segment 上。
- Group By：当表需要 Group By，但是 Group By 的字段不是分布键时，为了使 Group By 的字段在同一个库中，Greenplum 会分两个 Group By 操作来执行，首先，在单库上执行一个 Group By 操作，从而减少需要重分布的数据量；然后将结果数据按照 Group By 字段重分布，之后再做聚合获得最终结果。
- 开窗函数：跟 Group By 类似，开窗函数（Window Function）的实现也需要将数据重分布到每个节点上进行计算，不过其实现比 Group By 更复杂一些。

### (4) 切片 (Slice)

Greenplm 在实现分布式执行计划的时候，需要将 SQL 拆分成多个切片（Slice），每一个 Slice 其实是单库执行的一部分 SQL，上面描述的每一个 motion 都会导致 Greenplum 多一个 Slice 操作，而每一个 Slice 操作子节点都会发起一个进程来处理数据。

所以应该尽量控制 Slice 的个数，将太复杂的 SQL 拆分，减少进程数，在执行计划中，最常见的 Slice 关键字的地方就是广播跟重分布，如下：

```
Broadcast Motion 6:6  (slice1)
Gather Motion 6:1  (slice1)
```

## 5.3.3 两种聚合方式

HashAggregate 和 GroupAggregate 这两种聚合方式在 5.7 节介绍执行计划原理时会给出详细的讲解，这里主要从占用内存方面简单介绍。

### (1) HashAggregate

对于 Hash 聚合来说，数据库会根据 Group By 字段后面的值计算 Hash 值，并根据前面使用的聚合函数在内存中维护对应的列表，然后数据库会通过这个列表来实现聚合操作，效率相对较高。

### (2) GroupAggregate

对于普通聚合函数，使用 Group 聚合，其原理是先将表中的数据按照 Group By 的字段排序，这样同一个 Group By 的值就在一起，只需要对排好序的数据进行一次全扫描就可以得到聚合的结果了。

### 5.3.4 关联

Greenplum 中的关联的实现比较多，有 Hash Join、NestLoop、Merge Join，实现方式跟普通的 PostgreSQL 数据库方式一样。由于 Greenplum 是分布式的，所以关联可能会涉及表的广播或重分布，这个在 5.7.3 中有详细的讲解。下面通过实际的执行计划来分析这 3 种关联在 Greenplum 上的简单实现，首先建立两张表以方便我们查看后面的执行计划：

```
testDB=# create table test1 (id int,values varchar(256)) distributed by (id);
CREATE TABLE
testDB=# create table test2 (id int,values varchar(256)) distributed by (id);
CREATE TABLE
```

#### 1. Hash Join

Hash Join（Hash 关联）是一种很高效的关联方式，简单地说，其实现原理就是将一张关联表按照关联键在内存中建立哈希表，在关联的时候通过哈希的方式来处理。学过数据结构的读者应该都清楚，哈希表是一种非常高效的数据结构。

下面是一个 Hash Join 的例子：

```
testDB=# explain select * from test1 a,test2 b where a.id=b.id;
          QUERY PLAN
-----
Gather Motion 6:1  (slice1)  (cost=0.01..0.05 rows=3 width=300)
  -> Hash Join  (cost=0.01..0.05 rows=3 width=300)
      Hash Cond: a.id = b.id
      -> Seq Scan on test1 a  (cost=0.00..0.00 rows=1 width=150)
      -> Hash  (cost=0.00..0.00 rows=1 width=150)
          -> Seq Scan on test2 b  (cost=0.00..0.00 rows=1 width=150)
(6 rows)
```

#### 2. Hash Left Join

通过 Hash Join 的方式来实现左连接，在执行计划中的体现就是 Hash Left Join：

```
testDB=# explain select * from test1 a left join test2 b on a.id=b.id;
          QUERY PLAN
-----
Gather Motion 6:1  (slice1)  (cost=0.01..0.05 rows=3 width=300)
  -> Hash Left Join  (cost=0.01..0.05 rows=3 width=300)
      Hash Cond: a.id = b.id
      -> Seq Scan on test1 a  (cost=0.00..0.00 rows=1 width=150)
      -> Hash  (cost=0.00..0.00 rows=1 width=150)
          -> Seq Scan on test2 b  (cost=0.00..0.00 rows=1 width=150)
(6 rows)
```

### 3. NestedLoop

NestedLoop 关联是最简单，也是最低效的关联方式，但是在有些情况下，不得不使用 NestedLoop，例如笛卡儿积：

```
testDB=# explain select * from test1 ,test2;
                                         QUERY PLAN
-----
Gather Motion 6:1  (slice2)  (cost=0.08..0.20 rows=3 width=300)
-> Nested Loop  (cost=0.08..0.20 rows=3 width=300)
    -> Seq Scan on test1  (cost=0.00..0.00 rows=1 width=150)
    -> Materialize  (cost=0.08..0.14 rows=6 width=150)
        -> Broadcast Motion 6:6  (slice1)  (cost=0.00..0.07 rows=6 width=150)
            -> Seq Scan on test2  (cost=0.00..0.00 rows=1 width=150)
Settings: enable_seqscan=off
(7 rows)
```

由于是笛卡儿积，因此 SQL 一定是采取 Nestedloop 关联。在 Grenplum 中，如果采取 Nestedloop，关联的两张表中有一张表必须广播，否则无法关联，一般是数据量比较小的表会广播。

### 4. Merge Join 和 Merge Left Join

Merge Join 也是两表关联中比较常见的关联方式，这种关联方式需要将两张表按照关联键进行排序，然后按照归并排序的方式将数据进行关联，效率比 Hash Join 差。

下面的例子先通过设置两个参数来强制执行计划，采取的是 Merge Join 方式：

```
testDB=# set enable_hashjoin =off;
SET
testDB=# set enable_mergejoin =on;
SET
testDB=# explain select * from test1 a join test2 b on a.id=b.id;
                                         QUERY PLAN
-----
Gather Motion 6:1  (slice1)  (cost=0.02..0.05 rows=3 width=300)
-> Merge Join  (cost=0.02..0.05 rows=3 width=300)
    Merge Cond: a.id = b.id
    -> Sort  (cost=0.01..0.02 rows=1 width=150)
        Sort Key: a.id
        -> Seq Scan on test1 a  (cost=0.00..0.00 rows=1 width=150)
    -> Sort  (cost=0.01..0.02 rows=1 width=150)
        Sort Key: b.id
        -> Seq Scan on test2 b  (cost=0.00..0.00 rows=1 width=150)
Settings: enable_hashjoin=off; enable_mergejoin=on; enable_seqscan=off
(10 rows)
```

伴随 Merge Join 的肯定是两张表关联键的排序。

## 5. Merge Full Join

如果关联使用的是 full outer join，则执行计划使用的就是 Merge Full Join。在 Greenplum 中其他的关联方式都无法进行全关联。

```
testDB=# explain select * from test1 a full outer join test2 b on a.id=b.id;
          QUERY PLAN
-----
Gather Motion 6:1  (slice1)  (cost=0.02..0.05 rows=3 width=300)
-> Merge Full Join  (cost=0.02..0.05 rows=3 width=300)
    Merge Cond: a.id = b.id
    -> Sort  (cost=0.01..0.02 rows=1 width=150)
        Sort Key: a.id
        -> Seq Scan on test1 a  (cost=0.00..0.00 rows=1 width=150)
    -> Sort  (cost=0.01..0.02 rows=1 width=150)
        Sort Key: b.id
        -> Seq Scan on test2 b  (cost=0.00..0.00 rows=1 width=150)
Settings: enable_seqscan=off
(10 rows)
```



**注意** 在 Oracle 10g 中，a full outer join b 的实现方式是对 a 和 b 做一个左外连接，然后对 b 和 a 做一个反连接（在关联时，匹配的剔除，不匹配的保留），再对两个结果直接进行 union all 操作。但是在 Greenplum 中没有执行这个优化，所以只能采取 Merge Join。Nestloop 只能用于内连接，对外连接无能为力。

## 6. Hash EXISTS Join

关联子查询 exist 之类的 SQL 会被改写成 inner join，如果 SQL 被改写了，则会出现 Hash EXISTS Join。

```
testDB=# explain select * from test1 a where exists(select 1 from test2 b where
a.id=b.id);
          QUERY PLAN
-----
Gather Motion 6:1  (slice1)  (cost=0.01..0.05 rows=3 width=150)
-> Hash EXISTS Join  (cost=0.01..0.05 rows=3 width=150)
    Hash Cond: a.id = b.id
    -> Seq Scan on test1 a  (cost=0.00..0.00 rows=1 width=150)
    -> Hash  (cost=0.00..0.00 rows=1 width=4)
        -> Seq Scan on test2 b  (cost=0.00..0.00 rows=1 width=4)
(6 rows)
```

### 5.3.5 SQL 消耗

在每个 SQL 的执行计划中，每一步都会有（cost=0.01..0.05 rows=3 width=150）这 3 项表示 SQL 的消耗，后面会介绍消耗具体的计算方法，这里先介绍这 3 个字段的含义。

#### (1) Cost

以数据库自定义的消耗单位，通过统计信息来估计 SQL 的消耗。具体消耗的单位可以参考 PostgreSQL 的官方文档：<http://www.pgsqldb.org/pgsql/doc-8.1c/runtime-config-query.html>

#### (2) Rows

根据统计信息估计 SQL 返回结果集的行数。

#### (3) Width

返回结果集每一行的长度，这个长度值是根据 pg\_statistic 表中的统计信息来计算的。

### 5.3.6 其他术语

这里列举一些执行计划中常见的关键术语。

#### (1) Filter 过滤

Where 条件中的筛选条件，在执行计划中就是 Filter 关键字。

```
Filter: relfilenode = 1249::oid
```

#### (2) Index Cond

如果在查询的表中 where 筛选的字段中有索引，那么执行计划会通过索引定位，提高查询的效率。Index Cond 就是定位索引的条件。

```
Index Scan using pg_class_oid_index on pg_class(cost=0.00..200.27 rows=1
width=205)
    Index Cond: oid = 1259::oid
```

#### (3) Recheck Cond

在使用位图扫描索引的时候，由于 PostgreSQL 里面使用的是 MVCC 协议，为了保证结果的正确性，要重新检查一下过滤条件。

```
-> Bitmap Heap Scan on test1  (cost=100.37..103.48 rows=7 width=12)
    Recheck Cond: a >= 1 AND a <= 50
    -> Bitmap Index Scan on idx_test1  (cost=0.00..100.37 rows=7 width=0)
        Index Cond: a >= 1 AND a <= 50
```

#### (4) Hash Cond

执行 Hash join 的时候的关联条件：

```
-> Hash Join  (cost=40.87..119.60 rows=1683 width=24)
    Hash Cond: y.b = x.a
```

## (5) Merge

在执行排序操作时数据会在子节点上各自排好序，然后在 Master 上做一个归并操作。

```
Gather Motion 6:1  (slice1)  (cost=0.01..0.02 rows=1 width=150)
  Merge Key: id
    -> Sort  (cost=0.01..0.02 rows=1 width=150)
```

## (6) Hash Key

在数据重分布时候指定的重算 hash 值的分布键：

```
-> Redistribute Motion 6:6  (slice1)  (cost=0.00..53.49 rows=1683 width=12)
  Hash Key: y.b
    -> Seq Scan on test2 y  (cost=0.00..19.83 rows=1683 width=12)
```

## (7) Materialize

将数据保存在内存中，避免多次扫描磁盘带来的开销。这个要重点注意，由于将数据保存在内存中，会占用很大的内存，而执行计划是按照统计信息来计算的，如果统计信息丢失或者错误，有可能会将一张很大的表保存在内存中，直接导致内存空间不足，进而导致 SQL 执行失败：

```
-> Materialize  (cost=147.74..248.72 rows=10098 width=12)
    -> Broadcast Motion 6:6  (slice1)  (cost=0.00..137.64 rows=10098 width=12)
        -> Seq Scan on test2 y  (cost=0.00..19.83 rows=1683 width=12)
```

## (8) Join Filter

对数据关联后再进行筛选，如：

```
-> Nested Loop  (cost=147.74..467528.25 rows=314721 width=24)
    Join Filter: x.a < y.a AND x.a > (y.a + 1)
```

## (9) Sort, Sort key

如果执行计划中出现了 Sort 关键字，则说明有排序的操作，排序的字段为 Sort Key。

```
-> Sort  (cost=0.01..0.02 rows=1 width=150)
  Sort Key: id
    -> Seq Scan on test1  (cost=0.00..0.00 rows=1 width=150)
```

## (10) Window, Partition By, Order by

这个是使用开窗函数（Window Function）时，执行计划中显示了使用分析函数的标识：

```
testDB=# explain select * from ( select row_number() over (partition by id
order by values) rn from test1) t where rn=1 ;
          QUERY PLAN
-----
Gather Motion 6:1  (slice1)  (cost=0.01..0.03 rows=1 width=8)
  -> Subquery Scan t  (cost=0.01..0.03 rows=1 width=8)
      Filter: rn = 1
        -> Window  (cost=0.01..0.02 rows=1 width=150)
```

```

Partition By: id
Order By: "values"
-> Sort (cost=0.01..0.02 rows=1 width=150)
    Sort Key: id, "values"
    -> Seq Scan on test1 (cost=0.00..0.00 rows=1 width=150)
Settings: enable_seqscan=off
(10 rows)

```

### ( 11 ) Limit

当在 SQL 中只取前几行的时候，就使用 Limit 语句。

```
Limit (cost=0.00..0.85 rows=1 width=205)
```

### ( 12 ) Append

将结果直接汇总起来：

```

-> Append (cost=0.00..2.02 rows=1 width=13998)
    -> Append-only Scan on offer_1_prt_p20100801 offer
    -> Append-only Scan on offer_1_prt_p20100802 offer
    -> Append-only Scan on offer_1_prt_p20100803 offer

```

## 5.4 数据库统计信息收集

Greenplum 与 Oracle 等数据库一样，都是根据 CBO 优化器来选择一个好的执行计划的，尤其是在识别广播或者重分布的时候，统计信息十分重要，其准确与否直接决定了执行计划的好坏。

### 5.4.1 Analyze 分析

统计信息的命令如下：

```
ANALYZE [ VERBOSE ] [ table [ (column [, ...] ) ] ]
```

如果没有参数，ANALYZE 检查当前数据库中所有表。如果有参数，ANALYZE 只检查参数指定的那个表。还可以给出一列字段名字，这个时候只收集给出的字段的统计信息。

ANALYZE 收集表内容的统计信息，表级别的信息（表的数据量及表大小）保存在 pg\_class 中的 reltuples 和 relpages 字段中，然后把字段级别的结果保存在系统表 pg\_statistic 中，字段级信息通常包括每个字段最常用数值的列表以及显示每个字段中数据近似分布的包线图。（这些内容详见第 4 章。）

对于大表，ANALYZE 采集表内容的一个随机抽样进行统计，而不是检查每一行，这样就保证了即使是在很大的表上也需要很少时间就可以完成分析。不过，要注意的是，统计只是近似的结果，而且每次运行 ANALYZE 都会导致 EXPLAIN 显示的规划器的预期开销有一些小变化，即使表内容实际上没有改变。

在 Greenplum 中会对表自动进行统计信息收集。控制自动收集的参数是 `gp_autostats_mode`，这个参数有三个值：`none`、`on_change`、`on_no_stats`。这三个参数值的含义如表 5-1 所示。

表 5-1 `gp_autostats_mode` 的参数值解释

参数值	解    释
<code>none</code>	不自动收集统计信息
<code>on_change</code>	当目标表变更的数据量超过阀值 ( <code>gp_autostats_on_change_threshold</code> ) 时收集统计信息。 <code>gp_autostats_on_change_threshold</code> 这个值控制数据变化的行数
<code>on_no_stats</code>	当使用 <code>create table as select</code> 、 <code>insert</code> 、 <code>copy</code> 时，如果在目标表中没有收集过统计信息，那么 Greenplum 就会使用 <code>Analyze</code> 收集统计信息

这个参数在 Master 上修改，然后通过 `gpstop -u` 重新加载 `postgresql.conf` 这个配置文件即可。

Greenplum 默认使用 `on_no_stats`。使用这个参数，对数据库的消耗比较小，但是对于不断变更的表，数据库在第一次收集统计信息之后就不会再收集了，对于这种表，需要 DBA 定时手工运行 `Analyze` 收集统计信息。



说明 在 Greenplum 3.2.4 版本中，收集统计信息很慢，因为是对表进行全表扫描而不是进行数据采样，消耗比较大。在 3.3.x 版本之后，对统计信息收集进行了优化，只对表进行抽样分析，速度快了很多。在 4.x 版本中，Greenplum 在收集统计信息的时候建立临时表进行分析，临时表都是通过 `random` 函数抽取其中一定的百分比数据来进行分析的。

## 5.4.2 固定执行计划

通过上面的介绍，我们知道 Greenplum 是通过统计信息来生成执行计划的。如果发现数据库有一个表的统计信息收集不完整，导致执行计划出错，我们可以通过重新收集统计信息来解决，如果无法解决，则可以修改数据字典中的统计信息来达到固化执行计划的目的。

一般对执行计划影响最大的是 `pg_class` 的 `relpages` 和 `reltuples` 这两个字段，`reltuples` 是表的数据量，而 `relpages` 则表示表大小除以 32k，即：

```
select pg_relation_size(tablename)/32/1024;
```

比方说，将一个表的数据量变成原来的 10 倍，可以看出 `cost` 跟查询出来的行数都翻了 10 倍，如下：

```
testDB=# explain select * from test01;
          QUERY PLAN
-----
 Gather Motion 6:1  (slice1; segments: 6)   (cost=0.00..340.28 rows=4188
```

```

width=103)
-> Append-only Scan on test01  (cost=0.00..340.28 rows=4188 width=103)
(2 rows)
testDB=# update pg_class set reltuples=reltuples*10,relpages=relpages*10 where
relname='test01';
UPDATE 1
testDB=# explain select * from test01;
               QUERY PLAN
-----
Gather Motion 6:1  (slice1; segments: 6)   (cost=0.00..3402.80 rows=41880
width=103)
-> Append-only Scan on test01  (cost=0.00..3402.80 rows=41880 width=103)
(2 rows)

```



Greenplum 4.2 之后的版本，不允许对数据字典进行修改操作，在上述操作中，直接更新 pg\_class 表只能在较早的 Greenplum 版本中进行，Greenplum 4.2 之后的版本只能通过 analyze 命令重新收集统计信息。

在 Greenplum 4.3 中，报如下错误：

```
ERROR: permission denied: "pg_class" is a system catalog
```

## 5.5 控制执行计划的参数介绍

在 Oracle 中，我们可以只用 Hint 控制更改 SQL 的执行计划。在 Greenplum 中，也提供了类似的方法，但是比较简单。与 Oracle 的 Hint 不同，在 Greenplum 中，控制执行计划的参数是在会话级别，对于同一会话中的所有 SQL 生效，前面介绍执行计划的时候已经使用过了。

这些配置参数提供了影响查询优化器选择查询规划的原始方法。如果优化器为特定的查询选择的默认规划并不是最优，那么我们就可以通过使用这些配置参数强制优化器选择一个更好的规划来临时解决这个问题。这些参数一般是在某个会话级别对某个 SQL 进行设置的，不建议在全局中修改，会影响其他正常 SQL 的执行计划。

表 5-2 控制执行计划的参数列表

参 数	参数介绍	默认值
enable_bitmapscan	打开或者关闭规划器对位图扫描规划类型的使用	on
enable_hashagg	打开或者关闭规划器对 Hash 聚集规划类型的使用	on
enable_hashjoin	打开或者关闭规划器对 Hash 连接规划类型的使用	on
enable_indexscan	打开或者关闭规划器对索引扫描规划类型的使用	on
enable_mergejoin	打开或者关闭规划器对融合连接规划类型的使用	off

(续)

参数	参数介绍	默认值
enable_nestloop	打开或者关闭规划器对嵌套循环连接规划类型的使用。我们不可能完全消除嵌套循环连接，但是把这个变量关闭就会让规划器在存在其他方法的时候优先选择其他方法	off
enable_seqscan	打开或者关闭规划器对顺序扫描规划类型的使用。我们不可能完全消除顺序扫描，但是把这个变量关闭会让规划器在存在其他方法的时候优先选择其他方法	on
enable_sort	打开或者关闭规划器使用明确的排序步骤。我们不可能完全消除明确的排序，但是把这个变量关闭可以让规划器在存在其他方法的时候优先选择其他方法	on
enable_tidscan	打开或者关闭规划器对 TID 扫描规划类型的使用	on

以上这些参数都是 session 级别的，只要在 SQL 运行前设置一下就可以了，如果需要全局修改，只要在 Master 上改就可以了，然后执行 gpstop-u 重新加载配置文件即可生效，无需重启。

## 5.6 规划器开销的计算方法

在选择合理的执行计划的时候，Greenplum 会遍历所有的执行计划，计算其开销，即 Cost 值，并选择最小的执行路径执行 SQL。这里所描述的开销可以按照任意标准度量。我们只关心其相对值，因此以相同的系数缩放它们将不会对规划器产生任何影响。一般，Greenplum/PostgreSQL 以抓取顺序页的开销作为基准单位，也就是说将 seq\_page\_cost 设为 1.0，同时其他开销参数对照它来设置。当然也可以使用其他基准，比如以毫秒计的实际执行时间。表 5-3 是 Greenplum 中衡量数据库消耗的各个变量及默认参数。



现在还没有定义得很合理的方法用于判断下面出现的“开销”变量族的理想数值。如表 5-3 所示，每个变量的数值最好按照某个特定安装的平均查询开销来衡量。这意味着仅仅根据很少量的试验结果来修改它们是很危险的。

表 5-3 各种数据库消耗的各个变量

参数	说 明	默认值
seq_page_cost	计算一次顺序磁盘页面抓取的开销	1
random_page_cost	计算一次非顺序磁盘页面抓取的开销	100
cpu_tuple_cost	计算在一次查询中处理一个数据行的开销	0.01
cpu_index_tuple_cost	计算在一次索引扫描中处理每条索引行的开销	0.005
cpu_operator_cost	计算在一次查询中执行一个操作符或函数的开销	0.0025
gp_motion_cost_per_row	计算数据在 motion 操作中将一行数据从一个 Segment 传递到另外一个节点的开销。如果值为 0，则默认开销为 cpu_tuple_cost 的两倍	0
effective_cache_size	为规划器设置在一次索引扫描中可用的磁盘缓冲区的有效大小	512MB

减小 random\_page\_cost 值（相对于 seq\_page\_cost）将导致更倾向于使用索引扫描，而增加这个值将导致更倾向于使用顺序扫描。可以通过同时增加或减少这两个值来调整磁盘 I/O 相对于 CPU 的开销。

下面将使用几个例子来介绍如何计算执行计划的开销。在阅读这一节的时候，建议先阅读 PostgreSQL 文档的 13.1 节（使用 EXPLAIN）。这里直接介绍 Greenplum 的 cost 值计算。以下面这个表关联（A/B 表的分布键均为字段 id）为例进行介绍。

两个表的建表语句如下：

```
testDB=# create table A as select id,id+1 as id2 from generate_series(1,59000)
id distributed by (id);
SELECT 59000
testDB=# create table B as select id,id+1 as id2 from generate_series(1,10000)
id distributed by (id);
SELECT 10000
```

查看两表进行内连接的执行计划：

```
testDB=# explain select * from A,B where A.id=B.id2;
                                         QUERY PLAN
-----
Gather Motion 6:1  (slice2; segments: 6)  (cost=437.00..1365.50 rows=1667
width=16)
-> Hash Join  (cost=437.00..1365.50 rows=1667 width=16)
    Hash Cond: a.id = b.id2
    -> Seq Scan on a  (cost=0.00..656.00 rows=9834 width=8)
    -> Hash  (cost=312.00..312.00 rows=1667 width=8)
        -> Redistribute Motion 6:6  (slice1; segments: 6)
        (cost=0.00..312.00 rows=1667 width=8)
        Hash Key: b.id2
        -> Seq Scan on b  (cost=0.00..112.00 rows=1667 width=8)
(8 rows)
```

执行计划以统计信息的内容为准，这里先了解这两张表的统计信息：

```
testDB=# select relname,relpages,reltuples from pg_class where relname in
('a','b');
  relname | relpages | reltuples
-----+-----+-----
b      |     12 |     10000
a      |     66 |     59000
(2 rows)
```

接下来看看每个 cost 值是如何计算得到的：

```
-> Seq Scan on b  (cost=0.00..112.00 rows=1667 width=8)
```

表 B 有 12 个数据页，10000 行数据，那么 cost 中 112 单位消耗的计算方法如下：

$$12 * \text{seq\_page\_cost} + 10000 * \text{cpu\_tuple\_cost} = 12 + 100 = 112$$

```
-> Redistribute Motion 6:6    (slice1; segments: 6)   (cost=0.00..312.00
rows=1667 width=8)
```

这里是数据重分布，总共有 10000 行数据需要重分布，代价为：

$10000 * \text{gp\_motion\_cost\_per\_row} = 10000 * 2 * \text{cpu\_tuple\_cost} = 200$

加上前面顺序扫描的 112，总消耗就是 312。

对表 B 重分布之后，就是顺序扫描表 A：

```
-> Seq Scan on a  (cost=0.00..656.00 rows=9834 width=8)
```

顺序扫描表 A 使用同样的计算方法，结果是 656。

使用 Hash Join 将两表关联起来：

```
-> Hash Join  (cost=437.00..1365.50 rows=1667 width=16)
```

在开始进行关联的时候，表 B 要先保存在内存的 Hash 表中，所以 437 的 cost 中包括了计算 Hash 值的 312 的 cost，加上了 Hash 关联的其他 cost，开始返回第一条数据的消耗是 437。

实际消耗在关联上的 cost 是  $1365.5 - 437 - 656 = 272.5$ 。

## 5.7 各种执行计划原理分析

### 5.7.1 详解关联的广播与重分布

当两张表关联的时候，如果有一张表关联键不是分布键，那么就会发生表的广播或者重分布，将数据移动到一个节点上进行关联，从而获得数据。这里将详细介绍什么时候用广播，什么时候用重分布。

分布式的关联有两种：

- 单库关联。关联键与分布键一致，只需要在单个库关联后得到结果即可。
- 跨库关联。关联键与分布键不一致，数据需要重新分布，转换成单库关联，从而实现表的关联。

采用广播或者重分布首先必须保证结果的准确性，其次才考虑性能方面。下面将从内连接、左连接、全连接三种情况进行分析：

下面分析 SQL 是通过表 5-4 中的两个表来进行的。

表 5-4 测试表 A 和 B 的定义

表名	字段	分布键	数据量
A	id,id2	id	M
B	id,id2	id	N

## 1. 内连接

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

分布键与关联键相同，属于单库关联，不会造成广播或者重分布。

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

表 A 的关联键是分布键，表 B 的关联键不是分布键，那么可以通过两种方法来实现表的关联。

- 将表 B 按照 id2 字段将数据重分布到每一个节点上，然后再与表 A 进行关联。重分布的数据量是 N。
- 将表 A 广播，每一个节点都放一份全量数据，然后再与表 B 关联得到结果。广播的数据量是  $M \times$  节点数。

所以当  $N > M \times$  节点数的时候，选择表 A 广播，否则选择表 B 重分布。

下面来看实际的例子。

$M=10000, N=50000$ , 节点数 =6:

```
testDB=# explain select * from A,B where A.id=B.id2;
          QUERY PLAN
-----
Gather Motion 6:1 (slice2; segments: 6) (cost=237.00..2088.60 rows=2222 width=16)
  -> Hash Join (cost=237.00..2088.60 rows=2222 width=16)
      Hash Cond: b.id2 = a.id
      -> Redistribute Motion 6:6 (slice1; segments: 6) (cost=0.00..1560.00
rows=8334 width=8)
          Hash Key: b.id2
          -> Seq Scan on b (cost=0.00..560.00 rows=8334 width=8)
          -> Hash (cost=112.00..112.00 rows=1667 width=8)
              -> Seq Scan on a (cost=0.00..112.00 rows=1667 width=8)
(8 rows)
```

$M=10000, N=60100$ , 节点数 =6:

```
testDB=# explain select * from A,B where A.id=B.id2;
          QUERY PLAN
-----
Gather Motion 6:1 (slice2; segments: 6) (cost=1422.25..2830.61 rows=2976 width=16)
  -> Hash Join (cost=1422.25..2830.61 rows=2976 width=16)
      Hash Cond: a.id = b.id2
      -> Broadcast Motion 6:6 (slice1; segments: 6) (cost=0.00..812.00
rows=10000 width=8)
          -> Seq Scan on a (cost=0.00..112.00 rows=1667 width=8)
          -> Hash (cost=671.00..671.00 rows=10017 width=8)
              -> Seq Scan on b (cost=0.00..671.00 rows=10017 width=8)
(7 rows)
```

情况 3: select \* from A,B where A.id2=B.id2;

对于这种情况，两个表的关联键及分布键都不一样，那么还有两种做法：

- 将表 A 与表 B 都按照 id2 字段，将数据重分布到每个节点，重分布的代价是 M+N。
- 将其中一张表广播后再关联，当然选取小表广播，代价小，广播的代价是  $\min(M,N) \times$  节点数。

所以当  $N+M > \min(M,N) \times$  节点数的时候，选择小表广播，否则选择两个表都重分布。

$M=10000$ ,  $N=50000$ , 节点数 =6, 两个表都重分布:

```
testDB=# explain select * from A,B where A.id2=B.id2;
          QUERY PLAN
-----
Gather Motion 6:1 (slice3; segments: 6) (cost=437.00..2247.00 rows=1667 width=16)
  -> Hash Join  (cost=437.00..2247.00 rows=1667 width=16)
      Hash Cond: b.id2 = a.id2
      -> Redistribute Motion 6:6 (slice1; segments: 6) (cost=0.00..1560.00
rows=8334 width=8)
          Hash Key: b.id2
          -> Seq Scan on b  (cost=0.00..560.00 rows=8334 width=8)
          -> Hash  (cost=312.00..312.00 rows=1667 width=8)
              -> Redistribute Motion 6:6 (slice2; segments: 6)
(cost=0.00..312.00 rows=1667 width=8)
          Hash Key: a.id2
          -> Seq Scan on a  (cost=0.00..112.00 rows=1667 width=8)
(10 rows)
```

$M=10000$ ,  $N=50100$ , 节点数 =6, 将表 A 广播:

```
testDB=# explain select * from A,B where A.id2=B.id2;
          QUERY PLAN
-----
Gather Motion 6:1 (slice2; segments: 6) (cost=1187.25..2400.02 rows=1672 width=16)
  -> Hash Join  (cost=1187.25..2400.02 rows=1672 width=16)
      Hash Cond: a.id2 = b.id2
      -> Broadcast Motion 6:6 (slice1; segments: 6) (cost=0.00..812.00
rows=10000 width=8)
          -> Seq Scan on a  (cost=0.00..112.00 rows=1667 width=8)
          -> Hash  (cost=561.00..561.00 rows=8350 width=8)
              -> Seq Scan on b  (cost=0.00..561.00 rows=8350 width=8)
(7 rows)
```

## 2. 左连接

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

单库关联，不涉及数据跨库关联。

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

由于左表的分布键是关联键，鉴于左连接的性质，无论表 B 数据量多大，都必须将表 B 按照字段 id2 重分布数据。

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

左表的关联键不是分布键，由于左连接 A 表肯定是不能被广播的，所以有两种方式：

□ 将表 A 按照 id2 重分布数据，转换成情况 A，代价为 M。

□ 将表 B 广播，代价为  $N \times$  节点数。

$M = 60100$ ,  $N = 10000$ , 节点数 = 6, 将表 B 广播：

```
testDB=# explain select * from A left join B on A.id2=B.id;
                                         QUERY PLAN
-----
Gather Motion 6:1 (slice2; segments: 6) (cost=1562.00..3135.25 rows=10017 width=16)
  -> Hash Left Join  (cost=1562.00..3135.25 rows=10017 width=16)
      Hash Cond: a.id2 = b.id
      -> Seq Scan on a  (cost=0.00..672.00 rows=10017 width=8)
      -> Hash  (cost=812.00..812.00 rows=10000 width=8)
          -> Broadcast Motion 6:6  (slice1; segments: 6)
(cost=0.00..812.00 rows=10000 width=8)
          -> Seq Scan on b  (cost=0.00..112.00 rows=1667 width=8)
(7 rows)
```

$M = 59000$ ,  $N = 10000$ , 节点数 = 6, 将表 A 广播：

```
testDB=# explain select * from A left join B on A.id2=B.id;
                                         QUERY PLAN
-----
Gather Motion 6:1 (slice2; segments: 6) (cost=237.00..2835.50 rows=9834 width=16)
  -> Hash Left Join  (cost=237.00..2835.50 rows=9834 width=16)
      Hash Cond: a.id2 = b.id
      -> Redistribute Motion 6:6  (slice1; segments: 6)  (cost=0.00..1836.00
rows=9834 width=8)
      Hash Key: a.id2
      -> Seq Scan on a  (cost=0.00..656.00 rows=9834 width=8)
      -> Hash  (cost=112.00..112.00 rows=1667 width=8)
          -> Seq Scan on b  (cost=0.00..112.00 rows=1667 width=8)
(8 rows)
```

情况 4: `select * from A left join B on A.id2=B.id2;`

还是有两种方式：

□ 将表 A 与表 B 都按照 id2 字段将数据重分布一遍，转换成情况 1。代价是  $M+N$ 。

□ 表 A 不能被广播，只能将表 B 广播，代价是  $N \times$  节点数。

对于有多种情况的，Greenplum 总是聪明地选择代价小的方式来执行 SQL。

### 3. 全连接

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

对于关联键都是分布键的情况，前面已经讲到，在 Greenplum 中全连接只能采用 Merge Join 来实现。

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

将不是关联键不是分布键的表重分布数据，转换成情况1来解决。无论A、B大小分别为多少，为了实现全连接，不能将表广播，只能是重分布。

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

将两张表都重分布，转换成情况1进行处理。

从前面可以看到，理论上Greenplum选择广播或者重分布这种方法是没有问题的，但是表的大小是怎么得到的呢？这就只能根据统计信息来判断了，如果统计信息出现错误，那么就会导致执行计划出错，5.8节案例分析中将会分析统计信息丢失或者不详细导致的执行计划出错的问题。

将表A和表B进行关联，如果其中一个表需要重分布数据，那么重分布的Hash Key是否一定是关联键呢？

不一定，看下面的执行计划：

```
testDB=# explain select * from A,B where A.id=B.id2 and A.id=1;
                                         QUERY PLAN
-----
Gather Motion 6:1  (slice2; segments: 6)  (cost=0.00..940.53 rows=1 width=16)
  -> Nested Loop  (cost=0.00..940.53 rows=1 width=16)
    -> Redistribute Motion 6:6  (slice1; segments: 6)  (cost=0.00..137.02
rows=1 width=8)
      Hash Key: 1
      -> Seq Scan on b  (cost=0.00..137.00 rows=1 width=8)
          Filter: l = id2
      -> Seq Scan on a  (cost=0.00..803.50 rows=1 width=8)
          Filter: id = 1
(8 rows)
```

由于关联的时候，表A的分布键上有一个过滤条件，即id=1，表示其实数据只会在其中一个节点上有，因此只需要将表B的数据值全部移动到这个节点上就可以完成关联。在重分布的时候Hash Key为1，整个计算都只在一个节点上进行，一般数据量比较小。

## 5.7.2 HashAggregate与GroupAggregate

在PostgreSQL/Greenplum数据库中，聚合函数有两种实现方式：HashAggregate与GroupAggregate。

我们现在通过一个最简单的SQL来分析这两种聚合的区别及其应用场景。

```
select count(1) from pg_class group by oid;
```

### 1. 两种实现算法的比较

#### (1) HashAggregate

对于hash聚合来说，数据库会根据Group By字段后面的值算出hash值，并根据前面使用的聚合函数在内存中维护对应的列表。如果select后面有两个聚合函数，那么在内存中就

会维护两个对应的数据。同样的，有 n 个聚合函数就会维护 n 个同样的数组。对于 hash 聚合来说，数组的长度肯定是大于 Group By 的字段的 distinct 值的个数的，且与这个值应该呈线性关系，Group By 后面的字段重复值越少使用的内存也就越大。

执行计划如下：

```
testDB=# explain select count(1) from pg_class group by oid;
          QUERY PLAN
-----
      HashAggregate  (cost=1721.40..2020.28 rows=23910 width=4)
        Group By: oid
        ->  Seq Scan on pg_class  (cost=0.00..1004.10 rows=143460 width=4)
    Settings: enable_seqscan=on
(4 rows)
```

## (2) GroupAggregate

对于普通聚合函数，使用 GroupAggregate，其原理是先将表中的数据按照 Group By 的字段排序，这样同一个 Group By 的值就在一起，只需要对排好序的数据进行一次全扫描，并进行对应的聚合函数的计算，就可以得到聚合的结果了。

执行计划如下：

```
testDB=# set enable_hashagg = off;
SET
testDB=# explain select count(1) from pg_class group by oid;
          QUERY PLAN
-----
      GroupAggregate  (cost=13291.66..14666.48 rows=23910 width=4)
        Group By: oid
        ->  Sort  (cost=13291.66..13650.31 rows=143460 width=4)
            Sort Key: oid
            ->  Seq Scan on pg_class  (cost=0.00..1004.10 rows=143460 width=4)
    Settings: enable_hashagg=off; enable_seqscan=on
(6 rows)
```

从上面两个执行计划的消耗来说，GroupAggregate 由于需要排序，效率很差，消耗是 HashAggregate 的 7 倍。所以在 Greenplum 中，对于聚合函数的使用，采用的都是 HashAggregate。

## 2. 两种实现的内存消耗

先建立一张测试表，并且向里面插入数据，通过每个字段的数据重复率不一致，还有聚合函数的个数来观察 HashAggregate 与 GroupAggregate 在内存的消耗情况及实际的计算时间的比较。

### (1) 表结构

建立测试表，该表结构如下：

```
create table test_group(
    id integer
    ,col1 numeric
    ,col2 numeric
    ,col3 numeric
    ,col4 numeric
    ,col5 numeric
    ,col6 numeric
    ,col7 numeric
    ,col8 numeric
    ,col9 numeric
    ,col11 varchar(100)
    ,col12 varchar(100)
    ,col13 varchar(100)
    ,col14 varchar(100)
)distributed by(id);
```

## (2) 插入数据

通过 random 函数，实现每个字段数据重复率都不同：

```
testDB=# insert into test_group
testDB# select generate_series(1,100000),
testDB#           (random()*200)::int,
testDB#           (random()*800)::int,
testDB#           (random()*1600)::int,
testDB#           (random()*3200)::int,
testDB#           (random()*6400)::int,
testDB#           (random()*12800)::int,
testDB#           (random()*40000)::int,
testDB#           (random()*100000)::int,
testDB#           (random()*1000000)::int,
testDB#           'hello',
testDB#           'welcome',
testDB#           'haha',
testDB#           'chen';
INSERT 0 100000
```

表大小为：

```
testDB=# select pg_size_pretty(pg_relation_size('test_group'));
pg_size_pretty
-----
12 MB
(1 row)
```

## (3) 使用 explain analyze 来观察实际数据库消耗的内存差异

以下是根据底层单个节点来计算的，避免了广播的时间及内存消耗。

HashAggregate：

```
testDB=# explain analyze select sum(col1),sum(col2),sum(col3),sum(col4),sum(co
```

```
15),sum(col6),sum(col7),sum(col8),sum(col9) from test_group group by col5;
                                         QUERY PLAN
-----
HashAggregate  (cost=4186.96..5432.88 rows=38336 width=62)
  Group By: col5
  Rows out: 6401 rows with 289 ms to first row, 295 ms to end, start offset by 0.143 ms.
  Executor memory: 2818K bytes.
-> Seq Scan on test_group  (cost=0.00..1480.56 rows=108256 width=62)
  Rows out: 100000 rows with 0.023 ms to first row, 48 ms to end, start
offset by 0.218 ms.
  Slice statistics:
    (slice0)  Executor memory: 2996K bytes.
  Settings: enable_seqscan=off
  Total runtime: 296.283 ms
(10 rows)
```

### GroupAggregate:

```
testDB=# explain analyze select sum(col1),sum(col2),sum(col3),sum(col4),sum(co
15),sum(col6),sum(col7),sum(col8),sum(col9) from test_group group by col5;
                                         QUERY PLAN
-----
GroupAggregate  (cost=10532.97..14755.93 rows=38336 width=62)
  Group By: col5
  Rows out: 6401 rows with 306 ms to first row, 585 ms to end, start offset by
0.092 ms.
  Executor memory: 8K bytes.
-> Sort  (cost=10532.97..10803.61 rows=108256 width=62)
  Sort Key: col5
  Rows out: 100000 rows with 306 ms to first row, 342 ms to end, start
offset by 0.093 ms.
  Executor memory: 19449K bytes.
  Work_mem used: 19449K bytes.
-> Seq Scan on test_group  (cost=0.00..1480.56 rows=108256 width=62)
  Rows out: 100000 rows with 0.021 ms to first row, 46 ms to end,
start offset by 0.116 ms.
  Slice statistics:
    (slice0)  Executor memory: 19623K bytes.  Work_mem: 19449K bytes max.
  Settings: enable_hashagg=off; enable_seqscan=off
  Total runtime: 586.114 ms
(15 rows)
```

通过这种方法，可以看出，消耗的内存与实际执行时间的比例。在不同的字段重复率下，计算如下 9 个聚合函数，在分别使用 HashAggregate 与 GroupAggregate 这两种方式进行聚合操作的情况下，它们使用内存及执行时间的比较如表 5-5 所示。

```
explain analyze select sum(col1),sum(col2),sum(col3),sum(col4),sum(col5),sum(co
16),sum(col7),sum(col8),sum(col9) from test_group group by id;
```

表 5-5 9 个聚合函数的 HashAggregate 与 GroupAggregate 比较

	group by 字段	col1	col2	col3	col4	col5	col6	col7	col8	col9	id
HashAgg- regate	Executor memory	554KB	786KB	1074KB	1715KB	2996KB	5469KB	13691KB	21312KB	29428KB	29476KB
	时间 (ms)	266	272	275	281	296	323	357	359	352	340
GroupAgg- regate	Executor memory	19623KB	19615KB								
	时间 (ms)	500	533	547	568	589	609	636	652	649	387

在不同的字段重复率情况下，计算 27 个聚合函数，在分别使用 HashAggregate 与 GroupAggregate 这两种方式进行聚合操作时，它们使用内存及执行时间的比较如表 5-6 所示，测试 SQL 如下：

```
explain analyze select sum(col1),sum(col2),……,sum(col19),
max(col1),max(col2),……,max(col19),
avg(col1),avg(col2),……,avg(col19) from test_group group by id;
```

表 5-6 27 个聚合函数的 HashAggregate 与 GroupAggregate 比较

	Group By 字段	col1	col2	col3	col4	col5	col6	col7	col8	col9	id
Hash Aggregate	Executor memory	514KB	1299KB	2340KB	4405KB	8504KB	19687KB	69947KB	93859KB	106419KB	106876KB
	时间 (ms)	504.9	511.0	523.3	559.8	616.9	937.7	1179.0	1395.6	1391.2	1391.1
Group Aggregate	Executor memory	19687KB	19687KB								
	时间 (ms)	759.5	782.5	802.4	838.0	880.3	939.5	1104.7	1256.9	1365.6	1142

可以看出，对于 GroupAggregate 来说，消耗的内存基本上是恒定的，无论对哪个字段进行 Group By。当聚合函数较少的时候，速度也相对较慢，但是相对稳定。

HashAggregate 在少数聚合函数时表现优异，但是对于很多聚合函数的情况，性能和消耗的内存差异很明显。尤其是受 Group By 字段唯一性的影响，字段 count (district) 值越大，HashAggregate 消耗的内存越多，性能下降越明显。

所以在 SQL 中有大量聚合函数，Group By 的字段重复值比较少的时候，应该用 GroupAggregate，而不能用 HashAggregate。

### 5.7.3 Nestloop Join、Hash Join 与 Merge Join

#### (1) Nestloop join

Nestloop join 就是所谓的笛卡尔积，这种关联的效率极差。比方说，有两张 100 万的表

关联，则会产生  $100\text{万} \times 100\text{万} = 1\text{万亿}$  的数据量，如果还有筛选，再对结果集进行筛选，这是相当可怕的。尤其是对于 Greenplum 这种资源独占的系统，一个 SQL 就可以将数据库所有的资源消耗完，造成其他的 SQL 无法运行，如果对两张大表做笛卡尔积，会产生极大的中间表，随时有可能将数据库的存储耗光，导致整个数据库垮掉，后果相当严重，所以在数据库中，应该杜绝这种笛卡尔积。如果在执行计划中，看到了 Nestloop，那就要小心了，一般都是 SQL 有问题。

### (2) Hash join

这是在关联时候采用的一种很高效的方法，它先对其中一张关联的表计算 hash 值，在内存中用一个散列表保存，然后对另外一张表进行全表扫描，之后将每一行与这个散列表进行关联。对于散列表来说，在理想情况下，每一行的关联都只有  $O(1)$  常数的消耗，从而使得表关联达到很高的性能。在一般情况下，Greenplum 都是使用这个关联方式进行等值连接的。

### (3) Merge Join

这种方法是对两张表都按照关联字段进行排序，然后按照排序好的内容顺序遍历一遍，将相同的值连接起来，从而实现了连接。使用这种方法，最大的消耗是对两表进行排序，快速排序至少也要  $O(n \log n)$  的时间复杂度。Greenplum 默认将 Mergejoin 给关闭掉了。



注意 在 PG 文档中，有一节叫做“明确的 JOIN 控制规划器”，这一节说明了执行计划生产过程中表关联的一些规则，建议读者仔细阅读。

## 5.7.4 分析函数：开窗函数和 grouping sets

### 1. 开窗函数

对于如下的 SQL：

```
explain select
row_number() over(partition by offer_type order by join_from)
, row_number() over(partition by member_id order by gmt_create)
from offer;
```

执行计划的图示如图 5-6 所示。

这段 SQL 代码中有两个开窗函数。开窗函数的实现与 Group By 相似，需要把分组 (partitionby) 的字段分布到一个节点上计算，这个表的分布键是 offer\_id，而 offer\_id 不是开窗函数的分区字段，故都要将数据进行重分布才能计算，步骤如下：

- 1) 顺序扫描 appenonly 的 offer 表。

- 2) 按照 member\_id 字段进行重分布。
- 3) 对数据重分布之后按照 member\_id 和 gmt\_create 对其进行排序，然后将排好顺序的数据进行编号，即完成了这个 row\_number 的开窗函数。
- 4) 再按照 offer\_type 对数据进行重分布，用同样的方法计算另外一个开窗函数的值。

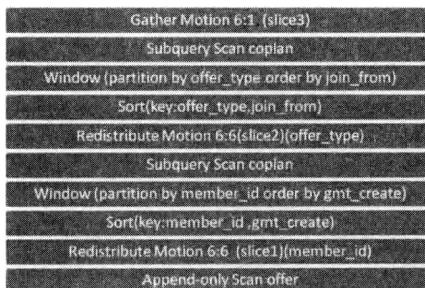


图 5-6 开窗函数的执行计划图示

由于分区字段不是分布键，所以数据全部都要重分布一遍，如果开窗函数太多，会导致数据重分布的次数非常多，每一次重分布每一个 Segment 都要发起一个进程来处理，这会给操作系统和网络都带来一定的压力，所以开窗函数尽量少用，或者用分区键作为分布键，这样也可以减少数据库的消耗。

如果开窗函数是对整个数据进行排序，没有 partition 字段，那么为了维护一个全局的序列，所有数据都必须汇总到 Master 上进行计算，然后再重新分发到每一个节点上，这个性能瓶颈会出现在 Master 上，效率会很差。例如下面这段 SQL 代码：

```
testDB=# explain
testDB# insert into offer2
testDB# select offer_id, row_number() over(order by gmt_create)
testDB#   from offer1;
                                         QUERY PLAN
-----
-----  

Insert (slice0; segments: 6)  (rows=1 width=40)
-> Redistribute Motion 1:6 (slice2; segments: 1)  (cost=0.01..0.06 rows=6 width=40)
    Hash Key: "*SELECT*".offer_id
    -> Subquery Scan "*SELECT*"  (cost=0.01..0.06 rows=6 width=40)
        -> Window  (cost=0.01..0.04 rows=1 width=40)
            Order By: offer1.gmt_create
            -> Gather Motion 6:1 (slice1; segments: 6)
(cost=0.01..0.04 rows=1 width=40)
                Merge Key: offer1.gmt_create
                -> Sort  (cost=0.01..0.02 rows=1 width=40)
                    Sort Key: offer1.gmt_create
                    -> Seq Scan on offer1 (cost=0.00..0.00 rows=1 width=40)
(11 rows)
```

## 2. grouping sets

第2章介绍了grouping sets。使用分析函数grouping sets、cube、rollup可进行多维度分析，如下面的SQL语句：

```
explain select a,b,count(1) from cxa group by grouping sets((a),(b),(a,b));
```

其执行计划图如图5-7所示。

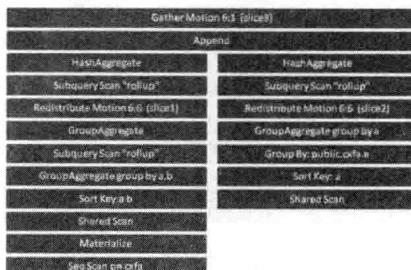


图 5-7 grouping sets 的执行计划图示

grouping sets 的执行步骤如下。

- 1) 顺序扫描 cxa 表，然后将其保存在内存中，之后分两个分支进行。
- 2) 分支 1：读取在内存中的数据，按照 a 执行 GroupAggregate，计算出 a 字段汇总结果（a 是分布键）。
- 3) 分支 2：读取内存中的数据，按照 a、b 执行 GroupAggregate，计算出这两个字段的汇总结果，然后按照 b 字段重分布，再计算出 b 字段的汇总结果。
- 4) 将分支 1 与分支 2 的结果都进行重分布，然后分别执行 HashAggregate。
- 5) 将结果在 Master 上汇总起来。

## 5.8 案例

### 5.8.1 关联键强制类型转换，导致重分布

两表的关联键 id 的类型都是一样的，都是 integer 类型。如果强制将两个 integer 类型转换成其他类型，会导致两个表都要重分布。

正常关联的执行计划如下：

```
testDB=# explain select * from test1 a join test2 b on a.id=b.id;
          QUERY PLAN
```

```
Gather Motion 6:1 (slice1) (cost=0.01..0.05 rows=3 width=300)
-> Hash Join (cost=0.01..0.05 rows=3 width=300)
```

```

Hash Cond: a.id = b.id
-> Seq Scan on test1 a  (cost=0.00..0.00 rows=1 width=150)
-> Hash  (cost=0.00..0.00 rows=1 width=150)
    -> Seq Scan on test2 b  (cost=0.00..0.00 rows=1 width=150)
Settings: enable_seqscan=off
(7 rows)

```

强制将两个表的执行计划转换成 numeric 之后的执行计划：

```

testDB=# explain select * from test1 a join test2 b on a.id::numeric=b.
id::numeric;
                                         QUERY PLAN
-----
-----
Gather Motion 6:1  (slice3)  (cost=0.03..0.10 rows=3 width=300)
-> Hash Join  (cost=0.03..0.10 rows=3 width=300)
    Hash Cond: a.id::numeric = b.id::numeric
    -> Redistribute Motion 6:6 (slice1)  (cost=0.00..0.02 rows=1 width=150)
        Hash Key: a.id::numeric
        -> Seq Scan on test1 a  (cost=0.00..0.00 rows=1 width=150)
    -> Hash  (cost=0.02..0.02 rows=1 width=150)
        -> Redistribute Motion 6:6 (slice2)  (cost=0.00..0.02 rows=1 width=150)
            Hash Key: b.id::numeric
            -> Seq Scan on test2 b  (cost=0.00..0.00 rows=1 width=150)
Settings: enable_seqscan=off
(11 rows)

```

可以看出，由于两个表刚开始的时候都是按照 integer 的类型进行分布的，但是关联的时候强制将类型转换成 numeric 类型，由于 integer 与 numeric 的 hash 值是不一样的，所以数据需要重分布到新的节点进行关联。

## 5.8.2 统计信息过期

前面已经讲到，当统计信息过期的时候，会导致执行计划出错。选择一个糟糕的执行计划，会导致很大的数据库开销。

一般的解决办法就是将表重新使用 analyze 分析一下，重新收集统计信息。或者使用 vacuum full analyze 对表中的空洞进行回收，从而提高性能。

## 5.8.3 执行计划出错

有时候统计信息是正确的，但是由于信息不够全面，或者执行的优化器还不够精准，可能会对结果集大小的估计有很大的偏差。

例如，在 SQL 中加入了一个无用的条件：`id::integer&0=0;`

```

testDB=# select count(1) from offer_2 ;
      count
-----

```

```

1000000
(1 row)
testDB=# select count(1) from offer_2 where id::integer&0=0;
      count
-----
1000000
(1 row)

```

以上两个 SQL 的数据量是一样的，但是执行计划看起来却有很大的区别：

```

testDB=# explain select * from offer_2 ;
          QUERY PLAN
-----
Gather Motion 6:1 (slice1; segments: 6) (cost=0.00..30654.00 rows=166667 width=3729)
  -> Append-only Scan on offer_2  (cost=0.00..30654.00 rows=166667 width=3729)
(2 rows)

testDB=# explain select * from offer_2 where id::integer&0=0;
          QUERY PLAN
-----
--- Gather Motion 6:1  (slice1; segments: 6)  (cost=0.00..38154.00 rows=167
width=3729)
  -> Append-only Scan on offer_2  (cost=0.00..38154.00 rows=167 width=3729)
    Filter: (id::integer & 0) = 0
(3 rows)

```

从 cost 值可以看出，数据库对结果集的估计，一个是 167，一个是 166667，相差将近 1000 倍。如果数据库通过这个估计值去判断表是进行广播还是重分布，就有可能会将大表广播，数据量太大，会导致内存占用过高、关联的性能变差，进而会导致执行太慢，或者因为内存不足而导致 SQL 报错。

```

testDB=# explain select * from offer_2 a,member b where a.member_id=b.member_id;
          QUERY PLAN
-----
-----
Gather Motion 6:1  (slice3; segments: 6)  (cost=177901.00..6989899.10
rows=84930667 width=7448)
  -> Hash Join  (cost=177901.00..6989899.10 rows=84930667 width=7448)
    Hash Cond: b.member_id::text = a.member_id::text
      -> Redistribute Motion 6:6 (slice1; segments: 6) (cost=0.00..97955.57
rows=165570 width=3719)
        Hash Key: b.member_id::text
        -> Seq Scan on member b (cost=0.00..78087.19 rows=165570 width=3719)
      -> Hash  (cost=50654.00..50654.00 rows=166667 width=3729)
        -> Redistribute Motion 6:6 (slice2; segments: 6)
(cost=0.00..50654.00 rows=166667 width=3729)
        Hash Key: a.member_id::text
        -> Append-only Scan on offer_2 a (cost=0.00..30654.00
rows=166667 width=3729)
(10 rows)

```

这个是正常的执行计划，将两张表都重分布，但是如果加上了条件 `a.id::integer&0=0`，则 Greenplum 会以为结果集很小，从而误将大表广播：

```
testDB=# explain select * from offer_2 a,member b where a.member_id=b.member_id
and a.id::integer&0=0;
                                         QUERY PLAN
-----
-----+
Gather Motion 6:1  (slice2; segments: 6)  (cost=38299.00..131609.34 rows=84931
width=7448)
-> Hash Join  (cost=38299.00..131609.34 rows=84931 width=7448)
    Hash Cond: b.member_id::text = a.member_id::text
    -> Seq Scan on member b  (cost=0.00..78087.19 rows=165570 width=3719)
    -> Hash  (cost=38224.00..38224.00 rows=1000 width=3729)
        -> Broadcast Motion 6:6  (slice1; segments: 6)
(cost=0.00..38224.00 rows=1000 width=3729)
        -> Append-only Scan on offer_2 a  (cost=0.00..38154.00
rows=167 width=3729)
                Filter: (id::integer & 0) = 0
(8 rows)
```

`cost` 值似乎变小了。Greenplum 认为 `offer_2` 表在筛选之后只有 1000 条，相对 `member` 表数据量非常小，因此执行计划认为将 `offer_2` 表广播的代价比较小，就采用了广播的策略，而实际上是将 100 万记录广播了。广播 `offer_2` 的代价非常的大，对于有些 SQL，有可能因为广播使用内存过多，造成内存不足，从而导致 SQL 运行报错。

对于这种执行计划出错，没有很好的办法，只能将 SQL 拆分，物化一张新的表来实现。

```
create table offer_tmp as select * from offer_2 where id::integer&0=0;
```

通过物化，让 Greenplum 重新收集 `offer_tmp` 的信息，然后再与 `member` 表进行关联，才能得到正确的执行计划。

#### 5.8.4 分布键选择不恰当

分布键选择不当一般有两种情况：

- 随便选择一个字段作为分布键，导致关联的时候需要重分布一个表来关联。
- 分布键导致数据分布不均，SQL 都卡在一个 Segment 上进行计算。

对于第一种情况，我们可以通过查询执行计划来得知。当执行计划出现 `Redistribute Motion` 或 `Broadcast Motion` 时，我们就知道重新分布了数据，这个时候就要留意分布键选择是否有误，进而导致多余的重分布，比如一个表用了字段 `id` 来分布，另外一个表通过 `id` 和 `name` 两个字段来分布，然后通过 `id` 来进行关联，这个时候也会导致数据重分布。

第二种情况就比较麻烦，因为在执行计划中，我们看不出 SQL 有什么问题，往往要到 SQL 执行非常慢的时候才意识到有问题。在数据分布不均中，有一个特例，就是空值，这是一个比较常见的问题。我们将在第 10 章中详细介绍如何排除这种问题。

下面将介绍几个方法来判断表是否分布不均。

1) 每个表都有一个隐藏字段 `gp_segment_id`, 表示数据是在哪个 Segment 上的, 我们可以对这个字段进行 Group By 来查看每个节点的数据量。

```
testDB=# select gp_segment_id, count(1) from test01 group by 1 order by 1 ;
gp_segment_id | count
-----+-----
 0 | 3948
 1 | 3576
 2 | 5448
 3 | 4020
 4 | 4740
 5 | 3396
(6 rows)
```

2) 对于 `appendonly` 表, 我们还可以通过 `get_ao_distribution` 函数来获取数据分布的信息。

```
testDB=# select * from get_ao_distribution('test01') order by 1;
segmentid | tupcount
-----+-----
 0 |      3948
 1 |      3576
 2 |      5448
 3 |      4020
 4 |      4740
 5 |      3396
(6 rows)
```

3) 在第 10 章中, 会介绍视图 `all_seg_sql`, 通过这个视图, 我们可以查看子节点上正在运行的所有 SQL。

如果我们在数据库中发现一条 SQL 执行了很长时间, 但是在执行计划中看不出有什么问题, 这个时候, 我们可以查出这条 SQL 的 `sess_id`, 然后通过这个 `sess_id`, 用下面的 SQL 查询所有节点 SQL 的运行情况。如果只发现其中小部分节点还在运行, 则表示大多数都是数据分布不均导致的。

```
select * from all_seg_sql where sess_id=xxxx;
```

还有很多种数据分布不均的情况很难发现, 如果 SQL 比较复杂, 我们可以查询表是否分布均匀, 但是由于有重分布, 而对于 Greenplum 来说, 重分布并不会考虑数据是否均衡, 因此会导致原表可能是分布均匀的, 中间却发生了重分布 (关联或者是聚合引起的)。这样就更难定位到问题了, 如果通过 `all_seg_sql` 观察到有数据不均, 那就要根据 SQL 业务逻辑的理解或者将 SQL 拆分成小的 SQL 来进行分析, 看看到底是哪一步导致的数据分布不均。

## 5.8.5 计算 `distinct`

在 SQL 中使用 `distinct` 一般有两种办法。

第一种是将全部数据按照使用 `distinct` 那个字段排序，然后执行一个 `unique` 操作去掉重复的数据，这样效率是比较差的。

```
testDB=# explain select distinct coll from test10;
                                         QUERY PLAN
-----
-----
Gather Motion 6:1  (slice2)  (cost=858107.25..880419.21 rows=137665 width=6)
  Merge Key: coll
    -> Unique  (cost=858107.25..880419.21 rows=137665 width=6)
      Group By: coll
        -> Sort  (cost=858107.25..869263.23 rows=4462392 width=6)
          Sort Key (Distinct): coll
            -> Redistribute Motion 6:6 (slice1)  (cost=0.00..140812.76 rows=4462392
width=6)
              Hash Key: coll
                -> Seq Scan on test10 (cost=0.00..51564.92 rows=4462392 width=6)
Settings: enable_seqscan=off
(10 rows)

Time: 0.958 ms
```

第二种是按照使用 `distinct` 哪个字段来计算 hash 值，然后放到一个 hash 数组中，同样的值会得到相同的 hash 值，从而实现去重的功能。

对于如下的执行计划，从开销来看，只使用了不到第一种执行计划 1/10 的开销。

```
testDB=# explain select coll from test10 group by coll;
                                         QUERY PLAN
-----
-----
Gather Motion 6:1  (slice2)  (cost=67195.01..68571.66 rows=137665 width=6)
  -> HashAggregate  (cost=67195.01..68571.66 rows=137665 width=6)
    Group By: test10.coll
      -> Redistribute Motion 6:6 (slice1)  (cost=62720.90..65474.20
rows=137665 width=6)
        Hash Key: test10.coll
        -> HashAggregate  (cost=62720.90..62720.90 rows=137665 width=6)
          Group By: test10.coll
            -> Seq Scan on test10 (cost=0.00..51564.92 rows=4462392 width=6)
(8 rows)
```

在 Greenplum4.1 中，使用 `distinct` 只能采用第一种方法，这种方式的效率很差。所以我们建议在使用 `distinct` 的时候将 SQL 改写为 `Group By` 形式，这样就能够使用第二种方式，以大大提高性能。在 Greenplum 4.3 版本中，`distinct` 跟 `group by` 两种方式都采用了 `HashAggregate` 这种方式，性能上就区别不大了。

而对于 `count(distinct)`，则没有这个问题，在规划期会选择 `HashAggregate` 来执行这段 SQL：

```

testDB=# explain select count(distinct col1),count(distinct col2) from test10;
          QUERY PLAN
-----
-----
Nested Loop  (cost=179660.54..179660.59 rows=1 width=24)
-> Aggregate  (cost=89608.02..89608.03 rows=1 width=12)
    -> HashAggregate  (cost=85591.56..87599.79 rows=133882 width=12)
        Group By: public.test10.col2
            -> Gather Motion 6:1  (slice1)  (cost=73876.88..83248.62
rows=133882 width=12)
                -> HashAggregate (cost=73876.88..73876.88 rows=133882 width=12)
                    Group By: public.test10.col2
                        -> Seq Scan on test10 (cost=0.00..51564.92 rows=4462392
width=12)
-> Aggregate  (cost=90052.52..90052.53 rows=1 width=12)
    -> HashAggregate  (cost=85922.57..87987.54 rows=137665 width=12)
        Group By: public.test10.col1
            -> Gather Motion 6:1 (slice2)  (cost=73876.88..83513.43 rows=137665
width=12)
                -> HashAggregate (cost=73876.88..73876.88 rows=137665 width=12)
                    Group By: public.test10.col1
                        -> Seq Scan on test10 (cost=0.00..51564.92 rows=4462392
width=12)
Settings: enable_seqscan=off
(16 rows)

```

## 5.8.6 union 与 union all

注意，如果使用 union，会进行去重。在 Greenplum 中，如果不是分布键，去重的就要涉及数据的重分布，而在 Greenplum 中则更加特殊，因为这个去重是以整行数据为分布键的，这样分布键很长，一般 Union 的结果会插入到另外一张表中，又会造成一次数据重分布，效率会较差。

```

testDB=# create table t01
testDB-# as select * from pg_tables
testDB-# distributed by(tablename);
SELECT 254
testDB=# create table t02
testDB-# as select * from pg_tables
testDB-# distributed by(tablename);
SELECT 255
testDB=# create table t03
testDB-# as select * from pg_tables limit 0
testDB-# distributed by(tablename);
SELECT 0

testDB=# explain insert into t03 select * from t01 union select * from t02;
          QUERY PLAN

```

```

-----
Insert (slice0; segments: 6) (rows=85 width=259)
-> Redistribute Motion 6:6 (slice2; segments: 6) (cost=45.06..55.24
rows=85 width=259)
    Hash Key: tablename
-> Unique (cost=45.06..55.24 rows=85 width=259)
    Group By: schemaname, tablename, tableowner, tablespace,
hasindexes, hasrules, hastriggers
-> Sort (cost=45.06..46.34 rows=85 width=259)
    Sort Key (Distinct): schemaname, tablename, tableowner,
tablespace, hasindexes, hasrules, hastriggers
-> Redistribute Motion 6:6 (slice1; segments: 6)
(cost=0.00..22.18 rows=85 width=259)
    Hash Key: schemaname, tablename, tableowner,
tablespace, hasindexes, hasrules, hastriggers
-> Append (cost=0.00..22.18 rows=85 width=259)
-> Seq Scan on t01 (cost=0.00..8.54 rows=43
width=259)
-> Seq Scan on t02 (cost=0.00..8.55 rows=43
width=259)
(12 rows)

```

从执行计划中可以看到，Greenplum 会按照所有的字段作为 key 来去重分布数据，然后按照全部的字段去排序，再去重，从而实现 Union 的操作。



**注意** 分布键字段很多会造成 CPU 消耗比较高，笔者简单测试了一下，分布键为一个字段和 98 个字段的重分布在执行时间上差不多，但是在消耗的 CPU 上，一个字段重分布的时候 CPU 的使用率大概是 35%，98 个字段重分布的 CPU 使用率大概是 45%，网络的消耗基本一致。

使用 Union All 可能会造成不必要的数据重分布，还是用上面的 3 张表。

```

testDB=# create view view01
testDB-# as
testDB-# select * from t01
testDB-# union all
testDB-# select * from t02;
CREATE VIEW
testDB=# explain insert into t03
testDB-# select * from view01;
                                         QUERY PLAN
-----
Insert (slice0; segments: 6) (rows=85 width=259)
-> Redistribute Motion 6:6 (slice1; segments: 6) (cost=0.00..22.18
rows=85 width=259)

```



```
Gather Motion 6:1  (slice3)  (cost=133.43..212.16 rows=1682 width=84)
 -> Hash Left Join  (cost=133.43..212.16 rows=1682 width=84)
     Hash Cond: a.col1::text = b.col2::text
     Filter: b.col2 IS NULL
 -> Redistribute Motion 6:6  (slice1)  (cost=0.00..53.49 rows=1683
 width=16)
     Hash Key: a.col1::text
     -> Seq Scan on test10 a  (cost=0.00..19.83 rows=1683 width=16)
-> Hash  (cost=112.39..112.39 rows=1683 width=68)
     -> HashAggregate  (cost=78.73..95.56 rows=1683 width=6)
         Group By: test10.col2
         -> Redistribute Motion 6:6  (slice2)  (cost=24.04..57.70
 rows=1683 width=6)
             Hash Key: test10.col2
             -> HashAggregate  (cost=24.04..24.04 rows=1683
 width=6)
                 Group By: test10.col2
                 -> Seq Scan on test10  (cost=0.00..19.83
 rows=1683 width=6)

Settings: enable_seqscan=off
(16 rows)
Time: 13.396 ms
```

从 SQL 中的 cost 我们看出，这两个实现一样功能的 SQL 在性能上有极大的差异，对于相同的数据量，一个使用了 30 多秒，另外一个只用了 92 毫秒：

```
testDB=# select count(1) from (select * from test10 where col1 not in (select col2 from test10))t;
   count
-----
 10000
(1 row)
Time: 3278.033 ms

testDB=# select count(1) from (select * from test10 a left join(select col2
from test10 group by col2) b on a.col1=b.col2 where b.col2 is null)t;
   count
-----
 10000
(1 row)
Time: 92.180 ms
```

以上关于 not in 不同写法在性能上的差异在 Greenplum 4.1 中很明显，经过测试，在新版本 Greenplum 4.3 中对其进行了优化，采用了 Hash Left Anti Semi Join 的关联算法来实现了 not in 的语义，性能与改写成 left join 的形式差不多。在 Greenplum 4.3 中，执行计划如下：

```
testDB=# explain select * from test10 where col1 not in (select col2 from test10);
```

```

-----
      Gather Motion 4:1  (slice2; segments: 4)  (cost=12137.44..15541.32 rows=24953
      width=18)
      -> Hash Left Anti Semi Join (Not-In)  (cost=12137.44..15541.32 rows=24953
      width=18)
          Hash Cond: public.test10.col1::text = "NotIn_SUBQUERY".col2::text
          -> Seq Scan on test10  (cost=0.00..1158.12 rows=24953 width=18)
              Filter: col1::text IS NOT NULL
          -> Hash  (cost=7146.84..7146.84 rows=99812 width=274)
              -> Broadcast Motion 4:4  (slice1; segments: 4)
                  (cost=0.00..7146.84 rows=99812 width=274)
                  -> Subquery Scan "NotIn_SUBQUERY"  (cost=0.00..2156.24
                  rows=24953 width=274)
                      -> Seq Scan on test10  (cost=0.00..1158.12
                      rows=24953 width=5)
(9 rows)

```



如果读者想在 Greenplum 4.3 中模拟 4.1 中这种比较慢的执行计划，可以设置参数 enable\_hashjoin=off，然后再运行 SQL 即可。

## 5.8.8 聚合函数太多导致内存不足

在 Greenplum 4.1 数据库中，SQL 进行很多的聚合运算时，有时候会报如下的错误：

```
Error 7 (ERROR: Unexpected internal error: Segment process received signal
SIGSEGV (postgres.c:3360) (seg43 slice1 sdw19-4:30003 pid=26345) (cdbdisp.c:1457))
```

这段 SQL 其实就是占用内存太多，进程被操作系统发出信号干扰导致的报错。

查看执行计划，发现是 HashAggregate 搞的鬼。一般来说，数据库会根据统计信息来选择 HashAggregate 或 GroupAggregate，但是有可能统计信息不够详细或 SQL 太复杂而选错执行计划。

一般遇到这种问题，有两种方法：

- 1) 拆分成多个 SQL 来执行，减少 HashAggregate 使用的内存。
- 2) 在执行 SQL 之前，先执行 enable\_hashagg = off；将 HashAggregate 参数关掉。强制不采用 HashAggregate 这种聚合方式，则数据库会采用 GroupAggregate，虽然增加了排序的代价，但是内存使用量是可控的，建议用这种方式，比较简单。

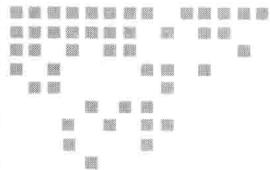
下次如果再遇到这种内存不足的报错，并且 SQL 中有很多的聚合函数，建议采用这两种方法改脚本重新运行。

## 5.9 小结

对于所有数据库来说，学会阅读执行计划，可以让我们了解整个数据库的运行方式。对于 SQL 调优来说，执行计划是一个强有力的利器，本章介绍了以下几个方面：

- 阅读执行计划。
- 统计信息对执行计划的影响。
- 各种执行计划的原理。
- 执行计划案例分析。

Greenplum 与其他数据库在执行计划上的最大区别就是广播与重分布，而这两个过程又严重依赖于统计信息的完整性，对于因统计信息不完善而导致的执行计划出错，就需要将 SQL 拆分来实现。Greenplum 的执行计划相对其他数据库的执行计划更容易出错，而且广播大表有可能耗尽数据库所有的资源，因此在分析执行时间过长的 SQL 时，应当首先从执行计划入手。



## 第 6 章 Greenplum 高级应用

本章将介绍一些 Greenplum 的高级特性，主要是与其他关系型数据库有区别的地方。通过本章的介绍，读者会对 Greenplum 中针对数据仓库 OLAP 类型的一些优化有一个更加深入的了解。

当今的数据处理大致可以分成两大类：联机事务处理 OLTP (On-Line Transaction Processing)、联机分析处理 OLAP (On-Line Analytical Processing)。OLTP 是传统的关系型数据库的主要应用，主要是基本的、日常的事务处理，例如银行交易。OLAP 是数据仓库系统的主要应用，支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。表 6-1 列出了 OLTP 与 OLAP 之间的比较。

表 6-1 OLTP 与 OLAP 的比较

	OLTP	OLAP
用户	前台用户	ETL、BI、分析人员
功能	日常操作处理	分析决策
DB 设计	面向应用	面向主题
数据	当前的、最新的、细节的、二维的、分立的	历史的、聚集的、多维的、集成的、统一的
存取	读/写数十条记录	读/写上百万条记录
工作单位	简单的事务	复杂的查询
DB 大小	100MB ~ 100GB	100GB 以上

看完了上面 OLTP 和 OLAP 在应用上的比较，现在让我们来看一下它们在硬件的使用特性上有哪些明显的区别，如表 6-2 所示。

表 6-2 OLTP 和 OLAP 在硬件性能上的差异

	OLTP	OLAP
磁盘 IO	大部分随机 IO，数据量小，重响应时间	顺序 IO，数据量大，重吞吐
内存	数据量小，可以将热点数据缓存在内存中，加快响应时间。 缓存表数据，每次查询都共用这些数据	数据量大，每次读取是全表扫描，内存主要用在排序、hash join 等操作上，还有缓存数据字典 缓存 SQL 中间结果数据，SQL 执行结束后，这部分内存就会被下一个 SQL 挤出内存
网络	每次返回结果集一般较小，网络主要在于延时时间小。 当 SQL 很多时，网络吞吐也有较高要求	对响应时间没有太高要求，但是对于 Greenplum，重分布和广播要求网卡要有很高的吞吐
CPU	TPS 高时，CPU 需要处理大量的 SQL，CPU 消耗较高	依据 SQL 类型的不同，一般 CPU 使用较小，瓶颈在 IO。但对于某些计算性 SQL，CPU 消耗也可以很大

## 6.1 Appendonly 表与压缩表

对于数据仓库应用来说，由于数据量大，而且每次的分析几乎都是全表扫描，因此磁盘无论在存储上还是在吞吐上，都是一个瓶颈，为了缓解这个问题，Greenplum 引入了 Appendonly 表和压缩表的概念。

### 6.1.1 应用场景及语法介绍

压缩表必须是 Appendonly 表。Appendonly 表顾名思义，就是只能不断追加的表，不能进行更新和删除。

#### (1) 压缩表的应用场景

- 1) 业务上不需要对表进行更新和删除操作，用 truncate+insert 就可以实现业务逻辑。
- 2) 访问表的时候基本上是全表扫描，不需要在表上建立索引。
- 3) 不能经常对表进行加字段或修改字段类型，对 Appendonly 表加字段比普通表慢很多。

#### (2) 语法介绍

建表的时候加上 with (appendonly=true) 就可以指定表是 Appendonly 表。如果需要建压缩表，则加上 with (appendonly=true, compresslevel=5)，其中 compresslevel 是压缩率，取值为 1 ~ 9，一般选择 5 已经足够了。

### 6.1.2 压缩表的性能差异

由于数据仓库的大部分应用都是 IO 密集型的，当然，这与具体的业务场景有关。对于数据仓库，每次的查询基本上都是全表扫描，大量的顺序读写。

下面简单测试普通表与压缩表在性能上的差距，由于测试的都是 IO 密集型的 SQL，因此性能差异基本上是与压缩率成正比的。压缩率是跟实际数据类型相关的，在我们的系统中，大部分的表压缩率大概是在 3 ~ 4 之间。

一张简单的商品表，总数据量 500 万条，常见的数据类型有 date、numeric、integer、varchar、text 等，普通表与压缩表性能比较结果如表 6-3 所示。

表 6-3 普通表与压缩表的性能比较

	堆表	压缩表 (compresslevel=5)
表大小	12 GB	2774 MB
全表扫描时间	29406.261 ms	57369.278 ms
导入时间	82368.702 ms	529447.779 ms



测试数据与硬件环境、测试方法及业务场景有很大的关系，故测试数据仅供参考。

### 6.1.3 Appendonly 表特性

对于每一张 Appendonly 表，创建的时候都附带创建一张 pg\_aoseg 模式下的表，表名一般是 pg\_aoseg\_ + 表的 oid，这张表中保存了这张 Appendonly 表的一些原数据信息，下面将做详细的介绍。

首先建一张 Appendonly 的表：

```
testDB=# create table test_appendonly with (appendonly=true , compresslevel=5)
testDB=# as
testDB=# select generate_series(0,1000) a,'helloworld'::varchar(50) b
testDB=# distributed by (a);
SELECT 1001
```

查一下表的 oid：

```
testDB=# select oid from pg_class where relname='test_appendonly';
      oid
-----
23119
(1 row)

testDB=# select oid,oid::regclass from pg_class where relname='test_appendonly'
or relname like '%23119%';

      oid |          oid
-----+-----
23121 | pg_toast.pg_toast_23119
23122 | pg_toast.pg_toast_23119_index
23119 | test_appendonly
23124 | pg_aoseg.pg_aoseg_23119
```

```
23125 | pg_aoseg.pg_aoseg_23119_index
(5 rows)
```

aoseg 表的字段信息：

```
testDB=# \d pg_aoseg.pg_aoseg_23119
?o? "pg_aoseg.pg_aoseg_23119"
   Column      |      Type
-----+-----
segno      | integer
eof        | double precision
tupcount   | double precision
varblockcount | double precision
eofuncompressed | double precision
```

这个表的每个字段的含义在 Greenplum 的文档中没有，测试后每个字段信息如表 6-4 所示。

表 6-4 pg\_aoseg 表的字段信息

字段名	字段含义
segno	一个编号，在 pg_aoseg 表，可能会有多行记录，通过 segno 来标识
eof	压缩后文件大小
tupcount	表的数据量
varblockcount	表占用的块数量
eofuncompressed	未压缩表的大小

在第 4 章中，我们介绍了一个函数 gp\_dist\_random，通过这个函数，我们可以查询子节点的信息，可以知道 Greenplum 底层每个数据节点的数据量、数据文件大小的情况，这样我们就能够分析表的数据分布情况、算出倾斜率等。

在 Greenplum 中，提供了两个函数 get\_ao\_compression\_ratio 和 get\_ao\_distribution，用于查询 Appendonly 表的压缩率和每个子节点数据量，其实这两个函数就是利用 pg\_aoseg 这个表来查询的，下面演示一下这两个函数的结果。

get\_ao\_compression\_ratio：查询压缩表的压缩率，与使用 gp\_dist\_random 查询底层 pg\_aoseg 前缀的表的结果是一样的。

```
testDB=# select * from get_ao_compression_ratio('test_appendonly');
get_ao_compression_ratio
-----
5.74
(1 row)
testDB=# select sum(eofuncompressed)/sum(eof) as compression_ratio from gp_
dist_random('pg_aoseg.pg_aoseg_23119');
compression_ratio
-----
5.73555166374781
```

(1 row)

**get\_ao\_distribution:** 查询每个子节点的数据量。

```
testDB=# select * from get_ao_distribution('test_appendonly') order by segmentid;
segmentid | tupcount
-----+-----
 0 |      161
 1 |      161
 2 |      159
 3 |      161
 4 |      181
 5 |      178
(6 rows)

testDB=# select gp_segment_id,tupcount from gp_dist_random('pg_aoseg.pg_aoseg_23119') order by gp_segment_id;
gp_segment_id | tupcount
-----+-----
 0 |      161
 1 |      161
 2 |      159
 3 |      161
 4 |      181
 5 |      178
(6 rows)
```

基于 Appendonly 表，开发了两个 Greenplum 的函数：get\_table\_count 和 get\_table\_info。

get\_table\_count 是获取 Appendonly 表的行数的，Appendonly 表有一个字段用于保存这些信息。这个函数获取表的行数快，支持分区表。

```
testDB=# select get_table_count('public.test_appendonly');
get_table_count
-----
 1001
(1 row)
```

get\_table\_info 用于获取更加详尽的表的信息，有表的行数、大小、数据倾斜率（最大节点数据量 / 平均节点数据量）、压缩率，对于分区表，既有汇总信息，也有每个单表的数据信息。

对于非分区表，就只有一行数据。对于分区表，第一行则是所有分区的汇总数据，其他是每一个分区的具体信息，效果如图 6-1 所示。

tablename	subparname	tablecount	tablesize	pretty_size	max_div_avg	compression_ratio
public.test_appendonly		1001	4568	4568 bytes	1.08491508492	5.73555166375

tablename	subparname	tablecount	tablesize	pretty_size	max_div_avg	compression_ratio
public.hello_partition	##ALL##	66200	263680	258 kb		8.543386
public.hello_partition	p20101230	10000	39792	39 kb	1.0062	8.552700
public.hello_partition	p20101231	11000	43832	43 kb	1.00690909091	8.53987954006
public.hello_partition	p20110101	11300	45008	44 kb	1.00672586372	8.54337006754
public.hello_partition	p20110102	11300	45024	44 kb	1.00672566372	8.54033404407
public.hello_partition	p20110103	11300	45008	44 kb	1.00672566372	8.54337006754
public.hello_partition	p20110104	11300	45016	44 kb	1.00672566372	8.54185178603

图 6-1 get\_table\_info 使用效果

创建 get\_table\_info 函数的代码如下：

-- 该函数只对 Appendonly 表有效

```
create type public.table_info as (
    tablename text                                -- 表名
    ,subparname text                             -- 分区名
    ,tablecount bigint                         -- 表的行数
    ,tablesize bigint                          -- 表大小(单位:字节)
    ,pretty_size text                           -- 格式化大小输出
    ,max_div_avg float                         -- 斜率, 最大节点数据量 / 平均节点数据量
    ,compression_ratio float                  -- 压缩率
);
-- 将 regclass 类型转换成 text 类型
CREATE or replace FUNCTION regclass2text(a regclass)
RETURNS text
AS $$
return a;
$$ LANGUAGE plpython;

-- 获取表信息的函数
CREATE or replace FUNCTION public.get_table_info(tablename text)
RETURNS setof table_info
AS $$
def one_table_info(plpy,tablename,subparname,aosegname,privilege):
    aosegsql="";
    #plpy.info(privilege);
    if privilege=='1':
        aosegsql='''
            select '%s' tablename,'%s' subparname,
                   COALESCE(sum(tupcount)::bigint,0) tablecount,
                   COALESCE(sum(eof)::bigint,0) tablesize,
                   pg_size_pretty(COALESCE(sum(eof)::bigint,0)) pretty_size,
                   COALESCE(max(tupcount)::bigint,1)/(case when
COALESCE(avg(tupcount),1.0)=0 then 1 else COALESCE(avg(tupcount),1.0) end) max_div_avg,
                   COALESCE(sum(eofuncompressed),1)/(case when
COALESCE(sum(eof),1.0)=0 then 1 else COALESCE(sum(eof),1.0) end) compression_ratio
                   from gp_dist_random('%s');
                   '''%(tablename,subparname,aosegname)
    else :
```

```

        aosegsql='''
        select '%s' tablename,'%s' subparname,
        0 tablecount,
        0 tablesizer,
        'permission denied' prettysize,
        0 max_div_avg,
        0 compression_ratio;
        """%(tablename,subparname)

        result_rv=plpy.execute(aosegsql);
        #plpy.info(result_rv[0]);
        return result_rv[0];

    try:
        table_name = tablename.lower().split('.')[1]
        table_schema = tablename.lower().split('.')[0]
    except (IndexError):
        plpy.error('Please input put "tableschema.table_name"');

#check version of database
check_version_sql = """
    select substring(version(), 'Database (.* build') as version;
"""
rv = plpy.execute(check_version_sql);
version = rv[0]['version'];
plpy.execute("set enable_seqscan=off");
#get table oid
get_table_oid='';
if version>'3.4.0':
    get_table_oid = """
        select a.oid,reloptions,b.segrelid,regclass2text(b.
segrelid::regclass) aosegname,
        relstorage,
        case has_table_privilege(user,b.segrelid,'select')
when 't' then '1' else '0' end privilege
        from pg_class a left join pg_appendonly b on a.oid=b.relid
        where a.oid='%s'::regclass """%(tablename)
else:
    get_table_oid = """ select oid,reloptions,relaosegrelid,regclass2te
xt(relaosegrelid::regclass) aosegname,
        relstorage ,
        case has_table_privilege(user,relaosegre
lid,'select')  when 't' then '1' else '0' end privilege
        from pg_class
        where oid='%s'::regclass
        """%(tablename)

try:
    rv_oid = plpy.execute(get_table_oid, 5)
    if not rv_oid:

```

```

        plpy.error('Did not find any relation named "' + tablename +'".')
except (Error):
    plpy.error( 'Did not find any relation named "' + tablename +'".');

#
table_oid = rv_oid[0]['oid']
if rv_oid[0]['relstorage']!='a':
    plpy.error(tablename + ' is not appendonly table ,this function
only support appendonly table');
    #plpy.info('table_oid');

#check if table is partition table
check_par_table="select count(*) from pg_partition where
parrelid=%s%(table_oid);
if version>'3.4.0':
    tablecount_sql1=""""
        SELECT regclass2text(pp.parrelid::regclass) tabname,pr1.
parname,parruleord,pa.segrelid,regclass2text(pa.segrelid::regclass) aosegname ,
        case has_table_privilege(user,pa.segrelid,'select') when
't' then '1' else '0' end privilege
        FROM pg_partition pp, pg_partition_rule pr1,pg_appendonly pa
        WHERE pp.paristemplate = false
            AND pp.parrelid = %s
            AND pr1.paroid = pp.oid
            AND pa.relid=pr1.parchildrelid
            order by pr1.parruleord;
    """%(table_oid)
else:
    tablecount_sql1=""""
        SELECT regclass2text(pp.parrelid::regclass) tabname,pr1.
parname,parruleord,pc.relaosegrelid,regclass2text(pc.relaosegrelid::regclass)
aosegname ,
        case has_table_privilege(user,pc.relaosegrelid,'select')
when 't' then '1' else '0' end privilege
        FROM pg_partition pp, pg_partition_rule pr1,pg_class pc
        WHERE pp.paristemplate = false
            AND pp.parrelid = %s
            AND pr1.paroid = pp.oid
            AND pc.oid=pr1.parchildrelid
            and relaosegrelid <> 0
            order by pr1.parruleord;
    """%(table_oid)
#    tablecount_sql2=""""
#        select '%s' tablename,null as subparname,pg_relation_size(%s)
tablesize,pg_size_pretty(pg_relation_size(%s)) prettysize;
#    """%(tablename,table_oid,table_oid)

rv=plpy.execute(check_par_table);

```

```

if rv[0]['count']==1:
    a1=plpy.execute(tablecount_sql1);
    result_rv=[];
    rv_tmp=[];

    totalcount=0
    totalsize=0
    unzipsize=0
    compression_ratio=1;
    for i in a1:
        rv_ao = one_table_info(plpy,tablename,i['parname'],i['aosegname'],
                               str(i['privilege']));
        rv_tmp.append(rv_ao);
        totalsize = totalsize + rv_ao['tableszie'];
        totalcount = totalcount + rv_ao['tablecount'];
        unzipsize = unzipsize + rv_ao['tableszie']*rv_ao['compression_ratio'];
    if totalsize==0:
        compression_ratio=1;
    else :
        compression_ratio=unzipsize/totalsize;

    total_count_sql=""""
                    select '%s' as tablename,'###ALL###' as subparname,%d as
tablecount,
                    %d as tableszie,pg_size_pretty(%d::bigint) prettysize,
                    null as max_div_avg,
                    %f as compression_ratio;
                """%(tablename,totalcount,totalsize,totalsize,compression_ratio)
    a2=plpy.execute(total_count_sql);
    result_rv.append(a2[0])

    for i in rv_tmp:
        result_rv.append(i);
    return result_rv;
else :
    result_rv=[];
    rv_ao=one_table_info(plpy,tablename,'',rv_oid[0]
['aosegname'],str(rv_oid[0]['privilege']));
    result_rv.append(rv_ao);
    return result_rv;
$$ LANGUAGE plpythonu;

```

#### 6.1.4 相关数据字典

在 Greenplum3.\* 中，aoseg 表的信息记录在 pg\_class 中的 relaosegrelid 字段中，可用以下 SQL 进行查询：

```
testDB=# select relaosegrelid from pg_class where relname = 'test_appendonly';
relaosegrelid
```

```
-----
383665
(1 row)
```

在 Greenplum4.\* 中, pg\_class 的 relaosegrelid 字段已经取消了, aoseg 表的信息保留在 pg\_appendonly 表中的 segrelid 字段中。

pg\_appendonly 表的字段描述如表 6-5 所示。

表 6-5 pg\_appendonly 表字段描述

字段名	描    述
relid	对应该表的 oid
blocksize	块大小, 8KB~2MB, 默认是 32KB
safefswritesize	安全写入数据的块大小
compresslevel	压缩率
checksum	是否有校验, 当 gp_appendonly_verify_block_checksums 参数被设置成 true 的时候, 则在创建一个块的时候会计算校验码, 在读取的时候也会计算校验码, 判断数据是否正确, 默认是关闭的
compressstype	数据压缩的类型
columnstore	是否是列存储
segrelid	pg_aoseg 表的 oid
segidxit	pg_aoseg 表索引的 oid



Greenplum 4.3 之后对 Appendonly 表进行了优化, 以前的 Appendonly 表只能插入数据, 不能够更新和删除数据。Greenplum4.3 之后, 将 Appendonly 表改成了 Append-Optimized 表, 用法跟原来的是一样的, 建表的时候还是制定 appendonly=true。Append-Optimized 表可以更新和删除数据。与 Heap 表一样, Greenplum 在 Update 数据的时候, 其实都是将原有的数据标志位删除, 然后新增了一条数据, 所以当 Append-Optimized 经过一定时间的更新之后, 需要使用 vacuum 命令对其空间进行回收, 用法与 Heap 表的一样。

## 6.2 列存储

Appendonly 表还有一种特殊的形式, 那就是列存储。关于列存储的相关介绍, 读者可以参考网络上的一些资料。列存储主要是针对数据库优化的一种数据存储类型。

### 6.2.1 应用场景

列存储一般适用于宽表 (即字段非常多的表)。在使用列存储时, 同一个字段的数据都连

续保存在一个物理文件中，所以列存储的压缩率比普通压缩表的压缩率要高很多，另外在多字段中筛选其中几个字段时，需要扫描的数据量很小，扫描速度比较快。因此，列存储尤其适合在宽表中对部分字段进行筛选的场景。

### 6.2.2 数据文件存储特性

列存储在物理上存储，一个列对应一个数据文件，文件名是原文件名加上.n（n是一个数字）的形式来表示的，第一个字段，n=1，第二个字段n=129，以后每增加一个字段，n都会增加128。

为什么每增加一个字段，n不是递增的而是中间要间隔128个数字呢？这大概是由于，在Greenplum中，每一个数据文件最大是1GB，如果一个字段在一个Segment中的数据量超过了1GB，就要重新生成一个新的文件，每个字段中间剩余的这127个数字，就是预留给字段内容很大，单个文件1GB放不下，拆分成多个文件时，可以使用预留的127个数字为数据文件编号，如xxxx.1是第一个字段的数据文件，如果这个文件达到了1GB，则产生xxxx.2数据文件，继续保存第一个字段的数据。

从这个物理的存储特性就可以很明显地看到列存储的缺点，那就是会产生大量的小文件，假设一个集群中有30个节点，我们建了一张有100个字段宽表的，假设这张表是按照每天分区的一张分布表，保留了一年的数据，那么将会产生的文件数是：

$$30 \times 100 \times 365 = 1095000$$

即产生将近110万的文件，这样我们对一个表进行DDL操作就会比较慢，所以，列存储应该在合适的场景下才能使用，而不能滥用列存储，否则会得不偿失。

### 6.2.3 如何使用列存储

列存储的表必须是appendonly表，创建一个列存储的表只需要在创建表的时候，在with子句中加入appendonly=true,ORIENTATION=column即可。一般appendonly表会结合压缩一起使用，在with子句中增加compresslevel=5启用压缩，如下：

```
testDB=# create table test_column_ao (
testDB(#     id bigint
testDB(#     ,name varchar(128)
testDB(#     ,value varchar(128))
testDB-# with (appendonly=true,ORIENTATION=column,compresslevel=5)
testDB-# distributed by (id);
CREATE TABLE
```

### 6.2.4 性能比较

这里我们将简单比较一下列存储和普通压缩表在性能上的区别，让大家对列存储有一个

直观的认识，希望通过这个比较，使读者对是否需要使用列存储有一个大概的印象，但是由于硬件环境和测试的业务场景对测试的结果有很大的影响，故读者应该根据自己具体的业务场景和集群环境进行测试，以获取比较真实的结果数据。

由于列存储主要应用在宽表上，故这次测试都是针对字段比较多的表。下面对两个不同的数据进行比较，对于每个数据，除了表的存储类型，其他包括字段数、字段类型和数据全部一模一样。通过 count 具体字段（数据库需要检查字段是否非空），可以使测试的字段的数据文件都被扫描，测试的查询 SQL 如下：

```
select count(col1),count(col2),... from tablename;
```

## 测试数据 1

纯随机数据，总共有 121 个字段，1 个 int 字段作为 ID，其他字段都是字符串类型，字符串字段的内容都是随机的，值是由 md5(random()) 生成的，数据非常均匀且随机。数据量为 500 000，表的压缩属性都是 compresslevel=5。比较结果如表 6-6 所示。

表 6-6 普通压缩表与列存储测试结果 1

	普通压缩表	列存储
字段数	121	与普通表相同
压缩率	实际压缩率 =1.74	实际压缩率 =1.7
压缩后大小	1152 MB	1116 MB
查询 10 个字段消耗时间	4278.839 ms	423.139 ms
查询 20 个字段消耗时间	4072.795 ms	811.920 ms
查询 50 个字段消耗时间	4923.737 ms	2101.356 ms
查询全部字段消耗时间	4999.031 ms	5247.297 ms
count(*)	3909.924 ms	174.418 ms

通过表 6-6 可以看出：

- ①完全随机的数据压缩率比较低，使用两种方式的压缩结果差不多。
- ②当只查询表的数据量的时候，普通压缩表需要全表扫描，但是列存储只需要查询其中一个字段，速度非常快。
- ③随着查询字段的增加，普通压缩表查询的速度基本上变动不大，因为扫描的数据量没有变化。而列存储则随着查询的字段增加，消耗时间增长明显。
- ④在测试的时候瓶颈在于磁盘的 IO，这是因为测试集群磁盘性能较差。在 CPU 消耗上，列存储的消耗略大于普通表的消耗。

## 测试数据 2

这是一个实际业务场景的表，主要是保存商品描述信息的表，字段数为 98 个，数

据类型有 date、text、varchar、int、numeric，数据量为 3100 万，表的压缩属性都是 compresslevel=5。

表 6-7 普通压缩表与列存储测试结果 2

	普通压缩表	列存储
压缩率	实际压缩率 = 3.84	实际压缩率 = 5.98
压缩后大小	17 GB	10 GB
查询 10 个字段消耗时间	172707.491 ms	10817.341 ms
查询 20 个字段消耗时间	179532.098 ms	19811.137 ms
查询 50 个字段消耗时间	217994.797 ms	49127.229 ms
查询全部字段消耗时间	251559.494 ms	125667.730 ms
count(*)	161226.188 ms	2931.648 ms

采用这两种数据方式加载数据时都使用外部表，加载时的瓶颈在于文件服务器的网卡，所以使用这两种方式进行数据加载的速度都差不多。

通过表 6-7 的结果可以看出，测试的结果与测试数据 1 的结果大致相同，但是还有一些区别，主要是：

①测试的数据是真实的数据，每个字段的内容都会有些相似，当每个字段的数据比较相似时，使用列存储，这样在压缩文件时，可以获得更高的压缩率，节省磁盘的空间占用。

②瓶颈还是磁盘 IO，由于列存储有更好的压缩率，所以在查询全部字段的情况下，列存储消耗的时间还是比普通压缩表小很多的，不像测试数据 1，由于压缩率差不多，查询所有字段的性能也差不多。

## 6.3 外部表高级应用

对于数据库应用，数据导入导出是必不可少的，在 Greenplum 中，提供了外部表方便用户对数据进行导入导出。本节将介绍外部表的原理及外部表的一些高级特性。

### 6.3.1 外部表实现原理

本地模式数据是在 Master 所在的主机上的，与使用 copy 命令导入数据一样，都要通过 Master 传到每个 Segment 上。两者在性能上差别不大，原理及使用方法也都比较简单，这里就不详细描述了。

在第 2 章中，我们已经介绍 gpfldist 的架构，其实 gpfldist 也可以看做一个 http 服务，当我们启动 gpfldist 的时候，可以用 wget 命令去下载 gpfldist 的文件，将创建外部表命令时使用的 url 地址中的 gpfldist 换成 http 即可，如：

```
wget http://10.20.151.59:8081/inc/1.dat -O 2.dat
```

当查询外部表时，所有的 Segment 都会连上 gpfdist，然后 gpfdist 将数据随机分发给每个节点。

gpfdist 的架构如图 6-2 所示。

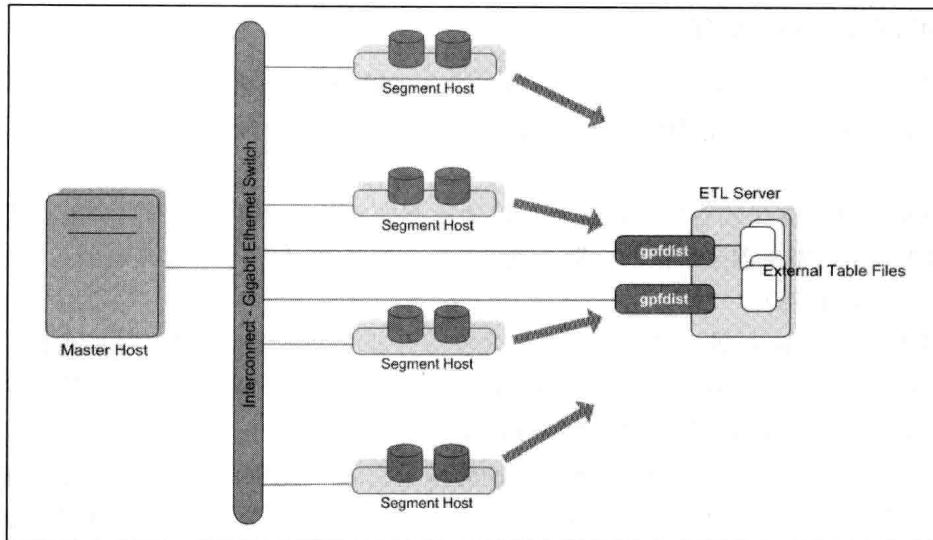


图 6-2 Greenplum 外部表架构

## 1. gpfdist 的工作流程

- 1) 启动 gpfdist，并在 Master 上建表。表建好之后并没有任何的数据流动，只是定义好了外部表的原数据信息。
- 2) 将外部表插入到一张 Greenplum 的物理表中，开始导入数据。
- 3) Segment 根据建表时定义的 gpfdist url 个数，启动相同的并发到 gpfdist 获取数据，其中每个 Segment 节点都会连接到 gpfdist 上获取数据。
- 4) gpfdist 收到 Segment 的连接并要接收数据时开始读取文件，顺序读取文件，然后将文件拆分成多个块，随机抛给 Segment。
- 5) 由于 gpfdist 并不知道数据库中有多少个 Segment，数据是按照哪个分布键拆分的，因此数据是随机发送到每个 Segment 上的，数据到达 Segment 的时间基本上是随机的，所以外部表可以看成是一张随机分布的表，将数据插入到物理表的时候，需要进行一次重分布。
- 6) 为了提高性能，数据读取与重分布是同时进行的，当数据重分布完成后，整个数据导入流程结束。

## 2. gpfdist 最主要的功能

- ①负载均衡：每个 Segment 分配到的数据都是随机的，所以每个节点的负载都非常均衡。

②并发读取，性能高：每台 Segment 都同时通过网卡到文件服务器获取数据，并发读取，从而获取了比较高的性能。相对于 copy 命令，数据要通过 Master 流入，使用外部表就消除了 Master 这个单点问题。

### 3. 如何提高 gpfldist 性能？

想提高一个工具的性能，首先要了解这个工具使用的瓶颈在哪里。对于数据导入，衡量性能最重要的就是数据分发时的吞吐，其中有两个地方容易成为瓶颈。

#### (1) 文件服务器

一般文件服务器比较容易出现瓶颈，因为所有的 Segment 都连接到这个节点上获取数据，所以如果文件服务器是单机的，那么就很容易在磁盘 IO 和网卡上出现瓶颈。

#### (2) Segment 节点

相对来说，Segment 的机器数一般会比文件服务器多很多，而且网卡性能也比较好，因此 Segment 一般不容易出现瓶颈。当 Segment 出现瓶颈的时候，可能不只是数据导入出现瓶颈，应该是整个数据库的性能都已经出现了瓶颈。

对于文件服务器，当磁盘 IO 出现瓶颈的时候，我们可以使用磁盘阵列来提高磁盘的吞吐能力，如果条件允许，还可以采用分布式文件系统来提高整体的性能，例如 MooseFS 等。对于网卡出现瓶颈，第一种方法是更换网卡为万兆网卡，但是这种方法还需要各环节的网络环境满足条件，如交换机支持等，成本较高，第二种方法就是通过多网卡机制来保证。

如图 6-3 所示，就是多网卡的一个例子，我们将文件拆分成多个文件，并启动多个 gpfldist 分别读取不同的文件，然后将 gpfldist 绑定到不同的网卡上以提高性能。

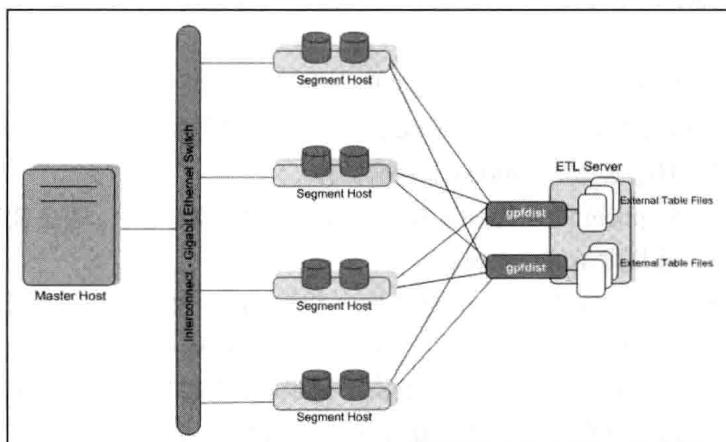


图 6-3 外部表多网卡并行加载架构图

在创建表的时候指定多个 gpfldist 的地址，例如：

```

CREATE EXTERNAL TABLE table_name
( column_name data_type [, ...] | LIKE other_table )
LOCATION ('gpfdist://filehostip1[:port1]/file_pattern1',
'gpfdist://filehostip2[:port2]/file_pattern2',
'gpfdist://filehostip3[:port3]/file_pattern3',
'gpfdist://filehostip4[:port4]/file_pattern4' )
.....

```

当然，我们也可以只启动一个 gpfdist，但是通过不同的 ip 连接到 gpfdist 上，这样读取文件的 gpfdist 只有一个，不能实现 IO 的并发，但是网卡却可以使用多张，从而来消除单网卡的瓶颈。



### 如何控制并发度？

可以在 postgresql.conf 中修改参数 `gp_external_max_segs`，默认是 64，这个参数用于控制同时有多少个 Segment 连接到 gpfdist 上。

## 6.3.2 可写外部表

前面介绍的是只读外部表，在 Greenplum 中，还有一种可写外部表。

语法如下：

```

CREATE WRITABLE EXTERNAL TABLE table_name
( column_name data_type [, ...] | LIKE other_table )
LOCATION ('gpfdist://outputhost[:port]/filename' [, ...])
| ('gphdfs://hdfs_host[:port]/path')
FORMAT 'TEXT'
[ ([ [DELIMITER [AS] 'delimiter']
[NULL [AS] 'null string']
[ESCAPE [AS] 'escape' | 'OFF']] )
| 'CSV'
[ ([QUOTE [AS] 'quote']
[DELIMITER [AS] 'delimiter']
[NULL [AS] 'null string']
[FORCE QUOTE column [, ...]] ]
[ESCAPE [AS] 'escape']) ]
[ ENCODING 'write_encoding' ]
[ DISTRIBUTED BY (column, [ ... ]) | DISTRIBUTED RANDOMLY ]

```

在语法上可写外部表与普通外部表主要有两个地方不一样：

①可写外部表没有错误表，不能指定允许有多少行数据错误。因为外部表的数据一般是从数据库的一张表中导出的，格式肯定是正确的，一般不会有异常数据。

②可写外部表在建表时可以指定分布键，如果不指定分布键，默认为随机分布（distributed randomly）。

下面将通过一个例子（4张表，如表 6-8 所示）介绍下分布键是怎么用的。

表 6-8 分布键不同的外部表

表 名	分布键
test001	id
test001_wext1	randomly
test001_wext2	id
test001_wext3	name

查看执行计划：

```
testDB=# create writable external table test001_wext1(like test001) location('gpfdist://10.20.151.7:8888/test001_wext1.dat') format 'text' distributed randomly ;
CREATE EXTERNAL TABLE
testDB=# explain insert into test001_wext1 select * from test001;
Insert (slice0; segments: 6) (rows=16667 width=9)
    -> Redistribute Motion 6:6 (slice1; segments: 6) (cost=0.00..1138.00
rows=16667 width=9)
        -> Seq Scan on test001 (cost=0.00..1138.00 rows=16667 width=9)
(3 rows)

testDB=# explain insert into test001_wext2 select * from test001;
Insert (slice0; segments: 6) (rows=16667 width=9)
    -> Seq Scan on test001 (cost=0.00..1138.00 rows=16667 width=9)
(2 rows)

testDB=# explain insert into test001_wext3 select * from test001;
Insert (slice0; segments: 6) (rows=16667 width=9)
    -> Redistribute Motion 6:6 (slice1; segments: 6) (cost=0.00..1138.00
rows=16667 width=9)
        Hash Key: test001.name
        -> Seq Scan on test001 (cost=0.00..1138.00 rows=16667 width=9)
(4 rows)
```

可以看到，随机分布及分布键与原表不一致都会导致数据重分布：数据先重新按照规则打散到每一个 Segment 上，然后再并发写入 gpfdist 中。因此，在使用外部表的时候，尽量采用与原表一致的分布键。

使用可写外部表的注意事项：

- ① 可写外部表不能中断 (truncate)。当我们把数据写入可写外部表时，如果可写外部表中途断开了，要想重新运行必须手动将原有的文件删除，否则那一部分数据会重复。
- ② 可写外部表指定的分布键应该与原表的分布键一致，避免多余的数据重分布。
- ③ gpfdist 必须使用 Greenplum 4.x 之后的版本。可写外部表是 Greenplum 4.x 之后加入的新功能，在其他机器上启动 gpfdist，只需要将 \$GPHOME/bin/gpfdist 复制过去即可。

### 6.3.3 HDFS 外部表

Greenplum 可以直接读取 HDFS 文件，这样可以将 Greenplum 和 HDFS 整合在一起，将 Greenplum 作为一个计算节点，计算后的数据可以直接写入到 HDFS 中，提供对外的服务。HDFS 外部表的架构图如图 6-4 所示。

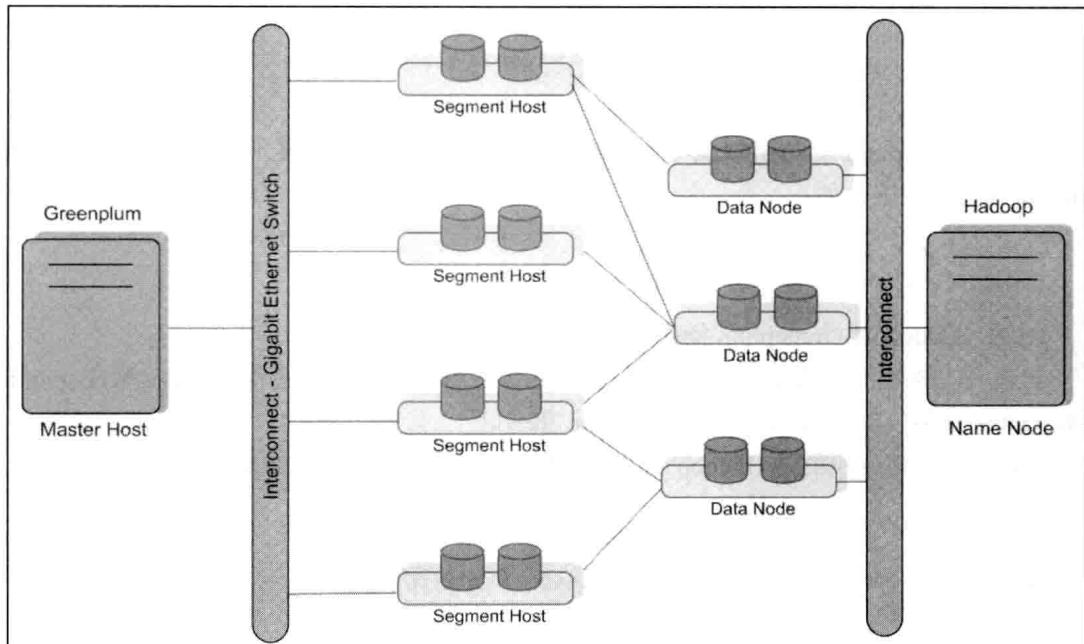


图 6-4 Greenplum 外部表读写 HDFS 架构



HDFS 外部表是 Greenplum 4.x 版本才加入的新功能，在第 3 章中也介绍到了这方面的内容。

本节将介绍如何使用 HDFS 外部表，但由于笔者很少用到 HDFS 外部表，因此这里只是简单介绍一下如何使用，具体的一些细节以及性能，需要读者自行测试。Greenplum 4.1 和 Greenplum 4.2 以上版本，在使用 HDFS 外部表的时候用法变化了，下面将对这两个版本的 HDFS 外部表分开介绍。

#### 1. Greenplum 4.1

在 Greenplum 4.1 上安装、配置 HDFS 外部表的步骤如下：

- 1) 在所有 Segment 上安装 Java1.6 或以上版本。
- 2) 在所有 Segment 上安装 Hadoop 0.21.0 或者 0.20.1。

如果使用的是 Hadoop 0.21.0，那么直接跳到步骤 3)，如果使用的是 Hadoop 0.20.1，那么还需要以下额外的步骤：

□ 将 \$GPHOME/bin 目录下的 HDFSReader.class 和 HDFSWriter.class 重命名成 \*.class.0.21.0。

□ 将 bin 目录下的 \*.class.0.20.1 的后缀 .0.20.1 去掉。

3) 在所有的 Segment 节点上在 ~/.bashrc 中设置以下的环境变量。

JAVA\_HOME: Java 的安装目录

HADOOP\_HOME: Hadoop 的安装目录

HADOOP\_VERSION: Hadoop 的版本

4) 在 Greenplum 中向用户赋予可以创建 gphdfs 外部表的权限，readable 为可读权限，writable 为可写权限，如：

```
ALTER ROLE hdfsadmin CREATEEXTTABLE
(type= 'writable', protocol='gphdfs');
```

5) 在 Hadoop 对 Greenplum 操作系统用户配置响应的权限。

创建 HDFS 外部表的语法与普通外部表的语法基本一致，就是将 url 头改写成 gphdfs，将 hostname 和端口改成 hdfs 对应的 hostname 和端口。

下面这个示例使用的是 Hadoop 0.20.2 版本，~/.bashrc 的参数配置如下：

```
export JAVA_HOME=/hadoop/jdk1.6.0_20
export HADOOP_HOME=/home/gpadmin/hadoop-0.20.2
export PATH=$PATH:$JAVA_HOME/bin/:$HADOOP_HOME/bin
export HADOOP_VERSION=0.20.1
```

每一台机器上的 \$GPHOME/bin 目录下的所有 .class 都更换成 0.20.1 版本的。

现在，将 HDFS 中的一个文件导入到 Greenplum 中，文件的内容如下：

```
$ hadoop fs -cat /data/gpext/1.dat
101,tom
102,cat
103,marry
104,john
```

首先，创建外部表：

```
testDB=# CREATE EXTERNAL TABLE hdfs_test ( id int, name varchar(128) )
testDB-# LOCATION ('gphdfs://10.20.151.7:9000/data/gpext/1.dat')
testDB-# FORMAT 'TEXT' (DELIMITER ',');
```

第一次执行时，报如下错误：

```
testDB=# select * from hdfs_test;
ERROR: external table hdfs_test command ended with error. sh: java: command
not found (seg4 slice1 sdw3:33000 pid=14991)
DETAIL: Command: gphdfs://10.20.151.7:9000/data/gpext/1.dat
```

发现系统没有找到 java 的命令，这是由于刚刚配置的 JAVA\_HOME 的环境变量还没有生效，需要重启数据库，让数据库重新获取环境变量。重启数据库使用命令 gpstop -afr。

重启数据库后，操作正常：

```
testDB=# select * from hdfs_test;
 id | name
-----+
 104 | john
 101 | tom
 103 | marry
 102 | cat
(4 rows)
```

创建可写外部表：

```
testDB=# CREATE writable EXTERNAL TABLE hdfs_test_write ( id int, name
varchar(128) )
testDB=# LOCATION ('gphdfs://10.20.151.7:9000/data/gpext/hdfs_test_write')
testDB=# FORMAT 'TEXT' (DELIMITER ',');
CREATE EXTERNAL TABLE
testDB=# insert into hdfs_test_write select * from hdfs_test;
INSERT 0 4
```

在写入外部表的时候，Greenplum 在 Hadoop 上创建了一个文件夹保存一个表的数据，下面验证数据是否已经正确写入：

```
$ hadoop fs -ls /data/gpext/hdfs_test_write
Found 4 items
-rw-r--r-- 3 gpadmin supergroup          8 2012-02-27 10:40 /data/gpext/hdfs_
test_write/0_1330309393-0000000013
-rw-r--r-- 3 gpadmin supergroup          8 2012-02-27 10:40 /data/gpext/hdfs_
test_write/1_1330309393-0000000013
-rw-r--r-- 3 gpadmin supergroup          9 2012-02-27 10:40 /data/gpext/hdfs_
test_write/4_1330309393-0000000013
-rw-r--r-- 3 gpadmin supergroup         10 2012-02-27 10:40 /data/gpext/hdfs_
test_write/5_1330309393-0000000013

$ hadoop fs -cat /data/gpext/hdfs_test_write/*
102,cat
101,tom
104,john
103,marry
```

当这个外部表的数据不断追加的时候，Greenplum 会在这个文件夹下创建新的文件来存放这些新的数据。将外部表删除的时候，并不会删除对应的 HDFS 文件，在使用过程中一定要留意这一点，需要定期删除这些已经没用的文件。

## 2. Greenplum 4.2 及以上版本

在 Greenplum 4.2 及以上版本安装、配置 HDFS 外部表的步骤如下：

- 1) 在所有 Segment 上安装 Java1.6 或以上版本。
- 2) Greenplum 数据库包括以下与 Hadoop 相关版本：
  - Prvotal HD1.0 (gphd-2.0): Hadoop 2.0 版本。
  - Greenplum HD (gphd-1.0,gphd-1.1 和 gphd-1.2): 默认版本，使用其他组件可以通过设置 gp.hadoop\_target\_version 这个参数进行切换。
  - Greenplum MR 组件 (gpmr-1.0 和 gpmr-1.2): Greenplum 版本的 Mapreduce。
  - Cloudera Hadoop 版本连接 (cdh3u2 和 cdh4.1): Cloudera 版本的 Hadoop。
- 3) 需要读者自己安装 Hadoop (Greenplum 只支持以上所列的版本)，安装之后，需确保 Greenplum 的系统用户有读和执行 Hadoop 相关 lib 库的权限。
- 4) 设置环境变量 JAVA\_HOME、HADOOP\_HOME。
- 5) 设置 gp.hadoop\_target\_version 和 gp.hadoop\_home 参数如表 6-9 所示。

表 6-9 gp.hadoop\_target\_version 和 gp.hadoop\_home 参数设置

字段名	描述	默认值
gp.hadoop_target_version	选择其中一个 hadoop 版本： gphd-1.0 gphd-1.1 gphd-1.2 gphd-2.0 gpmr-1.0 gpmr-1.2 cdh3u2 cdh4.1	gphd-1.1
gp.hadoop_home	跟 HADOOP_HOME 配置一样	NULL

笔者采用 Hadoop 2.0 版本进行测试，所以参数设置如下：

```
gp.hadoop_target_version='gphd-2.0'
gp.hadoop_home='/home/hadoop/soft/hadoop-2.0.0'
```

6) 重启数据库。

在使用 HDFS 外部表的时候，有两层权限关系：

对 Greenplum 的 HDFS 的使用用户赋予 SELECT 和 INSERT 权限：

```
testDB=# GRANT INSERT ON PROTOCOL gphdfs TO gpadmin;
GRANT
testDB=# GRANT SELECT ON PROTOCOL gphdfs TO gpadmin;
GRANT
```

确保 Greenplum 对应的 OS 用户有 Hadoop 上 HDFS 文件的读写权限。

Greenplum 在 HDFS 上支持三种文件格式：

- Text，文本格式，同时支持读写，跟介绍 Greenplum 4.1 版本的 HDFS 外部表一样。
- gphdfs\_import，只支持只读外部表。
- gphdfs\_export，只支持可写外部表。

创建外部表进行读取：

```
testDB=# CREATE readable EXTERNAL TABLE hdfs_test_read ( id int, name
varchar(128) )
testDB=# LOCATION ('gphdfs://10.20.151.7:9000/data/gpext/hdfs_test.dat')
testDB=# FORMAT 'TEXT' (DELIMITER ',');
CREATE EXTERNAL TABLE
testDB=# select * from hdfs_test_read;
 id | name
----+-----
 104 | john
 101 | tom
 103 | marry
 102 | cat
(4 rows)
```

在 \$GPHOME/lib/hadoop 目录下，Greenplum 自带了各种 Hadoop 版本的 jar 包，这些包中定义了 gphdfs\_import 跟 gphdfs\_export 的数据格式。

```
$ ls $GPHOME/lib/hadoop/
cdh3u2-gnet-1.1.0.0.jar  gphd-1.0-gnet-1.0.0.1.jar  gphd-1.2-gnet-1.1.0.0.jar
gpmr-1.0-gnet-1.0.0.1.jar  hadoop_env.sh
cdh4.1-gnet-1.2.0.0.jar  gphd-1.1-gnet-1.1.0.0.jar  gphd-2.0.2-gnet-1.2.0.0.jar
gpmr-1.2-gnet-1.0.0.1.jar
```

Greenplum 在读写这种格式的时候，需要自己编写 MapReduce 任务。Greenplum 4.3 的管理员手册中有对这些格式的解析，有兴趣的读者可以自己看下，这里就不讲解了。

### 6.3.4 可执行外部表

普通外部表其实可以理解成可执行外部表的一种特例，利用可执行外部表的功能也可以实现一个类似 gpfdist 并发加载数据的工具。

可执行外部表的语法如下：

```
CREATE [READABLE] EXTERNAL WEB TABLE table_name
( column_name data_type [, ...] | LIKE other_table )
  LOCATION ('http://webhost[:port]/path/file' [, ...])
  | EXECUTE 'command' [ON ALL
    | MASTER
    | number_of_segments
    | HOST ['segment_hostname']
    | SEGMENT segment_id ]
FORMAT 'TEXT'
```

```

[ ( [HEADER]
    [DELIMITER [AS] 'delimiter' | 'OFF']
    [NULL [AS] 'null string']
    [ESCAPE [AS] 'escape' | 'OFF']
    [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
    [FILL MISSING FIELDS] )]
  | 'CSV'
  [ ( [HEADER]
      [QUOTE [AS] 'quote']
      [DELIMITER [AS] 'delimiter']
      [NULL [AS] 'null string']
      [FORCE NOT NULL column [, ...]]
      [ESCAPE [AS] 'escape']
      [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
      [FILL MISSING FIELDS] )]
  [ ENCODING 'encoding' ]
  [ [LOG ERRORS INTO error_table] SEGMENT REJECT LIMIT count
    [ROWS | PERCENT] ]

```

在 gp\_toolkit (Greenplum 4 中自带的一个工具集) 中就有可执行外部表的例子，可以查询所有 Segment 的日志信息。

该外部表的定义如下，读者可以参考着写。

```

CREATE EXTERNAL WEB TABLE gp_toolkit.__gp_log_segment_ext
(
    logtime timestamp with time zone,
    loguser text,
    logdatabase text,
    .....
)
EXECUTE E'cat $GP_SEG_DATADIR/pg_log/*.csv'
FORMAT 'CSV' (DELIMITER AS ',' NULL AS '' QUOTE AS '');

```

在使用外部表的时候，每个 Segment 的参数可能会略有不同，或者是在脚本中需要获取一些系统的参数。表 6-10 是 Greenplum 自带的一些常见参数，可以供脚本使用。

表 6-10 可执行外部表常用参数

参数变量	描述
\$GP_DATABASE	数据库名
\$GP_DATE	外部表执行的日期
\$GP_MASTER_HOST	Master 的 hostname
\$GP_MASTER_PORT	Master 的端口
\$GP_SEG_DATADIR	Segment 的数据目录
\$GP_SEG_PG_CONF	Segment 的 postgresql.conf 这个文件位置
\$GP_SEG_PORT	Segment 的端口
\$GP_SEGMENT_COUNT	primary segment 的个数

(续)

参数变量	描述
\$GP_SEGMENT_ID	Segment 的 dbid 编号
\$GP_SESSION_ID	外部表 SQL 的 sess_id
\$GP_SN	一个序列名
\$GP_TIME	外部表执行的时间
\$GP_USER	数据库用户名
\$GP_XID	事务 ID

如果是自定义的一些脚本，那么这些脚本在每一个 Segment 上都必须部署好。同样的，外部表也支持可写可执行外部表，脚本通过标准输入流来获取数据。下面举一个最简单的例子，将数据直接输出到文本中。

```
testDB=# CREATE WRITABLE EXTERNAL WEB TABLE wext_web (output text)
testDB=# EXECUTE 'cat >> /home/gpadmin/data/wext_web_$GP_SEGMENT_ID'
testDB-# FORMAT 'TEXT'
testDB-# DISTRIBUTED RANDOMLY;
CREATE EXTERNAL TABLE
testDB=# insert into wext_web select sname from student;
INSERT 0 9
```

可以查询每个节点生成的文件，不同 Segment 的数据在该 Segment 的 dbid 对应的文件中。

```
=> 11
[sdw3] total 8
[sdw3] -rw----- 1 gpadmin gpadmin 10 Feb 27 13:57 wext_web_4
[sdw3] -rw----- 1 gpadmin gpadmin  6 Feb 27 13:57 wext_web_5
[sdw1] total 8
[sdw1] -rw----- 1 gpadmin gpadmin  9 Feb 27 13:57 wext_web_0
[sdw1] -rw----- 1 gpadmin gpadmin  6 Feb 27 13:57 wext_web_1
[sdw2] total 8
[sdw2] -rw----- 1 gpadmin gpadmin  5 Feb 27 13:57 wext_web_2
[sdw2] -rw----- 1 gpadmin gpadmin 12 Feb 27 13:57 wext_web_3
```

## 6.4 自定义函数——各个编程接口

本节将介绍 Greenplum 中的自定义函数，主要涉及 pgsql、plpython 和 C 语言，具体将介绍普通 UDF（用户自定义函数）、聚合函数、一变多函数。由于这些内容在 PostgreSQL 的文档中都有很详细的描述，故这里不会对具体的语法进行介绍，而着重介绍每个语言的特性，更加适合在什么场景下使用。

### 6.4.1 pl/pgsql

对于熟悉 Oracle 的读者来说，pl/pgsql 的语法应该不陌生，它的语法跟 Oracle 的 pl/sql 语法相当类似，只不过功能没有那么强。

相比起 Oracle 的 pl/sql，pl/pgsql 主要有如下限制：

- 1) 最大的子事务个数只能是 100，每一个异常捕获，都会造成一个子事务。
- 2) 在函数执行的过程中不能执行 commit。
- 3) 由于不能执行 commit，因此一个函数中不要处理太多的逻辑，否则容易导致程序出错，这一点与 Oracle 不一样。如果有比较复杂的逻辑，要以外部的 shell 调用来实现，不要在函数中实现。
- 4) 所有的 pl/pgsql 都必须以函数的形式存在，没有 Oracle 中存储过程的概念，所以每次使用都必须建立函数，然后再执行这个函数，使用起来不方便。

由于这些限制，很多 Oracle 的存储过程不能直接迁移过来，因此要用其他外部脚本来替代，以实现响应的功能。

下面介绍一个 pl/pgsql 的例子。

在日常数据库使用中，经常需要重建一些表，为了能够回滚，一般会将表重命名，保留一段时间后再删除。Greenplum 重命名表会有一个问题，就是将表重命名后，表上的视图还是指向原来的表，因此表重建后也需要将表上的视图重建，但是在重命名过程中，Greenplum 并没有这个提示。为了避免这种情况，我们封装一个函数，实现这个功能，函数的描述如下：

- 检查表是否存在，不存在则报错。
- 检查表上有没有视图，有视图则将这个表上的视图显示出来，然后报错。
- 表上如果没有视图，则将表重命名，并向 rename\_table\_log 表中插入一条记录，方便以后将这个表删除。
- 为了避免重命名后的表有重名，根据时间自动生成一个随机数据作为表名后缀。

函数代码如下：

```
create table PUBLIC.rename_table_log(
    schemaname      character varying(128)          not null
    ,ori_name       character varying(256)          not null
    ,tgt_name       character varying(256)          not null
    ,dw_ins_date   timestamp(0) without time zone default now()
    ,isdel          boolean                         default false
)Distributed by (ori_name);
CREATE INDEX idx_rename_table_log ON PUBLIC.rename_table_log USING btree
(schemaname, ori_name, tgt_name);

create or replace function public.prc_rename_tab(schema_name character
varying,table_name character varying)
RETURNS void
AS
```

```

$BODY$
declare
    l_schema_name  VARCHAR(80) := schema_name;
    l_table_name   VARCHAR(80) := table_name ;
    l_cnt          INTEGER;
    l_sql          text;
    l_viewname     varchar(256);
    l_table_rename VARCHAR(256);
    l_views_on_tables boolean;
BEGIN
    select count(*) into l_cnt
        from pg_catalog.pg_tables
        where schemaname=lower(l_schema_name) and tablename=lower(l_table_name);
    l_views_on_tables:='f';
    l_sql:='select ev_class::regclass from pg_rewrite where oid in (
                select b.objid
                    from pg_depend a,pg_depend b
                    where a.refclassid=1259
                        and b.deptype=''i''
                        and a.classid=2618
                        and a.objid=b.objid
                        and a.classid=b.classid
                        and a.refclassid=b.refclassid
                        and a.refobjid<>b.refobjid
                        and a.refobjid='''||l_schema_name||'.'||l_table_
name||'''::regclass)';
    if(l_cnt>0) then
        select substr(cast(to_char(current_timestamp(0), 'ddhhmiss') as
integer)/random(),1,6)||'_rename' into l_table_rename;

        for l_viewname IN execute l_sql LOOP
            RAISE NOTICE 'view (%) depend on table (%.%)', l_
viewname,l_schema_name, l_table_name;
            l_views_on_tables:='t';
        END LOOP;
        if(l_views_on_tables='t') then
            RAISE EXCEPTION 'Please drop the views on tables';
        end if;
        insert into PUBLIC.rename_table_log(schemaname,ori_name,tgt_
name,dw_ins_date,isdel)
            values (l_schema_name,l_table_name,l_table_name||l_table_
rename,current_timestamp,false);
        l_sql:=$$alter table $$ || l_schema_name || $$.$$ || l_table_
name || $$ rename to $$||l_table_name||l_table_rename;
        EXECUTE l_sql;
    ELSE
        RAISE EXCEPTION 'table does not exists';
    end if;
    RETURN;
END;

```

```
$BODY$  
LANGUAGE 'plpgsql' VOLATILE;
```

使用上面的函数会有以下几种效果。

1) 表不存在, 报错:

```
testDB=# select prc_rename_tab('public','dsfdsfsf');  
ERROR: table does not exist
```

2) 表上有视图, 报错:

```
testDB=# select prc_rename_tab('public','cxf');  
NOTICE: view (v_cxf) depend on table (public.cxf)  
ERROR: Please drop the views on tables
```

3) 重命名成功:

```
testDB=# select prc_rename_tab('public','cxf2');  
prc_rename_tab  
-----  
(1 row)
```

4) 在 rename\_table\_log 表中的记录:

```
testDB=# select * from rename_table_log;  
schemaname | ori_name | tgt_name | dw_ins_date | isdel  
-----+-----+-----+-----+-----  
public | cxf2 | cxf2139926_rename | 2012-03-03 23:10:48 | f  
(1 row)
```

## 6.4.2 C 语言接口

由于 Greenplum 是使用 C 语言写的, 因此使用 C 语言来编写自定义函数的效率会很高, 而且结合 PostgreSQL 开源的特性可以在 PostgreSQL 中通过一些内部函数来简单实现我们需要开发的接口。这一小节, 我们将介绍如何使用 C 语言在 Greenplum 中开发函数。

关于语法方面的内容, PostgreSQL 的文档已经解释得很清楚了, 这里就不再介绍了, 大家在阅读这个章节之前可以先去看一下 PostgreSQL 的官方文档关于这部分内容的解释。

下面将通过一个例子来介绍如何使用 C 语言来开发函数。

在 Greenplum/PostgreSQL 中, 将字符串转换成时间是很方便的, 很多种格式的时间字符串都可以自动转换成 date 或 timestamp 类型。如果是时间格式不正确, SQL 会报错, 例如:

```
testDB=# select '2011-13-10 10:10:10'::date;  
ERROR: date/time field value out of range: "2011-13-10 10:10:10"  
LINE 1: select '2011-13-10 10:10:10'::date;  
^  
HINT: Perhaps you need a different "datestyle" setting.
```

```
testDB=# select 'df'::date;
ERROR: invalid input syntax for type date: "df"
LINE 1: select 'df'::date;
```

现在有一个需求，就是验证一个字符串是否是有效的时间。查询了很多 PostgreSQL 的文档，都没有发现这样一个函数。但是，在进行类型转换的时候，数据库确实是做了这个验证的，只不过没有实现这个函数调用而已。所以看一下 PostgreSQL 的源码，把这个函数抽取出来，新建一个函数来实现这个功能。

在 src/backend/utils/adt/date.c 这个代码中，有这个函数：

```
/* date_in()
 * Given date text string, convert to internal date format.
 */
Datum
date_in(PG_FUNCTION_ARGS)
{
...
    dterr = ParseDateTime(str, workbuf, sizeof(workbuf), field, ftype,
MAXDATEFIELDS, &nf);
    if (dterr == 0)
        dterr = DecodeDateTime(field, ftype, nf, &dtype, tm, &fsec, &tzp);
    if (dterr != 0)
        DateTimeParseError(dterr, str, "date");
...
}
```

这个函数调用了 ParseDateTime 和 DecodeDateTime 这两个函数，就可以识别出这个字符串是否是时间字符。我们可以通过引用头文件 utils/datetime.h，直接使用这两个函数，就可以简单地实现这一个功能，避免重复判断字符串。这个函数的 C 语言代码如下：

```
//is_date.c
#include "postgres.h"
#include "funcapi.h"
#include "utils/datetime.h"
#include "fmgr.h"
#ifndef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
Datum is_date(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(is_date);
Datum
is_date(PG_FUNCTION_ARGS)
{
    VarChar *arg1 = PG_GETARG_VARCHAR_P(0);
    struct pg_tm tt,
                    *tm = &tt;
    fsec_t      fsec;
    char        *str = (char *)palloc(VARSIZE(arg1) - VARHDRSZ +1);
```

```

memcpy(str, VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
str[VARSIZE(arg1) - VARHDRSZ]=0;
int nf;
int dtype;
int dterr;
int tzp;
char workbuf[MAXDATELEN + 1];
char *field[MAXDATEFIELDS];
int ftype[MAXDATEFIELDS];
elog(INFO, str);
//elog(INFO, arg1->vl_len_);
dterr = ParseDateTime(str, workbuf, sizeof(workbuf), field, ftype,
MAXDATEFIELDS, &nf);

if (dterr == 0)
    dterr = DecodeDateTime(field, ftype, nf, &dtype, tm, &fsec, &tzp);
if (dterr == 0){
//    elog(INFO, "true");
    PG_RETURN_BOOL(true);
}
else
{
//    elog(INFO, "false");
    PG_RETURN_BOOL(false);
}
}

```

接下来我们将上面的 C 语言代码编译成 .so 的库文件：

```

gcc -m64 -I$GPHOME/include -I$GPHOME/include/postgresql -I$GPHOME/include/
libpq -I$GPHOME/include/postgresql/server -I$GPHOME/include/postgresql/internal
-O2 -Wall -Wmissing-prototypes -Wpointer-arith -Winline -Wdeclaration-after-
statement -Wendif-labels -fno-strict-aliasing -fwrapv -fPIC -I. -c -o is_date.o
is_date.c
gcc -m64 -shared is_date.o -L$GPHOME/lib -L$GPHOME/lib/postgresql -o is_date.so

```

将 is\_date.so 这个库文件通过 scp 复制到 Greenplum 集群的每一台机器上，放到 /home/gpadmin1/cxf/is\_date 目录下。

创建函数：

```

CREATE or replace FUNCTION is_date(varchar)
RETURNS bool AS '/home/gpadmin1/cxf/is_date'
LANGUAGE C IMMUTABLE ;

```

is\_date 函数的使用效果：

```

testDB=# select is_date('2011-12-10 10:10:10');
 is_date
-----
 t

```

```
(1 row)
testDB=# select is_date('2011-13-10 10:10:10');
 is_date
-----
 f
(1 row)
testDB=# select is_date('sfd');
 is_date
-----
 f
(1 row)
```



**说明** 如果字符串是时间格式，并且时间正确，则返回 true，否则返回 false。

### 6.4.3 plpython

笔者对 Python 比较了解，本书大部分函数的例子都是使用 plpython 来写的。熟悉脚本语言的人应该知道，python 无论在语法上还是在功能上，都非常完善，再加上 python 有着非常丰富且实用的类库，使用起来非常方便。与上面介绍的几个方法一样，语法方面就不一一介绍了，这里直接使用 plpython 来编写 UDF，快速实现我们的需求。

#### 1. 将字符串反转

```
CREATE or REPLACE FUNCTION public.reverse(str text)
RETURNS text
AS $$ 
    if str!=None:
        return str[::-1]
    else:
        return None
$$ LANGUAGE plpython;
```

#### 2. 字符串拼接

类似 string\_agg 的功能，在 Greenplum 3.x 版本中，没有这个函数，我们可以自己写一个。这里利用了 PostgreSQL 的聚合函数，具体用法可以参考官方文档。

```
CREATE or replace FUNCTION public.strcat_sfunc(str1 varchar,str2
varchar,delimiter varchar)
RETURNS varchar
AS $$ 
    if str1:
        return str1+delimiter+str2;
    else:
        return str2
$$ LANGUAGE plpython;
```

```

else:
    return str2;
$$ LANGUAGE plpythonu;

drop AGGREGATE PUBLIC.STRCAT(VARCHAR,varchar);
CREATE AGGREGATE PUBLIC.STRCAT(VARCHAR,varchar)
(
    SFUNC=strcat_sfunc,
    STYPE=VARCHAR
);

```

下面是一个示例，查询 test001 表的字段名，并用逗号分隔开：

```

testDB=# select STRCAT(attname::varchar,',') from pg_attribute where
attnum>0;
strcat
-----
id,name
(1 row)

```

### 3. Json 解析

在标准 SQL 语法中，有普通函数 UDF（1 对 1，如 to\_char、substr 等），聚集函数 UDAF（多对一，如 sum、count 等），现在介绍如何编写（一对多）函数，即一行变多行。同时，在这个例子上将介绍使用 python 中类库来解析 Json 的格式，感受下 python 使用类库的方便。

假设表 dormitory 用于保存宿舍成员的名字，其中名字用 Json 的格式保存，格式如下：

```

{
    "people" : [
        {
            "firstName" : "Brett",
            "lastName" : "McLaughlin"
        },
        {
            "firstName" : "Jason",
            "lastName" : "Hunter"
        },
        {
            "firstName" : "Elliotte",
            "lastName" : "Harold"
        }
    ]
}

```

现在我们要在这个 Json 格式中将所有成员的名字取出来，宿舍成员个数不固定，函数的结果是每一行数据代表一个宿舍成员。

```

CREATE OR REPLACE FUNCTION public.json_parse(data text)
RETURNS SETOF TEXT
AS $$
```

```

import json
try:
    mydata=json.loads(data)
except:
    return ['parse json error']
returndata=[]
try:
    for people in mydata['people']:
        returndata.append(people['firstName'] + " " + people['lastName'])
except:
    return ['get data error']
return returndata
$$ LANGUAGE plpythonu;

```

`json_parse` 函数的使用方法如下：

```

testDB=# select dormname,json_parse(people) from dormitory;
dormname | json_parse
-----+-----
401   | Brett McLaughlin
401   | Jason Hunter
401   | Elliotte Harold
403   | Mac Apple
403   | Lucy Red
403   | Lei Li
402   | Tom Green
402   | Xiaofeng Chen
402   | Meihua Li
(9 rows)

```

强大的 Python 类库使编写 UDF 变得十分方便，读者可以尝试使用 Python 编写一些自定义函数。

## 6.5 Greenplum MapReduce

MapReduce 是 Google 提出的一种海量数据并发处理的一种编程模型，可以让程序员在多台机器上快速的编写出自己的代码。MapReduce 的基本思想是将大数据量拆分成一行一行 (`key,value`) 对，然后分发给 Map 模块，通过自定义的 Map 函数，生成新的 (`key,value`) 对，通过排序汇总，生成 (`key, output_list`) 发给自定义的 Reduce 函数处理，每一个 Reduce 函数处理一个唯一的 `key` 及这个 `key` 对应的所有 `value` 的列表，如果有需要，还可以将 Reduce 的结果作为下一个 Map 的数据再进行计算。每一个机器都可以启动多个 Map 和 Reduce 来计算结果。

Greenplum MapReduce 允许用户自己编写 map 和 reduce 函数，然后提交给 Greenplum 并发处理引擎处理，Greenplum 在分布式的环境下运行 MapReduce 代码，并处理所有 Segment 的报错信息。由于笔者很少用到 MapReduce 进行计算，故这里只是简单介绍一下如

如何使用 MapReduce，具体的一些细节，需要读者自行测试。Greenplum 中的 MapReduce 基本框架如图 6-5 所示。

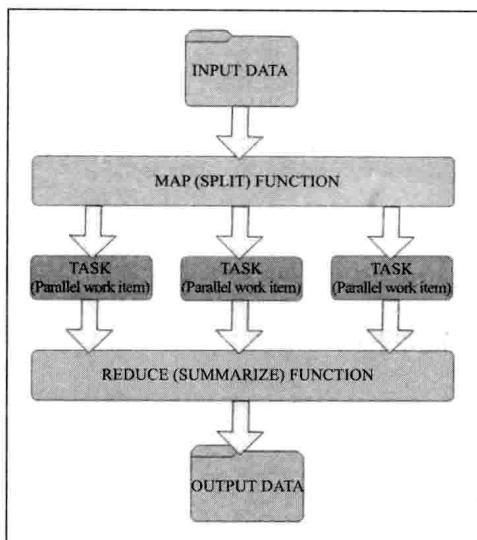


图 6-5 Greenplum 中的 MapReduce 基本框架

从图 6-5 中可以看到 MapReduce 的组成部分如表 6-11 所示。

表 6-11 MapReduce 的组成部分描述

组成部分	描述
INPUT	可以是数据库中的一张表、外部表读取外部文件，或者是一个 SQL
Map 函数	用户自定义，可以使用 Python、C、perl 等语言编写
Reduce 函数	用户自定义，可以使用 Python、C、perl 等语言编写，或者是使用一个系统自带的函数
OUTPUT	可以输出到数据库的某种表中，可以是标准输出，也可以是输出到一个外部文件中

在 Greenplum 中，执行 MapReduce 函数需要编写 yaml 文件。在 yaml 文件中，主要有以下几个部分需要定义。

## 1. INPUT

INPUT 有多种类型：可以是一个外部文本数据文件、一个数据库中的表或查询等，示例如下：

### (1) 外部文件

```

- INPUT:
  NAME: my_file_input
  FILE: seghostname:/var/data/gpfiles/employees.txt
  COLUMNS
    - first_name text
  
```

```

- last_name text
- dept text
- hire_date text
FORMAT: TEXT
DELIMITER: |

```

## (2) 数据库表

```

- INPUT:
NAME: my_table_input
TABLE: sales

```

## (3) 数据库查询

```

- INPUT:
NAME: my_query_input
QUERY: SELECT vendor, amt FROM sales WHERE region='usa';

```

## (4) 通过 gpfdist 导入外部文件

```

- INPUT:
NAME: my_distributed_input
# specifies the host, port and the desired files served
# by gpfdist. /* denotes all files on the gpfdist server
GPFDIST:
- gpfdisthost:8080/*
COLUMNS
- first_name text
- last_name text
- dept text
- hire_date text
FORMAT: TEXT
DELIMITER: '|'

```

## (5) 操作系统命令

```

- INPUT:
NAME: my_query_input
EXEC: /var/load_scripts/get_log_data.sh
COLUMNS
- url text
- date timestamp
FORMAT: TEXT
DELIMITER: '|'

```

类型(4)和(5)也可以采用在数据库中建成外部表的形式，就是可以当成数据库中的表操作，跟(2)一样，这两个类型的功能跟外部表有点重复了。

## 2. Map

Map是输入一行数据，有0个或多个数据输出，可以用Python或perl实现。

### 3. Reduce

也可以用 Python 或 perl 实现。

Greenplum 提供几个预定义 Reduce 函数：

IDENTITY	-	返回没有变化的 (key,value) 对
SUM	-	计算数字类型的和
AVG	-	计算数字类型的平均值
COUNT	-	计算输入数据的 count 数
MIN	-	计算数字类型的最小值
MAX	-	计算数字类型的最大值
( 函数对 value 进行相应操作 )		

### 4. OUTPUT

OUTPUT 可以是数据表也可以是文本文件，示例如下。

(1) 输出到文本文件

```
- OUTPUT:
NAME: gpmr_output
FILE: /var/data/mapreduce/wordcount.out
```

(2) 输出到数据表

```
- OUTPUT:
NAME: gpmr_output
TABLE: wordcount_out
KEYS:
- value
MODE: REPLACE (注：若是放此表并重建表，则用 REPLACE，若是想追加输出的数据，则用 APPEND。)
```

### 5. Task

Task 的描述是选择性的，主要用在多级的 MapReduce 任务中，一个 Task 的输出可以作为下一个 Task 或 Map 的输入。

例如：

```
- TASK:
NAME: document_prep
SOURCE: documents
MAP: document_map
```

### 6. EXECUTE

EXECUTE 就是将上面定义的步骤串联起来。

EXECUTE:

```
- RUN:
SOURCE: input_or_task_name
TARGET: output_name
MAP: map_function_name
REDUCE: reduce_function_name
```

MapReduce 最简单的程序一般都是 WordCount，现在就用 Greenplum 来编写一个简单的 WordCount，代码如下：

```
%YAML 1.1
---
VERSION: 1.0.0.1
DATABASE: testDB
USER: gpadmin
HOST: 10.20.151.7
PORT: 2345
DEFINE:
  - INPUT:
    NAME: sentences
    TABLE: sentences2
  - MAP:
    NAME: wordsplit
    LANGUAGE: python
    PARAMETERS: [id, sentence]
    FUNCTION: |
      for word in sentence.lower().split():
        yield([word,1])
    OPTIMIZE: STRICT IMMUTABLE
    RETURNS:
      - key text
      - value integer
  - OUTPUT:
    NAME: wordcount_result
    TABLE: wordcount_result
    MODE: REPLACE
EXECUTE:
  - RUN:
    SOURCE: sentences
    MAP: wordsplit
    REDUCE: SUM
    TARGET: wordcount_result
```

其中 sentences2 是原表，表中有 100 万行测试数据，每一行都是一个文本，Map 阶段将这些句子拆分成一个个单词，Reduce 阶段则执行系统自带的 SUM 命令，统计出每个单词出现的次数。执行这个 MapReduce 任务的命令及时间消耗如下：

```
$ time gpmapreduce -f wordcount.yaml
mapreduce_26132_run_1
DONE
```

```
real    1m35.634s
user    0m0.002s
sys     0m0.004s
```

可以看出，这个 SQL 消耗了 95 秒，在运行的过程中，CPU 的占有率都是 100%。为了让读者有一个比较，下面用 SQL 来实现这个逻辑，然后比较两者的速度。

首先，为了用 SQL 来实现这个逻辑，必须编写一个 UDF 函数，将句子拆分成一个个的单词，为了提升性能，在 UDF 中先对数据进行统计，代码如下：

```
create type wordcount_type as (word varchar(128),cnt int);
CREATE or REPLACE FUNCTION public.word_split(sentence text)
RETURNS setof wordcount_type
AS $$

dict={}
for word in sentence.lower().split():
    if dict.has_key(word):
        dict[word] = dict[word]+1
    else:
        dict[word] = 1
return dict.items()
$$ LANGUAGE plpythonu;
```

函数的执行效果如下：

```
testDB=# select * from word_split('hello hello cxf world cxf');
 word | cnt
-----+---
 world | 1
 hello | 2
 cxf  | 2
(3 rows)
```

现在句子已经跟 MapReduce 一样，被拆分成一个个单词了。可以使用 SQL 来实现这个逻辑了，语句如下：

```
testDB=# insert into wordcount_sqlresult
testDB-# select (ws).word,sum((ws).cnt)
testDB-#   from (
testDB(#       select word_split(sentence) ws
testDB(#           from sentences2
testDB(#           ) t
testDB-#   group by (ws).word;
INSERT 0 444
Time: 64894.717 ms
```

在 SQL 执行的过程中，Segment 上 CPU 的使用率也都是 100%，这个 SQL 只用了 65 秒。比使用 MapReduce 的效率高，为什么会这样呢？下面我们来分析下这两种方法的异同。

相同点：首先，无论使用 Greenplum 的 SQL 执行，还是 MapReduce，都是并行执行的，

所有的机器都同时在工作，同时，计算都是在数据所在节点上进行的，以提升性能。其次，对于 SQL 来说，先将句子拆分成单词，然后再对数据进行统计，这与 MapReduce 的思路也是一样的。

不同点：对于 MapReduce 来说，从 Map 到 Reduce 的阶段，需要对数据进行排序，然后把相同 key 的数据抛给 Reduce 执行。但是对于 SQL 来说，通过执行计划可以看到，在进行聚合的时候，数据库其实使用的是 Hash 聚合，效率比排序要高，所以这里使用 SQL 的效率要高一些。（关于 Hash 聚合的原理可参考第 5 章）。

为了验证是否如此，可以将 SQL 采取 Group 聚合，这样也都是使用排序来进行聚合了，使用 Group 聚合消耗的时间是 146 秒，这个时间明显比 MapReduce 的消耗要慢，会出现这种情况还是因为 SQL 的模型比 MapReduce 的模型要更加复杂，SQL 要考虑到事务、锁等问题，模型比较复杂，消耗的时间也肯定会更多。

通过上面的比较，读者应该对 MapReduce 有一个比较清楚的了解。对于 Greenplum 来说，大部分应用使用的还是 SQL，当然也有一些场景更加适合使用 MapReduce，由于两者的编程模型不一样，比如在实现聚合的方式上。如何选择更合适的方法，获取更好的性能，读者可针对应用场景进行测试后再决定。

## 6.6 小结

本章分别介绍了 Greenplum 的一些高级的特性，这些高级特性都是与 PostgreSQL 不一样的。能够合理地利用这些特性，可以获得更高的易用性，并获得性能的提升，当然，在使用每一种新特性的情况下，都必须对其架构及原理有一定的了解，清楚这些特性的优缺点，从而应用在相应的场景上以获得更佳的性能。

**压缩表：**对数据采用压缩存储的方法，可以利用 CPU 来节省存储空间，降低 IO 的消耗，对于数据库应用性能有着重要的影响。笔者所在公司 Greenplum 数据库的大部分数据表都是通过压缩表进行保存的。

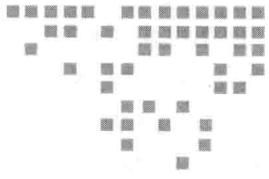
**列存储：**一种特殊的压缩表，将数据按列存储，可以提升压缩率，对于数据库这种 IO 密集型的应用，可以大大提升性能。但是列存储同时也带来了大量的小文件，会对文件系统造成一定的压力，故不能大规模使用。具体的性能差异，读者可以根据实际应用场景进行测试。

**外部表：**Greenplum 通过外部表可以非常方便地进行数据的导入导出，通过并行处理，性能非常高。可执行外部表提供了一种可拓展的方法，使得数据导入导出更加灵活。同时，现在 Hadoop 非常火，Greenplum 通过提供与 HDFS 快速交互的方式，使数据库与 Hadoop 可以灵活地结合在一起使用。

**自定义函数：**要使数据库使用起来更加方便，自定义函数在拓展 SQL 上有着很重要的作用，通过自定义函数，再结合 SQL，可以实现很多功能。在自定义函数上，Greenplum 提

供了非常灵活的语言实现，读者可以针对具体的应用场景以及对语言的熟悉程度，选择合适的语言编写自己的 UDF。

**MapReduce**：当下非常流行的一种编程思想，它简化了编程模型，通过这种编程思路，可以解决很多 SQL 无法完成的功能。Greenplum 的 MapReduce 很好地与数据库进行了结合，提供了另外一种处理数据的方法。



第 7 章

Chapter 7

## Greenplum 架构介绍

本章开始讲解 Greenplum 的架构，主要从并行计算和并行数据库入手来分析 Greenplum 架构的特性。在进行 Greenplum 开发、管理及大数据分析架构选型前，了解 Greenplum 的架构是必不可少的。

Greenplum 架构在前面几章（第 2 章和第 5 章）中都有简单介绍，本章将更详细系统地介绍 Greenplum 架构的特点及优势，最后将介绍其他大型分布式数据库的特点。

### 7.1 并行和分布式计算

#### 1. 并行计算

并行计算（Parallel computing）一般是指许多指令同时进行的计算模式。相对于串行计算，并行计算可以划分成时间并行和空间并行。时间并行即流水线技术，空间并行使用多个处理器执行并发计算，当前研究的主要问题是空间的并行问题。空间上的并行导致两类并行机器的产生，即单指令流多数据流（SIMD）和多指令流多数据流（MIMD）。MIMD 类的机器又可分为常见的五类：并行向量处理机（PVP）、对称多处理机（SMP）、大规模并行处理机（MPP）、工作站机群（COW）、分布式共享存储处理机（DSM）。我们简单看一下 SMP、DSM（NUMA）、MPP。

##### （1）SMP

所谓对称多处理器，是指服务器中多个 CPU 对称工作，无主次或从属关系，各 CPU 共享相同的物理内存，每个 CPU 访问内存中的任何地址所需时间是相同的，因此 SMP 也被称

为一致存储器访问结构 (Uniform Memory Access, UMA)。对 SMP 服务器进行扩展的方式包括增加内存、使用更快的 CPU、增加 CPU、扩充 I/O (槽口数与总线数) 以及添加更多的外部设备 (通常是磁盘存储)。SMP 服务器的主要特征是共享，系统中所有资源 (CPU、内存、I/O 等) 都是共享的。也正是由于这种特征，导致了 SMP 服务器的主要问题，那就是它的扩展能力非常有限。对于 SMP 服务器而言，每一个共享的环节都可能造成 SMP 服务器扩展时的瓶颈，而最受限制的则是内存。由于每个 CPU 必须通过相同的内存总线访问相同的内存资源，因此随着 CPU 数量的增加，内存访问冲突将迅速增加，最终会造成 CPU 资源的浪费，使 CPU 性能的有效性大大降低。实验证明，SMP 服务器 CPU 利用率最好的情况是 CPU 为 2 ~ 4 个。

### (2) NUMA

NUMA (Non-Uniform Memory Access) 服务器的基本特征是具有多个 CPU 模块，每个 CPU 模块由多个 CPU (如 4 个) 组成，并且具有独立的本地内存、I/O 槽口等。由于其节点之间可以通过互联模块 (又称为 Crossbar Switch) 进行连接和信息交互，因此每个 CPU 可以访问整个系统的内存 (这是 NUMA 系统与 MPP 系统的重要差别)。显然，访问本地内存的速度将远远高于访问远程内存 (系统内其他节点的内存) 的速度，这也是非一致存储访问 NUMA 的由来。但 NUMA 技术同样有一定缺陷，由于访问远程内存的延时远远超过本地内存，因此当 CPU 数量增加时，系统性能无法线性增加。

### (3) MPP

MPP 由多个 SMP 服务器通过一定的节点互联网络进行连接，协同工作，完成相同的服务，从用户的角度来看是一个服务器系统。其基本特征是由多个 SMP 服务器 (每个 SMP 服务器被称做节点) 通过节点互联网络连接而成，每个节点只访问自己的本地资源 (内存、存储等)，是一种完全无共享 (Shared-Nothing) 结构，因而扩展能力最好，理论上其扩展无限制。在 MPP 系统中，每个 SMP 节点也可以运行自己的操作系统、数据库等。但和 NUMA 不同的是，它不存在异地内存访问的问题。换言之，每个节点内的 CPU 不能访问另一个节点的内存。节点之间的信息交互是通过节点互联网络实现的，这个过程一般称为数据重分配 (Data Redistribution)。但是 MPP 服务器需要一种复杂的机制来调度和平衡各个节点的负载和并行处理过程。目前一些基于 MPP 技术的服务器往往通过系统级软件 (如数据库) 来屏蔽这种复杂性。

## 2. 分布式计算

分布式系统 (distributed system) 是建立在网络之上的软件系统。分布式系统具有高度的内聚性和透明性。因此，网络和分布式系统之间的区别更多的在于高层软件 (特别是操作系统)，而不是硬件。

内聚性是指每一个数据库分布节点高度自治，有本地的数据库管理系统。透明性是指每一个数据库分布节点对用户的应用来说都是透明的，看不出是本地还是远程。在分布式数据

库系统中，用户感觉不到数据是分布的，即用户无须知道关系是否分割、有无副本、数据存于哪个站点以及事务在哪个站点上执行等。

分布式计算就是研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给许多计算机进行处理，最后把这些计算结果综合起来得到最终的结果。

MapReduce 就是分布式计算中最典型的一种编程方法，是 Google 提出的一个软件架构，用于大规模数据集（大于 1TB）的并行运算。关于“Map”和“Reduce”，其中 Map（映射）函数，用来把一组键值对映射成一组新的键值对，Reduce（化简）函数，用来保证所有映射的键值对中的每一个共享相同的键组。很多数据计算都可以拆分成 Map 和 Reduce，通过 MapReduce 让用户方便地在分布式系统中运行数据处理程序。

## 7.2 并行数据库

并行数据库要求尽可能并行执行所有的数据库操作，从而在整体上提高数据库系统的性能。根据所在计算机的处理器（Processor）、内存（Memory）及存储设备（Storage）的相互关系，并行数据库可以归纳为三种基本的体系结构（这也是并行计算的三种基本体系结构），即共享内存结构（Shared-Memory）、共享磁盘结构（Shared-Disk）和无共享资源结构（Shared- Nothing）。

### （1）Shared-Memory 结构

该结构包括多个处理器、一个全局共享的内存（主存储器）和多个磁盘存储，各个处理器通过高速通信网络（Interconnection Network）与共享内存连接，并均可直接访问系统中的一个、多个或全部的磁盘存储，在系统中，所有的内存和磁盘存储均由多个处理器共享。在并行数据库领域，Shared-Memory 结构很少被使用。

### （2）Shared-Disk 结构

该结构由多个具有独立内存（主存储器）的处理器和多个磁盘存储构成，各个处理器相互之间没有任何直接的信息和数据的交换，多个处理器和磁盘存储由高速通信网络连接，每个处理器都可以读写全部的磁盘存储。Shared-Disk 结构的典型代表是 Oracle 集群。

### （3）Shared-Nothing 结构

该结构由多个完全独立的处理节点构成，每个处理节点具有自己独立的处理器、独立的内存（主存储器）和独立的磁盘存储，多个处理节点在处理器由高速通信网络连接，系统中的各个处理器使用自己的内存独立地处理自己的数据。

在这种结构中，每一个处理节点就是一个小型的数据库系统，多个节点一起构成整个的分布式的并行数据库系统。由于每个处理器使用自己的资源处理自己的数据，不存在内存和磁盘的争用，从而提高了整体性能。另外这种结构具有优良的可扩展性，只需增加额外的处理节点，就可以以接近线性的比例增加系统的处理能力。Shared-Nothing 结构的典型代表是 Teradata、Vertica、Greenplum、Aster Data、IBM DB2 和 MySQL 的集群也使用了这种结构。

### 7.3 Greenplum 架构分析

Greenplum 的高性能得益于其良好的体系结构。Greenplum 的架构采用了 MPP（大规模并行处理）。在 MPP 系统中，每个 SMP 节点也可以运行自己的操作系统、数据库等。换言之，每个节点内的 CPU 不能访问另一个节点的内存。节点之间的信息交互是通过节点互联网络实现的，这个过程一般称为数据重分配（Data Redistribution）。与传统的 SMP 架构明显不同，在通常情况下，MPP 系统因为要在不同处理单元之间传送信息，所以它的效率要比 SMP 要差一点，但是这也不是绝对的，由于 MPP 系统不共享资源，因此对它而言，资源比 SMP 要多，当需要处理的事务达到一定规模时，MPP 的效率要比 SMP 好。视通信时间占用计算时间的比例来判定，如果通信时间比较多，那 MPP 系统就不占优势了，相反，如果通信时间比较少，那么 MPP 系统可以充分发挥资源的优势，达到高效率。在当前使用的 OTLP 程序中，用户访问一个中心数据库，如果采用 SMP 系统结构，它的效率要比采用 MPP 结构快得多。而 MPP 系统在决策支持和数据挖掘方面显示了优势，可以这样说，如果操作相互之间没有什么关系，处理单元之间需要进行的通信比较少，采用 MPP 系统就要好，相反就不合适了。

Greenplum 是一种基于 PostgreSQL（开源数据库）的分布式数据库，其采用的 Shared-Nothing 架构（MPP）、主机、操作系统、内存、存储都是自我控制的，不存在共享。Greenplum 架构主要由 Master Host、Segment Host、Interconnect 三大部分组成，如图 7-1 所示。

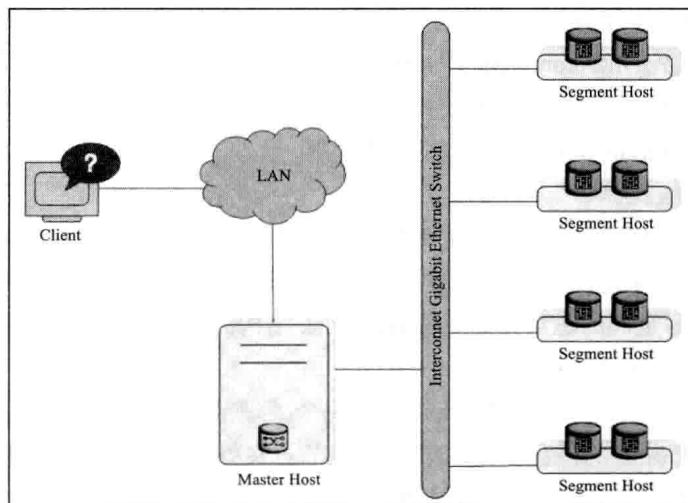


图 7-1 Greenplum 架构

#### (1) Master Host

Master Host 是 Greenplum 数据库系统的入口，它接受客户端的连接请求、负责权限认证、处理 SQL 命令（SQL 的解析并形成执行计划）、分发执行计划、汇总 Segment 的执行结

果、返回执行结果给客户端。由于 Greenplum 数据库是基于 PostgreSQL 数据库的，终端用户通过 Master 同数据库交互就如同操作一个普通的 PostgreSQL 数据库。用户可以使用 PostgreSQL 或者 JDBC、ODBC 等应用程序接口连接数据库。Greenplum Master 不存储业务数据，仅存储数据字典。

### (2) Segment Host

Segment Host 负责业务数据的存储和存取、用户查询 SQL 的执行。Greenplum 数据库的性能由一组 Segment 服务中最慢的 Segment 决定，因此要确保基本的运行 Greenplum 数据的硬件与操作系统在同一个性能级别，同样建议在 Greenplum 数据系统中的所有的 Segment 机器有一样的资源与配置。

### (3) Interconnect

Interconnect 是 Greenplum 数据库的网络层，在每个 Segment 中起到一个 IPC 的作用（Inter-Process Communication）。Greenplum 数据库推荐使用标准的千兆以太网交换机来做 Interconnect。在默认情况下，Interconnect 使用的是 UDP 协议来进行传输，因为在 Greenplum 的软件当中，没有其他包去检查和验证 UDP，所以 UDP 协议在可靠性上等同于 TCP 协议，并且超过了 TCP 的性能和可扩展性，而且使用 TCP 协议会有一个限制，最大只能使用 1000 个 Segment 实例。

## 7.4 冗余与故障切换

Greenplum 数据库配置了镜像节点之后，当主节点不可用时会自动切换至镜像节点，集群仍然保持可用状态。当主节点恢复并启动之后，主节点会自动恢复期间的变更。只要 Master 不能连接上 Segment 实例时，就会在系统表中将此实例标识为不可用，并用镜像节点来代替。当然如果在配置服务器集群时，没有开启镜像功能，任何一个 Segment 实例不可用，整个集群将变得不可用，大大降低集群可用性。镜像节点一般需要和主节点位于不同服务器上，如图 7-2 所示。

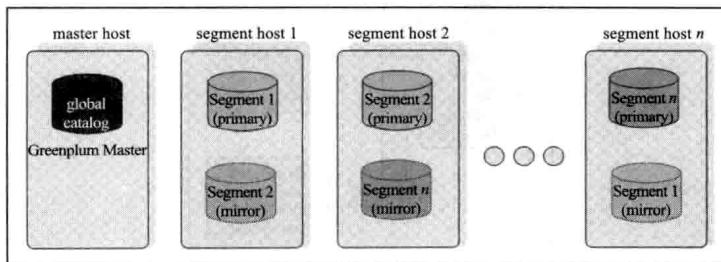


图 7-2 Greenplum 的 Segment 镜像配置

除了像图 7-2 这样为 Segment 实例配置镜像节点，我们也可以为 Master 节点配置镜像，

确保系统的变更信息不会丢失，提升系统的健壮性，如图 7-3 所示。

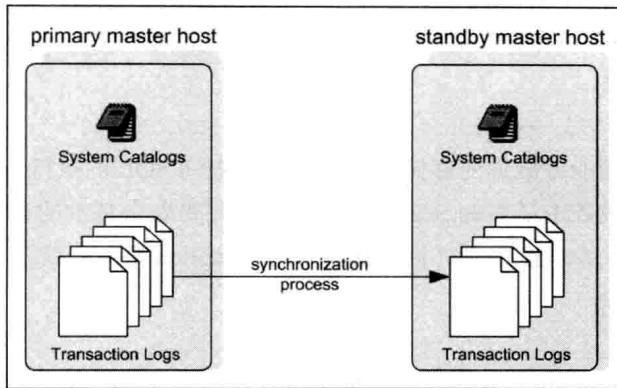


图 7-3 Greenplum 的 Master 镜像配置

另外，我们还需要从网络配置上确保节点之间的网络交互的高可用。

## 7.5 数据分布及负载均衡

Greenplum 是一个分布式数据库系统，故其所有的业务数据都是物理存放在集群的所有 Segment 实例数据库上。这些看似独立的 PostgreSQL 数据库通过网络相互连接，并和 Master 节点协同构成整个数据库集群。在学习和使用 Greenplum 数据库时，我们必须理解数据在集群中是如何存放的。下面我们以一个简单数据仓库星型模型为例，形象地介绍数据是如何存放的，如图 7-4 所示。

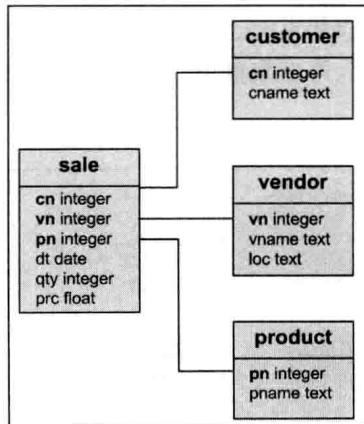


图 7-4 示例模型

在 Greenplum 数据库中所有表都是分布式的，所以每一张表都会被切片，每个 Segment

实例数据库会存放相应的数据片段。切片规则可由用户定义，可选的方案有根据用户对每一表指定的 Hash Key 进行 Hash 分片或者选择随机分片。图 7-5 中的业务场景在 Greenplum 数据库中的存放规则如图 7-5 所示。sale、customer、product、vendor 四张表的数据都会切片存放在所有的 Segment 上，当我们需要进行数据分析时，所有 Segment 实例同时工作，由于每个 Segment 只需要计算一部分数据，所以计算效率将会大大提升。这正是 Greenplum 数据库分布式计算提升性能的关键所在。接下来我们简单介绍一下 Greenplum 数据库在创建或者修改表时可选的数据分散策略——Hash Distribution 和 Random Distribution。

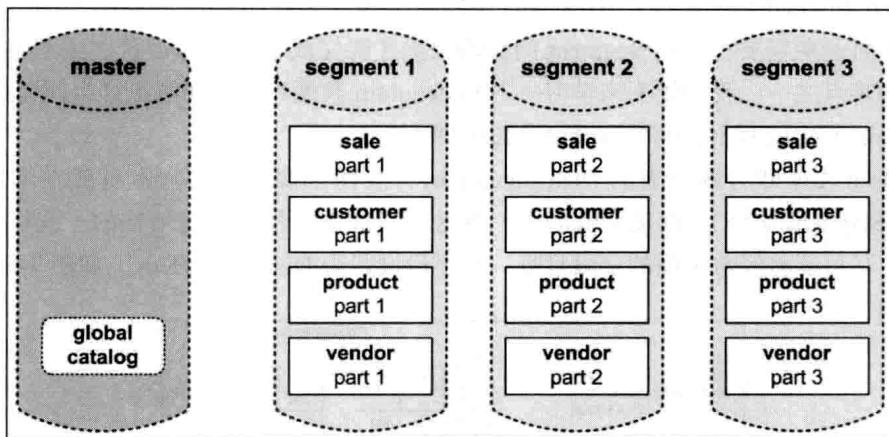


图 7-5 Greenplum 数据分布

### (1) Hash Distribution

当选择 Hash Distribution 策略时，可指定表的一列或者多列组合。Greenplum 会根据指定的 Hash Key 列计算每一行数据对应的 Hash 值，并映射至相应的 Segment 实例。当选择的 Hash Key 列的值唯一时，数据将会均匀地分散至所有 Segment 实例。Greenplum 数据库默认会采用 Hash Distribution，如果创建表时未指定 Distributed Key，则会选择 Primary Key 作为 Distributed Key，如果 Primary Key 也不存在，就会选择表的第一列作为 Distributed Key。

### (2) Random Distribution

当选择 Random Distribution 时，数据将会随机分配至 Segment，相同值的数据行不一定会分发至同一个 Segment。虽然 Random Distribution 策略可以确保数据平均分散至所有 Segment，但是在进行表关联分析时，仍然会按照关联键重分布数据，所以 Random Distribution 策略通常不是一个明智的选择（除非你的 SQL 只有对单表进行全局的聚合操作，即没有 group by 或者 join 等需要数据重分布的操作，此时这种分布模式可以避免数据倾斜，而且性能更高）。

在数据建模时，表的分片规则选取一定要慎重，尽可能选择唯一且常用于 Join 的列作为 Distribution Key。这样数据才会均匀分散至所有 Segment 实例，相应的查询及计算的负载

也会平摊至整个集群，进而最大程度体现分布式计算的优势。假如 Distribution Key 选取不当（如选择性别列作为 Distribution Key），数据将只会分散至少数几个 Segment，这样将只有少数 Segment 处理相应的计算请求，完全不能发挥整个集群的计算资源，实现并行计算。

## 7.6 跨库关联

Greenplum 数据库将表数据分散至所有 Segment 实例，当需要进行表关联分析时，由于各个表的 Distributed Key 不同，相同值的行数据可能分布在不同服务器的不同 Segment 实例，因此不可避免需要在不同 Segment 间移动数据才能完成 Join 操作。跨库关联也正是分布式数据库的难点之一。我们接下来学习一下 Greenplum 数据库是如何解决这个问题的。

### (1) Join 操作的两个表的 Distributed Key 即 Join Key

由于 Join Key 即为两个表的 Distributed Key，故两个表关联的行本身就在本地数据库（即同一个 Segment 实例），直接关联即可。在这种情况下，性能也是最佳的。我们在进行模型设计时，尽可能将经常关联的字段且唯一的字段设置为 Distributed Key，如图 7-6 所示。

No Redistribution needed.		
SELECT	...	
FROM	Table1 T1	
INNER JOIN	Table2 T2	
ON	T1.A = T2.A;	
	T1	
	A   B   C	
	DK	
	100   214   433	
	T2	
	A   B   C	
	DK	
	100   725   002	

图 7-6 Distributed Key 和 Join Key 相同

### (2) Join 操作的两个表中的一个 Distributed Key 与 Join Key 相同

由于其中一个表的 Join Key 和 Distributed Key 不一致，故两个表关联的行不在同一个数据库中，便无法完成 Join 操作。在这种情况下就不可避免地需要数据跨节点移动，将关联的行组织在同一个 Segment 实例，最终完成 Join 操作。Greenplum 可以选择两种方式将关联的行组织在同一个 Segment 中，其中一个方式是将 Join Key 和 Distributed Key 不一致的表按照关联字段重分布（即 Redistribute Motion），另一种方式是可以将 Join Key 和 Distributed Key 不一致的表在每个 Segment 广播（即 Broadcast Motion），也就是每个 Segment 都复制一份全量，如图 7-7 所示。

Redistribution needed.		
SELECT	...	
FROM	Table3 T3	
INNER JOIN	Table4 T4	
ON	T3.A = T4.B;	
	T3	
	A   B   C	
	DK	
	255   345   225	
	T4	
	A   B   C	
	DK	
	867   255   566	
Redistribute T4 rows on column B.		
	A   B   C	
	DK	
	867   255   566	

图 7-7 T4 需要重分布

### (3) Join 操作的两个表的 Distributed Key 和 Join Key 都不同

由于两个表的 Join Key 和 Distributed Key 都不一致，故两个表关联的行不在同一个数据库中，便无法完成 Join 操作。同样在这种情况下，一种方式将两个表都按照关联字段重分布（即 Redistribute Motion），另一种方式可以将其中一个表在每个 Segment 广播（即 Broadcast Motion），也就是每个 Segment 都复制一份全量，如图 7-8 所示。

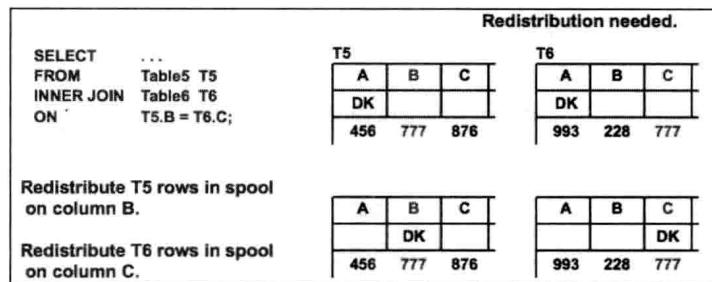


图 7-8 两个表皆重分布

综合上面讨论的三种情况，我们可以看出，Greenplum 主要采用了 Redistribute Motion 和 Broadcast Motion 这两方式来解决跨节点关联这个难点。在 Greenplum 具体分析及选择执行计划时，选择哪一种方式会基于执行计划的成本，这部分内容在第 5 章中已经详细介绍了。

## 7.7 分布式事务

分布式事务处理是指一个事务可能涉及多个数据库操作，而分布的关键在于两阶段提交（Two Phase Commit, 2PC）。两阶段提交用于确保所有分布式事务能够同时提交或者回滚，以便数据库能够处于一致性状态。

分布式事务处理的关键是必须有一种方法可以知道事务在任何地方所做的所有动作，提交或回滚事务必须产生一致的结果（全部提交或全部回滚），所以就需要有一个事务协调器来负责处理每一个数据库的事务。在 Greenplum 中，Master 就充当了这样一个角色。

两阶段提交顾名思义把事务提交分成两个阶段：

第一阶段，Master 向每个 Segment 发出“准备提交”请求，数据库收到请求后执行相同的数据修改和日志记录等处理，处理完成后只是把事务的状态改成“可以提交”，然后把执行的结果返回给 Master。

第二阶段，Master 收到回应后进入第二阶段，如果所有 Segment 都返回成功，那么 Master 向所有的 Segment 发出“确认提交”请求，数据库服务器把事务的“可以提交”状态改为“提交完成”状态，这个事务就算正常完成了，如图 7-9 所示；如果在第一阶段内有 Segment 执行发生了错误，Master 收不到 Segment 回应或者 Segment 返回失败，则认为事务失败，回撤事务，Segment 收到 Rollback 的命令，即将当前事务全部回滚，如图 7-10 所示。

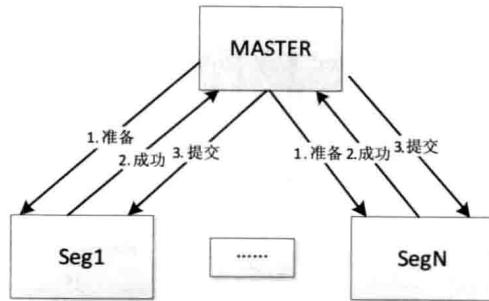


图 7-9 两阶段提交正常的流程

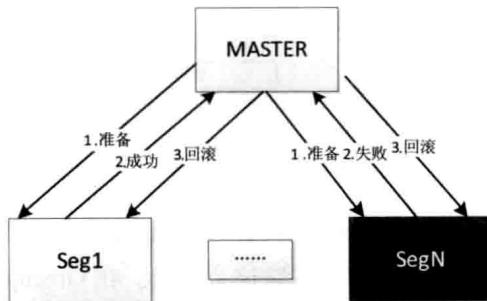


图 7-10 两阶段提交失败的流程

从图 7-10 的流程可以看出，两阶段提交并不能保证数据一定会恢复到一致性状态。例如，当 Master 向 Segment 发出提交命令的时候，在提交过程中，有一个 Segment 失败了，但是其他 Segment 已经提交成功了，那么这个事务是不能再次回滚的，这样就会造成不一致的情况。

两阶段提交的核心思想就是将可能发生不一致的时间降低到最小，因为提交过程对数据库来说应该是一个瞬间完成的动作，而且发生错误的概率极小，危险期比较短。

在 `gp_distributed_xacts` 视图中，记录了正在进行的分布式事务的状态。`gp_distributed_log` 记录了分布式事务的历史信息，这个视图中应该包括提交跟回滚事务的信息，但是在实际测试过程中，却只包括了提交的事务信息。下面做个实验演示下。

现在的 Greenplum 有一个连接，就是当前的连接，没有其他连接。当还没有事务时，如图 7-11 所示。

distributed_xid	distributed_id	state	gp_session_id	xmin_distributed_snapshot
823	1335169489-0000000823	Active Not Distributed	104	823

图 7-11 无事务状态

当前没有事务，所以显示“Active Not Distributed”。接下来启动一个事务，创建一个表后将其提交，如图 7-12 所示。

```

testDB=# begin;
BEGIN
testDB=# select * from gp_distributed_xacts;
distributed_xid      distributed_id | state          | gp_session_id | xmin_distributed_snapshot
-----+-----+-----+-----+-----+
(1 row)   824 | 1335169489-0000000824 | Active Distributed | 104 | 824
testDB=# create table test_trans(a int) distributed by(a);
CREATE TABLE
testDB=# end;
COMMIT

```

图 7-12 开启事务

这个时候事务正在运行，状态变成了“Active Distributed”。在事务成功提交后，我们查询下 `gp_distributed_log` 中的状态，如图 7-13 所示。

```

testDB=# select dbid,distributed_xid,distributed_id,status
testDB-#   from gp_distributed_log
testDB-#   where distributed_id='1335169489-0000000824';
dbid | distributed_xid | distributed_id | status
-----+-----+-----+-----+
1     | 824           | 1335169489-0000000824 | Committed
(1 row)

```

图 7-13 事务日志

再看一下每个 Segment 的状态，如图 7-14 所示。

```

testDB=# select dbid,distributed_xid,distributed_id,status
testDB-#   from gp_dist_random('gp_distributed_log')
testDB-#   where distributed_id='1335169489-0000000824';
dbid | distributed_xid | distributed_id | status
-----+-----+-----+-----+
7     | 824           | 1335169489-0000000824 | Committed
6     | 824           | 1335169489-0000000824 | Committed
3     | 824           | 1335169489-0000000824 | Committed
5     | 824           | 1335169489-0000000824 | Committed
2     | 824           | 1335169489-0000000824 | Committed
4     | 824           | 1335169489-0000000824 | Committed
(6 rows)

```

图 7-14 所有 Segment 的事务日志

所有的状态都显示了这个事务已提交。

## 7.8 其他大数据分析方案

在大数据时代，基本上有两大阵营，分析型 RDBMS 和 Hadoop 生态系统。RDBMS 主要用于结构化数据分析，商业产品有 Oracle Exadata、EMC Greenplum、IBM Netezza、SAP SybaseIQ、HP Vertica、Teradata 等，开源项目有 C-Store、MonetDB、VectorWise、Infobright 等。而 Hadoop 生态系统主要用于非结构化数据分析，核心是 Map/Reduce。接下来我们简单分析几个大数据分析方案。

### (1) Hadoop

Hadoop 是 Apache 下的一个项目，由 HDFS、Map/Reduce、HBase、Hive 和 ZooKeeper 等成员组成，如图 7-15 所示。其中，HDFS 和 MapReduce 是两个最基础、最重要的成员。

HDFS 是 Google GFS 的开源版本，一个高度容错的分布式文件系统，它能够提供高吞吐量的数据访问，适合存储海量（PB 级）的大文件（通常超过 64MB），其原理示意如图 7-16 所示。

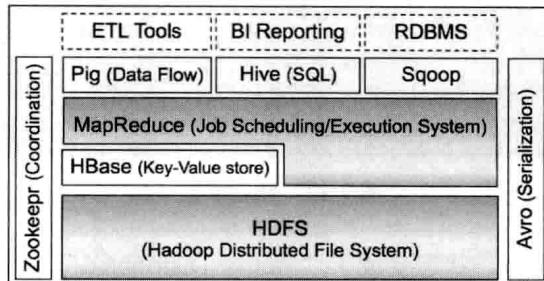


图 7-15 Hadoop 生态系统

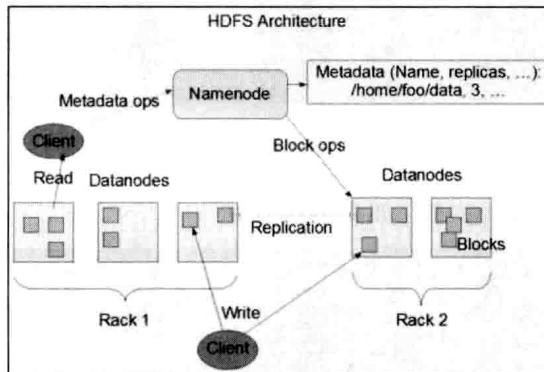


图 7-16 HDFS 体系结构

Hadoop Map/Reduce 是一个使用简易的软件框架，基于它写出来的应用程序能够运行在由上千个商用机器组成的大型集群上，并以一种可靠容错的方式并行处理上 T 级别的数据集。Map 负责将数据打散，Reduce 负责对数据进行聚集，用户只需要实现 Map 和 Reduce 两个接口，即可完成 TB 级数据的计算，常见的应用包括：日志分析和数据挖掘等数据分析应用。另外，还可用于科学数据计算，如圆周率 PI 的计算等。Hadoop Map/Reduce 的实现也采用了 Master/Slave 结构。Master 叫做 JobTracker，而 Slave 叫做 TaskTracker。用户提交的计算叫做 Job，每一个 Job 会被划分成若干个 Tasks。JobTracker 负责 Job 和 Tasks 的调度，而 TaskTracker 负责执行 Tasks。

## (2) Teradata

Teradata 是用于世界上最大的商用数据库的关系数据库管理系统。目前的技术允许数据库有数百 Terabyte 字节的容量，这就使 Teradata 成为一个大型数据库应用的正确选择。而 Teradata 数据库系统也可以只有 10GB 那么小。由于并行性能和可扩展能力，Teradata 可以使一个系统通过线性扩展从一个单一的节点开始扩展为多个节点的系统。Teradata 在整体上是按 Shared-Nothing 架构体系进行组织的，它的定位就是大型数据库系统，定位比较高。由于 Teradata 通常被用于 OLAP 应用，因此单机的 Teradata 系统很少见，即使是单机系统，Teradata 也建议使用 SMP 结构尽可能地提供更好的数据库性能。

根据 Shared-Nothing 的组成结构特点，在物理布局上，Teradata 系统主要包括处理节点（Node）、用于节点间通信的内部高速互联（InterConnection）、数据存储介质（通常是磁盘阵列）。每个节点都是 SMP（对称多处理器结构）结构的单机，多个节点一起构成一个 MPP（海量并行处理器结构）系统，多个节点之间的内部高速互联是通过一种被称为 BYNET 的硬件来实现的，整个系统的组成如图 7-17 所示。

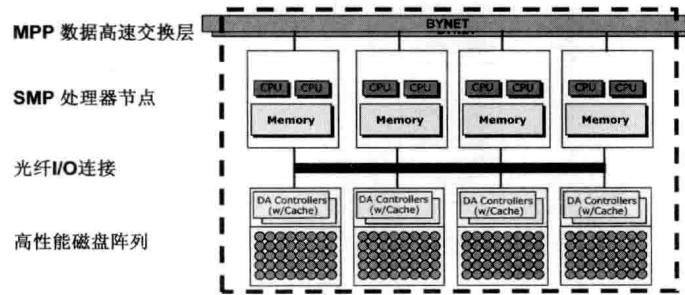


图 7-17 Teradata 系统整体结构

### (3) IBM Netezza

Netezza 是专门的数据仓库数据库。Netezza 将存储、处理、数据库和分析融入到一个高性能数据仓库设备中，该设备专为使大数据高级分析更简单、更迅捷和更易用而设计。Netezza 是软件与硬件不可分离的紧密结合体，无缝地整合数据库管理系统（DBMS）、服务器（Server）和存储设备（Storage），不需要复杂配置和调优就可以取得非常优异性能。

Netezza 的 AMPP（Asymmetric Massively Parallel Processing）是一个两层结构，如图 7-18 所示，专门为了处理多用户的大数据量查询而设计。AMPP 结构是 SMP 前端和 Shared-Nothing 的 MPP 后端的完美结合。前端是 SMP 的高性能 Linux 主机，其主要功能是通过标准的接口（SQL、ODBC、JDBC、OLE DB）对外提供服务。SMP 主机负责编译从应用程序发出的查询请求，生成优化过的可执行代码片段，称之为 snippet，然后分发这些代码片段到所有的 S-Blades 上并行执行。当所有的 S-Blades 都执行完毕时，SMP 主机汇总结果后把最终的结果返回给应用程序。后端由大量的 S-Blades 组成。主要的数据操作过程都是在 S-Blades 上完成的。S-Blades 之间是相互独立的，每个 S-Blades 都会占有自己磁盘和数据片（data slice），在并行处理的时候并不会相互影响。这种结构的好处是可以通过增加 S-Blades 节点和其所使用的磁盘来使性能得到近似线性的提升。Netezza 1000 的 S-Blades 数量可以扩展到 120 个。

### (4) Oracle Exadata

Oracle Exadata 也叫（HP Oracle Exadata）是一个高性能的存储软件和硬件产品系列，克服了传统存储系统的局限性，它通过采用大量的并行架构，显著增加了数据库服务器和存储系统之间的数据带宽。此外，智能存储软件卸载了 Oracle 11g 服务器的数据密集型查询处

理，并使查询处理更贴近数据，其结果是，通过更高的带宽连接加快了并行数据处理并减少了数据迁移量。

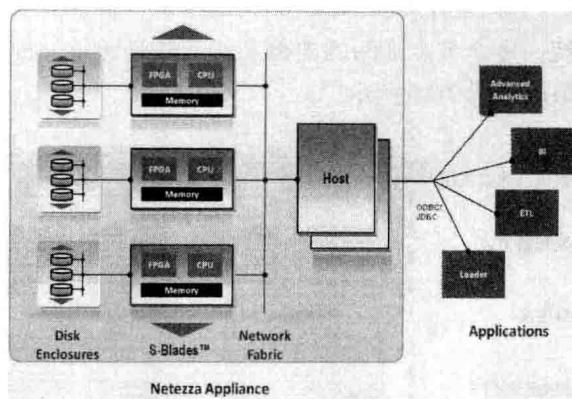


图 7-18 AMPP 架构

Oracle Exadata 提供一种混合式的数据库架构，即 Shared-Nothing 与 Shared-Disk 架构相结合，有效解决了两者的冲突，吸取两种架构长处：既可以满足 OLTP 的高并发、高可用特点；又可以满足 OLAP 的大数据量处理要求。整体架构如图 7-19 所示。

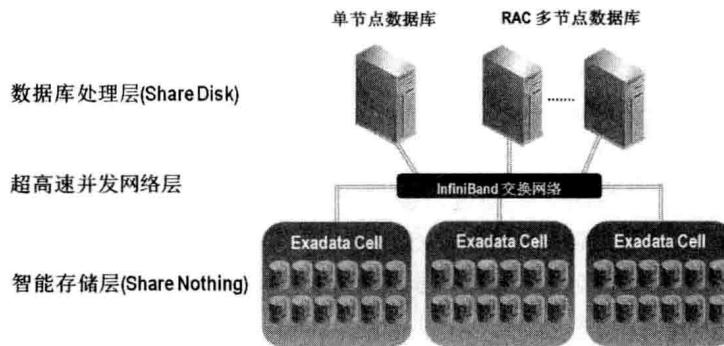
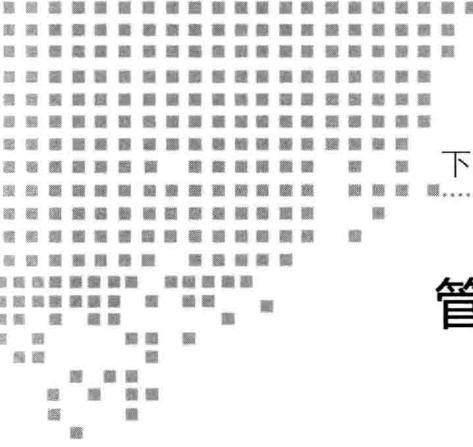


图 7-19 Exadata 架构

## 7.9 小结

本章从并行计算和并行数据库出发，讲解了 Greenplum 的总体架构，分析了如何实现高可用、跨库关联以及如何处理分布式事务等，最后还简单介绍了其他大数据分析方案，比如 Hadoop、Teradata 等。经过本章的学习，我们能够知道数据是如何在集群中存放，了解如何才能最大的发挥分布式计算的并行优势。



下篇

*Part 3*

## 管理篇

- 第 8 章 Greenplum 线上环境部署
  - 第 9 章 数据库管理
  - 第 10 章 数据库监控及调优
  - 第 11 章 解读 Greenplum 维护脚本
  - 第 12 章 备份及恢复策略
  - 第 13 章 数据库扩容
  - 第 14 章 基于 Greenplum 的海量数据实时分析服务平台
  - 第 15 章 使用 Greenplum 的常见报错及小技巧
- \*\*\*\*\*

## Greenplum 线上环境部署

本章开始讲解如何搭建一个高性能、安全可靠、可扩展、可管理的 Greenplum 集群。Greenplum 数据库的性能由最慢的 Segment 决定，因此要确保基本的运行 Greenplum 数据库的硬件与操作系统在同一个性能级别，并建议 Greenplum 数据库系统中所有的 Segment 机器有一样的资源和配置。

### 8.1 服务器硬件选型

数据库服务器硬件选型应该遵循以下几个原则。

#### (1) 高性能原则

保证所选购的服务器，不仅能够满足现有应用的需要，而且能够满足一定时期内业务量增长的需要。对于数据库而言，数据库性能依赖于硬件的性能和各种硬件资源的均衡，CPU、内存、磁盘、网络这几个关键组件在系统中都很关键，如果过分突出某一方面硬件资源则会造成大量的资源浪费，而任何一个硬件性能差都会成为系统的瓶颈，故在硬件选择的时候，需要根据应用需求在预算中做到平衡。

#### (2) 可靠性原则

不仅要考虑服务器单个节点的可靠性或稳定性，而且要考虑服务器与相关辅助系统之间连接的整体可靠性。

#### (3) 可扩展性原则

需要服务器能够在相应时间根据业务发展的需要对其进行相应的升级，如 CPU 型号升级、内存扩大、硬盘扩大、更换网卡等。

#### (4) 可管理性原则

需要服务器的软硬件对标准的管理系统提供支持。

### 8.1.1 CPU

目前，在做 CPU 选择方案时，主要考虑以下两个方面。

- Intel 还是 AMD
- 更快的处理器还是更多的处理器

Intel 作为单核运算速度方面的领导者，其 CPU 及相关组件相对较贵，而 AMD 在多核处理技术方面表现优异，仍不失为一个较好的选择。

如果仅有较少进程运行在单个处理器上，则可以考虑配备更快的 CPU。相反，如果大量并发进程运行在所有处理器上，则考虑配备更多的核。

一台服务器上 CPU 的数量决定部署到机器上的 Greenplum 数据库 Segment 实例的数量，比如一个 CPU 配置一个主 Segment。

### 8.1.2 内存

内存是计算机中重要的部件之一，它是与 CPU 进行沟通的桥梁。计算机中所有程序的运行都是在内存中进行的，因此内存的性能对计算机的影响非常大。内存的选择总体上来说是越大越好，不过当内存足以容纳所有业务数据时，则必须提升 CPU 性能才能获得性能的提升。

在 Greenplum 中，内存主要用于在 SQL 执行过程中保存中间结果（如排序、HashJoin、数据广播等），如果内存不够，Greenplum 会选择使用磁盘来缓存数据，这样就会大大降低 SQL 执行的性能。

另外，对于处理像数据库这样的海量数据，并且内存远小于数据存储的情况，如果内存已经够用，通过增加内存来获取性能的提升则非常细微。对于 Greenplum 这种数据库系统，磁盘的吞吐量决定着 Greenplum 的性能，磁盘吞吐越高性能越好。

### 8.1.3 磁盘及硬盘接口

硬盘分为固态硬盘（SSD）和机械硬盘（HDD），SSD 采用闪存颗粒来存储，HDD 采用磁性碟片来存储。

硬盘接口主要有如下几类。

#### (1) ATA

全称 Advanced Technology Attachment，是用传统的 40-pin 并口数据线连接主板与硬盘的。外部接口速度最大为 133MB/s。因为并口线的抗干扰性太差，且排线占空间，不利计算机散热，将逐渐被 SATA 所取代。

### (2) IDE

全称 Integrated Drive Electronics，即电子集成驱动器，是把“硬盘控制器”与“盘体”集成在一起的硬盘驱动器。

### (3) SCSI 小型计算机系统接口

同 IDE 和 ATA 完全不同的接口，IDE 接口是普通 PC 的标准接口，而 SCSI 并不是专门为硬盘设计的接口，是一种广泛应用于小型机上的高速数据传输技术。SCSI 接口具有应用范围广、任务多、带宽大、CPU 占用率低，以及热插拔等优点，但较高的价格使得它很难如 IDE 硬盘般普及，因此 SCSI 硬盘主要应用于中、高端服务器中。

### (4) SATA

全称 Serial Advanced Technology Attachment（串行高级技术附件，一种基于行业标准的串行硬件驱动器接口），是由 Intel、IBM、Dell、APT、Maxtor 和 Seagate 公司共同提出的硬盘接口规范。

### (5) SAS

全称 Serial Attached SCSI（串行连接 SCSI），缩写为 SAS，是新一代的 SCSI 技术。和现在流行的 Serial ATA（SATA）硬盘相同，都采用串行技术以获得更高的传输速度，并通过缩短连接线改善内部空间等。

### (6) SSD

全称 Solid State Disk，即固态硬盘。目前的硬盘（ATA 或 SATA）都是磁碟型的，数据就储存在磁碟扇区里，固态硬盘数据就储存在芯片里。

目前较为流行的选择主要集中在 SATA 和 SAS 上，这两者的主要区别如表 8-1 所示。

表 8-1 SATA 与 SAS 主要区别

类型名称	转速	容量	价格
SATA	7200	大	低
SAS	10000 ~ 15000	小	高

在选择硬盘和硬盘控制器时，确认选择的硬盘控制器能够包括硬盘带宽之和。假如你有 20 个 70Mb/s 内部带宽的硬盘，为了获得硬盘最佳性能，需要最少支持 1.4Gb/s 的硬盘控制器。

要提升服务器 I/O 吞吐量、可用性及存储容量，常见的方法是做 RAID，即独立冗余磁盘阵列（Redundant Array of Independent Disk，RAID）。几种常见的 RAID 技术如下。

### (1) RAID 0

从严格意义上说，RAID 0 不是 RAID，因为它没有数据冗余和校验。RAID 0 技术只是实现了条带化，具有很高的数据传输率，最高的存储利用率，但是 RAID 中硬盘数越多，安全性越低。

### (2) RAID 1

通常称为 RAID 镜像。RAID 1 主要是通过数据镜像实现数据冗余，在两对分离的磁盘

上产生互为备份的数据，因此 RAID 1 具有很高的安全性。但是 RAID 1 空间利用率低，磁盘控制器负载大，因此只有当系统需要极高的可靠性时，才选择 RAID 1。

#### (3) RAID 1+0

RAID 0+1 至少需要 4 块硬盘才可以实现，不过它综合了 RAID 0 和 RAID 1 的特点，独立磁盘配置成 RAID 0，两套完整的 RAID 0 互相镜像。它的读写性能出色，安全性也较高。但是，构建 RAID 0+1 阵列的成本投入大，数据空间利用率只有 50%，因此还不能称为经济高效的方案。

#### (4) RAID 5

RAID 5 是目前应用最广泛的 RAID 技术。各块独立硬盘进行条带化分割，相同的条带区进行奇偶校验（异或运算），校验数据平均分布在每块硬盘上。RAID 5 具有数据安全、读写速度快、空间利用率高等优点，应用非常广泛。但不足之处是，如果 1 块硬盘出现故障，整个系统的性能将大大降低。

### 8.1.4 网络

Greenplum 数据库互联的性能与 Segment 上网卡的网络负载有关，所以 Greenplum 服务器一般由一组带有多个网卡的硬件组成。为了达到最好性能，Greenplum 建议为 Segment 机器上的每一个主 Segment 配置一个千兆网卡，或者配置每台机器都有万兆网卡。

如果 Greenplum 数据库网络集群中有多个网络交换机，那么交换机之间均衡地分配子网较为理想，比如每个机器上的网卡 1 和 2 用其中一个交换机，网卡 3 和 4 使用另一个交换机。

## 8.2 服务器系统参数调整

对于不同的应用场景，每一种硬件都有不同的参数配置，通过参数调整，可以极大地提高读写性能。例如，对于 OLTP 数据库而言，关注的是磁盘的随机读写，提高单次读写的性能；而对于 OLAP 数据库而言，最关注的是磁盘的顺序读写，重点在于提高每次磁盘读写的吞吐量。

Greenplum 目前支持的操作系统有以下几种。

- SUSE Linux SLES 10.2 or higher
- CentOS 5.0 or higher
- RedHat Enterprise Linux 5.0 or higher
- Oracle Unbreakable Linux 5.5
- Solaris x86 v10 update 7

一般情况下，需要在操作系统中修改以下三种类型的参数。

#### (1) 共享内存

Greenplum 只有在操作系统中内存大小配置适当的时候才能正常工作。大部分操作系统

的共享内存设置太小，不适合 Greenplum 的场景，因为这样可以避免进程因为内存占用过高被操作系统停止。

### (2) 网络

Greenplum 的数据分布在各个节点上，因此在计算过程中经常需要将数据移动到其他节点上进行计算，这时，合理的网络配置就显得格外的重要。

### (3) 系统对用户的限制

操作系统在默认情况下会对用户进行一些资源的限制，以避免某个用户占用太多资源导致其他用户资源不可用。对于数据库来说，只会有一个操作系统用户，这些限制都必须取消掉，例如 Greenplum 会同时打开很多文件句柄，而操作系统默认的文件句柄数一般很小。

在笔者所在公司，用得最多的是 RedHat 和 Solaris，不同的操作系统版本，参数配置是不一样的，下面就列举 Solaris 跟 RedHat 这两种操作系统进行配置。

## 8.2.1 Solaris 参数修改

在 Solaris 下，分别修改一下这些参数文件使得 Greenplum 能够发挥出更好的性能。

### (1) 修改 /etc/system 文件

```
set rlim_fd_cur=65536
set zfs:zfs_arc_max=0x600000000
set pcplusmp:apic_panic_on_nmi=1
set npanicdebug=1
set zfs:zfs_prefetch_disable = 0
```

其中 set rlim\_fd\_cur=65536 修改了文件句柄数，而 set zfs:zfs\_arc\_max=0x600000000 用于限制 ZFS 使用的最大内存。

最后一个参数 zfs:zfs\_prefetch\_disable = 0 是用于优化 ZFS 文件系统的读取性能的，通过这个参数，可以打开 ZFS 文件系统的预读功能。在进行磁盘读取的时候，预先将当前读取的数据块之后的数据块也读取进来。了解磁盘原理的读者应该清楚，在顺序读取文件时使用预读功能可以大大提高磁盘的吞吐，而 Greenplum 中的大部分数据都是连续存储的，很符合这种场景。

### (2) 修改 /etc/project 文件

将 default:3:::: 这一行修改为：

```
default:3:default project:::process.max-filedescriptor=(priv,252144,deny);project.max-sem-ids=(priv,1024,deny);project.max-shm-memory=(priv,21474836480,deny)
```

其中 project.max-shm-memory=(priv,21474836480,deny) 用于限制共享内存使用，需大于 shared\_buffers 总和。

### (3) 修改 /etc/user\_attr 文件，配置 gpadmin 的用户权限

```
gpadmin::::defaultpriv=basic,dtrace_user,dtrace_proc
```

#### (4) 修改 /etc/hosts

这个文件必须包括集群中所有服务器，并且每台服务器的每一张网卡都必须分配一个 hostname，所有服务器的 hosts 文件保持一致。以下的 hosts 文件对应的每台机器上分别有 4 张网卡，每张网卡都有对应的 hostname，mdw、smdw 分别是主备 Master，sdw1、sdw2 分别是两个 Segment：

```
127.0.0.1 localhost loghost
10.1.18.254    greenplum-1  gml  mext1
192.168.0.254          m1    mdw
192.168.1.254          m2
192.168.2.254          m3
192.168.3.254          m4
10.1.18.253    greenplum-2  gm2  smext1
192.168.0.253          sm1    smdw
192.168.1.253          sm2
192.168.2.253          sm3
192.168.3.253          sm4
10.1.18.1     greenplum-3  gm3  sdw1-ext1
192.168.0.1      sdw1-1  sdw1
192.168.1.1      sdw1-2
192.168.2.1      sdw1-3
192.168.3.1      sdw1-4
10.1.18.2     greenplum-4  gm4  sdw2-ext1
192.168.0.2      sdw2-1  sdw2
192.168.1.2      sdw2-2
192.168.2.2      sdw2-3
192.168.3.2      sdw2-4
```

#### (5) 创建用户组

```
groupadd -g 3030 gpadmin
groupadd -g 3040 gpmon
```

#### (6) 创建数据库用户

```
useradd -u 3030 -g gpadmin -d /home/gpadmin -s /bin/bash -m gpadmin
useradd -u 3040 -g gpmon -d /home/gpmon -s /bin/bash -m gpmon
```

#### (7) 配置时钟同步

Network Time Protocol (NTP) 是使计算机时间同步化的一种协议，它可以使计算机对其服务器或时钟源做同步化，它可以提供高精准度的时间校正。在 Linux 中，自带了实现 NTP 的服务，名为 ntpd。通过这个服务，可以将所有 Segment 的机器上的时间都调整成同一个时间，避免因为时间错乱导致 Greenplum 不可用。

修改 /etc/inet/ntp.conf，配置如下：

```
# server 0.0.0.0
server 127.127.1.0
```

```

enable auth monitor
driftfile /var/ntp/ntp.drift
statsdir /var/ntp/ntpstats/
filegen peerstats file peerstats type day enable
filegen loopstats file loopstats type day enable
filegen clockstats file clockstats type day enable
keys /etc/inet/ntp.keys
trustedkey 0
requestkey 0 controlkey

```

在 Master 启动 NTP 服务：

```

touch /var/ntp/ntp.drift
svcadm enable ntp

```

前面已经配置好了 Master 上面的 NTP 服务，接下来修改 Segment 服务器的 NTP 服务，使得 Segment 上的时间与 Master 上一致，配置如下（配置完成后，重启 NTP 服务即可）：

```

server 192.168.1.253
server 192.168.1.254

```

#### (8) 配置 ZFS 存储池

Solaris 的 ZFS 文件系统可以很方便地对数据进行快照，对于数据的备份恢复很有用，是一个很优秀的文件系统。下面将介绍如何创建 ZFS 存储池，示例机器中有 14 张磁盘，创建了两个 raidz 组，每个组有 6 张盘，总共 12 张盘，剩下 2 张盘作为备盘，以备有磁盘损坏时可以马上切换到备盘上。创建命令如下：

```

zpool create -f data \
raidz c1t2d0 c1t3d0 c1t4d0 c1t5d0 c1t6d0 c1t7d0 \
raidz c1t9d0 c1t10d0 c1t11d0 c1t12d0 c1t13d0 c1t14d0 \
spare c1t8d0 c1t15d0

```

在 Zpool 上创建 ZFS 文件系统：

```

zfs create data/mast
zfs create data/p1
zfs create data/p2
zfs create data/m1
zfs create data/m2

```

### 8.2.2 Linux 参数修改

在 Linux 下，对应共享内存、网络、用户限制的参数修改如下。

□ /etc/sysctl.conf，修改共享内存及网络 ipv4 的配置：

```

kernel.sem = 250 64000 100 512
kernel.shmmmax = 500000000
kernel.shmmni = 4096

```

```

kernel.shmall = 4000000000
kernel.sem = 250 64000 100 512
kernel.sysrq = 1
kernel.core_uses_pid = 1
kernel.msgmnb = 65536
kernel.msgmax = 65536
net.ipv4.tcp_syncookies = 1
net.ipv4.ip_forward = 0
net.ipv4.conf.default.accept_source_route = 0
net.ipv4.tcp_tw_recycle=1
net.ipv4.tcp_max_syn_backlog=4096
net.ipv4.conf.all.arp_filter = 1
net.ipv4.conf.default.arp_filter = 1
net.core.netdev_max_backlog=10000
vm.overcommit_memory=2

```

#### □ 修改文件数，进程数的限制：

```

* soft nofile 65536
* hard nofile 65536
* soft nproc 131072
* hard nproc 131072

```

### 8.2.3 系统参数及性能验证

Greenplum 中提供了 gpcheck 脚本对系统参数进行校验，确保配置合理，同时还提供了 gpcheckperf 对机器进行性能测试，包括网络性能测试、磁盘 I/O 吞吐测试、内存带宽测试等。

(1) 使用 gpcheck 脚本对系统参数进行验证，命令如下：

```

[root@test /]#gpcheck -h dw-greenplum-1
20111115:14:24:24:gpcheck:dw-greenplum-1:root-[INFO]:-dedupe hostnames
20111115:14:24:24:gpcheck:dw-greenplum-1:root-[INFO]:-Detected platform:
Generic Solaris Cluster
20111115:14:24:24:gpcheck:dw-greenplum-1:root-[INFO]:-generate data on servers
20111115:14:24:25:gpcheck:dw-greenplum-1:root-[INFO]:-copy data files from
servers
20111115:14:24:25:gpcheck:dw-greenplum-1:root-[INFO]:-delete remote tmp files
20111115:14:24:25:gpcheck:dw-greenplum-1:root-[INFO]:-Using gpcheck config file:
/opt/greenplum/greenplum-db-4.1.1.1/etc/gpcheck.cnf
'default:3::::project.max-sem-ids=(priv,1024,deny);process.max-file-
descriptor=(priv,252144,deny)'
20111115:14:24:25:gpcheck:dw-greenplum-1:root-[INFO]:-gpcheck completing...

```

这个脚本会配置一些需要校验的参数，在 \$GPHOME/etc/gpcheck.cnf 文件中，如果参数配置与 gpcheck.cnf 中的不一致，则会报出一条 ERROR。例如，在上面这个例子中，错误显示在 /etc/project 中没有配置相应的参数。

## (2) 网络测试

利用 Greenplum 自带的 gpcheckperf 工具可以很方便地测试各节点间网络连通情况，下面通过示例来进行介绍。

网络测试的机器名列表如下 (host.1 文件中):

```
[gpadmin@dw-greenplum-3 /export/home/gpadmin]#cat host.1
sdw1-1
sdw2-1
sdw3-1
sdw4-1
sdw5-1
sdw6-1
```

运行 gpcheckperf 进行测试:

```
[gpadmin@dw-greenplum-3 /export/home/gpadmin]#gpcheckperf -d /storagepool/
upload -r N -f host.1
/opt/greenplum/greenplum-db-4.1.1.1/bin/gpcheckperf -d /storagepool/upload -r N
-f host.1

-----
-- NETPERF TEST
-----

=====
== RESULT
=====

Netperf bisection bandwidth test
sdw1-1 -> sdw2-1 = 112.500000
sdw3-1 -> sdw4-1 = 112.540000
sdw5-1 -> sdw6-1 = 111.570000
sdw2-1 -> sdw1-1 = 111.890000
sdw4-1 -> sdw3-1 = 112.160000
sdw6-1 -> sdw5-1 = 111.760000

Summary:
sum = 672.42 MB/sec
min = 111.57 MB/sec
max = 112.54 MB/sec
avg = 112.07 MB/sec
median = 112.16 MB/sec
```

一组，因此当服务器节点为奇数个时，其中一组间测试结果会不理想，需要单独验证，例如下面是 5 台服务器节点的情况：

```
[gpadmin@dw-greenplum-3 /export/home/gpadmin]#cat host.1
sdw1-1
sdw2-1
sdw3-1
```

```

sdw4-1
sdw5-1

[gpadmin@dw-greenplum-3 /export/home/gpadmin]#gpcheckperf -d /storagepool/
upload -r N -f host.1
/opt/greenplum/greenplum-db-4.1.1.1/bin/gpcheckperf -d /storagepool/upload -r N
-f host.1

-----
-- NETPERF TEST
-----

=====
== RESULT
=====
Netperf bisection bandwidth test
sdw1-1 -> sdw2-1 = 112.030000
sdw3-1 -> sdw4-1 = 112.380000
sdw5-1 -> sdw1-1 = 85.720000
sdw2-1 -> sdw1-1 = 104.230000
sdw4-1 -> sdw3-1 = 112.010000
sdw1-1 -> sdw5-1 = 111.420000

Summary:
sum = 637.79 MB/sec
min = 85.72 MB/sec
max = 112.38 MB/sec
avg = 106.30 MB/sec
median = 112.01 MB/sec

```

[Warning] connection between sdw5-1 and sdw1-1 is no good

和 sdw1-1 的网络不是很好，需要单独验证，命令如下：

```

[gpadmin@dw-greenplum-3 /export/home/gpadmin]#gpcheckperf -d /storagepool/
upload -r N -h sdw5-1 -h sdw1-1
/opt/greenplum/greenplum-db-4.1.1.1/bin/gpcheckperf -d /storagepool/upload -r N
-h sdw5-1 -h sdw1-1

-----
-- NETPERF TEST
-----

=====
== RESULT
=====
Netperf bisection bandwidth test
sdw5-1 -> sdw1-1 = 112.260000
sdw1-1 -> sdw5-1 = 112.780000

Summary:

```

```
sum = 225.04 MB/sec
min = 112.26 MB/sec
max = 112.78 MB/sec
avg = 112.52 MB/sec
median = 112.78 MB/sec
```

### (3) 文件系统性能验证

利用 Greenplum 自带的 gpcheckperf 工具可以很方便地测试文件系统的读写性能，示例代码如下：

```
[gpadmin@greenplum-3 /export/home/gpadmin]#gpcheckperf -d /storagepool/
gptempp1 -d /storagepool/gptempp2/ -d /storagepool/gptempp3/ -d /storagepool/
gptempp4/ -d /storagepool/gptempm1/ -d /storagepool/gptempm2/ -d /storagepool/
gptempm3/ -d /storagepool/gptempm4/ -r ds -D -f host.1
/opt/greenplum/greenplum-db-4.1.1.1/bin/gpcheckperf -d /storagepool/gptempp1 -d
/storagepool/gptempp2/ -d /storagepool/gptempp3/ -d /storagepool/gptempp4/ -d
/storagepool/gptempm1/ -d /storagepool/gptempm2/ -d /storagepool/gptempm3/ -d
/storagepool/gptempm4/ -r ds -D -f host.1

-----
-- DISK WRITE TEST
-----

-----
-- DISK READ TEST
-----

-----
-- STREAM TEST
-----

=====
== RESULT
=====

disk write avg time (sec): 93.12
disk write tot bytes: 257603665920
disk write tot bandwidth (MB/s): 2644.07
disk write min bandwidth (MB/s): 489.97 [sdw1-1]
disk write max bandwidth (MB/s): 556.07 [sdw4-1]
-- per host bandwidth --
    disk write bandwidth (MB/s): 530.89 [sdw5-1]
    disk write bandwidth (MB/s): 552.01 [sdw3-1]
    disk write bandwidth (MB/s): 556.07 [sdw4-1]
    disk write bandwidth (MB/s): 489.97 [sdw1-1]
    disk write bandwidth (MB/s): 515.14 [sdw2-1]

disk read avg time (sec): 64.93
disk read tot bytes: 257603665920
disk read tot bandwidth (MB/s): 3789.42
```

```

disk read min bandwidth (MB/s): 727.26 [sdw5-1]
disk read max bandwidth (MB/s): 811.60 [sdw3-1]
-- per host bandwidth --
disk read bandwidth (MB/s): 727.26 [sdw5-1]
disk read bandwidth (MB/s): 811.60 [sdw3-1]
disk read bandwidth (MB/s): 731.38 [sdw4-1]
disk read bandwidth (MB/s): 760.47 [sdw1-1]
disk read bandwidth (MB/s): 758.71 [sdw2-1]

stream tot bandwidth (MB/s): 38014.82
stream min bandwidth (MB/s): 7130.14 [sdw2-1]
stream max bandwidth (MB/s): 8458.92 [sdw3-1]
-- per host bandwidth --
stream bandwidth (MB/s): 7536.51 [sdw5-1]
stream bandwidth (MB/s): 8458.92 [sdw3-1]
stream bandwidth (MB/s): 7398.58 [sdw4-1]
stream bandwidth (MB/s): 7490.66 [sdw1-1]
stream bandwidth (MB/s): 7130.14 [sdw2-1]

```

## 8.3 计算节点分配技巧

Greenplum 软件安装及数据库初始化可参看第 2 章，这里简单介绍一下计算节点分配技巧。

如果配置了 mirror 节点，其会分布在所有 Segment 节点上，在默认情况下同一服务器上主节点对应的所有备节点会分配在一台服务器上（这种方式称为 Grouped Mirror，可参考 13.1.1 节的介绍），这样一旦某一台计算节点死机，所有备节点会在同一台服务器上，致使性能降低 50%，且不利于数据恢复。在初始化数据库时，可以指定 -S 参数，将同一服务器上主节点对应的备节点打散至集群不同服务器上（这种方式称为 Spread Mirror，可参考 13.1.1 节的介绍）。

## 8.4 数据库参数介绍

数据库优化主要从两个方面着手。一方面是提升 CPU、内存、磁盘、网络等集群服务器的硬件配置，另一方面是优化提交到数据库的语句。在这里我们简单介绍一下影响数据库性能的参数及其他常用的配置参数。

### (1) shared\_buffers

数据距离 CPU 越近效率就越高，而离 CPU 由近到远的主要设备有寄存器、CPU cache、RAM、Disk Drives 等。CPU 的寄存器和 cache 是没办法直接优化的，为了避免磁盘访问，只能尽可能将更多有用信息存放在 RAM 中。Greenplum 数据库的 RAM 主要用于存放如下信息。

□ 执行程序。

- 程序数据和堆栈。
- PostgreSQL shared buffer cache。
- kernel disk buffer cache。
- kernel。

因此最大化地保持数据库信息在内存中而不影响其他区域才是最佳的调优方式，但这常常不是一件容易的事情。

PostgreSQL 并非直接在磁盘上进行数据修改，而是将数据读入 shared buffer cache，进而 PostgreSQL 后台进程修改 cache 中的数据块，最终再写回磁盘。后台进程如果在 cache 中找到相关数据，则直接进行操作，如果没找到，则需要从 kernel disk buffer cache 或者磁盘中读入。PostgreSQL 默认的 shared buffer 较小，将此 cache 调大则可降低昂贵的磁盘访问。但是前面提到，修改此参数时一定要避免 swap 发生，因为内存不仅仅用于 shared buffer cache。刚开始可以设置一个较小的值，比如总内存的 15%，然后逐渐增加，过程中监控性能提升和 swap 的情况。

#### ( 2 ) effective\_cache\_size

设置优化器假设磁盘高速缓存的大小用于查询语句的执行计划判断，主要用于判断使用索引的成本，此参数越大越有机会选择索引扫描，越小越倾向于选择顺序扫描，此参数只会影响执行计划的选择。

#### ( 3 ) work\_mem

当 PostgreSQL 对大表进行排序时，数据库会按照此参数指定大小进行分片排序，将中间结果存放在临时文件中，这些中间结果的临时文件最终会再次合并排序，所以增加此参数可以减少临时文件个数进而提升排序效率。当然如果设置过大，会导致 swap 的发生，所以设置此参数时仍然需要谨慎。同样刚开始仍可设置为总内存的 5%。

#### ( 4 ) temp\_buffers

temp\_buffers 即临时缓冲区，用于数据库访问临时表数据，Greenplum 默认值为 1M。可以在单独的 session 中对该参数进行设置，在访问比较大的临时表时，对性能提升有很大帮助。

#### ( 5 ) client\_encoding

设置客户端字符集，默认和数据库 encoding 相同。

#### ( 6 ) client\_min\_messages

控制发送至客户端的信息级别，每个级别包括更低级别的消息，越是低的消息级别发送至客户端的信息越少。例如，warning 级别包括 warning、error、fatal、panic 等级别的信息，而 panic 则只包括 panic 级别的信息。此参数主要用于错误调试。

#### ( 7 ) cpu\_index\_tuple\_cost

设置执行计划评估每一个索引行扫描的 CPU 成本。同类参数还包括 cpu\_operator\_cost、cpu\_tuple\_cost、cursor\_tuple\_fraction。

#### (8) debug\_assertions

打开各种断言检查，这是调试助手。如果遇到了奇怪的问题或数据库崩溃，那么可以将这个参数打开，便于分析错误原因。

#### (9) debug\_print\_parse

当需要查看查询语句的分析树时，可以设置开启此参数，默认为 off。

#### (10) debug\_print\_plan

当需要查看查询语句的执行计划时，可以设置开启此参数，默认为 off。同类参数包括 debug\_print\_prelim\_plan、debug\_print\_rewritten、debug\_print\_slice\_table。

#### (11) default\_tablespace

指定创建对象时的默认表空间。

#### (12) dynamic\_library\_path

在创建函数或者加载命令时，如果未指定目录，将会从这个路径搜索需要的文件。

#### (13) enable\_bitmapscan

表示是否允许位图索引扫描，类似的参数还有 enable\_groupagg、enable\_hashagg、enable\_hashjoin、enable\_indexscan、enable\_mergejoin、enable\_nestloop、enable\_seqscan、enable\_sort、enable\_tidscan。这些参数主要用于控制执行计划。

#### (14) gp\_autostats\_mode

指定触发自动搜集统计信息的条件。当此值为 on\_no\_stats 时，create table as select 会自动搜集统计信息，如果 insert 和 copy 操作的表没有统计信息，也会自动触发统计信息搜集。当此值为 on\_change 时，如果变化量超过 gp\_autostats\_on\_change\_threshold 参数设置的值，会自动触发统计信息搜集。此参数还可设为 none 值，即不自动触发统计信息搜集。

#### (15) gp\_enable\_gpperfmon

要使用 Greenplum Performance Monitor 工具，必须开启此参数。

#### (16) gp\_external\_max\_segs

设置外部表数据扫描可用 segments 数目。

#### (17) gp\_fts\_probe\_interval

设置 ftsprobe 进程对 segment failure 检查的间隔时间。

#### (18) gp\_fts\_probe\_threadcount

设置 ftsprobe 线程数，此参数建议大于等于每台服务器 segments 的数目。

#### (19) gp\_fts\_probe\_timeout

设置 ftsprobe 进程用于标识 segment down 的连接 Segment 的超时时间。

#### (20) gp\_hashjoin\_tuples\_per\_bucket

此参数越小，hash tables 越大，可提升 join 性能，相关参数还有 gp\_interconnect\_hash\_multiplier、gp\_interconnect\_queue\_depth。

( 21 ) `gp_interconnect_setup_timeout`

此参数在负载较大的集群中，应该设置较大的值。

( 22 ) `gp_interconnect_type`

可选值为 TCP、UDP，用于设置连接协议。TCP 最大只允许 1000 个节点实例。

( 23 ) `gp_log_format`

设置服务器日志文件格式。可选值为 csv 和 text。

( 24 ) `gp_max_databases`

设置服务器允许的最大数据库数，相关参数还有 `gp_max_filespaces`、`gp_max_packet_size`、`gp_max_tablespaces`

( 25 ) `gp_resqueue_memory_policy`

此参数允许 none 和 auto 这 2 个值，当设置为 none 时，和 Greenplum 4.1 版本以前的策略一致；而设置为 auto 时，查询内存使用受 `statement_mem` 和资源队列的内存限制，而 `work_mem`、`max_work_mem` 和 `maintenance_work_mem` 这三个参数将失效。

( 26 ) `gp_resqueue_priority_cpus_per_segment`

指定每个 Segment 可用的 CPU 单元

( 27 ) `gp_segment_connect_timeout`

设置网络连接超时时间。

( 28 ) `gp_set_proc_affinity`

设置进程是否绑定至 CPU。

( 29 ) `gp_set_read_only`

设置数据库是否允许写。

( 30 ) `gp_vmem_idle_resource_timeout`

设置数据库会话超时时间，超过此参数值会话将释放系统资源（比如 shared memory）。此参数越小，集群并发度支持越高。

( 31 ) `gp_vmem_protect_limit`

设置服务器中 postgres 进程可用的总内存，建议设置为  $(X * \text{physical\_memory}) / \text{primary\_segments}$ ，其中 X 可设置为 1.0 和 1.5 之间的数字，当 X=1.5 时容易引发 swap，但是会减少因内存不足而失败的查询数。

( 32 ) `log_min_duration_statement`

当查询语句执行时间超过此值时，将会记录日志，相关参数参考 `log_` 开头的所有参数。

( 33 ) `maintenance_work_mem`

设置用于维护的操作可用的内存数，比如 vacuum、create index 等操作将受到这个参数的影响。

( 34 ) `max_appendonly_tables`

最大可并发处理的 appendonly 表的数目。

## (35) max\_connections

最大连接数，Segment 建议设置成 Master 的 5 ~ 10 倍。

## (36) max\_statement\_mem

设置单个查询语句的最大内存限制，相关参数是 max\_work\_mem。

## (37) random\_page\_cost

设置随机扫描的执行计划评估成本，此值越大越倾向于选择顺序扫描，越小越倾向于选择索引扫描。

## (38) search\_path

设置未指定 schema 时，按照这个顺序选择对象的 schema。

## (39) statement\_timeout

设置语句终止执行的超时时间，0 表示永不终止。

更多参数请详细参考 PostgreSQL 及 Greenplum 文档。

## 8.5 数据库集群基准测试

在集群搭建完成正式应用上线之前，对集群整体性能做一个基准测试是很有必要的，比如验证数据库参数设置是否恰当、集群稳定性如何等。如果大家做过 PostgreSQL 数据库的基准测试，相信对 pgbench 是比较熟悉的。这个工具能非常方便地对 PostgreSQL 数据库做一个整体的性能测试，但是 pgbench 是 OLTP-like (TCP-B-like stress test) 类型的测试工具，对 Greenplum 集群这种侧重于 DSS 或数据库类型的应用场景是不恰当的。在这里简单介绍一下 TPC-H (商业智能计算测试) 这个工具，它主要用于 ad-hoc、决策支持等系统的基准测试。

虽然 TPC Council 提供的 TPC-H 这个工具不支持 Greenplum，但是我们发现其支持 TDAT (Teradata)，所以我们对它适当修改即可用于测试我们的 Greenplum 集群。

## (1) TPC-H 下载及安装

从 <http://www.tpc.org/tpch/default.asp> 下载 tpch-queries.tgz 包，解压出来之后准备 Makefile 文件（可以将文件夹中的 makefile.suite 作为模板），Makefile 文件的第 109 行左右有项要适当的修改，如 CC=gcc、DATABASE= TDAT、MACHINE=LINUX、WORKLOAD=TPCH，修改完成之后，运行 make 命令进行编译即可。

## (2) 测试数据生成

我们需要自动创建测试数据。编译之后会生成 dbgen 执行文件，用 TPC-H 即可方便地生成自定义大小的测试数据。和 pgbench 工具一样，TPC-H 可通过参数控制测试数据的大小。在这里我们需要生成 10GB 数据用于测试，采用 \$ ./dbgen -s 10 命令即可，例如下面的代码，生成的 8 个文件分别对应我们测试的 8 个实体表。

```
[dcplatform@de112 dbgen]$ ls -lrth *.tbl
-rw-rw-r-- 1 dcplatform dcplatform 14M Feb  9 08:33 supplier.tbl
```

```
-rw-rw-r-- 1 dcplatform dcplatform 389 Feb 9 08:33 region.tbl
-rw-rw-r-- 1 dcplatform dcplatform 1.2G Feb 9 08:33 partsupp.tbl
-rw-rw-r-- 1 dcplatform dcplatform 233M Feb 9 08:33 part.tbl
-rw-rw-r-- 1 dcplatform dcplatform 1.7G Feb 9 08:33 orders.tbl
-rw-rw-r-- 1 dcplatform dcplatform 2.2K Feb 9 08:33 nation.tbl
-rw-rw-r-- 1 dcplatform dcplatform 7.3G Feb 9 08:33 lineitem.tbl
-rw-rw-r-- 1 dcplatform dcplatform 234M Feb 9 08:33 customer.tbl
```

注意，这里自动生成的文件每行会多余一个'|'分隔符，我们需要通过如下脚本简单处理一下，以生成相应的.csv 文件：

```
for i in `ls *.tbl`; do sed 's/|$//' $i > ${i/tbl/csv}; echo $i; done;
```

### (3) 测试表创建

我们先创建表用于测试数据库：

```
createdb tpch
```

在 tpch 库上，我们创建表实体，如果在 TPC-H specification 文档中有表实体的表结构，我们可以按照如下方式一一进行创建：

```
CREATE TABLE PART (
    P_PARTKEY      BIGINT,
    P_NAME          VARCHAR(55),
    P_MFGR          CHAR(25),
    P_BRAND         CHAR(10),
    P_TYPE          VARCHAR(25),
    P_SIZE          INTEGER,
    P_CONTAINER     CHAR(10),
    P_RETAILPRICE   DECIMAL,
    P_COMMENT        VARCHAR(23)
) WITH (APPENDonly=TRUE, COMPRESSlevel=5)
distributed BY(P_PARTKEY);
```

### (4) 测试数据导入

前面我们已经准备好了 8 个.csv 测试文件，这里直接用 copy 命令分别将 csv 数据文件导入相应的表中：

```
psql tpch -c "COPY part FROM '/home/dcplatform/hy/dbgen/part.csv' WITH
DELIMITER '|'"
```

### (5) 测试脚本准备

TPC-H 提供了 22 个测试 SQL，位于 queries 目录下，这 22 个查询 SQL 中，有些 SQL 包含 correlated subquery（相关子查询），其语法是 Greenplum 不支持的，我们需要适当修改以便符合 Greenplum 特性，发挥最大的性能优势。

修改 SQL 之后，生成基准测试的工作脚本，如下：

```
for r in `seq 1 10`  
do  
    rn=$(((`cat /dev/urandom|od -N3 -An -i` % 10000))  
    DSS_QUERY=queries ./qgen 3 5 -r $rn >> benchmark.sql  
    Done
```

上面脚本会选取 3.sql 和 5.sql，重复 10 次执行，每次选取随机数进行 SQL 拼装，生成的脚本需要简单处理一下再运行。

#### (6) 运行基准测试脚本

数据库类型应用的典型场景是运行特定并发度的 OLAP 型 SQL，所以在这里简单将上面生成的脚本复制 5 份，然后并行执行，运行脚本 benchmark.sh，代码如下：

```
for i in `seq 1 5`  
do  
    /usr/bin/time -f "runtime=%e" psql tpch < benchmark$i.sql |grep "runtime=" &  
done;  
  
for p in `jobs -p`  
do  
    wait $p;  
done;
```

## 8.6 小结

本章简单介绍了 Greenplum 线上环境部署应该关注的几个主要方面，其中服务器硬件选型和服务器系统参数调整尤为重要，读者一定要参考系统管理或 Greenplum 官方技术支持的建议。在正式上线应用前，服务器整体性能及数据集群的基准测试也是必不可少的步骤。数据库参数调整也在一定程度上影响集群性能，所以作为数据库管理员，务必根据应用情况参考 Greenplum 官方文档和 PostgreSQL 文档进行参数调整。

## 第 9 章 数据库管理

本章将介绍数据库管理方面的内容，包括用户、权限控制、资源队列等，还将介绍 Greenplum 自带的一些工具包，方便数据库管理员维护 Greenplum。

### 9.1 用户及权限管理

对于一个完整的数据库系统来说，用户和权限管理都是必不可少的，下面将介绍 Greenplum 的权限管理系统。Greenplum 的权限管理基本上与 PostgreSQL 一样，大家可以参考 PostgreSQL 的官方文档（8.2.3）——第 18 章数据库角色和权限。

#### 9.1.1 Greenplum 数据库逻辑结构

在介绍权限之前，我们先了解下 Greenplum 数据库的逻辑结构组成，如图 9-1 所示。

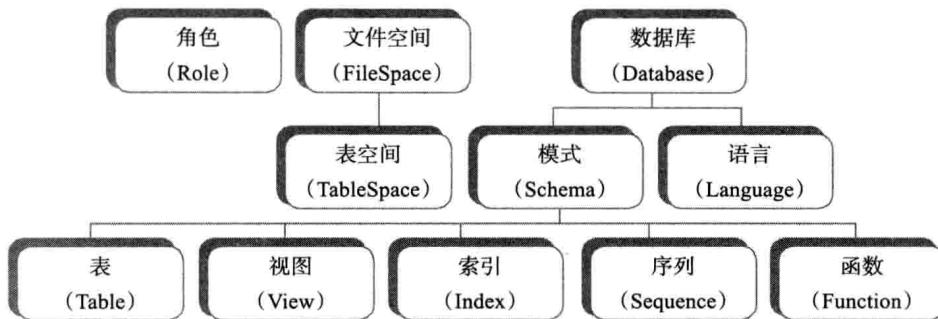


图 9-1 Greenplum 数据库逻辑结构组成

在 Greenplum/PostgreSQL 中，角色（Role）、模式（Schema）、数据库（DataBase）是 3 个不同的概念，不像在 MySQL 中，DataBase 等同于 Schema，在 Oracle 中 Role 等同于 Schema。

在 Greenplum 中：

- 1) 一个 Database 下可以有多个 Schema，一个 Schema 只属于一个 Database。Schema 在 Greenplum 中也叫做 Namespace，不同 Database 之间的 Schema 没有关系，可以重名。
- 2) Language 在使用前必须创建，一个语言只属于一个 Database。
- 3) Table、View、Sequence、Function 必须属于一个 Schema。
- 4) 一个 FileSpace 可以有多个 TableSpace，一个 TableSpace 只属于一个 FileSpace，FileSpace 与 Role 没有关系。
- 5) TableSpace 与 Table 是一对多的关系，一个 Schema 下的表可以分布在多个 TableSpace 下。
- 6) 在图 9-1 中，除了 FileSpace 之外，其他的权限管理都是通过 Role 来实现，在这些层次结构中，用户必须对上一层有访问权限，才能够访问该层的内容。
- 7) Group 与 Role 是一样的概念，在 Greenplum 中，使用 Role 就可以了，虽然 Group 在语法上还可以用，实际上已经被废弃了。

### 9.1.2 Grant 语法

与普通数据库一样，赋权通过 Grant 来实现，Greenplum 中的权限控制相比其他数据库会更细一点。

创建数据库语法为：

```
testDB=# \h create role
Command: CREATE ROLE
Description: define a new database role
Syntax:
CREATE ROLE name [[WITH] option [ ... ]]
where option can be:
    SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEEXTTABLE | NOCREATEEXTTABLE
    [ ( attribute='value'[, ...] ) ]
        where attributes and values are:
            type='readable'|'writable'
            protocol='gpfdist'||'http'||'gphdfs'
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
```

```

| IN ROLE rolename [, ...]
| ROLE rolename [, ...]
| ADMIN rolename [, ...]
| RESOURCE QUEUE queue_name

```

从语法上看，参数配置主要有以下几种，配置可以叠加使用。

- 1) 超级用户 (SUPERUSER|NOSUPERUSER)：最高用户权限，不受资源队列控制，拥有所有的权限，可以对数据库进行任何操作，一般只有 DBA 可以拥有这个权限。
- 2) 创建数据库权限 (CREATEDB|NOCREATEDB)。
- 3) 创建用户权限 (CREATEROLE|NOCREATEROLE)：这个用户可以创建其他用户。
- 4) 登录权限 (LOGIN|NOLOGIN)：可以指定该用户登录的连接数控制。
- 5) 创建外部表权限 (CREATEEXTTABLE|NOCREATEEXTTABLE)：属性配置中也可以对外部表有更细的权限控制，如只读、可写外部表权限等。
- 6) 用户继承 (INHERIT|NOINHERIT)：子用户可以拥有父用户的所有权限。
- 7) 资源队列控制 (RESOURCE QUEUE)：这个在 9.3 节有更详细的描述。
- 8) 密码控制 (ENCRYPTED|UNENCRYPTED)：还可以指定密码以及失效时间。

下面是一个创建用户的例子，testrole1 这个用户可以有登录权限，以及创建数据库与创建用户的权限：

```

testDB=# create role testrole1 createdb createrole login ;
NOTICE:  resource queue required -- using default resource queue "pg_default"
CREATE ROLE

```

Testrole2 这个用户的密码为 test，有效期只到“2013-12-21: 00:00:00”，连接数限制为 5，并且继承了 testrole1 的所有权限。

```

testDB=# create role testrole2 password 'test' valid until '2013-12-21
00:00:00' connection limit 5 inherit in role testrole1 login ;
NOTICE:  resource queue required -- using default resource queue "pg_default"
CREATE ROLE

```

赋权命令 Grant，在语法上与其他数据库类似，其基本语法结构是：

GRANT 权限类型 ON Relation (如表，视图，函数，Schema 等) TO 用户或用户组；

Grant 的语法示例如图 9-2 所示（由于权限比较多，因此语法比较长，这里只截取其中一段，读者可以详细读下语法，通过语法可以了解整个数据库的权限控制）。

```

testDB# \h GRANT
Command: GRANT
Description: define access privileges
Syntax:
GRANT [ { SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER }
        [, ...] | ALL [ PRIVILEGES ] ]
        ON [ TABLE ] tablename [, ...]
        TO { username | GROUP groupname | PUBLIC } [, ...] [ WITH GRANT OPTION ]

```

图 9-2 Grant 语法示例

回收权限的语法 Revoke，基本上与 Grant 是一一对应的，下面举几个例子说明下如何使

用回归权限。

为 testrole1 赋予查询 test\_table1 的权限：

```
testDB=# grant select on test_table1 to testrole1;
GRANT
```

回收这个权限：

```
testDB=# revoke select on test_table1 from testrole1;
REVOKE
```

所有的权限信息都保存在数据字典的 acl 字段中，我们可以通过下面的 SQL 得到哪些数据字典记录了权限控制的信息。

```
testDB=# select c.nspname as schame,a.relname as table,b.attname as column
testDB-#   from pg_class a,pg_attribute b,pg_namespace c
testDB-#  where a.oid=b.attrelid
testDB-#    and a.relnamespace=c.oid
testDB-#    and a.relkind='r'
testDB-#    and c.nspname='pg_catalog'
testDB-#    and b.attname like '%acl%';
schame |      table      | column
-----+-----+-----+
pg_catalog | pg_proc          | proacl
pg_catalog | pg_class          | relacl
pg_catalog | pg_language       | lanacl
pg_catalog | pg_namespace      | nspacl
pg_catalog | pg_database       | datacl
pg_catalog | pg_tablespace     | spcacl
pg_catalog | pg_pltemplate    | tmplacls
pg_catalog | pg_foreign_data_wrapper | fdwacl
pg_catalog | pg_foreign_server | srvacl
(9 rows)
```

## 9.2 登录权限控制

客户端认证是由一个配置文件（通常名为 pg\_hba.conf）控制的，它存放在数据库集群的数据目录中。HBA 是“Host-Based Authentication”的缩写，即基于主机的认证，可以限制登录机器的 IP 段。

在第 2 章的时候简单介绍了 pg\_hba.conf 文件控制用户的登录权限，Greenplum 中的登录权限与 PostgreSQL 中是一样的，在 PostgreSQL 的官方文档中已经讲得很清楚了，建议读者自行浏览 PostgreSQL 8.2.3 中文档——20.1. pg\_hba.conf。

### 9.3 资源队列及并发控制

在 Greenplum 4.x 之后的版本中，加入了资源队列的概念，本节将简单介绍一些与队列控制相关的内容。

资源负载管理是为了限制系统中活动的 SQL 对使用资源的消耗，避免由于 SQL 将系统资源（如 CPU、I/O、内存）耗尽而造成系统缓慢或崩溃。资源队列可以限制活动 SQL 的个数，以及 SQL 各种消耗的大小。每一个用户会对应到一个资源队列中。通过对用户消耗资源的控制，DBA 可以尽量避免系统出现过负载。

资源队列在 Greenplum 中是如何工作的？

资源调度在系统安装的时候已经默认打开了，所有的数据库用户都必须对应一个资源队列，如果配置具体的资源队列，默认的资源队列是 pg\_default。

在 Greenplum 中，资源队列可以实现如下的限制。

- 活动的 SQL 数，在这个资源队列下最多能够运行的 SQL 数。
- 能够消耗的最大内存。
- SQL 优先级，与其他队列的比较，主要限制在 CPU 的资源上。
- SQL 的 cost 值。

在一个资源队列中，当一个新的 SQL 要执行的时候，如果发现队列中已经运行的 SQL 占用的资源总和超过了指定限制，那么这个 SQL 会处于等待状态，等待队列中的其他 SQL 执行完，将资源释放出来后再继续执行，如图 9-3 所示。

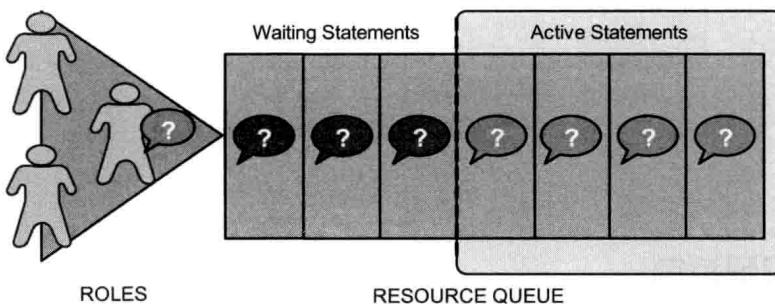


图 9-3 用户与资源队列的限制

下面分别从内存、CPU 优先级、cost 等方面讲解

#### 1. 内存

如果一个资源队列中限制了最大使用内存是 2000MB，同时设置了同时执行的 SQL 数为 10 个，那么每一个 SQL 最多使用的内存就是 200MB。同时，每个 SQL 消耗的内存，不能大于 statement\_mem 参数中设置的内存大小。当一个 SQL 运行的时候，这个内存大小就会被分

配出来，直到 SQL 执行结束后才释放。

## 2. CPU

CPU 优先级管理，每一个资源队列中，都有一个对应的 CPU 优先级。CPU 的优先级有三个等级。

- adhoc，低优先级。
- reporting，高优先级。
- executive，最高优先级。

当系统中有新的 SQL 进入的时候，各个 SQL 消耗 CPU 的资源会根据其优先级重新评估，如图 9-4 所示。

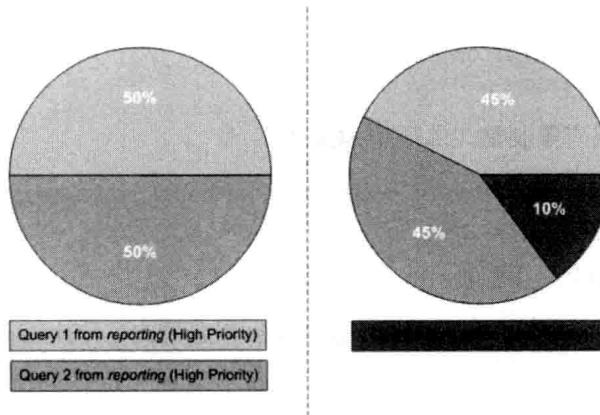


图 9-4 当新的 SQL 执行时，重新分配 CPU 资源

当 executive 优先级的 SQL 进入时，系统会将大部分的资源分配给它，如图 9-5 所示。

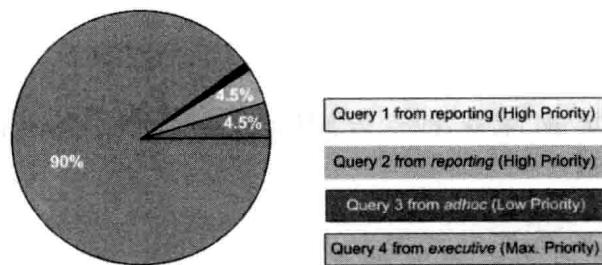


图 9-5 当最高优先级 SQL 执行时的资源分配

## 3. 语法介绍

并不是所有的 SQL 都会被限制在资源队列中，在默认情况下，SELECT、SELECT

INTO、CREATE TABLE AS SELECT 和 DECLARE CURSOR 会被限制在队列中。如果将参数 resource\_select\_only 设置成 off，那么 INSERT、UPDATE、DELETE 语句也会被限制在队列中。

下面介绍如何创建资源队列，以及如何使用资源队列，语法如下：

```
CREATE RESOURCE QUEUE name WITH (queue_attribute=value [, ... ])
where queue_attribute is:
    ACTIVE_STATEMENTS=integer
        [ MAX_COST=float [COST_OVERCOMMIT={TRUE|FALSE}] ]
        [ MIN_COST=float ]
        [ PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX} ]
        [ MEMORY_LIMIT='memory_units' ]
    | MAX_COST=float [ COST_OVERCOMMIT={TRUE|FALSE} ]
        [ ACTIVE_STATEMENTS=integer ]
        [ MIN_COST=float ]
        [ PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX} ]
        [ MEMORY_LIMIT='memory_units' ]
```

□ 创建一个队列只有限制最大的活动 SQL 数：

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

□ 创建一个队列加上内存限制：

```
testDB=# CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20,
testDB (# MEMORY_LIMIT='2000MB');
CREATE QUEUE
```



**注意** 如果只设置了 MEMORY\_LIMIT 和 ACTIVE\_STATEMENTS，那么每个 query 使用的内存是 MEMORY\_LIMIT/ACTIVE\_STATEMENTS，如果设置了 MAX\_COST，那么 query 使用的内存就是 MEMORY\_LIMIT \* (query\_cost / MAX\_COST)，query\_cost 为 query 在执行计划中的 cost 值

如果想对一个 SQL 进行特殊处理，增加其运行时的内存，那么可以设置 statement\_mem 参数，将它调大，例如：

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

□ 设置最大的 cost 值 (cost 值如何计算，请参考第 5 章)：

```
CREATE RESOURCE QUEUE webuser WITH (MAX_COST=10000.0);
```

当 cost 值超出资源限制时，会报错：

```
ERROR: statement requires more resources than resource queue allows
```

在默认情况下 COST\_OVERCOMMIT 参数为 false，这个时候，只要 cost 值超过 MAX\_COST 的 SQL 都会报错。如果 COST\_OVERCOMMIT 设置为 true，在当前资源队列中，没有其他的 SQL 在运行的时候，这个超过 MAX\_COST 的 SQL 也会被执行。

有一个与 MAX\_COST 对应的 MIN\_COST，如果一个 SQL 的 cost 值小于 MIN\_COST，那么这个 SQL 不管资源是什么情况，都会马上被执行。

#### □ 设置 CPU 优先级：

CPU 优先级有 5 个级别：MIN|LOW|MEDIUM|HIGH|MAX，可以根据不同的需求选择：

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

Greenplum 中提供了很多表和视图用于查看资源队列的情况。查看配置情况：

```
testDB=# select * from pg_resqueue_attributes;
rsqname | resname | ressetting | restypid
-----+-----+-----+-----
adhoc  | active_statements | 3 | 1
adhoc  | max_cost | -1 | 2
adhoc  | min_cost | 0 | 3
adhoc  | cost_overcommit | 0 | 4
adhoc  | priority | medium | 5
adhoc  | memory_limit | -1 | 6
```

查看现有的资源队列使用情况：

```
testDB=# select * from pg_resqueue_status;
rsqname | rsqcountlimit | rsqcountvalue | rsqcostlimit | rsqcostvalue | rsqwaiters | rsqholders
-----+-----+-----+-----+-----+-----+-----+
adhoc  | 3 | 0 | -1 |  |  |
0 | 0
pg_default | 20 | 0 | -1 |  |  |
0 | 0
(2 rows)
```

在 gp\_toolkit 中，还有几个视图可用于查看资源队列的使用情况：

```
testDB=# \dv gp_toolkit(gp_resq*
List of relations
Schema | Name | Type | Owner | Storage
-----+-----+-----+-----+-----+
gp_toolkit | gp_resq_activity | view | gpadmin | none
gp_toolkit | gp_resq_activity_by_queue | view | gpadmin | none
gp_toolkit | gp_resq_priority_backend | view | gpadmin | none
gp_toolkit | gp_resq_priority_statement | view | gpadmin | none
gp_toolkit | gp_resq_role | view | gpadmin | none
```

```
gp_toolkit | gp_resqueue_status           | view | gpadmin | none
(6 rows)
```

创建 / 修改用户指定资源队列：

```
testDB=# create role aquery RESOURCE QUEUE adhoc;
testDB=# alter role etl resource queue adhoc;
```

修改资源队列的语法如下，只有超级用户才可以修改资源组，ALTER RESOURCE QUEUE 的语法介绍很明显，这里就不多介绍。修改资源队列语法如下：

```
ALTER RESOURCE QUEUE name WITH ( queue_attribute=value [, ... ] )
where queue_attribute is:
ACTIVE_STATEMENTS=integer
MEMORY_LIMIT='memory_units'
MAX_COST=float
COST_OVERCOMMIT={TRUE|FALSE}
MIN_COST=float
PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX}
```

## 9.4 Greenplum 锁机制

Greenplum 的锁基本上与 PostgreSQL 的锁是一样的，这里将介绍锁的类型、分布式锁的一些概念，以及在 Greenplum 中有可能造成的死锁。

Greenplum 是一个分布式的数据库，其对应的锁机制肯定比普通的数据库还要复杂一些，因为需要统一每一个节点的锁，这个锁的控制是在 Master 上执行的。不幸的是，Greenplum 的锁机制还不够完善，在某些场景上可能会出现一些问题。

与 PostgreSQL 一样，Greenplum 中也对应了八种锁，如表 9-1 所示。

表 9-1 Greenplum 中的锁类型

锁类型	说 明
ACCESS SHARE	SELECT 命令在被引用的表上请求一个这种锁。通常，任何只读取表而不对它进行修改的命令都请求这种锁模式
ROW SHARE	SELECT FOR UPDATE 和 SELECT FOR SHARE 命令在目标表上需要一个这样模式的锁（加上在所有被引用却没有 ACCESS SHARE 的表上的 FOR UPDATE/FOR SHARE 锁）
ROW EXCLUSIVE	UPDATE、DELETE、INSERT 命令自动请求这种锁模式（加上所有其他被引用的表上的 ACCESS SHARE 锁）。通常，这种锁将被任何修改表中数据的查询请求
SHARE UPDATE EXCLUSIVE	VACUUM (不带 FULL 选项)、ANALYZE、CREATE INDEX CONCURRENTLY 请求这样的锁
SHARE	这个模式避免表的并发数据修改，CREATE INDEX (不带 CONCURRENTLY 选项) 语句需要这样的锁模式
SHARE ROW EXCLUSIVE	任何 PostgreSQL 命令都不会自动请求这种锁模式

(续)

锁类型	说 明
EXCLUSIVE	这种模式只允许并发 ACCESS SHARE 锁，也就是说，只有对表的读动作可以和持有这种锁模式的事务并发执行
ACCESS EXCLUSIVE	与所有模式冲突（包括其自身）。这种模式保证其所有者（事务）是可以访问该表的唯一事务。ALTER TABLE、DROP TABLE、TRUNCATE、REINDEX、CLUSTER、VACUUM FULL 命令需要这样的锁。在 LOCK TABLE 命令没有明确声明需要的锁模式时，它是默认锁模式

这几种锁的冲突如表 9-2 所示。

表 9-2 几种锁的冲突情况

Requested Lock Mode	Current Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								×
ROW SHARE							×	×
ROW EXCLUSIVE					×	×	×	×
SHARE UPDATE EXCLUSIVE				×	×	×	×	×
SHARE			×	×		×	×	×
SHARE ROW EXCLUSIVE			×	×	×	×	×	×
EXCLUSIVE		×	×	×	×	×	×	×
ACCESS EXCLUSIVE	×	×	×	×	×	×	×	×

在 4.7.10 节也简单介绍了如何通过 pg\_locks 这个视图查询数据库中锁的信息，在 Greenplum 4.x 中，pg\_locks 比 3.x 版本多了 Segment 的锁信息。

下面通过一个实际的例子来描述锁的过程。

1) 在一个会话中，将表 pg\_class 锁住，如图 9-6 所示。

```
testDB=# begin;
BEGIN
testDB=# lock pg_class in ROW SHARE mode;
LOCK TABLE
```

图 9-6 在一个事务中将表锁住

2) 然后在另外一个会话中，通过 sess\_id（通过 pg\_stat\_activity 获取）查询这个会话产生

的锁，如图 9-7 所示。

locktype	relation	transactionid	pid	mode	gp_segment_id	gpid
transactionid	1259	1086539	1177	ExclusiveLock	-1	
relation	1259		1177	RowShareLock	-1	
transactionid	1259	1099518	26801	RowShareLock	0	
relation	1259		26803	ExclusiveLock	1	0
transactionid	1259	1099513	26803	ExclusiveLock	1	
transactionid	1259	1099483	30858	ExclusiveLock	2	
relation	1259		30858	RowShareLock	2	
relation	1259	1099488	30860	ExclusiveLock	3	
relation	1259		30860	RowShareLock	3	
transactionid	1259	1099505	5753	RowShareLock	4	
relation	1259		5753	RowShareLock	4	
transactionid	1259	1099483	5755	ExclusiveLock	5	
relation	1259		5755	RowShareLock	5	
(14 rows)						

图 9-7 查询 pg\_locks 视图

locktype 表示锁住的内容，主要是 transactionid 和 relation。在 Greenplum 中，Master 到 Segment 的连接就是一个 transaction，只要一连接就会有这个锁信息，relation 字段中的 1259 对应 pg\_class 的 oid 字段。gpid 不等于 -1 就是代表每一个 Segment 的锁信息。

通过 lock 命令可以显式地将表锁住，前面已经试过将 pg\_class 锁住了，语法如下：

```
Command:      LOCK
Description:  lock a table
Syntax:
LOCK [ TABLE ] name [, ...] [ IN lockmode MODE ] [ NOWAIT ]
where lockmode is one of:

ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

## 9.5 数据目录结构

本节将介绍 Greenplum 的数据目录结构，文件在数据库中的存储特点，以及我们在维护 Greenplum 中要注意的一些问题。

如图 9-8 所示是 Greenplum 主节点（Master）的数据目录结构。

drwx-----	6	gpadmin	gpadmin	4096	Jan 8 23:08	base
drwx-----	2	gpadmin	gpadmin	4096	Jan 8 23:08	global
-r-----	1	gpadmin	gpadmin	128	Dec 11 19:48	gp_dbid
drwxrwxr-x	5	gpadmin	gpadmin	4096	Dec 11 19:43	gp_segmentid
drwxrwxr-x	2	gpadmin	gpadmin	4096	Dec 11 19:42	pg_canceled_transactions
drwxrwxr-x	2	gpadmin	gpadmin	4096	Dec 11 19:42	pg_cdc_log
drwxrwxr-x	2	gpadmin	gpadmin	4096	Dec 11 19:42	pg_cdc_xlog
drwxrwxr-x	2	gpadmin	gpadmin	4096	Dec 11 19:42	pg_cdc_xlog_index
drwxrwxr-x	2	gpadmin	gpadmin	4096	Dec 11 19:42	pg_cdc_xlog_map
drwxrwxr-x	1	gpadmin	gpadmin	4096	Dec 11 19:42	pg_cdc_xlog_min_lsn
drwxrwxr-x	1	gpadmin	gpadmin	4072	Dec 11 21:04	pg_hba.conf
-rw-----	1	gpadmin	gpadmin	1636	Dec 11 19:42	pg_ident.conf
drwxrwxr-x	2	gpadmin	gpadmin	4096	Jan 25 00:00	pg_logical
drwxrwxr-x	4	gpadmin	gpadmin	4096	Dec 11 19:42	pg_notify
drwxrwxr-x	2	gpadmin	gpadmin	4096	Dec 11 19:42	pg_notify_trigger
drwxrwxr-x	2	gpadmin	gpadmin	4096	Dec 11 19:42	pg_notify_trigger_index
drwxrwxr-x	2	gpadmin	gpadmin	4096	Dec 11 19:42	pg_notify_trigger_map
drwxrwxr-x	2	gpadmin	gpadmin	4096	Dec 11 19:42	pg_notify_trigger_min_lsn
drwxrwxr-x	2	gpadmin	gpadmin	4096	Dec 11 19:42	pg_notify_trigger_min_lsn_index
drwxrwxr-x	1	gpadmin	gpadmin	4	Dec 11 19:42	PG_VERSION
drwxrwxr-x	3	gpadmin	gpadmin	4096	Jan 8 23:08	postgresql.conf
-rw-----	1	gpadmin	gpadmin	19047	Dec 11 19:43	postgresql.opts
-rw-----	1	gpadmin	gpadmin	189	Dec 11 20:11	postmaster.opts
-rw-----	1	gpadmin	gpadmin	64	Dec 11 20:11	postmaster.pid

图 9-8 数据目录结构

其中：

- base 是数据目录，每个数据库在这个目录下，会有一个对应的文件夹。
- global 是每一个数据库公用的数据目录。
- gpperfmon 监控数据库性能时，存放监控数据的地方。
- pg\_changetracking 是 Segment 之间主备同步用到的一些原数据信息保存的地方。
- pg\_clog 是记录数据库事务信息的地方，保存了每一个事务 id 的状态，这个非常重要，不能丢失，一旦丢失，整个数据库就基本上不可用了。
- pg\_log 是数据库的日志信息。
- pg\_twophase 是二阶段提交的事务信息（关于二阶段提交的内容可参阅第 7 章中的介绍）
- pg\_xlog 是数据库重写日志保存的地方，其中每个文件固定大小为 64MB，并不断重复使用。
- gp\_dbid 记录这个数据库的 dbid 以及它对应的 mirror 节点的 dbid。
- pg\_hba.conf 是访问权限控制文件。
- pg\_ident.conf 是 Ident 映射文件。
- PG\_VERSION 是 PostgreSQL 的版本号。
- postgresql.conf 是参数配置文件。
- postmaster.opts 是启动该数据库的 pg\_ctl 命令。
- postmaster.pid 是该数据库的进程号和数据目录信息。

其中数据目录 base 下面的文件夹结构为：

```
#ls
1 10890 10891 16992 285346
```

其中，一个文件夹代表一个数据库，文件夹的名字就是数据库的 oid，可以通过 pg\_database 查询其对应关系：

```
testDB=# select oid,datname from pg_database order by oid;
oid      | datname
-----+-----
1       | template1
10890   | template0
10891   | postgres
16992   | testDB
285346  | gpperfmon
(6 rows)
```

接下来这一小节的内容将介绍 Greenplum 的数据文件是如何保存的（不涉及文件中具体的文件存储格式，主要是表的存储文件分布）。

## 9.6 数据文件存储分布

下面分别介绍表、索引、序列的文件存储分布（外部表和视图没有实际的数据，所以都没有数据文件生成），每一种类型对应数据库里面哪些文件。

### (1) 表

最普通的堆表只有一个数据文件，如果表中有大字段，那么这个表会多两个数据文件，分别是 toast 表和 toast 表索引。

怎样才算是大字段呢？下面通过一个实验来说明下，首先创建下面这些表：

```
create table test_varchar2037(id int,values varchar(2037)) distributed by (id);
create table test_varchar2036(id int,values varchar(2036)) distributed by (id);
create table test_text(id int,values text) distributed by (id);
```

接着查询 pg\_class 的结构：

```
testDB=# select oid,relname,reltoastrelid from pg_class where relname like
'test_varchar%' or relname = 'test_text';
      oid |          relname           | reltoastrelid
-----+-----+-----+
 316625 | test_varchar2037       |        316627
 316650 | test_varchar2036       |          0
 316696 | test_text              |        316698
(4 rows)
```

可以看出临界点 varchar 的大小就是 2036，当超过 2036 时，数据库就会为该表创建一个 toast 表，text 类型本来就是保存大文本字段的，所以一定有 toast 表。

如果原来表中最大的一个字段是 varchar(2036)，通过修改表结构将其改为 varchar(2037)，那么原来没有 toast 表的也会增加 toast 表。

```
testDB=# create table test_varchar2036_to_2037(id int,values varchar(2036))
distributed by (id);
CREATE TABLE
testDB=# select oid,relname,reltoastrelid from pg_class where relname ='test_
varchar2036_to_2037';
      oid |          relname           | reltoastrelid
-----+-----+-----+
 316743 | test_varchar2036_to_2037 |          0
(1 row)
testDB=# alter table test_varchar2036_to_2037    alter   values type
varchar(2037);
ALTER TABLE
testDB=# select oid,relname,reltoastrelid from pg_class where relname ='test_
varchar2036_to_2037';
      oid |          relname           | reltoastrelid
-----+-----+-----+
 316743 | test_varchar2036_to_2037 |        316778
(1 row)
```

默认 toast 表的名字为 pg\_toast\_+原表的 relfilenode，索引为 pg\_toast\_+原表的 relfilenode+\_index，如下（316696 是表 test\_text 的 oid）：

```
testDB=# select oid,relname from pg_class where relname like '%316696%';
      oid      |    relname
-----+-----
 316698 | pg_toast_316696
 316699 | pg_toast_316696_index
(2 rows)
```



**注意** 超大尺寸字段存储技术（TOAST，The Oversized-Attribute Storage Technique），详见 PostgreSQL 8.23 官方文档——52.2. TOAST。

如果是 Appendonly 表，那么会多 4 个文件——pg\_aoseg、pg\_aovisimap 表及其索引对应的数据文件，这个在第 6 章中介绍了。

### （2）索引

索引文件只有一个，可以通过索引名在 pg\_class 中查找。索引在创建的时候就分配了 32KB 的存储空间，等到这 32KB 用完了才开始扩大。

```
testDB=# create index test_varchar2036_idx on test_varchar2036(id);
CREATE INDEX
testDB=# select oid,relname,relfilenode from pg_class where relname = 'test_varchar2036';
      oid      |    relname      |    relfilenode
-----+-----+-----
 316480 | test_varchar2036 |        316480
(1 row)
```

可以在对应的数据目录中查到这个文件：

```
#ls -l 316606
-rw----- 1 gpadmin gpadmin 32768 Oct  6 21:26 316606
```

### （3）序列

序列（Sequence）与索引一样，也只有一个数据文件，在 pg\_class 中对应一条记录， relfilenode 字段就是文件名，这些不再说明。

如果一个表中有字段是 serial 类型的，即一个递增序列，那么这个表会自动创建一个序列，也就会多一个数据文件。

## 9.7 表空间管理

在 Greenplum 4.0 之后的版本中，Greenplum 加入了文件空间（Filespace）和表空间

(Tablespace) 的概念，在以前的版本中，所有数据都是放在 base 目录下的，每一个数据库一个文件夹（文件夹的名字为数据库的 oid）。

在系统初始化的时候，只有两个表空间 pg\_default 和 pg\_global，这两个表空间都在 pg\_system 这个文件系统下：

```
testDB=# select a.spcname,b.fsname from pg_tablespace a ,pg_filespace b where
spcfsoid=b.oid;
      spcname   | fsname
-----+-----
 pg_default | pg_system
 pg_global  | pg_system
(2 rows)
```

pg\_global 表空间保存的是各个数据库之间的通用信息，在 data\_directory/global 目录下，pg\_default 表空间保存的是每个数据库特有的数据，包括数据字典及用户数据。其中，每一个数据库都会有一个对应的数据目录，如果数据库中的表比较多，或者表分区比较多（每一个分区都相当于一张表），那么在一个目录下就会有非常多的文件，文件数太多会给文件系统带来非常大的压力。因此，当文件数增长到一定程度的时候，就必须使用表空间，将数据存放到多个目录下。

在 Greenplum 中，表空间必须创建在文件空间上，默认只有 pg\_system 一个文件空间，在这个文件空间上不能再创建其他的表空间了。下面介绍如何创建多一个文件空间。

首先，为每一个表空间创建系统目录，在 MASTER 和每一个 SEGMENT 上都要创建：

```
MASTER: /home/gpadmin/gpdata/master_fspc
Primary Segment: /home/gpadmin/gpdata/primary_fspc
Mirror Segment: /home/gpadmin/gpdata/mirror_fspc
```

然后运行 gpfilespace 脚本，根据提示输入文件系统的名字和每个 segment 的目录。

```
$ gpfilespace
...
> fs_test
Checking your configuration:
Your system has 1 hosts with 0 primary and 0 mirror segments per host.
Your system has 3 hosts with 2 primary and 2 mirror segments per host.
Configuring hosts: [mdw]
Configuring hosts: [sdw3, sdw1, sdw2]
Please specify 2 locations for the primary segments, one per line:
primary location 1> /home/gpadmin/gpdata/primary_fspc
primary location 2> /home/gpadmin/gpdata/primary_fspc

Please specify 2 locations for the mirror segments, one per line:
mirror location 1> /home/gpadmin/gpdata/mirror_fspc
mirror location 2> /home/gpadmin/gpdata/mirror_fspc

Enter a file system location for the master
```

```
master location> /home/gpadmin/gpdata/master_fspc
Creating configuration file... [created]
```

```
To add this filesystem to the database please run the command:
gpfspace --config /home/gpadmin/gpdata/gpfspace_config_20120603_210559
```

之后，会生成一个 `gp_filespace_config` 文件，这个文件中保存了每个 Segment 对应的数据目录（也可以开始手动编辑这个文件）：

```
$ cat /home/gpadmin/gpdata/gpfspace_config_20120603_210559
filespace:fs_test
mdw:1:/home/gpadmin/gpdata/master_fspc/gpseg-1
sdw3:14:/home/gpadmin/gpdata/master_fspc/gpseg-1
sdw3:6:/home/gpadmin/gpdata/primary_fspc/gpseg4
...
...
```

运行 `gpfspace` 创建文件系统：

```
$ gpfspace --config /home/gpadmin/gpdata/gpfspace_config_20120603_210559
```

创建好文件系统之后，我们就可以在上面创建表空间了：

```
testDB=# create tablespace tbs_test1 filesystem fs_test;
CREATE TABLESPACE
```

在表空间创建成功之后，在建表的时候就可以使用参数，指定表建在哪个表空间下面：

```
testDB=# create table test_01 (id int)
testDB-# tablespace tbs_test1 distributed by (id);
CREATE TABLE
```

在 `filespace` 的目录下，就可以看到刚刚创建的表的数据文件了：

```
$ tree
.
|-- gpseg0                         --Segment 名字
|   '-- 295089                      -- 表空间的 oid
|       '-- 16992                    -- 数据库的 oid
|           '-- 295101                -- 数据文件名，对应 pg_class 的 relfilenode
|               '-- PG_VERSION
`-- gpseg1
    '-- 295089
        '-- 16992
            '-- 295101
                '-- PG_VERSION
```

默认都是在 `default_tablespace`（参数）下面建表，这个参数默认是 `pg_default` 表空间，我们可以在配置文件 `postgresql.conf` 中修改这个参数，或者在当前窗口下修改这个参数。

```
set default_tablespace='tbs_test1';
```

这样，建的表就在这个表空间下了。我们还可以为每一个用户设置不同的表空间，同时必须给表空间赋权：

```
testDB=# alter role etl set default_tablespace='tbs_test2';
ALTER ROLE
testDB=# grant ALL on tablespace tbs_test2 to etl;
GRANT
```

更换表空间，将表 hello1 从表空间 tbs\_test2 更换到 tbs\_test3：

```
testDB=# alter table hello1 set tablespace tbs_test3;
ALTER TABLE
```



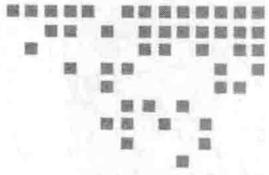
**注意** 更换表空间在 Greenplum 中其实是将数据复制了一份，而不是移动文件，所以对比较大的文件，性能比较差，会造成大量的 IO。

## 9.8 小结

本章介绍了数据库管理的相关内容，以及一些基本的数据库管理工具。

先介绍了数据库用户权限、登录权限管理；之后介绍了数据库的锁机制及 Greenplum 4.x 版本之后加入的资源队列控制新特性；最后介绍了 Greenplum 的数据目录结构以及表空间的使用和管理。

本章内容中的用户及权限管理部分都是与 PostgreSQL 中一样的，读者在阅读本章的同时，也多看看 PostgreSQL 的文档，可以更好地掌握本章内容。



第 10 章

## 数据库监控及调优

Chapter 10

要想将 Greenplum 应用在生产环境上，必要的数据监控是不可少的，因为通过监控及报警系统可以使 DBA 及早发现问题并及时处理。本章将简单介绍 Greenplum 数据的监控及维护，包括 Linux 下的几个通用的监控工具。

### 10.1 Linux 监控工具介绍

对于操作系统来说，要时刻监控机器的负载情况，避免因机器太忙导致数据库发生错误。对于 Linux 操作系统来说，其提供了很多的工具来监控服务器的性能状况。所有机器，无非就是通过磁盘 IO、网络、CPU 及内存等来分析自身的性能状况，下面简单介绍一下 Linux 下对应的监控工具。

#### 10.1.1 监控磁盘

监控磁盘的工具主要有 iostat，一般运行的命令为 iostat -x 1，如图 10-1 所示。重点观察最后一个参数 %util，它表示磁盘使用时间的占比，当这个数据是 100% 时，就说明磁盘已经很忙碌了。然后再看下 rsec/s、wsec/s 等参数，观察磁盘的吞吐。

对于 Solaris 来说，还可以使用 zpool iostat 查看磁盘的繁忙情况（如图 10-2 所示）。

#iostat rx 1											
avg-cpu:			%user	%nice	%System	%iowait	%steal	%idle			
			1.49	0.00	1.36	0.31	0.00	96.84			
Device:											
sda	0.22	6.74	0.62	3.93	66.95	85.34	33.46	0.17	36.56	5.08	2.31
sda1	0.00	0.00	0.00	0.00	0.00	0.02	4.30	0.00	33.10	13.74	0.01
sda2	0.01	0.97	0.05	0.50	1.49	11.78	24.30	0.01	24.68	9.45	0.52
sda3	0.01	1.26	0.05	0.39	1.39	12.77	31.83	0.01	22.45	9.72	0.43
sda4	0.00	0.00	0.00	0.00	0.00	0.00	2.00	0.00	10.50	10.50	0.00
sda5	0.00	0.60	0.00	1.59	0.01	16.06	11.11	0.01	35.81	14.22	0.96
sda6	0.00	0.11	0.01	0.21	0.56	2.61	14.22	0.00	16.84	14.22	0.72
sda7	0.04	0.06	0.01	0.00	0.44	0.49	78.59	0.00	18.77	3.96	0.00
sda8	0.16	3.80	0.50	1.33	63.06	41.02	56.70	0.09	47.77	6.86	1.26
sdb	0.29	15.39	0.25	0.43	89.34	126.50	319.55	0.08	121.72	3.30	0.22
sdb1	0.00	0.00	0.00	0.00	0.00	0.00	17.17	0.00	19.08	8.84	0.00
sdb2	0.00	0.00	0.00	0.00	0.00	0.00	18.5	0.00	3.85	2.00	0.00
sdb3	0.00	0.00	0.00	0.00	0.00	0.00	10.80	0.00	7.35	6.75	0.00
sdb4	0.00	0.00	0.00	0.00	0.00	0.00	6.74	0.00	18.26	18.26	0.00
sdb5	0.00	0.00	0.00	0.00	0.00	0.00	17.86	0.00	2.71	2.03	0.00
sdb6	0.29	15.38	0.25	0.42	89.33	126.44	321.28	0.08	122.31	3.31	0.22
sdc	0.57	22.45	0.54	0.47	175.82	183.35	354.22	0.15	150.68	3.77	0.38
sdc1	0.37	22.43	0.34	0.47	173.82	183.35	354.23	0.15	150.68	3.77	0.38

图 10-1 使用 iostat 查询磁盘运行情况

pool	capacity		operations		bandwidth	
	used	avail	read	write	read	write
storagepool	2.84T	1.69T	2	1	256K	99.8K
storagepool	2.84T	1.69T	0	0	0	0
storagepool	2.84T	1.69T	0	0	0	0
storagepool	2.84T	1.69T	0	0	0	0
storagepool	2.84T	1.69T	0	0	0	0
sc	0	0	0	0	0	0

图 10-2 在 Solaris 下可以使用 zpool iostat 查询 ZFS 文件系统的 IO 吞吐



在 Solaris 下，还有一个很强大的动态跟踪的工具——Dtrace，有兴趣的读者可以去了解下这个工具，这个工具可以分析出当前文件系统中哪一个文件正在被访问，访问的次数，已经被哪一个进程访问等，非常强大。除了 I/O 以外，几乎所有的操作系统相关的数据都可以通过 Dtrace 跟踪。Dtrace 是一个非常强大的调试分析工具，可以快速找到系统的性能瓶颈。

### 10.1.2 监控网络

监控网络一般可以通过 ifstat 命令来实现，这个命令可以列出每一个网卡的吞吐情况，如图 10-3 所示。

也可以通过 sar 命令监控网络流量（sar -n DEV 1 100），如图 10-4 所示。

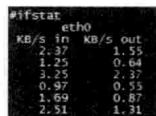


图 10-3 使用 ifstat 监控网络

IFACE	rxpck/s	txpck/s	rxbyt/s	txbyt/s	rxcmp/s	txcmp/s	rxmcst/s
06:25:26 PM eth0	2.02	2.02	876.77	876.77	0.00	0.00	0.00
06:25:26 PM eth0	10.10	6.06	1880.81	1133.33	0.00	0.00	0.00
06:25:26 PM eth1	0.00	0.00	0.00	0.00	0.00	0.00	0.00

图 10-4 使用 sar 查询网络流量



这两个工具不是系统自带的，需要重新安装。

### 10.1.3 监控 CPU

使用 mpstat 可以看每个 CPU 核心的情况，或者 CPU 整体的使用情况，主要看 CPU 空闲率 (idle)，如图 10-5 所示。

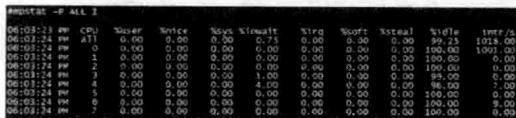


图 10-5 使用 mpstat 查询 CPU 消耗

#### 10.1.4 监控内存

Greenplum 在执行表广播、Hash join 等数据库操作时，会占用大量的内存，所以内存的监控显得格外重要。在 Linux 下，可以使用 free 和 vmstat 来查看内存的使用。

( 1 ) free

使用 free 查看内存使用情况如图 10-6 所示。



图 10-6 通过 free 查看内存使用情况

主要关注 buffers/cache 的 free 是否已经快没有了，如果是，那就说明内存使用已经很吃紧了。还有就是看 swap 是否已经开始使用了，如果 swap 开始使用，说明操作系统的内存已经不足。

## ( 2 ) vmstat

使用 vmstat 查看内存消耗情况如图 10-7 所示。

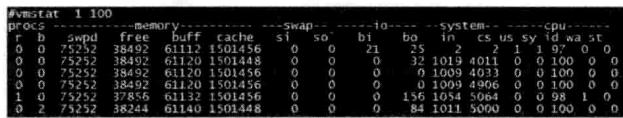


图 10-7 通过 vmstat 查看内存消耗情况

同样的，使用 vmstat 也可以监控内存的使用情况，其中几个参数如下。

- ❑ swpd: 虚拟内存使用情况, 单位: KB。
  - ❑ free: 空闲的内存, 单位 KB。
  - ❑ buff: 用作缓存的内存数, 单位: KB。

在内存使用过程中，要时刻监控空闲内存的大小，如果发现内存占用过高，尤其是 swap 开始使用时（swap 开始使用，会导致 Greenplum 性能大减），要及时发现并处理。

除此之外，要了解系统的参数还可以通过 top ( solaris 下自带 prstat)、sar、dstat 等工具。通过 dstat 查看系统各项信息的情况如图 10-8 所示。

total-cpu-usage				dsk/total				net/total				paging				system			
usr	sys	idl	wai	hiq	sqn	read	writ	recv	send	in	out	int	csw	in	out	int	csw		
1	0	0	0	0	0	332k	395k	0	0	2238	2508	1040	4564	0	0	0	0		
0	0	100	0	0	0	0	0	0	0	23508	15830	0	0	0	0	0	0		
0	0	100	0	0	0	0	0	0	0	16878	9966	0	0	1006	4126	0	0		
0	0	99	1	0	0	0	0	408k	12788	7058	0	0	1055	4020	0	0			
3	0	96	1	0	0	0	0	344k	54088	65578	0	0	1030	4011	0	0			
0	0	100	0	0	0	0	0	0	0	23818	10606	0	0	1009	3890	0	0		

图 10-8 通过 dstat 查看系统各项信息

## 10.2 安装 Performance Monitor

Greenplum 中自带了一个工具——Performance Monitor (在第 3 章的时候已使用，在 Greenplum 中简称为 gpperfmon)，使用这个工具可以收集当前数据库中 SQL 的信息和系统性能的指标，以图形的方式展现出来，还会收集 Master 和所有 Segment 的信息。其架构图如图 10-9 所示。

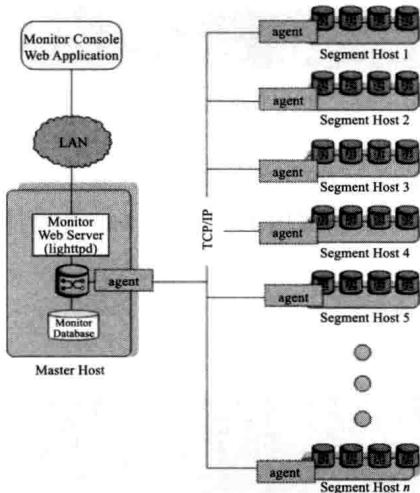


图 10-9 Performance Monitor 架构图

接下来先介绍如何安装 gpperfmon。安装分两个步骤如下。

- 创建监控数据库，数据库名为 gpperfmon，并且打开 gp\_enable\_gpperfmon 参数。
- 创建 gmon 用户。

□ 安装并配置监控控制台 (Greenplum Performance Monitor Console)。

先使用 gpperfmon\_install 命令打开 gp\_enable\_gpperfmon 参数 (使用 Performance Monitor 的时候需要修改 Master 和 Segment 的 postgresql.conf，将参数 gp\_enable\_gpperfmon 值设为 on，这个参数需要重启数据库才生效)。

```

$ gpperfmon_install --enable --password gpperfmon --port 2345
[INFO]:-PGPORT=2345 psql -f /opt/greenplum/greenplum-db/.lib/gpperfmon/
gpperfmon3.sql template1 >& /dev/null
[INFO]:-PGPORT=2345 psql -f /opt/greenplum/greenplum-db/.lib/gpperfmon/
gpperfmon4.sql gpperfmon >& /dev/null
[INFO]:-PGPORT=2345 psql -f /opt/greenplum/greenplum-db/.lib/gpperfmon/
gpperfmon41.sql gpperfmon >& /dev/null
[INFO]:-PGPORT=2345 psql -f /opt/greenplum/greenplum-db/.lib/gpperfmon/
gpperfmonC.sql template1 >& /dev/null
[INFO]:-PGPORT=2345 psql template1 -c "DROP ROLE IF EXISTS gpmon" >& /dev/null
[INFO]:-PGPORT=2345 psql template1 -c "CREATE ROLE gpmon WITH SUPERUSER
CREATEDB LOGIN ENCRYPTED PASSWORD 'gpperfmon'" >& /dev/null
[INFO]:-echo "local gpperfmon gpmon md5" >> /home/gpadmin/gpdata/gpmaster/
gpseg-1/pg_hba.conf
[INFO]:-echo "host all gpmon 127.0.0.1/28 md5" >> /home/gpadmin/gpdata/
gpmaster/gpseg-1/pg_hba.conf
[INFO]:-touch /home/gpadmin/.pgpass >& /dev/null
[INFO]:-mv -f /home/gpadmin/.pgpass /home/gpadmin/.pgpass.1336914387 >& /dev/
null
[INFO]:-echo "*:2345:gpperfmon:gpmon:gpperfmon" >> /home/gpadmin/.pgpass
[INFO]:-cat /home/gpadmin/.pgpass.1336914387 >> /home/gpadmin/.pgpass
[INFO]:-chmod 0600 /home/gpadmin/.pgpass >& /dev/null
[INFO]:-PGPORT=2345 gpconfig -c gp_enable_gpperfmon -v on >& /dev/null
[INFO]:-PGPORT=2345 gpconfig -c gpperfmon_port -v 8888 >& /dev/null
[INFO]:-PGPORT=2345 gpconfig -c gp_external_enable_exec -v on --masteronly >&
/dev/null
[INFO]:-gpperfmon will be enabled after a full restart of GPDB

```

在 `gpperfmon_install` 运行过程中，执行了很多 SQL 文件（如 `gpperfmon3.sql`），这些文件都在 `$GPHOME/lib/gpperfmon/` 目录下，主要是创建表和视图等。在完成创建后，要使配置生效，必须重启数据库 (`gpstop -afr`)。

接下来安装监控控制台（Greenplum Performance Monitor Console）。

安装包是 `greenplum-perfmon-web-4.1.1.3-build-4-RHEL5-x86_64.zip`。

下面是具体的安装过程。

### (1) 解压安装包

```

$ unzip greenplum-perfmon-web-4.1.1.3-build-4-RHEL5-x86_64.zip
Archive:  greenplum-perfmon-web-4.1.1.3-build-4-RHEL5-x86_64.zip
  inflating: greenplum-perfmon-web-4.1.1.3-build-4-RHEL5-x86_64.bin

```

### (2) 根据提示安装

```

$ ./greenplum-perfmon-web-4.1.1.3-build-4-RHEL5-x86_64.bin

```

安装后的目录如图 10-10 所示。

接下来配置跟启动控制台，首先在配置文件 `~/.bash_profile` 中增加以下命令，使 `gpperfmon` 的环境变量生效。

```

source /home/gpadmin/opt/gpperfmon/gpperfmon_path.sh

```



图 10-10 gpperfmon 安装后目录

接着运行 bin/gpperfmon --setup 进行安装：

```
An instance name is used by the Greenplum Performance monitor as
a way to uniquely identify a Greenplum Database that has the monitoring
components installed and configured. This name is also used to control
specific instances of the Greenplum Performance monitors web UI. Instance
names cannot contain spaces.

Please enter a new instance name. Entering an existing
instance name will reconfigure that instance:
> testGPPerf

The web component of the Greenplum Performance Monitor can connect to a
monitor database on a remote Greenplum Database.

Is the master host for the Greenplum Database remote? y|n (default=n):
> y
what is the hostname of the master:
> 10.20.151.7
what port does the Greenplum database use? (default=2345):
>
```

图 10-11 gpperfmon 配置过程

配置好后，运行 gpperfmon --start testGPPerf 启动 Web UI：

```
$ bin/gpperfmon --start testGPPerf
Starting instance testGPPerf... Done.
```



**注意** 启动时要修改 pg\_hba.conf，加入 gpmmon 用户的登录权限。

接下来就可以在本地使用浏览器连接了，输入网址：

<https://10.20.151.7:28080/>

输入用户名和密码，如图 10-12 所示。



图 10-12 输入 gpperfmon Web UI 登录信息

登录之后显示如图 10-13 所示的界面，其中左侧是 System Summary，展示系统层面的性能摘要信息，右侧是数据库的摘要信息。

单击第二个页面（SYSTEM METRICS），展示了系统相关的性能信息，在右上角的 Duration 框可以修改时间，查看最近不同时间段的性能变化情况，如图 10-14 所示。

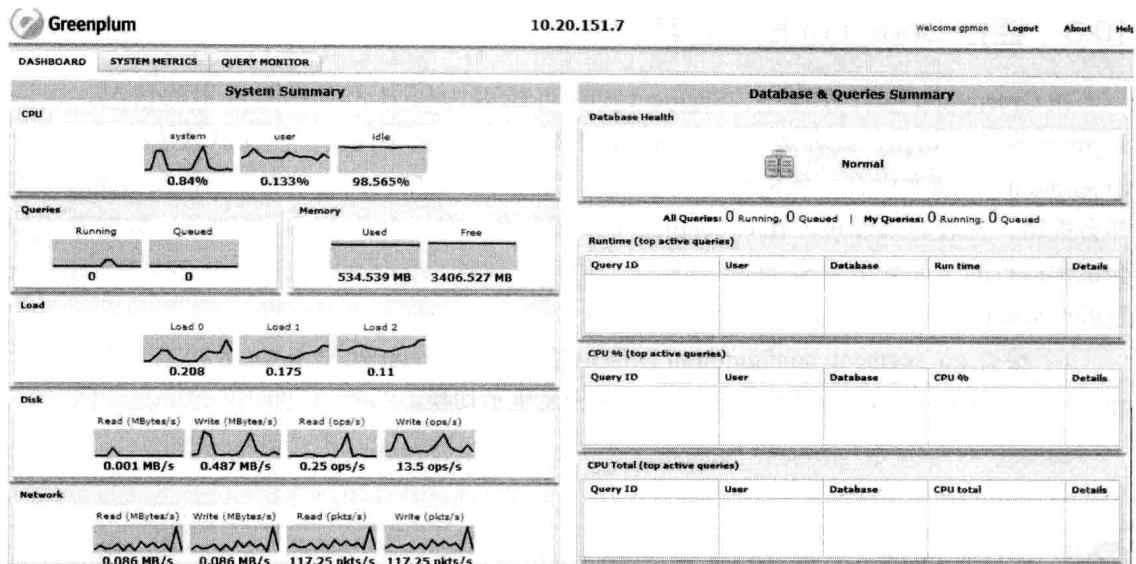


图 10-13 登录后第一个页面

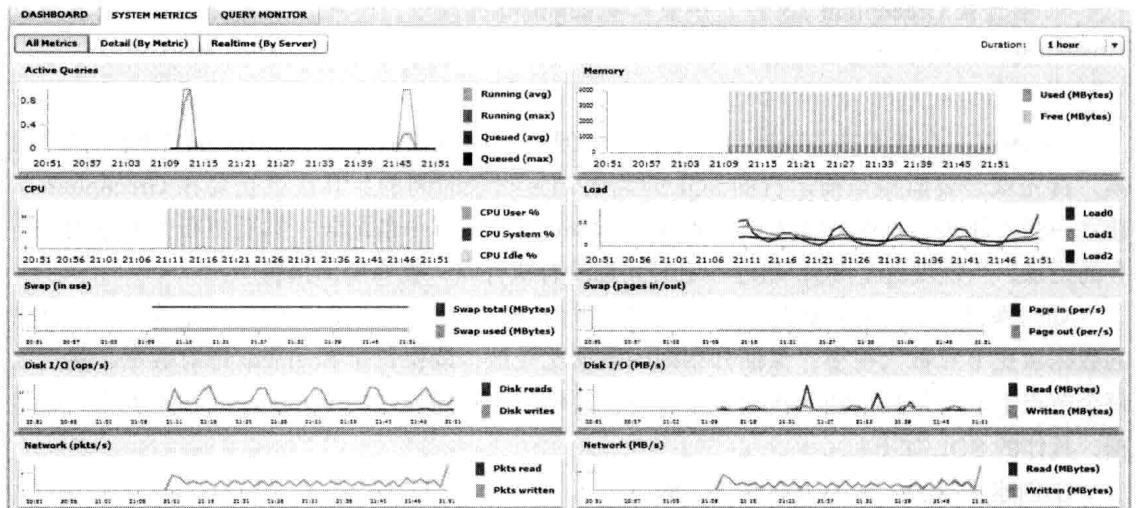


图 10-14 系统性能参数展示图

GPperfmon 还有很多的配置及特性，很多的参数，具体的使用，读者可以参考 GPPerfmonAdminGuide.pdf 这个文档（在安装目录下的 docs 目录中），该文档对 GPperfmon 的使用有很详细的描述。



**提示** Greenplum 4.3 最新的安装文件中没有自带 greenplum-perfmon-web 的更新安装包，经笔者实测，greenplum-perfmon-web-4.1.1.3 可以应用在 Greenplum 4.3 上，安装步骤流程都一样的，很方便。

### 10.3 监控 Segment 是否正常

在 Greenplum 运行过程中，Segment 很有可能因为压力大出现不可用的情况，主备 Segment 发现了切换，或者是主备 Segment 网络断开，数据不同步了。在默认情况下，如果 Greenplum 3.x 版本中有一个 Segment 失败了，数据库会切换到只读状态，运行在上面的任务都会报错，比较容易发现。在 Greenplum 4.x 版本中，有一个 Segment 失败了，数据库还是会正常运行的，如果是主 Segment 失败了，则切换到备 Segment 上，这样就必须对 Segment 是否正常加以监控，一般有以下两种监控方法：

1) 检查 `gp_segment_configuration` 以确定是否有 Segment 处于 down 的状态，或者查看 `gp_configuration_history` 以观察最近数据库是否发生了切换。

```
select * from gp_segment_configuration where status='d' or mode<>'s';
```

 **注意** 在 Greenplum 3.x 中，配置表为 `gp_configuration`，查询 SQL 为：`select * from gp_configuration where valid='f';`

不过在 Greenplum 3.x 中，一般只要 segment 失败了，相关任务就会报错，很容易发现，因此这个监控一般不用加。

2) 还有一个就是 Segment 已经卡住了，但是 Master 没有感知到 Segment 失败，这个时候，首先就是要监控当前运行的 SQL 是否有超过很长时间的。其次就是要在 Greenplum 中建立一张心跳表，这张心跳表至少要在每个 Segment 都有一条记录，然后不断去更新表中所有的记录，当发现这个 SQL 超过一定时间都没有执行完，就要发出报警。

如何保证每个 Segment 都有一条记录？下面介绍一个方法：首先创建一张临时表，其中的数据量比节点数大很多，保证分布键的数据比较散，确保每个 Segment 都有数据，然后在每个节点上取一条数据插入新的一张表中，这样就能够保证每一个 Segment 都至少有一条数据，具体的 SQL 如下：

① 创建临时表，插入 10000 条数据

```
create table xdual_temp
as
select generate_series(1,10000) id
distributed by (id);
```

② 建立心跳表，2 个字段，第二个字段是 timestamp 类型的，每次心跳检测数据会被更新

```
create table xdual(id int, update_time timestamp(0))
distributed by(id);
```

### ③往心跳表的每个 Segment 中插入一条数据

```
insert into xdual(id,update_time)
select id,now() from
  (select id, row_number() over(partition by gp_segment_id order by id) rn
   from xdual_temp
  )t
where rn=1;
```

效果如下：

```
testDB=# select gp_segment_id,* from xdual order by 1;
gp_segment_id | id | update_time
-----+-----+
 0 | 1 | 2012-08-12 19:40:43
 1 | 4 | 2012-08-12 19:40:43
 2 | 9 | 2012-08-12 19:40:43
 3 | 2 | 2012-08-12 19:40:43
 4 | 3 | 2012-08-12 19:40:43
 5 | 8 | 2012-08-12 19:40:43
(6 rows)
```

心跳检测的 SQL 就是：update xdual set update\_time=now(); 只要这个 SQL 运行正常，就是代表每一个 Segment 都是正常的。

## 10.4 VACUUM 系统表

Greenplum 是基于 MVCC 版本控制的，所有的 delete 并没有删除数据，而是将这一行数据标记为删除，而 update 其实就是 delete 加 insert。所以，随着操作越来越多，表的大小也会越来越大。对于 OLAP 应用来说，大部分的表都是一次导入后不再修改的，所以不会有这个问题，但是对于数据字典来说，就会随着时间表越来越大，其中的数据垃圾越来越多。

所以，Greenplum 提供一种 VACUUM 的工具，可以回收已删除行占据的存储空间。语法如下：

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

简单的 VACUUM (没有 FULL) 只是简单地回收空间且令其可以再次使用，通过简单的 VACUUM 可以缓解表的增长。这个命令执行时，其他操作仍可以对表的读写等并发操作，因为 VACUUM 没有请求排他锁。

VACUUM FULL 执行更广泛地处理，包括跨块移动行，以便把表压缩至使用最少的磁盘块数目存储。这种形式要慢许多且在处理的时候需要在表上施加一个排它锁。在 PostgreSQL 中，提供了自动 VACUUM 的功能，但是由于 Greenplum 中大部分的表是不需要 VACUUM 的，所以 AUTOVACUUM 是关闭的。

在 Greenplum 日常的维护过程中，需要对数据字典定期执行 VACUUM，这个可以在每天数据库比较空闲的时候进行。然后每隔一段比较长的时间（两三个月）对系统表执行一次 VACUUM FULL，这个操作需要停机，当表大的时候，这一步会比较慢，根据表的大小不同，有可能会长达几个小时。执行 VACUUM 之后，最好对表上的索引进行重建（reindex）。

 **注意** 在数据库中，保存事务号是一个 32 位整型，当系统消耗的事务 id 过多时，到达一个阀值，就会触发强制 AUTOVACUUM，回收事务号。AUTOVACUUM 是对整库的扫描，会比较影响性能，所以在平时维护中，要留意事务号可以通过下面这个 SQL 来看数据库中的事务号消耗：

```
select gp_segment_id, datname, datfrozenxid, age(datfrozenxid)
from gp_dist_random('pg_database') where datname='testDB';
```

如果 age(datfrozenxid) 超过 2 亿，就会触发 AUTOVACUUM。如果全部事务号只剩下 100w，那么数据库为了保护数据安全，整个数据库都会挂掉，无法启动，只能使用 standalone、单用户模式启动数据库，对整个数据库执行 VACUUM 回收事务号之后才能够启动。

下面通过一个例子来介绍 VACUUM 的效果。

测试的数据库从来没有进行过 VACUUM，先查询出 pg\_class 中最大的 ctid（可以看做是数据块的位置，在没有 VACUUM 的时候，ctid 一直是递增的）：

```
testDB=# select max(ctid) from pg_class;
      max
-----
(10,163)
(1 row)
```

看出最大值是（10,163）之后创建两张表，并查询其新的 ctid 值：

```
testDB=# create table test_vacuum1(a int) distributed by(a);
CREATE TABLE
testDB=# create table test_vacuum2(a int) distributed by(a);
CREATE TABLE
testDB=# select ctid, relname from pg_class where relname like 'test_vacuum%';
      ctid   |    relname
-----+-----
(10,164) | test_vacuum1
(10,165) | test_vacuum2
(2 rows)
```

可以看出，由于这个库从来没有进行过 VACUUM，因此新建的两张表的 ctid 值是在之前的值上递增的。然后对 pg\_class 进行 VACUUM，建表之后再测试。

```

testDB=# VACUUM pg_class;
VACUUM
testDB=# create table test_vacuum3(a int) distributed by(a);
CREATE TABLE
testDB=# select ctid,relname from pg_class where relname like 'test_vacuum%';
   ctid |    relname
-----+-----
(2,36) | test_vacuum3
(10,164)| test_vacuum1
(10,165)| test_vacuum2
(3 rows)

```

可以看出，在执行 VACUUM 之后，表的空间被回收了，新插入的 pg\_class 记录所在的数据块被重用了。但是 VACUUM 并不会对原有的记录做任何改变。

接着对 pg\_class 进行 VACUUM FULL，首先查出 pg\_class 的大小，然后进行 VACUUM FULL，之后观察表的大小变化。

```

testDB=# select pg_relation_size('pg_class');
 pg_relation_size
-----
 2523136
(1 row)
testDB=# VACUUM FULL pg_class ;
NOTICE: 'VACUUM FULL' is not safe for large tables and has been known to yield
unpredictable runtimes.
HINT: Use 'VACUUM' instead.
VACUUM
testDB=# select pg_relation_size('pg_class');
 pg_relation_size
-----
 1605632
(1 row)

```

继续观察，之前在 pg\_class 中几张表的数据中的 ctid 都发生了变化，比之前的值更小了：

```

testDB=# select ctid,relname from pg_class where relname like 'test_vacuum%';
   ctid |    relname
-----+-----
(2,36) | test_vacuum3
(2,127)| test_vacuum1
(2,128)| test_vacuum2
(3 rows)

```

## 10.5 数据倾斜排查

对于 Greenplum 这种并行执行的数据库，负载均衡其实是通过数据均匀分布来实现的，如果表的数据不均衡，则数据多的节点会比其他节点压力大。根据木桶原理，时间消耗会卡

在数据多的节点上。所以，使用 Greenplum 必须时刻保持数据均匀分布。

Greenplum 的数据分布是通过设置分布键来实现的，因此设置合理的分布键就非常重要。在第 2 章中简单介绍了分布键的使用，这里总结一下，分布键设置的一些原则如下。

- 一般使用数据比较均匀、空值很少的字段作为主键（最合适的就是主键）。
- 依据第 5 章执行计划中介绍的，两表关联时可能会造成数据广播或者重分布，所以选择分布键的时候要适当考虑这个表与其他表最常用的关联键。除此之外，group by、window function 都会造成数据重分布。
- 关联键与唯一键不一样要有所取舍。

下面来介绍如何看出一个表是否分布均匀。

### 1. 使用隐藏字段 `gp_segment_id`

通过使用 `gp_segment_id` 作为 group by 字段，可以看到每个 Segment 的数据量，SQL 如下（其中 Group by 1 中的 1 代表 select 的第一个字段，即 `gp_segment_id`）。

```
testDB=# select gp_segment_id, count(1) from offer_4 group by 1 order by 1;
gp_segment_id | count
-----+-----
 0 | 166523
 1 | 166898
 2 | 166332
 3 | 167442
 4 | 165990
 5 | 166815
(6 rows)
```

### 2. 使用 `get ao distribution`

在第 6 章讲到了 appendonly 表的特性，我们可以利用 `get ao distribution` 获取每个 Segment 的数据量。

```
testDB=# select * from get_ao_distribution('offer_4');
segmentid | tupcount
-----+-----
 1 | 166898
 2 | 166332
 0 | 166523
 4 | 165990
 3 | 167442
 5 | 166815
(6 rows)
```

### 3. 使用 `pg_relation_size` 和 `gp_dist_random`

利用这两个函数，可以查询表对应的数据文件大小，从而确定这个表是否分布均匀。

```

testDB=# select gp_segment_id,pg_relation_size(oid)
testDB-#   from gp_dist_random('pg_class')
testDB-#   where relname = 'offer_4';
      gp_segment_id | pg_relation_size
-----+-----
      1 |          115857464
      0 |          115690216
      5 |          115799064
      3 |          116315936
      2 |          115439608
      4 |          115282984
(6 rows)

```

如果数据库比较小，可以通过这个方法计算出数据库中哪一个表发生了比较严重的倾斜，首先定义一个数据倾斜率为：最大子节点数据量 / 平均节点数据量。为避免整张表数据量为空，同时对结果的影响很小，要将平均节点数据量加上一个很小的值。

SQL 如下：

```

select tablename,max(size)/(avg(size)+0.001) as max_div_avg ,sum(size) total_size
from (
    select gp_segment_id,oid::regclass tablename,pg_relation_size(oid) size
    from gp_dist_random('pg_class')
    where relkind='r' and relstorage in ('a','h')
)t group by tablename
order by 2 desc;

```

SQL 执行的效果如图 10-15 所示。

tablename	max_div_avg	total_size
pdp_problem	5.999999895368304	340640
test_offer_temp	5.9999989013673923	32768
pdp_request	5.9996250234360332	96
cxt	2.999999999999999828438	56754176
offer_01	1.1247125398618275229	2869316
offer_3	1.1247125398618275229	663408328
member	1.12016546505355333	2233237504

图 10-15 查询数据库中表的数据倾斜率

通过这种方法找出整个数据库中发生数据倾斜的表，比较简单，也很容易操作，相比起前两种效率高很多，但是当数据库中表非常多的时候，这个方法的性能也有问题。下面介绍另外一种方法。

#### 4. 对整个数据目录做 ls

每个表对应的文件名保存在 pg\_class 中的 relfilenode 字段中，这样我们可以通过操作系统命令 ls -l 对整个数据库目录中的所有文件进行遍历，生成数据文件，然后将文件导入数据库中，与 pg\_class 关联，然后再比较找出发生数据倾斜的表。这种方法比较麻烦，需要读者

自己编写脚本，如果数据库中有非常多的表，这种方法无疑是性能最高的。

 **注意** 使用这种方法需要处理数据文件超过 1GB 拆分的问题：当一个数据文件超过 1GB 的时候，数据库会将超过 1GB 的部分放在文件名 .1 的文件中，超过 2GB 的放在 .2 的文件中，以此类推（在前面介绍的，列存储也有这个问题，处理方法一样）。在编写脚本的时候如果要处理这个问题，将文件名后面的 .1、.2…去掉。

## 10.6 查看子节点的 SQL 运行状态

查询 Master 上面的 SQL 的执行，主要通过视图 pg\_stat\_activity 来完成。通过这个视图可以获取当前 Greenplum 的任务执行情况，如图 10-16 所示。

```
testDB=# \x
Expanded display is on.
testDB=# select * from pg_stat_activity;
-[ RECORD 1 ]-
datid          16992
datname        gpdb
procpid        1177
sess_id         88774
usesysid       10
username        gpadmin
current_query   select * from pg_stat_activity;
waiting         f
query_start     2012-07-29 18:27:50.089297+08
backend_start   2012-07-29 17:37:16.53126+08
client_addr     -
client_port     -1
application_name psql
xact_start      2012-07-29 17:43:50.195667+08
```

图 10-16 查询 SQL 运行情况

而 Greenplum 本身并没有提供一个工具用于很好地观察每个子节点 SQL 的执行状态，如果我们想看某一个子节点上运行的 SQL，需要使用 utility 模式连接到 Segment 上，然后再查看。这里将介绍如何在 Master 上建立一个视图，方便查询 Segment 上的 SQL。

查询每个子节点的 SQL 将用到前面 10.5 节介绍的 gp\_dist\_random 函数。在开始编写 all\_seg\_sql 视图之前，我们先来讨论一下，在 Greenplum 中的函数是如何执行的，是在 Master 上执行，还是在 Segment 上执行。

```
testDB=# select current_setting('data_directory');
           current_setting
-----
/home/gpadmin/gpdata/gpmaster/gpseg-1
(1 row)
```

current\_setting 函数是显示系统参数，data\_directory 代表数据目录。上述 SQL 显示了 Master 的数据目录，如果要查询 Segment 的数据目录呢，使用 gp\_dist\_random 可以吗？

```
testDB=# select current_setting('data_directory') from gp_dist_random('gp_id');
           current_setting
-----
```

```
/home/gpadmin/gpdata/gpmaster/gpseg-1
/home/gpadmin/gpdata/gpmaster/gpseg-1
/home/gpadmin/gpdata/gpmaster/gpseg-1
/home/gpadmin/gpdata/gpmaster/gpseg-1
/home/gpadmin/gpdata/gpmaster/gpseg-1
/home/gpadmin/gpdata/gpmaster/gpseg-1
(6 rows)
```

可以看出，查询出来的结果还是 Master 上的，说明这个函数是在 Master 上执行的，而要让这个函数在 Segment 上执行，方法有下面两个。

### 1. 函数的参数中带有查询表的某一个字段

```
testDB=# select current_setting(replace('data_directory'||gpname, gpname, ''))
testDB-#   from gp_dist_random('gp_id');
            current_setting
-----
/home/gpadmin/gpdata/gpdatap1/gpseg2
/home/gpadmin/gpdata/gpdatap1/gpseg4
/home/gpadmin/gpdata/gpdatap2/gpseg5
/home/gpadmin/gpdata/gpdatap1/gpseg0
/home/gpadmin/gpdata/gpdatap2/gpseg3
/home/gpadmin/gpdata/gpdatap2/gpseg1
(6 rows)
```

上面这个 SQL 将 data\_directory 字符串拼接到 gpname 字段，然后再把这个字段给替换成空，看似做了一个完全无用的动作，但是，由于参数中有查询的表的字段，这个函数放到了 Segment 上去执行，所以查询出来的结果就是 Segment 上的数据目录了。



**注意** 表 gp\_id 在每个 Segment 中只有一行数据，利用此特性，可以结合 gp\_dist\_random 查询每个 Segment 的信息。

### 2. 将函数封装多一层

创建一个函数，将 current\_settings 包起来，如下：

```
create or replace function public.get_setting(setname varchar)
RETURNS text
AS
$BODY$
begin
    return current_setting(setname);
end;
$BODY$
LANGUAGE 'plpgsql' ;
```

然后再使用 get\_setting 查询：

```
testDB=# select get_setting('data_directory') from gp_dist_random('gp_id');
          get_setting
-----
/home/gpadmin/gpdata/gpdatap1/gpseg4
/home/gpadmin/gpdata/gpdatap2/gpseg5
/home/gpadmin/gpdata/gpdatap2/gpseg3
/home/gpadmin/gpdata/gpdatap1/gpseg2
/home/gpadmin/gpdata/gpdatap1/gpseg0
/home/gpadmin/gpdata/gpdatap2/gpseg1
(6 rows)
```

利用以上两种方法都可以使函数在 Segment 上执行，有了这个，我们就可以在 Master 上创建 all\_seg\_sql 这个视图了。下面是创建 all\_seg\_sql 的三个步骤。

### 1) 创建 v\_active\_sql 视图方便查看 SQL

```
CREATE VIEW v_active_sql AS
SELECT pg_stat_activity.procpid, pg_stat_activity.sess_id, pg_stat_activity.
username, pg_stat_activity.waiting AS w, to_char(pg_stat_activity.query_start,
'mm-dd hh24:mi:ss'::text) AS query_start, to_char(now() - pg_stat_activity.
query_start, 'hh24:mi'::text) AS exec, pg_stat_activity.current_query
    FROM pg_stat_activity
   WHERE pg_stat_activity.current_query <> '<IDLE>'::text
   ORDER BY pg_stat_activity.datname, to_char(pg_stat_activity.query_start,
'yyyymmdd hh24:mi:ss'::text);
```

### 2) 创建获取 ip 的函数

```
CREATE or replace FUNCTION public.hostip()
RETURNS text
AS $$

import socket
return socket.gethostbyname(socket.gethostname())
$$ LANGUAGE plpythonu;
```

### 3) 创建 all\_seg\_sql 函数

```
Create view public.all_seg_sql
as
select hostip(),
       current_setting(replace('port'||current_query,current_query,'')) as port,
       current_setting(replace('gp_contentid'||current_query,current_query,''))
           as content,
*
from gp_dist_random('v_active_sql')
where current_query <> '<IDLE>';
```

查询效果如图 10-17 所示。

testDB=# select * from all_seg_sql;	hostip	port	content	procpid	sess_id	username	w	query_start	exec	current_query
	10.20.151.8	33000	0	8849	90056	gpadmin	f	07-29 21:13:22	00:00	select * from all_seg_sql;
	10.20.151.9	33001	5	17180	90056	gpadmin	f	07-29 21:13:22	00:00	select * from all_seg_sql;
	10.20.151.9	33001	1	8851	90056	gpadmin	f	07-29 21:13:22	00:00	select * from all_seg_sql;
	10.20.151.10	33000	4	17178	90056	gpadmin	f	07-29 21:13:22	00:00	select * from all_seg_sql;
	10.20.151.9	33000	2	12345	90056	gpadmin	f	07-29 21:13:22	00:00	select * from all_seg_sql;
(6 rows)	10.20.151.9	33001	3	12347	90056	gpadmin	f	07-29 21:13:22	00:00	select * from all_seg_sql;

图 10-17 all\_seg\_sql 查询结果

## 10.7 自动加分区

对于数据库应用来说，记历史是对业务分析的常见手段。对于这些记录历史表，通常按天分区，即每天都是一个分区，将每天的全量或增量数据保存到这个分区中。



记历史就是将每一个的记录都保留下来，然后可能通过分析每天的变化来预测业务的发展，或者分析当前业务的状态等。常用的算法是拉链记历史，在第 3 章简单介绍这种方法。

对于这种应用，需要有一个脚本自动增加每天的分区，如果手工加，当表多的时候会很麻烦。下面简单介绍下如何自动加分区，以及加分区的策略。

要实现自动加分区，分区必须有一定的规则才行。每个公司的应用场景不一样，有一些分区可能跟业务相关，不能够简单地实现自动加分区。这里只介绍最简单的分区，也是数据库中最常用到的，按时间分区（包括按天，按月分区）。具体规则如下。

1) 按天分区，分区名为 p+yyyymmdd (如 p20120801); 按月分区，分区名为 p+yyymmm (如 p201208)。

2) 分区类型支持 list 分区和 range 分区。

3) 自动加分区必须将原表先建好，而且建好的子分区数必须超过两个，这样脚本才能判断两个分区之间的时间间隔。

4) 分区键支持多种数据类型，但是这个数据类型必须能够通过类型转换，直接可以转换成 date 类型。如果是字符类型，那么时间格式为 yyymmmdd。

5) 子分区表的其他属性与父表一致，比如 appendonly=true,compresslevel=5 (注意，这些参数必须在增加分区之后指定，默认并不会从父表中继承)。

6) 这里只处理一层分区，两层的分区这里不考虑。

7) 分区键只有一个字段。

下面就是自动加分区的例子。

创建 range 分区，在子分区中可以指定字段的起始值和结束值。

```
create table public.tbl_partition_range_yyymmmdd(
```

```

        id          numeric
,yyyymmdd      date
)with(appendonly=true,compresslevel=5)
Distributed by (id)
PARTITION BY range(yyyymmdd)
(
    PARTITION p20120811 START ('2012-08-11'::date) END ('2012-08-12'::date),
    PARTITION p20120812 START ('2012-08-12'::date) END ('2012-08-13'::date)
);

```

创建 list 分区，每个分区对应一个具体的值：

```

create table public.tbl_partition_list_yyyymmdd(
        id          numeric
,yyyymmdd      varchar(128)
)with(appendonly=true,compresslevel=5)
Distributed by (id)
PARTITION BY list(yyyymmdd)
(
    PARTITION p20120811 values('20120811'),
    PARTITION p20120812 values('20120812')
);

```

创建一个辅助视图：

```

create view public.v_pg_add_partitions
as
SELECT pp.parrelid tableoid,pr1.parchildrelid,pr1.parname as partitionname,
CASE
    WHEN pp.parkind = 'h'::"char" THEN 'hash'::text
    WHEN pp.parkind = 'r'::"char" THEN 'range'::text
    WHEN pp.parkind = 'l'::"char" THEN 'list'::text
    ELSE NULL::text
END AS partitiontype,
translate(pg_get_expr(pr1.parlistvalues, pr1.parchildrelid), '-'::date
character varying bpchar numeric double precision timestamp without time zone',
'') AS partitionlistvalue,
substring(translate(pg_get_expr(pr1.parrangestart, pr1.parchildrelid),
'-''::date character varying bpchar numeric double precision timestamp without
time zone', ''),1,8) AS partitionrangestart,
.substring(translate(pg_get_expr(pr1.parrangeend, pr1.parchildrelid),
'-''::date character varying bpchar numeric double precision timestamp without
time zone', ''),1,8) AS partitionrangeend,
pr1.parruleord AS partitionposition,
substring(parlistvalues,'consttype ([0-9]+)::integer::regtype listtype',
substring(parrangeend,'consttype ([0-9]+)::integer::regtype rangetype
FROM pg_partition pp, pg_partition_rule pr1
WHERE pp.paristemplate = false AND pr1.paroid = pp.oid ;

```

这个视图的每一个字段的描述如表 10-1 所示。

表 10-1 v\_pg\_add\_partitions 视图的字段描述

字段名	数据类型	描述
tableoid	oid	对应分区表父表的 oid
parchildrelid	oid	对应子分区的 oid
partitionname	name	子分区的分区名
partitiontype	text	子分区的类型 (list 或者 range)
partitionlistvalue	text	List 类型分区的 values 值, 若为 range 分区则为空
partitionrangestart	text	range 分区的起始值, 若为 list 分区则为空, 格式为 yyymmdd 或者 yyymmm
partitionrangeend	text	range 分区的结束值, 若为 list 分区则为空, 格式为 yyymmdd 或者 yyymmm
partitionposition	smallint	该子分区位于分区表的序号
listtype	regtype	list 分区, 分布键的数据类型
rangetype	regtype	range 分区, 分布键的数据类型

创建生成增加分区的语句, 第一个参数表示分区表的 oid, 第二个参数是增加分区到今天之后的第几天:

```

CREATE or replace FUNCTION public.add_partition_info(tableoid oid, days_from_now
integer)
RETURNS setof text
AS $$

import datetime
def now():
    d = datetime.datetime.now()
    format='%Y%m%d'
    return datetime.datetime.strftime(d,format)

def add_day(d,n):
    format='%Y%m%d'
    d2 = datetime.datetime.strptime(d, format)
    d3 = d2 + datetime.timedelta(days = n)
    return datetime.datetime.strftime(d3,format)

def add_month(d,n):
    format='%Y%m%d'
    formatydm='%Y%m01'
    if d.__len__()==6:
        format='%Y%m'
        formatydm='%Y%m'
    d2 = datetime.datetime.strptime(d, format)
    d3 = d2 + datetime.timedelta(days = 31*n)
    return datetime.datetime.strftime(d3,formatydm)

relist=[]
sql="""select * from (
    select *,tableoid::regclass tablename,lead(case when

```

```

partitionrangeend <>'' then partitionrangeend else partitonlistvalue end)
                                over(partition by tableoid order by partitionposition
desc) as pre_value,
                                row_number() over(partition by tableoid order by
partitionposition desc ) rn
                                from v_pg_add_partitions
                                where substr(partitionname,1,3)='p20'
                                and tableoid=%s
                                )t where rn=1;""%(tableoid);
rv = plpy.execute(sql);
sql_relation="select array_to_string(reloptions,',') reloptions from pg_
class where oid=%s"%(tableoid)
rv_relation = plpy.execute(sql_relation)

if rv.nrows()!=1:
    return None

else:
    reloptions = rv_relation[0]['reloptions']
    tablename=rv[0]['tablename']
    partitiontype = rv[0]['partitiontype']
    partitionname = rv[0]['partitionname']
    pre_value = rv[0]['pre_value']
    now_add_7days = add_day(now(),days_from_now)

    # 处理 range 分区

    if partitiontype=='range':
        rangetype=rv[0]['rangetype']
        partitionrangestart = rv[0]['partitionrangestart']
        partitionrangeend = rv[0]['partitionrangeend']
        interval = int(partitionrangeend) - int(pre_value)

        # 按月分区

        if partitionname.__len__()==7:
            func_add = add_month
            interval = int(partitionrangeend[0:6]) - int(pre_value[0:6])

        # 按天分区

        elif partitionname.__len__()==9:
            func_add = add_day

        # 分区名不符合规范，不处理

    else:
        return None
    while partitionrangestart<now_add_7days:
        partitionrangestart = func_add(partitionrangestart,interval)

```

```

        partitionrangeend = func_add(partitionrangeend,interval)
        partitionname = "p" + func_add(partitionname[1:],interval)
        add_sql = "alter table %s add partition %s start ('%s):::%s) end
('%":::%"') "%(tablename,partitionname,partitionrangestart,rangetype,partition
ra-ngeend,rangetype)
        if reloptions !=None and reloptions!="":
            add_sql+='with(%s);'%reloptions
        else:
            add_sql+=";"
        relist.append(add_sql)

    if partitiontype=='list':
        listtype=rv[0]['listtype']
        partitonlistvalue=rv[0]['partitonlistvalue']
        interval = int(partitonlistvalue) - int(pre_value)

        # 按月分区

        if partitionname.__len__()==7:
            func_add = add_month

        # 按天分区

        elif partitionname.__len__()==9:
            func_add = add_day

        # 分区名不符合规范，不处理

    else:
        return None
    while partitonlistvalue<now_add_7days:
        partitonlistvalue = func_add(partitonlistvalue,interval)
        partitionname = "p" + func_add(partitionname[1:],interval)
        add_sql = "alter table %s add partition %s values('%s):::%s)"
        "%(tablename,partitionname,partitonlistvalue,listtype)
        if reloptions !=None and reloptions!="":
            add_sql+='with(%s);'%reloptions
        else:
            add_sql+=";"
        relist.append(add_sql)
    return relist
$$ LANGUAGE plpythonu;

```

使用效果如下，增加当前时间 3 天内的分区：

```

testDB=# select add_partition_info('tbl_partition_range_yyyyymmdd)::regclass,3);
add_partition_info
-----
 alter table tbl_partition_range_yyyyymmdd add partition p20120813 start
('20120813)::date) end ('20120814)::date) with(appendonly=tr
ue,compresslevel=5);

```

```

alter table tbl_partition_range_yyyyymmdd add partition p20120814 start
('20120814'::date) end ('20120815'::date) with(appendonly=true,
compresslevel=5);
alter table tbl_partition_range_yyyyymmdd add partition p20120815 start
('20120815'::date) end ('20120816'::date) with(appendonly=true,
compresslevel=5);
(3 rows)

```

## 10.8 自动赋权

一般数据库会有很多不同类型的用户，有一些用户只有对表的只读权限，有一些则有对表有读写的权限。有时，我们需要将所有特定的表的只读权限赋予其中一个用户。下面简单介绍下如何找出将某个 schema 下用户 aquery 没有权限的表，然后生成赋权语句，赋予 aquery 用户这些表的只读权限。

```

select 'grant select on'||nspname||'.'||relname||' to aquery;' 
from pg_class a,pg_namespace b
where relname not like '%_1_prt%' 
and relkind = 'r'
and has_table_privilege('aquery',a.oid,'select')='f'
and a.relnamespace=b.oid
and nspname not in ('pg_catalog','information_schema')
and nspname not like '%pg_temp%';

```

通过上面的 SQL，就可以直接生成赋权的 SQL，每天只需要调用这个 SQL，然后执行生成的赋权语句，就可以实现自动赋权了。

## 10.9 清理过期数据

数据库在正常使用的情况下一般会对用户提供开发测试功能。出于效率上的考虑，一般对测试的表没有很严格的限制，这样就会导致数据库存在很多临时表。如何对这些表进行清理呢？

由于开发测试一般都有周期性，而且表都是临时表，根据业务需求，一般一个月以前的表都是没有用的，可以删除的，但是开发人员一般会忘记删除表。对于 DBA 来说，就是希望把数据库中一个月前的临时表都找出来，然后配成定时任务，每天将这些表自动删除。

在第 4 章介绍数据字典的时候，讲过如何获取数据文件的操作时间，从而找出表的近似建立时间，通过这个时间，我们就可以将过期数据清理掉，如果开发测试建的表只能够建立在某一个 schema（比如 devtest）下面，那么生成删除表脚本的 SQL 如下：

```

select 'drop table '||schemaname||'.'||tablename||';'
from v_table_modify_time

```

```
where access<now()-'30 days'::interval  
and tablename not like '%_l_prt_p%'  
and schemaname='devtest'  
order by access ;
```

## 10.10 小结

本章主要介绍了几种数据库监控的方法，通过这个方法，数据库管理员可以及时发现数据库发生的问题，并且在问题出现后可以快速定位并解决问题。对于监控，主要介绍如下。

- 操作系统监控磁盘、CPU 等信息的工具
- Greenplum 自带的 GPmonitor 的安装及使用方法
- 监控 Segment 状态是否正常

介绍了日常管理中调优常用的方法，以及调优时的注意事项。在日常维护的时候，需要定时 Vacuum 数据字典表，避免数据字典快速膨胀。通过分析所有表的大小，找出数据库中发生数据倾斜的表，及早发现问题，避免因大量数据倾斜导致数据库性能很差，或者某个 Segment 因压力过大而失败。当数据库慢的时候，可以通过 all\_seg\_sql 来分析数据库每个 Segment 都在执行什么 SQL，有助于了解数据库运行、某个 Segment 压力过大的原因。

最后介绍了 DBA 日常使用的维护工具，以及自动加分区、赋权及清理过期数据的一些小技巧。

## 第 11 章

# 解读 Greenplum 维护脚本

Greenplum 自带了很多维护数据库的脚本，熟读这些脚本有助于理解 Greenplum 的架构，以及当 Greenplum 出现问题时，能够更快速地找到解决的办法。本章首先介绍在 Greenplum 中怎么安装 PostgreSQL 中自带的 contrib 插件，之后挑选几个比较重要的脚本进行解读，使读者在阅读源码时更容易理解。

## 11.1 添加 Greenplum Contrib 模块

PostgreSQL 的 contrib 中提供了很多有用的工具，我们都知道，Greenplum 是基于 PostgreSQL 开发的，所以在 Greenplum 中也可以安装这些工具，从而方便使用数据库。

下面介绍在 Greenplum 中如何编译及安装 Contrib 的插件。

Contrib 模块的源码在 PostgreSQL 的源码包中，由于 Greenplum 是基于 PostgreSQL8.2.x 开发的，因此我们这里相应地下载 PostgreSQL8.2.x 的源码，然后编译源码（不用 install）。

```
./configure  
make
```

Greenplum 与原生的 PostgreSQL 在库包上有所不同，如果要在 Greenplum 中使用 contrib 中的工具，必须使用 Greenplum 头文件和库文件。方法就是修改 src/Makefile.global 文件。

这里以 PostgreSQL-8.2.20 为例，讲解如何在 Makefile.global 中加入 Greenplum 4.1.1.1 的头文件和库文件。

首先，下载 PostgreSQL8.2 的源代码，设置环境变量 GPHOME ( Greenplum 集群中一般已经设定了)。

其次，修改 PostgreSQL8.2 源代码中的 src/Makefile.global 文件，在 # Compilers 和 CPP = gcc -E 之间加入下面这段内容，定义 Greenplum 的类库：

```
# Compilers
# add start
GPINCL = -I$(GPHOME)/include -I$(GPHOME)/include/postgresql -I$(GPHOME)/
include/libpq -I$(GPHOME)/include/postgresql/server -I$(GPHOME)/include/
postgresql/internal
GPLIB = -L$(GPHOME)/lib -L$(GPHOME)/lib/postgresql
ifndef USE_PGXS
CFLAGS = $(GPINCL)
LDFLAGS = $(GPLIB)
endif
# add end

CPP = gcc -E
```

避免上述增加的内容被覆盖掉，将下面的：

```
CFLAGS = -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Winline -Wdeclaration-
after-statement -Wendif-labels -fno-strict-aliasing -fwrapv
```

修改为：

```
CFLAGS += -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Winline
-Wdeclaration-after-statement -Wendif-labels -fno-strict-aliasing -fwrapv
即将 “=” 修改为 “+=”
```

将 LDFLAGS = 注释掉，避免覆盖掉上面的定义：

```
#LDFLAGS =
```

在 LDFLAGS 中，将 \$(LDFLAGS) 放在前面（前面已经设置了 LDFLAGS 为 GPLIB），从而保证 Greenplum 的库文件优先被使用。将下面的：

```
override LDFLAGS := -L$(libdir) $(LDFLAGS)
```

修改为：

```
override LDFLAGS := $(LDFLAGS) -L$(libdir) $(LDFLAGS)
```

将修改保存后就可以编译 contrib 中的模块了。下面以编译 tablefunc 为例，在 contrib/tablefunc 下运行 make，完成编译后会多两个文件：

```
-rw-rw-r-- 1 gpadmin gpadmin 2.2K May 13 19:01 tablefunc.sql
-rwxrwxr-x 1 gpadmin gpadmin 25K May 13 19:01 tablefunc.so
```

使用 gpscp 将文件 scp 到每一台机器的 Greenplum 的安装目录下：

```
gpscp -f ~/hostlist tablefunc.so =:$GPHOME/lib/postgresql/
```

运行 psql 进行安装：

```
psql -f tablefunc.sql
```

如果需要卸载，则运行 uninstall\_tablefunc.sql。

Tablefunc 的“使用帮助”按钮可以参考 README.tablefunc，Tablefunc 中的几个函数。下面简单检验下 normal\_rand 函数（生成正态分布的随机数函数）是否安装成功。

```
testDB=# SELECT * FROM normal_rand(5, 5, 3);
      normal_rand
-----
 9.84995374805584
 -0.299470922745508
 10.9453286308036
 6.20799777320577
 8.31545906031287
(5 rows)
```

## 11.2 启动和关闭脚本 gpstart 和 gpstop

在 Greenplum 的脚本中，启动和关闭数据库脚本 gpstart 和 gpstop，是最常用的。

### 1. gpstart

gpstart 脚本参数如表 11-1 所示。

表 11-1 gpstart 脚本参数

参 数	说 明
-a	不提示用户确认
-B	多少个并行，默认是 64 个并行，当然还取决于 Segment 的个数
-d	Master 的数据目录，默认读取环境变量 \$MASTER_DATA_DIRECTORY
-l	日志目录，默认是 ~/gpAdminLog
-m	使用 utility 模式只启动 Master，Segment 不会启动，单独连接 Master 使用 PGOPTIONS='--session_role=utility' psql
-q	安静模式，不在屏幕上打印信息
-R	限制模式，在启动数据库后，除了超级用户之外，其他用户不允许连接
-t   --timeout	使用多少时间（秒）等待一个 Segment 启动，如果超过这个时间，则自动将这个 Segment 的 PostgreSQL 进程杀掉，默认是 60s，但是如果 Segment 启动的时候是 Recovery 模式，则可能需要更长的时间
-v	相当于 debug 模式，打印出详细的启动信息
-y	不启动 master_standy，默认启动
-?   -h   --help	帮助
--version	版本信息

示例：

□ 只启动 master：

```
gpstart -m
```

□ 正常启动数据库，不提示确认信息，同时不启动 master\_standy：

```
gpstart -a -y
```

通过脚本解读可知，正常的启动流程如下。

- 1) 获取环境变量，如 GPHOME、MASTER\_DATA\_DIRECTORY 等信息。
- 2) 检查 Greenplum 的版本，如果 gpstart 脚本的版本与数据目录的 Greenplum 版本不一致，则报错退出。
- 3) 通过 Master 的 postgresql.conf 配置文件，获取端口等信息，并通过这个端口检查 Master 是否已经启动，主要检查“/tmp/.s.PGSQL. 端口号”这个文件，还要通过 netstat 检查端口是否存在，如果 Master 正常运行，则报错退出。如果 Master 端口已经停止但仍有残留文件，那么删除相关文件，启动 Master 并停止。
- 4) 通过 utility 模式启动 Master，然后通过 gp\_segment\_configuration 表获取 Segment 和 Standby Master 的信息，检查子节点是否有被标记为 Down 的情况。
- 5) 停止 Master，并启动 Standby Master（如果有）。
- 6) 通过 ping 命令检查每个 Segment 所在机器的网络连接是否正常，然后并行启动 Segment 进程（启动 Segment 通过 ssh 命令，调用 \$GPHOME/sbin/gpsegstart.py 命令启动，启动的环境变量跟 Master 一致），并行度可以设置，默认是 64 个并发。
- 7) 检查 Segment 启动情况，确定是否满足集群启动条件（所有的主备 Segment 至少有一个可以启动），如果有一个 Segment 的主备无法启动，则所有的 Segment 都会被停止，然后报错退出。
- 8) 前面步骤都正常则开始启动 Master，检查 Master 的运行状态（pg\_ctl status 获取状态信息，同时尝试连接 Master 并释放）（注意，如果 GP 由于异常关闭，进入 recovery 模式，则连接 Master 消耗时间会长一些）。
- 9) Greenplum 启动成功。

## 2. gpstop

gpstop 脚本参数说明如表 11-2 所示。

表 11-2 gpstop 脚本参数

参 数	说 明
-a/-B/-d/-l/-q/-t	同 gpstart 脚本参数
-M fast	快速停止数据库，中止当前所有事务并回滚，相当于 pg_ctl -m fast stop

(续)

参 数	说 明
-M immediate	立刻停止数据库，杀掉所有的进程，中止所有事务，但不会回滚，不建议使用。相当于 pg_ctl -m immediate stop
-M smart	默认停止数据库的模式，当前库中不能有任何连接或 SQL，只要当前有 SQL 在执行，就会报警退出
-u	不重启数据库，只是重新加载 pg_hba.conf 和 postgresql.conf 配置文件，对不需要重启数据库的参数有效
-y	不停止 master_standy

**示例：****□ 重启数据库：**`gpstop -ar`**□ 停止 master\_standy：**`gpstop -m`**□ 快速停止 master：**`gpstop -M fast -a`**或者**`gpstop -af`**□ 重新加载配置：**`gpstop -u`

关闭数据库的流程基本跟启动数据库的流程一致，这里简单描述一下。

- 1) 前面还是一样的检查环境变量，数据库版本等。
- 2) 在 Master 获取 Segment 信息，然后单独停止 Master。
- 3) 停止 Standby Master。
- 4) 同样检查网络是否正常，然后并行通过 \$GPHOME/sbin/gpsegstop.py 脚本停止所有的 Segment，先停止 Primary Segment，再停止 Mirror Segment。
- 5) 停止数据库成功，并打印出简单的报告。

### 11.3 初始化系统脚本 gpinitSystem

系统初始化脚本 `gpinitSystem` 与其他脚本不一样，这个脚本是用 shell 写的。下面讲解其使用方法及初始化系统所必需的步骤。

使用 `gpinitSystem` 的语法如下：

```
gpinitsystem -c <gpinitsystem_config>
    [-h <hostfile_gpinitsystem>] [-m]
    [-B <parallel_processes>]
    [-p <postgresql_conf_param_file>]
    [-s <standby_master_host>]
    [--max_connections=<number>] [--shared_buffers=<size>]
    [--locale=<locale>] [--lc_COLLATE=<locale>]
    [--lc_CTYPE=<locale>] [--lc_MESSAGES=<locale>]
    [--lc_MONETARY=<locale>] [--lc_NUMERIC=<locale>]
    [--lc_TIME=<locale>] [--su_password=<password>]
    [-S] [-a] [-q] [-l <logfile_directory>] [-D]
```

Gpinitsystem 脚本每个参数的含义如表 11-3 所示。

表 11-3 gpinitsystem 脚本参数

参 数	说 明
-a	直接安装，不需要提示用户确认
-B	初始化系统的并发度，默认是 4 个，如果初始化的 Segment 比较多，建议将这个参数调大
-c	初始化系统的配置文件
-h	保存 Segment Hostname 的配置文件
-s	Master Standy 的机器名
-m	只创建 Master 节点
--su_password=   -e	超级用户 (gpadmin) 的初始化密码
-p	如果需要事前配置好 postgres.conf 文件，可以在这里指定。这样在初始化的时候，就可以指定这个配置文件为初始化数据库
-q	不要打印在屏幕上
-S	将 mirror 打散在多台机器上，在默认情况下，如果机器 1 上有 4 个主节点，那么它的 4 个备结点会全部在机器 2 上。加了这个参数，4 个备结点就会分别放在机器 2、3、4、5 上。
其他	语法中列举了很多修改系统的默认参数，可以在 postgres.conf 中配置

下面简单描述下执行 gpinitsystem 脚本的步骤。

- 1 ) 读取配置信息，检查所有机器上的端口是否被占用，数据目录是否存在等，并且初始化系统的配置信息。
- 2 ) 调用 initdb 命令创建 Master 数据库。
- 3 ) 使用 utility 模式启动 Master，修改 postgres.conf 文件，更新响应的配置信息。
- 4 ) 在 Master 上初始化集群配置信息表，如 gp\_segment\_configuration、gp\_filespace\_entry 等。
- 5 ) 并行创建 Primary Segment，之后并行创建 Mirror Segment（也是使用 initdb 命令）。
- 6 ) 初始化 Standy Master。
- 7 ) 停止 Master 并启动数据库。
- 8 ) 安装 gp\_toolkit 工具箱。
- 9 ) 修改超级用户密码（用户名默认是操作系统用户名）。

## 11.4 集群操作脚本 gpssh 和 gpsc

对于 Greenplum 这种分布式的系统，经常需要对每台机器进行一样的操作，为此，Greenplum 为使用者提供了 gpssh 和 gpsc 脚本来方便同时操作多台机器。

使用 gpssh 和 gpsc 之前，需要使用 gpssh-exkeys 将每一台机器都打通，相关内容在第 2 章入门的时候已经讲过了，这里不再重复。

下面讲解这两个脚本的参数及使用。

### 1. gpssh

gpssh 脚本参数的含义如表 11-4 所示。

表 11-4 gpssh 脚本参数

-e	结果显示执行的命令
-f	Hostname 的配置文件
-h	gpssh 支持单台机器的操作，使用这个命令指定 host
<bash_command>	gpssh 后面可以直接跟执行的命令

示例：

□ 在当前机器上执行：

```
$ gpssh -h localhost -e pwd
[localhost] pwd
[localhost] /home/gpadmin
```

□ 在所有机器上执行：

```
$ gpssh -f hostlist pwd
[10.20.151.10] /home/gpadmin
[ 10.20.151.7] /home/gpadmin
[ 10.20.151.8] /home/gpadmin
[ 10.20.151.9] /home/gpadmin
```



要使用这个脚本的上下方向键翻查命令功能，必须先安装 Python 的 readline 模块（Greenplum 自带），在 gpssh 中执行的命令会被保存在 `~/.gshist` 这个文件中。特别注意，gpssh 不能处理需要交互的命令，如果运行了带交互式的命令，窗口就会停止响应。

### 2. gpsc

gpsc 的功能是将文件复制到每一台机器上，例如：

```
gpsc -f hostfile_gpssh .bashrc =:/home/gpadmin
```

以上代码是将 .bashrc 复制到每一台在 hostlist 中的机器中。注意，=: 与后面连接的目录之间没有空格。



**注意** gpscp 这个脚本只能 scp 文件，不能 scp 目录。

## 11.5 数据库状态检查脚本 gpstate

脚本 gpstate 是用来显示正在运行的 Greenplum 实例的运行状态。Greenplum 由多个 PostgreSQL 数据库组成，通过 gpstate 可以观察到每个数据库运行的状态，例如：

- 1) 哪些数据库失败了。
- 2) Master 和 Segment 的配置信息。
- 3) 数据库使用的端口。
- 4) 主 Segment 和备 Segment 的对应关系。
- 5) 在 Gprecovery 恢复后，数据的恢复进度。

使用 gpstate 脚本的语法：

```
gpstate [-d <master_data_directory>] [-B <parallel_processes>]
         [-s | -b | -Q | -e] [-m | -c] [-p] [-i] [-f]
         [-v | -q] [-l <log_directory>]
```

gpstate 的参数如表 11-5 所示。

表 11-5 gpstate 的参数列表

参 数	说 明
-b	显示 Greenplum 简要的运行状态信息
-B	并行度，与前面的脚本参数一样，默认是 60
-c	显示主 Segment 与备 Segment 的对应关系
-d	Master 的数据目录，默认是读取环境变量 \$MASTER_DATA_DIRECTORY
-e	显示主备同步有问题的节点的详细信息
	一般的问题有：
	1) 有一个 Segment 在更改跟踪模式 (change tracking mode) 下运行，说明有一个节点失败了 2) 有一个 Segment 在重新同步模式 (resynchronization mode) 下运行，说明有主备正在同步 3) Segment 的角色与默认角色不一样，说明主备节点切换了，有些机器上的负载不均衡了，需要重新做负载均衡
-f	显示 Master Standby 的详细信息
-i	显示所有 Segment 数据库的版本
-m	列出所有的备 Segment 节点
-p	显示所有节点占用的端口

(续)

参数	说 明
-Q	快速检测，只检测在 Master 上的系统字典上的信息，不去收集 Segment 的信息
-s	获取整个数据库的详细信息
-v	Debug 模式，打印出运行的详细信息

下面通过脚本的解读来介绍 gpstate 从哪里获取信息，让读者对 Greenplum 的运作有一个比较深入的了解。

□ 获取节点的信息：gpstate -m 或 gpstate -c 或 gpstate -Q 或 gpstate -p。

以上的命令都是查询 Master 的 gp\_segment\_configuration 获取信息。

□ 获取数据库的总体信息：gpstate -s 或 gpstate -e 或 gpstate -b。

以上的这几个命令，除了查询表 gp\_segment\_configuration，还要在每个 Segment 上运行 \$GPHOME/sbin/gpgetstatususingtransition.py 脚本收集每个 Segment 实例的进程信息和端口信息，这个脚本主要用于判断进程是否运行正常（通过检查文件及端口的方式）。还有这个脚本会运行 gp\_primarymirror 命令，获取 Segment 的主备同步信息等，如：

```
$ echo "getMirrorStatus" | $GPHOME/bin/gp_primarymirror -h sdw2 -p 33001
Success: mode:PrimarySegment
segmentState:Ready
dataState:InSync
faultType:NotInitialized
databaseStatus:Up
postmasterState:run
isFullResync:0
resyncNumCompleted:0
resyncTotalToComplete:0
changeTrackingBytesUsed:105528264
estimatedCompletionTimeSecondsSinceEpoch:0
```

□ 获取每个子节点的版本信息：gpstate -i。

还是调用 gpgetstatususingtransition.py 脚本，在这个脚本中也调用 gp\_primarymirror，不过获取的内容不一样，这里获取的是版本信息，命令如下：

```
echo getVersion | $GPHOME/bin/gp_primarymirror -h 127.0.0.1 -p 33001
Success:PostgreSQL 8.2.15 (Greenplum Database 4.1.1.1 build 1) on x86_64-
unknown-linux-gnu, compiled by GCC gcc (GCC) 4.4.2 compiled on May 12 2011
18:07:08
```

这个信息与执行 select version(); 的结果是一样的。

## 11.6 数据库升级脚本 gpmigrate

在 Greenplum 的版本升级中，如果数据库版本升级的变化较小，方法很简单，只需要在

停止数据库后使用新版本的程序启动数据库即可，而且这个升级过程是可以回滚的，只需要将使用老版本的程序重启数据库即可（如从 4.1.1 升级到 4.1.8）。如果数据库版本变化比较大，比如修改了数据字段，增加了一些视图或函数功能，就需要使用 gpmigrate 脚本来升级数据库。

在数据库版本中，差异最大的应该是 Greenplum 3.3.x 到 Greenplum 4.0，而且从 Greenplum 3.3.x 升级到 4.1，必须先升级到 Greenplum 4.0，然后再从 4.0 升级到 4.1。为什么说 3.3.x 到 4.0 的差异最大呢，这个版本升级过程到底改变了哪些？

- Segment 节点的备份策略修改，3.3.x 是逻辑上的主备同步，而 4.0 将数据库改成基于文件的同步，而且支持增量同步。
- Greenplum 4.0 在 3.3.x 上修改了很多数据字典，包括增加字段、新增表。
- 增加了 gp\_toolkit 工具包。



**注意** 逻辑上的同步意味着，只要主备 Segment 中随便一个停止，整个集群就进入了只读模式，如果设置成 continue 模式，那么之后主备 Segment 之间数据将不再同步，后续需要停机全量恢复。Greenplum 4.0 是基于文件同步的，如果有任何一个 Segment 停止，系统还能够正常运行，下次恢复的时候比对主备之间的数据文件就可以恢复响应的数据。

其中最大的变化就是备份策略的修改。下面将解读 gpmigrate 升级脚本，让读者对升级的步骤有更加清晰的了解。其他版本之间的升级，方法不外乎如此。

首先简单介绍下升级的必备条件。

- 1) 保证登录到 gpmaster 节点，并且是 gp superuser (gpadmin)。
- 2) 在所有 Greenplum 节点上安装 Greenplum 4.0 的 bin 文件。
- 3) 把用户自定义的模块（如一些函数、包等）复制到 Greenplum 4.0 的相应目录下。
- 4) 把一些额外增加的文件或文件夹复制或保存，因为只有 Greenplum 需要的文件才会被 gpmigrator 这个命令保存。
- 5) 在所有的数据库中运行 vacuum，并且删除所有的日志文件，这是非必需的，只是为了节省空间。
- 6) 如果存在 gp\_jetpack 的 schema，将其删除，不要在系统中用 pg\_ 或 gp\_ 前缀的 schema，如果存在，将其重命名。
- 7) 在 Greenplum 3.x 版本中建表（分区表、列存储）时，如果使用了 OIDS=TRUE，将其修改为 OIDS=False，因为前者在 Greenplum4.0 中不会被支持。
- 8) 使用 gpcheckcat 命令检验系统目录的完整性。
- 9) 备份原有的数据 (gpfdist 或 ZFS 快照)。
- 10) 删除 master standby 节点 (gpinitstandby -r)。
- 11) 关闭现有系统 (gpstop)。

12) 把环境升级为 Greenplum 4.0, 修改 `~/.bash_profile`。

13) 通知所有数据库用户升级过程中数据库不可用。

下面介绍具体的升级步骤。

1) 获取系统环境变量参数并解析传入参数。

2) 检查参数配置是否合理 (从 `postgresql.conf` 获取一些参数)。



**注意** 接下来进入正式的升级步骤 (第一阶段), 下面所有的 Segment 都是指 Primary Segment。

3) 检查版本是否可升级 (有些版本之间不能跳级, 比如 Greenplum 3.3.x 不能直接升级到 4.1)。

4) 检查 `gpmigrator/state` 文件 (这个文件会记录上次升级到哪个阶段), 判断以前是否运行过 `gpmigrator`, 如果运行过, 则分两种情况处理。

如果是 '`BACKUP_VALID`', 会回滚数据。

如果是 '`NEWDB_VALID`', 会继续升级步骤。

5) 从旧的 Greenplum 中获取我们所需要的全部信息: Segment 信息、数据库信息、各节点是否可连接等。

6) 创建升级用户 `gpmigrator`。

7) 检查是否有 Greenplum 4.0 不支持的索引类型 (hash、gin 索引), 检查旧数据库的字符集在 Greenplum 4.0 版本中是否不再支持, 如果有, 则退出, 不能升级。

8) 在每个 Segment (包括 Master) 的数据目录下创建 `upgrade` 和 `backup` 文件夹, 用于存放升级过程中的临时文件。

9) 对每个数据库执行 `gpcheckcat` 命令, 检查数据库的数据字典一致性, 如果发生不一致且无法修复, 则报错退出, 执行命令如下:

```
$GPHOME/bin/lib//gpcheckcat -p port -U gpadmin -B 16 dbname
```

10) 修改 `pg_hba.conf` 文件, 除了 `gpmigrator` 用户, 其他用户不能连接 (此过程会备份 `pg_hba.conf` 和 `pg_ident.conf` 文件)。

11) 检查是否有其他连接或 SQL 正在运行, 有则重启 Greenplum。

12) 创建保存配置信息的文件, 在之后的脚本中会使用到这些文件。

`gp_databases`: 写入数据库的 `oid` 和 `name`。

`gp_array_config`: 写入 `gparray` 类中的信息。

`gp_config`: 写入在 `self.ExtractInfo()` 中的 `config` 数组信息。

13) 从数据字典和数据目录中获取信息, 生成骨骼 (源码注释中写的是 `skeleton system`)

在每个数据库中都执行下面这个 SQL, 将结果写入 `mapfile` (命名为 Segment 标号 + \_

catfiles)。

```

SELECT d.oid as datoid,
       c1.relname as rel1, c1.relfilenode as node1,
       c2.relname as rel2, c2.relfilenode as node2,
       c3.relname as rel3, c3.relfilenode as node3
  FROM pg_class c1
    left outer join pg_namespace n
      on (c1.relnamespace = n.oid)
    left outer join pg_class c2
      on (c1.reltoastrelid = c2.oid)
    left outer join pg_class c3
      on (c2.reltoastidxid = c3.oid),
       pg_database d
 WHERE d.datname = 'dbname' and
       (nspname = 'pg_catalog' or
        nspname = 'information_schema') and
       c1.relkind in ('r', 'i') and
       not c1.relisshared;

```

□ 将一些数据目录文件名字也写到该外部文件 mapfile 中。

14) 生成几个 SQL 文件，在后续会使用。

□ aoseg\_template0\_rewrite.sql，用于写入 vacuum freeze；

□ persistent\_master.sql，用于写入 select gp\_persistent\_build\_db(false);

□ persistent\_segment.sql，用于写入 select gp\_persistent\_build\_db(true);

这里提示一下，有 Mirror Segment 的时候参数为 true，否则为 false。

15) 关闭数据库。

16) 使用 pg\_controldata 在每个主 Segment 上运行，获取下一个 OID 的值：

```
pg_controldata 'segment 的数据目录' | grep Latest_checkpoint's_NextOID
```

17) 开始创建 upgrade 文件夹内容（在 Step 8 中已经创建了目录），并且开始复制数据（在每个 Segment 和 Master 上），同时修改配置文件，配置文件格式在 Greenplum 4.0 版本中有所调整。

18) 运行 pg\_resetxlog，设置 highest oid 为在 Step 16 中获取到的值。

19) 通过新版本的 bin/lib/gpmodcatversion 脚本修改数据库版本信息。

20) 使用单用户模式启动每个 Segment，然后开始修改数据字典。

在 \$GPHOME/share/postgresql/upgrade 中保存着 upg2\_conversion32.sql.in 等升级文件。使用单用户模式启动 Segment 的命令如下：

```
postgres --single -O -c gp_session_role=utility -c gp_before_persistence_
work=true -c exit_on_error=true -E -D /gpdata/gp3.3/gpmaster/gpmigrator/
upgrade/gp-1 template1
```

21) 在 Master 上修改 pg\_filespace\_entry 并初始化 gp\_segment\_configuration 配置表，将

pg\_filespace\_entry 中的数据路径指向 upgrade 目录。

至此，大部分的升级操作已经完成，接下来就是升级 Mirror 和安装 gp\_toolkit，以及后续收尾操作。

22) 重建 Mirror Segmen。这一步在每个 Primary Segment 上都要执行，主要是调用 \$GPHOME/sbin/gpupgrademirror.py 这个脚本。这个脚本也挺复杂的，有几千行代码，主要的思想就是删除老的数据文件，然后从 Primary Segment 中复制文件到 Mirror 中。有兴趣的读者可以去详细分析下相关代码，篇幅问题，这里就不详细描述了。

23) 在每个 Segment 上生成 gp\_dbid 这个文件。

24) 至此，升级好的数据文件就在 upgrade 目录中了，这时需要开始还原这个目录。

创建 DATA\_DIR/upgrade → DATA\_DIR/backup 的硬连接。

创建 DATA\_DIR/upgrade → DATA\_DIR 的硬连接。

25) 还原 pg\_filespace\_entry 的数据目录配置。

26) 用 Greenplum 4.0 的环境启动数据库。

27) 使用 share/postgresql/gp\_toolkit.sql 文件在数据库中创建 gp\_toolkit 工具箱。

28) 删除 gpmigrator 用户，升级结束。

#### 升级注意事项：

1) 在升级过程中需要备份文件，因此需要大量的空间，如果此时本身数据库空间就快满了，那么升级必将失败，而且数据量大，复制文件的过程将十分缓慢。如果文件系统本身有备份（比如 ZFS 中有快照功能），建议修改 gpmigrator 将复制文件这步去掉。

2) 检查数据字典的一致性 gpcheckcat 脚本要检查的内容很多，如果数据库中数据文件很多或数据字典比较大，执行 gpcheckcat 脚本会很慢，如果系统运行时间比较长，建议将这个脚本单独取出来做，将升级时间平摊，数据字典不一致的现象还是很容易发生的，或无法修复，这些问题都会导致升级失败。

3) 在升级过程中，步骤非常多，涉及的环境也非常复杂，很容易出错，因此在升级前必须对脚本有足够的了解，以减少风险，同时，在升级前必须做充分的测试（建议复制两个 Segment 进行升级测试）。

4) 升级过程可能需要停机，这将会对业务有比较大的影响，因此要做好充足的准备，以防升级失败导致回滚。

5) 升级 Mirror 其实都是从 Primary 向 Mirror 复制数据，在实际升级过程中，可以将 Mirror 去掉，在 Primary 升级成功后，再使用 gpaddmirrors 脚本初始化 Mirror。

6) Greenplum4.0 相对于 3.3.x 在执行计划上有很多的优化，升级后少部分语句会发生执行计划出错，导致执行缓慢，这一点在升级过后要特别注意。

7) 版本变化越大，升级风险越高，必须非常谨慎。



在此强调，上述讲解的是从 Greenplum3.3.x 到 Greenplum4.0 的升级步骤，其他的版本之间的升级相对比较简单，读者可自行阅读相应的升级脚本，深入了解每个版本之间的差异。

## 11.7 参数修改脚本 gpconfig

在 Greenplum 中，有些参数在 Master 上修改就可以了，有些参数的每个 Segment 都要修改，如果 Segment 比较多，手动修改会非常麻烦。Greenplum 4.x 提供了这样的脚本——gpconfig，可以修改和查看 Greenplum 中参数的设置，从而很方便地修改 Segment 的 postgresql.conf 配置文件中的内容，比如端口、最大连接数等。

gpconfig 的用法如下：

```
gpconfig -c <param_name> -v <value> [-m <master_value> | --masteronly]
| -r <param_name> [--masteronly]
| -l
[--skipvalidation] [--verbose] [--debug]

gpconfig -s <param_name> [--verbose] [--debug]

gpconfig --help
```

gpconfig 也和其他脚本一样，通过以下几个环境变量去连接数据库：

PGHOST PGPORT PGUSER PGPASSWORD PGDATABASE

表 11-6 是 gpconfig 脚本中每个参数的含义。

表 11-6 gpconfig 脚本参数

参 数	说 明
-c   --change	要修改的参数名
-v   --value	要修改的参数值，Master 上的值与 Segment 上的值不一样，-v 表示 Segment 上的参数值
-m   --mastervalue	-m 表示 Master 上的参数值，这个参数必须与 -v 一起使用
-r	删除一个参数，后面跟着参数名
-l	列出所有可以通过 gpconfig 修改的参数列表
-s	显示当前 Master 和 Segment 上的参数值
--skipvalidation	跳过 gpconfig 允许的参数列表的检查，即不在 gpconfig -l 列表中的参数都可以设置，不管这个参数能否被数据库识别

示例：

□ 将 Master 的 work\_mem 改成 120MB：

```
gpconfig -c work_mem -v 120MB -masteronly
```

□ 在 Master 上将 max\_connections 修改成 10，在 Segment 上设置成 100：

```
gpconfig -c max_connections -v 100 -m 10
```

□ 删除 default\_statistics\_target 参数：

```
gpconfig -r default_statistics_target
```

□ 查看所有数据库实例的参数：

```
$ gpconfig -s port
GUC : port
Context: 2 Value: 33000
Context: 3 Value: 33001
Context: 0 Value: 33000
Context: 1 Value: 33001
Context: 5 Value: 33001
Context: 4 Value: 33000
Context: -1 Value: 2345
```

```
$ gpconfig -s max_connections
Values on all segments are consistent
GUC : max_connections
Master value: 25
Segment value: 125
```

## 11.8 数据库一致性检查脚本 gpcheckcat

前面介绍过：在升级脚本的过程中会调用 gpcheckcat 这个脚本，检查数据库数据字典的一致性，并且进行一些数据字典修复，这里将简单介绍几个数据字典一致性的检查问题。运行 gpcheckcat 脚本命令如下：

```
$GPHOME/bin/lib/gpcheckcat -v
```

(1) 随机分布类型表上不能有主键或唯一性约束

有唯一性需求的字段必须是分布键才能保证唯一性，如果数据字典中随机分布类型表带有唯一性约束，则数据字典就不一致了。检查“随机分布类型且有唯一性约束的表”的 SQL 如下：

```
select n.nspname, rel.relname, pk.conname as constraint
from pg_constraint pk
join pg_class rel on (pk.conrelid = rel.oid)
join pg_namespace n on (rel.relnamespace = n.oid)
join gp_distribution_policy d on (rel.oid = d.localoid)
where pk.contype in('p', 'u') and d.attrnums is null
```

## (2) 找出没有字段信息的表

这个 SQL 比较简单，只需要与 pg\_class 进行左连接，找出 pg\_class 中出现但 pg\_attribute 中没有出现的表：

```
SELECT relname, relkind, tc.oid as oid,
       reltoastreloid, reltoastidxid
  FROM pg_class tc left outer join
       pg_attribute ta on (tc.oid = ta.attrelid)
 WHERE ta.attrelid is NULL
```

类似的，可以检查其他表的外键关系的表中信息是否一致。

## (3) 检查临时 Schema 是否忘记回收

在 Greenplum 运行过程中可能会建立一些临时表，这些表存放在临时 Schema 中（schema 名为“pg\_temp\_ 数字”，后面的数字就是对应 pg\_stat\_activity 表中的会话 ID-sess\_id），临时表在当前会话结束后就会被删除掉。通过下面 SQL 就可以查出哪些临时 Schema 没有被删除：

```
SELECT distinct nspname as schema
  FROM (
    SELECT nspname, replace(nspname, 'pg_temp_', '')::int as sess_id
      FROM gp_dist_random('pg_namespace')
     WHERE nspname ~ '^pg_temp_[0-9]+'
  ) n LEFT OUTER JOIN pg_stat_activity x using (sess_id)
 WHERE x.sess_id is null
UNION
SELECT nspname as schema
  FROM (
    SELECT nspname, replace(nspname, 'pg_temp_', '')::int as sess_id
      FROM pg_namespace
     WHERE nspname ~ '^pg_temp_[0-9]+'
  ) n LEFT OUTER JOIN pg_stat_activity x using (sess_id)
 WHERE x.sess_id is null
```

## (4) 检查 Master 与 Segment 表的 Owner 不一致的问题

```
select distinct n.nspname, coalesce(o.relname, c.relname) as relname,
               a.rolname, m.rolname as master_rolname
  from gp_dist_random('pg_class') r
 join pg_class c on (c.oid = r.oid)
 left join pg_appendonly ao on (c.oid = ao.segrelid or
                                c.oid = ao.segidxid or
                                c.oid = ao.blkdirrelid or
                                c.oid = ao.blkdiridxid)
 left join pg_class t on (t.reltoastidxid = c.oid)
 left join pg_class o on (o.oid = ao.relid or
                                o.reltoastreloid = t.oid or
                                o.reltoastreloid = c.oid)
 join pg_authid a on (a.oid = r.relpowner)
```

```

join pg_authid m on (m.oid = coalesce(o.relpersistence, c.relpersistence))
join pg_namespace n on (n.oid = coalesce(o.relnamespace, c.relnamespace))
where c.relpersistence <> r.relpersistence

```

### (5) 检查 Master 和 Segment 索引不一致的问题

```

select distinct n.nspname, c.relname
from gp_dist_random('pg_class') r, pg_class c, pg_namespace n
where r.oid = c.oid and r.relhassindex <> c.relhassindex
and c.relnamespace = n.oid

```

### (6) 检查 gp\_persistent 表，确定主备 Segment 之间是否有数据不同步的问题

```

SELECT p.tablespace_oid, p.relfilenode_oid, p.segment_file_num,
       case when p.persistent_state = 0 then 'free'
             when p.persistent_state = 1 then 'create pending'
             when p.persistent_state = 2 then 'created'
             when p.persistent_state = 3 then 'drop pending'
             when p.persistent_state = 4 then 'abort create'
             when p.persistent_state = 5 then 'JIT create pending'
             else 'unknown state: ' || p.persistent_state
       end as persistent_state,
       case when p.mirror_existence_state = 0 then 'mirror free'
             when p.mirror_existence_state = 1 then 'not mirrored'
             when p.mirror_existence_state = 2 then 'mirror create pending'
             when p.mirror_existence_state = 3 then 'mirror created'
             when p.mirror_existence_state = 4 then 'mirror down before create'
             when p.mirror_existence_state = 5 then 'mirror down during create'
             when p.mirror_existence_state = 6 then 'mirror drop pending'
             when p.mirror_existence_state = 7 then 'mirror only drop remains'
             else 'unknown state: ' || p.mirror_existence_state
       end as mirror_existence_state
FROM gp_persistent_relation_node p
WHERE (p.persistent_state not in (0, 2)
       or p.mirror_existence_state not in (0, 1, 3))
       and p.database_oid in (
           SELECT oid FROM pg_database WHERE datname = current_database()
)

```

之后 gpcheckcat 脚本还会检查数据文件和数据字典中对应的文件列表是否一一对应。gpcheckcat 脚本中还有很多的数据字典一致性需要检查，这里就不一一介绍了，有兴趣的读者可以阅读这个脚本的源码，进而了解下脚本里面是如何修复这种数据不一致的（当然，不是所有的不一致都能修复）。

## 11.9 小结

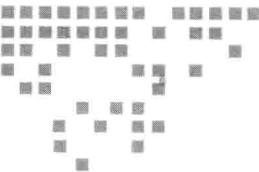
本章刚开始介绍了如何添加 PostgreSQL 的 Contrib 模块，以方便扩展 Greenplum 的功

能；之后主要介绍 Greenplum 提供的一部分脚本的使用，这些脚本除了二进制的 bin 文件外，大多都是用 Python 写的，也有少部分用 shell 写的。除了一些二进制的执行文件之外，其他都是可以看到源码的，通过这些代码，可以帮助我们更好地了解数据库的内部原理。

本章抽取了几个脚本对其执行步骤进行分析。

- 启动和关闭脚本 gpstart 和 gpstop，通过阅读源代码，可以帮助读者清楚掌握 Greenplum 启动和关闭过程的关键步骤。
- 初始化系统脚本 gpinitsystem，了解这个脚本可以帮助读者更好地了解 Greenplum 的架构，更容易找出安装过程中出现错误的原因。
- 数据库状态检查脚本 gpstate，通过这个脚本，可以监控 Greenplum 数据库的状态是否正常。
- 数据库升级脚本 gpmigrate，数据库升级是一个比较复杂的事情，尤其是环境变化等，学习这个脚本，会对升级更加有把握。
- 数据库一致性检查脚本 gpcheckcat，这是数据库升级中比较重要的一个脚本，理解这个脚本会让读者对 Greenplum 的数据字段有更加深入的理解。

这里只是让读者对这些脚本的内部原理有一个大概的了解。对于想要深入了解的读者，可以结合源码再读一遍。



## 第 12 章

## 备份及恢复策略

在数据库系统中，数据的备份及恢复必不可少，本章将通过 Greenplum 的两种主备策略及备份工具来讲述数据库的备份及恢复策略。

Greenplum 4.0 以上版本对 Segment 的备份策略有很大的修改，之前 Segment 的同步策略是基于逻辑复制的。在 4.0 以上版本中，Segment 之间的备份策略改成了基于文件的同步，下面先简要介绍两者在原理上的差异。

## 12.1 Greenplum 3.x

在 Greenplum 3.x 版本中，数据库的主备复制是基于逻辑复制的，如果系统中有一个节点挂掉了，那么整个系统将进入只读状态，因为再往里面写入，数据无法恢复，只能做全量拷贝。

### 1. 备份原理

前面已经介绍到，Greenplum 的 Segment 的同步是基于主备同步的，Mirror Segment 是 Primary Segment 的一个镜像，分别分布在不同的机器上可以应对有机器挂掉的情况。这一点在 Greenplum 所有版本中都是一样的（如图 12-1 所示）。

在 Greenplum 3.x 版本中，Primary Segment 和 Mirror Segment 的同步是基于逻辑同步的。何为逻辑同步，即如果系统在执行会产生数据的 SQL，如（`insert`、`create table`、`create as`）等语句之后，Greenplum 3.x 其实是将 SQL 在 Primary 和 Mirror 上各执行一遍，而非只是将最终结果传输到 Mirror 上，所以 Greenplum3.x 版本是基于逻辑上的复制，而非物理上的复制。

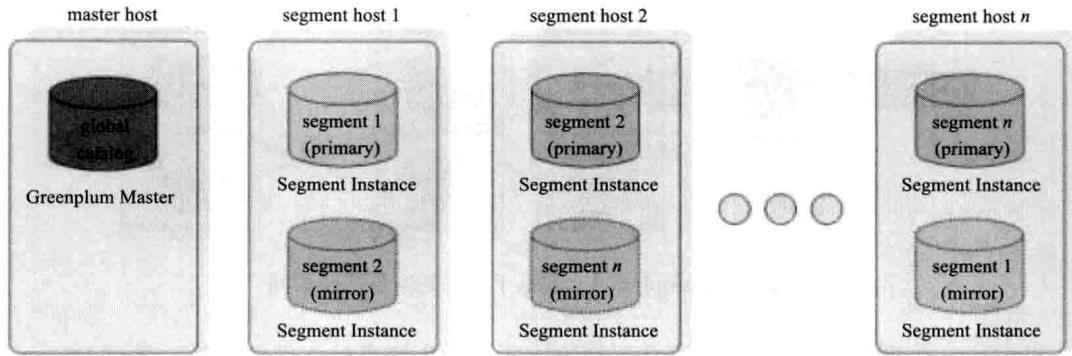


图 12-1 Greenplum 主备同步策略

**注意** 单纯的 Select 语句只会在 Primary Segment 上运行，因为查询语句不会产生新的数据，Mirror 上无须发生变更。

## 2. 失败节点恢复

由于是逻辑复制，当有一个 Segment 挂掉的时候，默认 Greenplum3.x 会变成 readonly 的状态，由参数 `gp_fault_action` 控制。此时 Greenplum 必须等待失败的 Segment 重新启动才能继续工作。

在将 `gp_fault_action` 改成 `continue` 模式的情况下，Greenplum 可以正常工作，但是有一个非常严重的问题，就是在 `continue` 模式下，失败的节点不会再接收新的数据，即使重新恢复，丢失的数据也不会再恢复。这个时候主备之间就出现了不一致的情况，如果想重新恢复一致的状态，必须进行全量拷贝，重建该 Segment 才行，代价很高。

## 12.2 Greenplum 4.x

在 Greenplum 4.x 版本中，主备之间的同步是基于文件级复制做的（如图 12.2 所示），在这种情况下，如果系统中有一个节点挂掉了，那么系统仍然可以继续运行，等到失败节点恢复的时候，再根据主备之间的文件差异来同步数据。

### 1. 备份原理

在正常情况下，Primary Segment 中发生的变更都会体现数据文件的变化，同步进程监控到此变化，就会记录文件的同步状态，并且将文件变化数据块发送到 Mirror 节点中。除非 Primary 发生故障，否则 Mirror 会一直处于非活动状态，如图 12-2 所示。

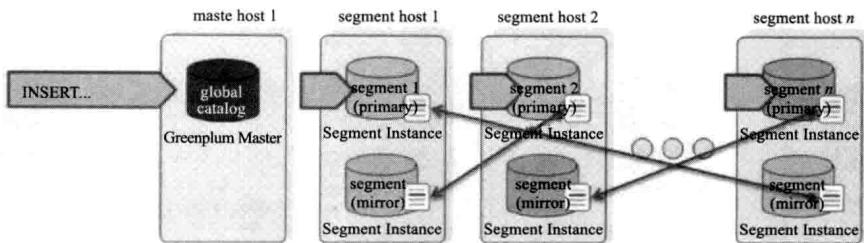


图 12-2 Greenplum 4.x 的基于文件的主备同步策略

当 Mirror 失败时，Primary 会记录下此阶段所有发生变更的文件数据块，等到 Mirror 节点恢复后开始同步。当 Primary 失败时，Mirror 节点会自动唤醒代替 Primary，此时两者角色会发生变化，系统状态会变为修改跟踪（Change Tracking）模式。待 Primary 恢复后，Primary 变成 Mirror 节点，并将这段时间的所有文件同步恢复。

## 2. 失败节点恢复

在 Greenplum 中，提供了 gprecoverseg 脚本用于对失败的 Segment 节点进行恢复。gprecoverseg 脚本参数说明如表 12-1 所示。

表 12-1 gprecoverseg 的参数列表

参 数	说 明
-a	不提示用户确认
-B	并行度，默认是 60
-d	Master 的数据目录，默认是读取环境变量 \$MASTER_DATA_DIRECTORY
-F	可选参数，会删除失败 Segment 上的所有数据，并全量数据同步整个 Segment
-i	指定恢复配置文件
-l	指定日志存放目录，默认是 ~/gpAdminLogs
-o	指定文件名，输出恢复配置文件信息，其中包括：当前失败的 Segment 以及它们默认的恢复目录，格式与 -i 参数指定的一样
-p	指定新增恢复的主机名，新增机器的配置必须跟其他的机器一样（如跟原有 Segment 一样的操作系统版本，都有 gpadmin 用户，数据目录已经被创建，并且网络配置一样，ssh 通道已经打通等）
-q	安静模式，命令输出信息只写日志文件，不会打印到屏幕上
-r	恢复所有的 Segment 到它们的默认角色，即如果主备发生过切换，则还原。执行这一步之前必须确认所有的 Segment 都是 Synchronized 状态的
-s	指定文件空间的配置文件，其中包括要恢复的 Segment 所在的 filespace 信息，配置文件的格式如下： pg_system=default_fselocation filespace1_name=filespace1_fselocation filespace2_name=filespace2_fselocation 如果在 Greenplum 中没有使用新增文件空间，那么默认就只有一个文件空间 pg_system
-S	输出默认的文件空间的配置信息
-v	显示版本信息

下面通过实际操作讲解如何使用 gprecoverseg 脚本。

### (1) 恢复所有失效的 Segment

```
$ gprecoverseg
```

直接运行 gprecoverseg 脚本，会将所有失效的节点打印在屏幕上。然后用户只需要选择“y”按钮即可恢复。gprecoverseg 脚本会检查 Primary 与 Mirror 之间的文件差异，这个操作会导致整个数据库卡住，无法执行命令，如果文件比较多，这一步会耗时比较久。待文件差异比较完之后，gprecoverseg 脚本就退出了，但这个时候，数据还没有完全同步成功，数据会在后台进行同步，同步状态会被修改成 Resynchronizing，同步完成后同步状态为 Synchronized。

在同步的过程中，可以通过 gpstate -m 命令查看同步的进度。

### (2) 还原所有 Segment 的角色

当 Primary 节点失效的时候，Mirror 节点会接替 Primary 节点，这时候主备的角色就发生切换。在角色发生切换之后，系统中某一些主机上的 Primary 就会比其他的主机多，会影响到负载均衡，通过 gprecoverseg -r 可以恢复所有 Segment 原来的角色。

首先，执行这一步之前必须确认所有的 Segment 都是 Synchronized 状态的：

```
$ gpstate -m
```

或者查看 gp\_segment\_configuration：

```
testDB=# select count(1) from gp_segment_configuration where status='d' or mode<>'s';
      count
-----
       12
(1 row)
```

等待数据全部同步完成之后，运行恢复命令：

```
$ gprecoverseg -r
```

同样的，如果重启数据库 (gpstop -afr) 也可以达到一样的效果。

### (3) 全量恢复

有时候 Segment 被破坏，可能导致增量同步失败，这个时候只能采用全量同步。在进行全量同步的时候，应该使用 gprecoverseg 先将能增量恢复的节点先恢复了，然后再开始全量恢复，减少全量恢复的数据量。

全量恢复命令如下：

```
gprecoverseg -F
```

通过 gpstate -m 查看恢复的进度，下面截取其中一段，其中 Estimated resync progress with mirror 这个指标即为恢复的百分比。

```

Segment Info
Hostname = sdw2
Address = sdw2
Datadir = /home/gpadmin/gpdata/gpdatap2/gpseg3
Port = 33001

Mirroring Info
Current role = Primary
Preferred role = Primary
Mirror status = Resynchronizing

Change Tracking Info
Change tracking data size = 103 MB

Resynchronization Info
Resynchronization mode = Full
Data synchronized = 27.3 MB
Estimated total data to synchronize = 10.3 GB
Estimated resync progress with mirror = 0.26%
Estimated resync end time = 2013-07-03 10:35:13

Status
PID = 30084
Configuration reports status as = Up
Database status = Up

```

 **注意** 在 Greenplum4.1 上查看恢复进度是使用命令 `gpstate-m`, 在 Greenplum4.3 中, 这一个功能变成了 `gpstate-s`, 内容是一样的, 只是用法的细微差异。

## 12.3 gp\_dump 和 pg\_dump

### 1. 数据备份

Greenplum 提供了两种备份的工具: `gp_dump` 和 `pg_dump`, 其中, `gp_dump` 是一个并行备份命令, 而 `pg_dump` 不支持并发备份。

`gp_dump` 的并发备份原理如图 12-3 所示, 在运行 `gp_dump` 的同时, Master 与所有的 Segment 节点都开始备份, 数据文件都存放在每个 Instance 所在的机器上。由于是并行备份的, 因此消耗的时间只跟数据量最大、时间最长的 Segment 有关, 而与集群中 Segment 的个数无关。

 **注意** Master 并不保存实际数据, 那么 Master 备份的数据是啥呢?

Master 主机上的备份文件包括所有的 DDL 命令生成的元数据信息, 以及一些全局的系统表 (如 `gp_segment_configuration`), 全局系统表的数据只会在 Master 上。

每个 Segment 上都会有对应的数据文件，其文件名包括 Segment 的 dbid 及一个 14 位的时间戳，这个时间戳在恢复数据的时候需要用到。

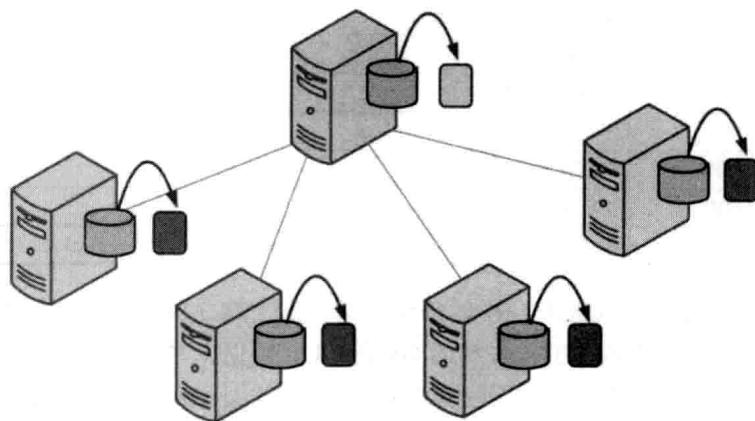


图 12-3 gp\_dump 备份原理

有一个与 `gp_dump` 的命令相关的命令 `gpcrondump`，该命令是 `gp_dump` 的一个封装，可以直接调用或从 `crontab` 中调用。

`gp_dump` 的语法如下，其中的脚本参数如表 12-2 所示。

```
gp_dump [OPTION]... [DBNAME]
```

表 12-2 gp\_dump 脚本参数

参 数	说 明
-a --data-only	只备份数据，不备份模式（数据定义）
-c --clean	在创建新的模式的时候，将原有的模式删除，即导出的数据定义中包括删除语句
-d --inserts	导出的数据格式为 Insert 语句，原先是用 copy 命令导出的纯数据文本
-D --column-inserts	导出的 Insert 语句中包括字段名
-E --encoding=ENCODING	导出的数据编码
-n --schema=SCHEMA	只导出某一个 schema
-o --oids	将表的 oid 字段也同时导出
-O --no-owner	不导出表的 owner 语句
-N --exclude-schema=SCHEMA	不导出某个 schema
-s --schema-only	只保留数据定义，不包括数据
-S --superuser=NAME	指定 superuser 用户
-t --table=TABLE	只导出匹配的表
-T --exclude-table=TABLE	不导出指定的表
-x --no-privileges	不导出权限信息

(续)

参 数	说 明
-h --host	Greenplum Master 主机名
-p --port	Greenplum Master 端口
-U --username	用户名
-W	强制使用密码
--gp-c	使用 gzip 压缩
--gp-d=BACKUPFILEDIR	指定备份的数据目录
--gp-r=REPORTFILEDIR	指定 report 的目录
--gp-s=BACKUPSET	指定 Segment 的 dbid 的列表，可以只备份某几个 Segments 的数据

而 pg\_dump 是与 PostgreSQL 一样的备份工具，不支持并发备份，在数据量比较大的情况下不切实际，这个工具多用于从常规的 PostgreSQL 数据库系统迁移到 Greenplum 系统的情况。下面通过一个简单的比较，让读者对 gp\_dump 和 pg\_dump 性能差异有一个比较直观的理解。例子中都只导出 member 表数据（该数据是随机生成的），Greenplum 集群为 4 台物理机，12 个 Segment，表大小大概为 11GB。

使用 pg\_dump 脚本导出，消耗时间为 52 秒：

```
$ time pg_dump -t member -c -f ~/member.dat
real    0m52.845s
user    0m2.968s
sys     0m7.883s
```

使用 gp\_dump 并行导出，消耗的时间为 22 秒：

```
$ time gp_dump --table=member --gp-d /home/gpadmin/gp_dump/ --gp-r=/home/
gpadmin/gp_dump/
# 中间数据结果略过
gp_dump utility finished successfully.
real    0m22.105s
user    0m0.005s
sys     0m0.004s
```

下面是使用 gp\_dump 之后产生的一个报告：

```
Greenplum Database Backup Report
Timestamp Key: 20130704075523
gp_dump Command Line: --table=member --gp-d /home/gpadmin/gp_dump/ --gp-r=/home/gpadmin/gp_dump/
Pass through Command Line Options: -t member
Compression Program: None

Individual Results
segment 7 (dbid 9) Host sdw3 Port 33001 Database testDB BackupFile
/home/gpadmin/gp_dump/gp_dump_0_9_20130704075523: Succeeded
```

```

segment 6 (dbid 8) Host sdw3 Port 33000 Database testDB BackupFile
/home/gpadmin/gp_dump/gp_dump_0_8_20130704075523: Succeeded
...
segment 0 (dbid 2) Host mdw Port 33000 Database testDB BackupFile
/home/gpadmin/gp_dump/gp_dump_0_2_20130704075523: Succeeded
Master (dbid 1) Host mdw Port 2345 Database testDB BackupFile /home
/gpadmin/gp_dump/gp_dump_1_1_20130704075523: Succeeded
Master (dbid 1) Host mdw Port 2345 Database testDB BackupFile /home
/gpadmin/gp_dump/gp_dump_1_1_20130704075523_post_data: Succeeded

gp_dump utility finished successfully.

```

在结果报告中有几个点需要注意。

**Timestamp Key** 是系统自动生成的时间戳字段，可以说是备份的唯一识别码。在进行数据恢复的时候，需要通过制定 **Timestamp Key** 来实现。

每台主机上对应的 Segment 节点都有对应的数据文件，文件名格式为：

```
gp_dump_0_<dbid>_< timestamp>
```

在这个文件中，只有一些参数配置和原表在该 Segment 上的数据，与 pg\_dump 单库导出格式基本一样（只导出数据），对应的导出日志文件为：

```
gp_dump_status_0_< dbid>_< timestamp>
```

在 Master 上，文件名的格式为：

```
gp_dump_1_<dbid>_< timestamp>
```

这个文件保存了表的定义等信息，没有原表的数据，如果表中有其他的相关对象，那么这些内容会被报错在 gp\_dump\_1\_<dbid>\_< timestamp>\_post\_data 这个文件中。

如果是全库备份，在 Master 上还有 gp\_catalog\_1\_<dbid>\_< timestamp>，用于保存系统日志表的结构和数据。gp\_catalog\_1\_<dbid>\_< timestamp> 则保存了数据库的创建语句。

gp\_dump 在每个 Segment（包括 Master）上启动 gp\_dump\_agent 的进程进行备份，在备份过程中，gp\_dump\_agent 会向 Master 的 gp\_dump 进程汇报，当所有的 Segment 都完成的时候，备份成功。

## 2. 数据恢复

和 gp\_dump 对应的并行恢复命令为 gp\_restore，这个命令通过 gp\_dump 产生时间戳来识别备份的集合，恢复数据库对象及数据到数据库中。gp\_restore 的参数基本与 gp\_dump 的参数对应，相对简单很多，读者可参考文档或命令帮助，这里不再列举。gp\_restore 的恢复原理如图 12-4 所示。

从图 12-4 上可以看出，在恢复的时候，数据库的 Segment 数量必须与原来的保持一致，如果是迁移数据库，那么为了能够并行恢复，目标数据库就必须与原有数据库的 Segment 一

致，否则无法使用 gp\_restore。

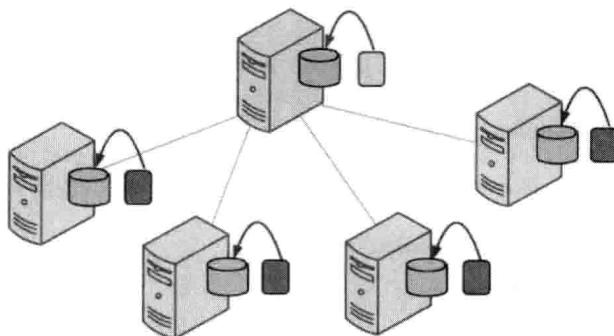


图 12-4 gp\_restore 恢复原理

要恢复上节使用 gp\_dump 导出的数据，可使用如下命令，其中 --gp-k 参数，即指定 gp\_dump 导出的时间戳：

```
gp_restore --gp-d /home/gpadmin/gp_dump/ --gp-r=/home/gpadmin/gp_dump/ --gp-k=20130704075523
```



在 Greenplum 官方文档中也介绍了如何将数据恢复到 Segment 上不同的数据库中，简单来说，就是将全部数据汇总到 Master 上，然后将所有操作交由 Master 执行，这样就将并行恢复改成串行恢复，性能瓶颈在 Master 上，所有数据必须流过 Master，性能很差，原理如图 12-5 所示。

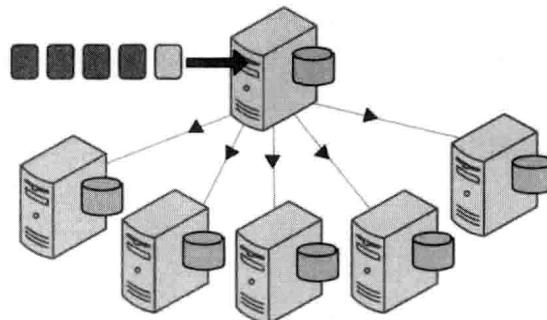


图 12-5 使用 gp\_dump 对数据进行非并行恢复

## 12.4 Greenplum Master 备份策略

Greenplum Master 同样也有一个备份，叫做 Standby Master，Primary Master 与 Standby

Master 之间通过 WAL 日志来进行同步，后台有一个运行在 Standby 主机上的名为 `gpsyncagent` 的进程负责数据的同步，如图 12-6 所示。Standby 在正常情况下都处于非活动状态，当 Master 发生故障无法恢复时，可激活 Standby 继续提供服务。

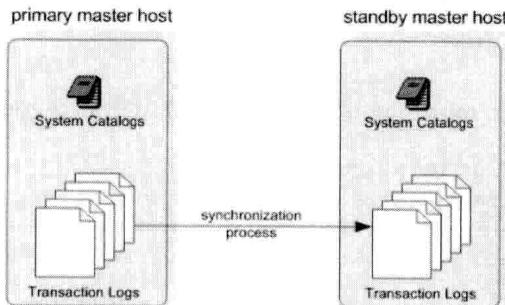


图 12-6 Greenplum Master 备份策略

### 12.4.1 增加 Standby Master

在初始化 Greenplum 的时候 (`gpinitsystem`) 可以配置 Standby——使用 `-s` 参数，具体用法可查询 `gpinitsystem` 脚本帮助。

如果在原有系统中没有设置 Standby，可以增加 Standby，如果 Standby Master 已经失效，可以将其删除。本节将讲述如何增加及删除 Standby。

增加 Standby Master 的步骤如下（在 Greenplum4.1.8 中测试通过，在 Greenplum 4.3 中的差异在本节最后会介绍）。

**Step1：**确保在 Standby 主机上已经正确配置好环境，如 `gpadmin` 用户、安装文件、`ssh` 通道互信、数据目录等。

**Step2：**运行 `gpinitstandby` 脚本，命令如下：

```
gpinitstandby -s mdw2
```

在初始化的过程中，Greenplum 会先用 `gpstop -a` 停止数据库，然后使用 utility 模式启动 Master，在数据字典 (`gp_segment_configuration`) 中增加 Standby Master 的配置，关闭 Master。接着将文件系统中的数据从 Master 上复制到 Standby Master 上，最后重新启动数据库即可。



**注意** 删除 Standby Master，则使用 `-r` 参数，如 `gpinitstandby -r`。

## 12.4.2 重新同步 Standby Master

如果系统中已经有 Standy Master，但是这个节点已经不同步了，那么可使用 -n 参数进行重新同步，使用这个参数的做法与初始化 Standby 基本一致，不同点是，数据字典中已经有了相关配置，重新同步不用再更新。

```
gpinitstandby -n
```

在数据字典 gp\_master\_mirroring 中，可以查看 Master 主备之间的同步状态，如：

```
testDB=# select * from gp_master_mirroring ;
summary_state | detail_state | log_time | error_message
-----+-----+-----+-----
Synchronized | | 2013-07-28 01:26:10+08 |
(1 row)
```

## 12.4.3 启用 Standby Master

如果 Master 已经失败且无法恢复（例如，不能重启），这时候就需要激活 Standby Master，替代原先 Master 的功能，步骤在官方文档中已经描述得很清楚了，这里将真实模拟 Master 挂掉的情况，并切换到 Standy Master 上。

首先，在一个已经启动的集群上，确保 Standy Master 已经配置并处于同步状态。

使用 gpstate -f 命令，查看 Master 状态，从 Summary State 中可以看出 Standy Master 正处于 Synchronized (同步) 状态，结果如图 12-7 所示。

```
[INFO]:-Starting gpstate with args: -f
[INFO]:-local Greenplum Version: postgres (Greenplum Database) 4.1.1.1 build
[INFO]:-Obtaining Segment details from master...
[INFO]:-Standby master details
[INFO]:-
[INFO]:-  Standby address      =
[INFO]:-  Standby data directory = /data/gpmaster/gpseg-
[INFO]:-  Standby port          =
[INFO]:-  Standby PID           =
[INFO]:-  Standby status        = Standby host passive
[INFO]:-gp_master_mirroring table
[INFO]:-
[INFO]:--Summary state: Synchronized
[INFO]:--Detail state:
[INFO]:--Log time: 2013-07-29 00:15:28+08
[INFO]:-
```

图 12-7 使用 gpstate 查看 Standy Master 状态

在确保 Master 处于同步状态之后，我们将停止 Master，模拟 Master 失效的情况。可以杀掉 Master 进程，或是用 gptop-m 命令关闭 Master，再或者直接关闭 Master 主机的电源，这里使用 gpstop -m。

```
$ gpstop -m
```

在 Master 关闭后，登录到 Standy Master 主机上，运行 gpactivatestandby 命令，激活

Standby。

```
gpactivatestandby -d $MASTER_DATA_DIRECTORY
```

激活过程如图 12-8 所示。

```
[INFO]::Stopping gpsync process...
[INFO]::Successfully shutdown sync process
[INFO]::Starting standby master database in utility mode...
[INFO]::Reading current configuration...
[INFO]::Updating catalog...
[INFO]::Database catalog updated successful
[INFO]::Stopping database...
[INFO]::Starting database in production mode...
[INFO]::-----
```

图 12-8 激活 Standy Master

接着，对整个数据库进行 ANALYZE，收集统计信息：

```
$ psql -d testDB -c "ANALYZE"
```

此时 Standy Master 已经启动完毕，但是所有的客户端连接的 IP 都需要修改成 Standy Master 的 IP。

如果需要重新切换到原有的 Master 上，那么使用 gpinitstandby 脚本，将原有的 Master 变成 Standby Master，然后再将其激活重新成为 Master 即可。



**注意** Greenplum 4.3 对 Master 主备同步进行了优化，使得主备同步更加稳定及易用，其中原来查询主备同步的数据字典 `gp_master_mirroring` 被删除了，替之以 `pg_stat_activity` 这个视图。读者可以通过下面 SQL 查询主备同步状态：

```
testDB=# SELECT procpid, state FROM pg_stat_replication;
 procpid | state
-----+-----
 26421 | streaming
(1 row)
```

在使用 `gpstate -f` 查看的时候，如果显示下面提示，则 Master 主备为同步状态：

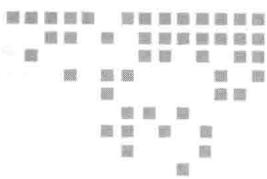
```
WAL Sender State: streaming
Sync state: sync
```

## 12.5 小结

本章介绍了在 Greenplum3.x 版本和 4.x 版本中，采用的 Segment 数据备份原理的差异，并且介绍了如何使用 `gprecoverseg` 脚本对失败节点进行恢复。之后介绍数据导出工具 `gp_dump` 和 `pg_dump`，并简单比较了它们的性能差异。最后介绍了 Master 的备份策略。

除了可以使用以上介绍的备份策略之外，还可以使用其他的备份策略，例如通过数据冗余，文件系统级别的数据备份（如使用 ZFS）等。

保证数据安全不丢失的重要性不言而喻，因此，用户在使用 Greenplum 之前，必须要对其数据备份策略有所了解，通过实验，建立起各种异常情况的应急方案，这将会大大降低问题发生之后的危害。



Chapter 13

## 第 13 章 数据库扩容

数据库在使用的过程中会随着数据量的增加而需要扩容，一般需要扩容的原因如下。

- 历史数据量增加，磁盘空间不足。
- 计算的数据量增加，计算性能跟不上（CPU 或磁盘 IO 吞吐限制）。
- 网络传输量增加，网卡限制。

在分布式数据库扩容中，最简单的就是升级机器硬件。升级机器硬件比较通用，与 Greenplum 的关系并不大，本章不再描述。另外一种扩容就是增加机器，将数据迁移到新的机器上。在 Greenplum 中，增加机器的方法有两种：1) 迁移计算节点；2) 增加计算节点。

迁移计算节点比较简单，而且代价较低，在笔者公司，一般采用这种方法，但是需要在数据库搭建的时候有容量的规划。为了增加计算节点，Greenplum 提供了 `gpexpand` 扩容脚本，还有一种方法就是对数据库机器进行升级，下面对这几种方法分别进行描述。

### 13.1 迁移计算节点

假设 Greenplum 的每一台机器上都有 4 个 Segment（两主两备），扩容的时候可以增加一倍的机器，将每台机器改成有两个 Segment（一主一备）。

如图 13-1 的扩容方案所示，Greenplum 在扩容前有两台机器（SDW1 和 SDW2），每台机器是两主两备，扩容时增加了一倍的机器（SDW3 和 SDW4），将 SDW1 中的一主（Seg2）一备（Seg4）迁移到 SDW3 中，同样的，将 SDW2 的两个 Segment 迁移到 SDW4 中。假设所有机器的配置都是一样的，那么 Greenplum 则成功从 2 台机器扩容到了 4 台机器，性能跟存储都翻了一倍。

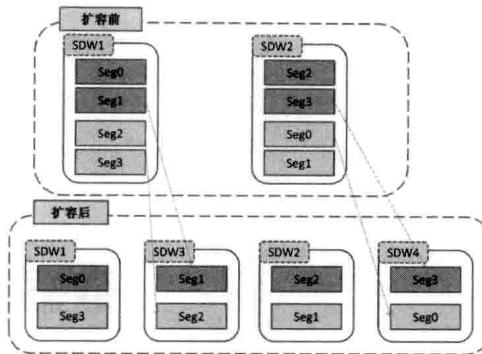
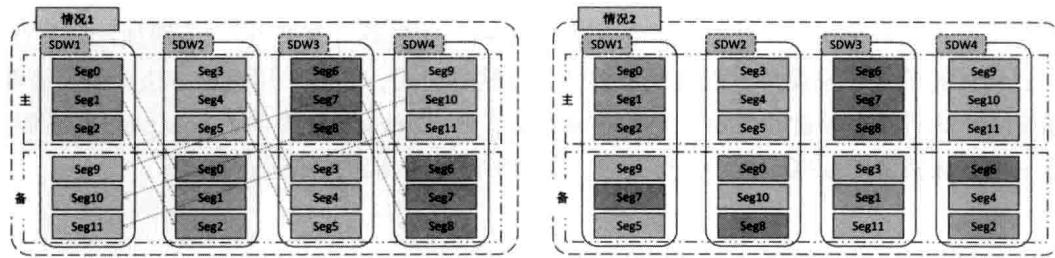


图 13-1 迁移节点扩容示例

图 13-1 只演示了最简单的方法，如果一台机器是三主三备，则增加的机器数就必须是原来的 1/2 (改成一机两主两备)，或者是原来的 2 倍 (改成一机一主一备)。如果是四主四备，情况就更多一些，读者可以按照这种方法继续推算。

### 13.1.1 两种备份方案

在 Greenplum 中，备份方案有两种，一种是 Grouped Mirror，一台机器的 Primary Segment 对应的 Mirror Segment 也同样放在同一台机器上，如图 13-2a 所示。另外一种是 Grouped Mirror，一台机器的 Primary Segment 对应的 Mirror Segment 按顺序打散在邻接的机器上，如图 13-2b 所示。



a) 主备设置方案 1—Grouped Mirror

b) 主备设置方案 2—Spread Mirror

图 13-2 Greenplum 主备设置的两种备份方案策略

下面介绍主备设置的两种备份方案的优缺点。

**Grouped Mirror：**假设 sdw1 死机，那么 sdw2 的 Mirror 将会作为 Primary 继续提供服务，这样 sdw2 上就会有 6 个 Primary Segment，由于 Primary 的消耗要比 Mirror 多很多，所以 sdw2 容易成为性能瓶颈。而 Spread Mirror 则可以将 sdw1 的 Mirror Segment 分布到另外 3 台机器上，这样就相当于剩下的 3 台机器每台机器有 4 个 Primary，刚好能够做到负载均衡。

**Spread Mirror**：从图 13-2b 中可以看出，如果 sdw1 死机，那么这时候系统不能再有任何一台机器死机，否则由于没有完整的 Primary Segment，整个数据库就会死机。而对于 **Grouped Mirror** 策略，此时 sdw3 再死机，系统仍可正常工作（sdw2 跟 sdw4 不能再死机）。

在一般情况下，在 Greenplum 中同时死机两台机器的概率很低，但是一台机器死机的概率较高，笔者建议用户部署 Greenplum 的时候，采用 Spread Mirror。



**注意** Greenplum 初始化的时候，默认是 Grouped Mirror，在初始化数据库的时候，可以加上 -S 参数，即可使用 Spread Mirror，使用方法如下：

```
$ gpinitsystem -s mdw2 -c initgp_config -S
```

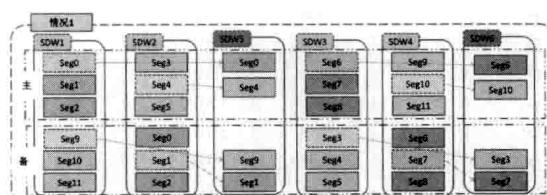
### 13.1.2 数据迁移实战

下面通过一个实验来介绍在实际中节点是如何迁移的，对于前面介绍的两种不同的备份策略，要耐心分析好需要迁移的节点，本例子在 Greenplum 4.1.8 中测试通过。

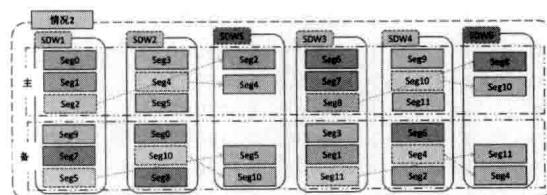
数据迁移的原则如下。

- 1) 迁移时一定要保证同一个数据节点的 Primary Segment 和 Mirror Segment 不能在同一个节点上。
- 2) 在同一个机器上，不能迁移使用同一个端口的 Segment。

图 13-3 则是对应 13.1.1 节中的两种备份策略。



a) Grouped Mirror 数据迁移方案



b) Spread Mirror 数据迁移方案

图 13-3 对应备份方案的两种数据迁移方案

接下来对图 13-3a 展示的 Grouped Mirror 的数据迁移方案为例，进行数据迁移实战，数据迁移在实际操作上有两种方案。

方案 1：如图 13-1 所述，停机将文件复制到对应的机器上（最简单的，可以使用 scp 命令），复制完了之后，修改 gp\_segment\_configuration 中对应节点的 hostname。

方案 2：首先修改 gp\_segment\_configuration，将要迁移的 Segment 对应的 Mirror 节点的 hostname 和 address 修改成新增机器对应的 hostname 和 address，使用 gprecoverseg -F 全量恢复 Mirror 节点。然后再用同样的方法迁移 Primary 节点。

下面通过操作实战来实现上面两个方案。

### 方案 1：使用 scp

1) 新增的机器已经安装好了，包括操作系统、网络及相关的软件等。然后利用 gpssh-exkeys 脚本，将 sdw5 和 sdw6 两台机器，跟原有的机器 ssh 通道打通。

原有 Greenplum 的集群配置如下：

```
testDB=# select dbid,content,role,mode,hostname,port from gp_segment_
configuration order by dbid;
dbid | content | role | mode |hostname| port
-----+-----+-----+-----+-----+
 1 |      -1 | p   | s    | mdw     | 2345
 2 |       0 | p   | s    | sdw1    | 33000
 3 |       1 | p   | s    | sdw1    | 33001
 4 |       2 | p   | s    | sdw1    | 33002
 5 |       3 | p   | s    | sdw2    | 33000
 6 |       4 | p   | s    | sdw2    | 33001
 7 |       5 | p   | s    | sdw2    | 33002
 8 |       6 | p   | s    | sdw3    | 33000
 9 |       7 | p   | s    | sdw3    | 33001
10 |      8 | p   | s    | sdw3    | 33002
11 |      9 | p   | s    | sdw4    | 33000
12 |     10 | p   | s    | sdw4    | 33001
13 |     11 | p   | s    | sdw4    | 33002
14 |       0 | m   | s    | sdw2    | 43000
15 |       1 | m   | s    | sdw2    | 43001
16 |       2 | m   | s    | sdw2    | 43002
17 |       3 | m   | s    | sdw3    | 43000
18 |       4 | m   | s    | sdw3    | 43001
19 |       5 | m   | s    | sdw3    | 43002
20 |       6 | m   | s    | sdw4    | 43000
21 |       7 | m   | s    | sdw4    | 43001
22 |       8 | m   | s    | sdw4    | 43002
23 |       9 | m   | s    | sdw1    | 43000
24 |      10 | m   | s    | sdw1    | 43001
25 |      11 | m   | s    | sdw1    | 43002
(25 rows)
```

使用 gpssh-exkeys 命令打通 ssh 通道，使所有机器互信：

```
$ cat segments
sdw1
sdw2
sdw3
sdw4
sdw5
sdw6
$ gpssh-exkeys -f segments
```

2) 在新增的两台机器上创建对应的数据目录，并利用 gp\_segment\_configuration 和 pg\_filespace\_entry 生成 scp 命令：

```
select 'scp -r ' || hostname || ':' || b.fselocation || ' sdw5:' ||
       b.fselocation
  from gp_segment_configuration a
    join pg_filespace_entry b on a.dbid = b.fsdbid
   where (content in (0, 4) and role = 'p') or (content in (9, 1) and role = 'm')
union all
select 'scp -r ' || hostname || ':' || b.fselocation || ' sdw6:' ||
       b.fselocation
  from gp_segment_configuration a
    join pg_filespace_entry b on a.dbid = b.fsdbid
   where (content in (6, 10) and role = 'p') or (content in (3, 7) and role = 'm');
```

3) 停止集群并开始复制 Segment 上所有的数据文件。

```
$ gpstop -af
$ scp -r sdw1:/data/gpdataapl/gpseg0 sdw5:/data/gpdataapl/gpseg0
...
```

4) 等待所有 scp 结束。

5) 只启动 Master (gpstart -m)，修改 gp\_segment\_configuration，修改前要先备份这个表，免得误操作导致数据无法恢复。

```
$ gpstart -m
$ PGOPTIONS="-c gp_session_role=utility" psql
testDB=# update gp_segment_configuration set hostname= 'sdw5',address='sdw5'
where (content in (0,4) and role = 'p') or (content in (9, 1) and role = 'm');
UPDATE 4
testDB=# update gp_segment_configuration set hostname= 'sdw6',address='sdw6'
where (content in (6,10) and role = 'p') or (content in (3, 7) and role = 'm');
UPDATE 4
testDB=# \q
$ gpstop -m
```

 **注意** 在 Greenplum4.1 之前版本，数据字典可以更新，但是 Greenplum4.2 之后的版本，数据字典都不能更新了。但是 Greenplum4.2 之后的版本自带了一个隐藏参数 `allow_system_table_mods`。在会话级别运行 `set allow_system_table_mods='dml'` 命令设置下，数据字典就可以更新的，更新数据字典有风险，操作必须十分谨慎。

6) 启动 Greenplum，并建表验证数据库是否正常，检查 Segment 节点的状态，如果有节点不同步，则运行 `gprecovery` 进行同步。

```
$gpstart -a
$ psql
psql (8.2.15)
Type "help" for help.
```

```
testDB=# create table test_1 as select * from pg_class distributed randomly;
SELECT 379
```

至此，数据库迁移成功。

 **注意** 一般迁移数据的时候，使用 `scp` 性能较差，可以编写一些并发传输的工具进行复制，并加入数据校验，这样可以大大提升性能及安全性。

## 方案 2：利用 `gprecoverseg -F`

在上一章中讲到，`gprecoverseg -F` 是在某个 Segment 无法进行增量恢复时用来进行全量恢复的工具，利用这个全量恢复工具可以避免使用 SCP 对文件进行复制，而直接使用 Greenplum 自带的数据同步工具，这种方案相对简单一些。

- 1) 与方案 1 的操作一致。
- 2) 只启动 Master (`gpstart -m`)，修改 `gp_segment_configuration`，修改前要先备份这个表，免得误操作导致数据无法恢复。

```
$ gpstart -m
```

使用 utility 模式登录 Master：

```
$ PGOPTIONS="-c gp_session_role=utility" psql
```

执行下面的 SQL 语句，修改 `gp_segment_configuration`。

将要迁移的节点的 `status` 标记成 d (即 down)：

```
update gp_segment_configuration set hostname='sdw5' , address='sdw5',status='d'
where (content in (0, 4) and role = 'p') or (content in (9, 1) and role = 'm');
update gp_segment_configuration set hostname='sdw6', address='sdw6',status='d'
```

```
where (content in (6, 10) and role = 'p') or (content in (3, 7) and role = 'm');
```

□ 然后将被标记成 down 的 Primary 节点对应的 Mirror 切换成 Primary:

```
update gp_segment_configuration set role=CASE WHEN preferred_role='p' THEN 'm'
ELSE 'p' END where (content in (0, 4));
update gp_segment_configuration set role=CASE WHEN preferred_role='p' THEN 'm'
ELSE 'p' END WHERE (content in (6,10));
```

□ 将同步状态 mode 修改为 c (Change Tracking) 模式, 即该节点不同步:

```
update gp_segment_configuration set mode='c' where (content in (0,4,9,1) AND
role = 'p');
update gp_segment_configuration set mode='c' where (content in (6,10,3,7) and
role = 'p');
```



**注意** 由于迁移的节点都没有相同的 Segment, 因此可以一起操作, 如果有相同的 Segment, 则必须分两次完成。

3) 使用 gpstart -a 启动数据库, 这个时候由于有 8 个数据库被迁移走, 被标记成失败状态, 故启动时自动跳过这 8 个节点, 如图 13-4 所示。

```
[INFO]: Process results...
[INFO]: -----
[INFO]: Successful segment starts = 16
[INFO]: Failed segment starts = 0
[WARNING]: Skipped segment starts (segments are marked down in configuration) = 8 <<<<<<
[INFO]: -----
[INFO]: -----
[INFO]: Successfully started 16 of 16 segment instances, skipped 8 other segments
[INFO]: -----
```

图 13-4 Greenplum 启动时有 8 个节点失败

4) 全量恢复这 8 个节点:

```
$ gprecoverseg -F
```

在同步过程中, 可以使用 gpstate -s 查看数据恢复进度, 如图 13-5 所示。

Port	= 43601
Current role	= Primary
Preferred role	= Mirror
Mirror status	= Resynchronizing
Change Tracking Info	
Change tracking data size	= 144 MB
Resynchronization Info	
Resynchronization mode	= Full
Data synchronized	= 1.94 GB
Total logical data to synchronize	= 7.98 GB
Estimated sync progress with mirror	= 19.7%
Estimated resync end time	= 2013-07-21 15:47:27
Status	
PID	= 95916
Configuration reports status as	= Up
Database status	= Up

图 13-5 使用 gpstate -s 查看全量数据恢复进度

此时数据在恢复过程中, 数据库可用, 就是比较慢, 等待数据恢复完成即可。



在迁移完数据之后，还需要对老的数据进行清理。为了避免误操作，在清理的时候，可以先关闭数据库，然后将要清理的 Segment 的数据目录重命名，之后启动数据库。如果数据库启动成功，则刚刚重命名的节点就是要删除的节点，没有操作失误，这样就可以安心删除重命名之后的数据文件了。

## 13.2 增加计算节点

对于有 Mirror 的 Greenplum 集群来说，图 13-2 描述的两种 Mirror 的分布策略对新增的主机数有如下的限制。

- Grouped Mirror：新增的主机数必须大于等于 2，确保新增 Primary Segment 的 Mirror Segment 不在同一台机器上，如图 13-6 所示。
- Spread Mirror：新增的主机数至少要比每个主机上 Primary Segment 的数量大 1，这样才能确保 Mirror 可以平均分配在其他的 Segment 节点上，如图 13-7 所示。



以上限制是在不迁移原有的 Segment 情况下，如果同时结合上一节迁移节点的方案进行，则只需要保证每一台主机上都没有相同 Segment 的主备节点，不受这个限制。

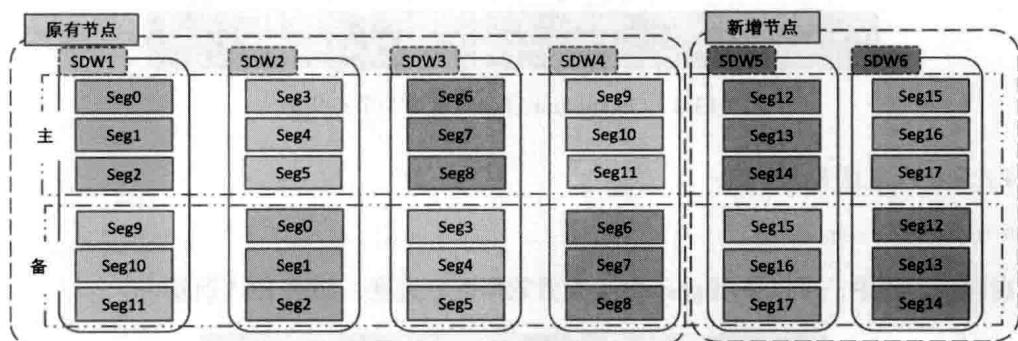


图 13-6 Grouped Mirror 的增加节点方案

Greenplum 管理员手册对 gpexpand 的一些准备工作进行了很清楚的讲解，这里不再复述。

Gpexpand 脚本有两个阶段，第一阶段是 Segment 初始化，第二个阶段是表重分布。

在 Segment 初始化阶段，gpexpand 接受一个配置文件，在这个配置文件中指定新的 Segment 的数据目录、对应的 dbid，以及一些其他的参数。用户可以自己手工创建这个配置文件，也可以根据 gpexpand 脚本，通过提示生成一个配置文件（与 gpfilespace 一样）。

如果用户选择根据提示生成配置文件，需要提供一个 hosts 文件，保存新增机器的 hostname，使用 -f 参数指定 hosts 文件。

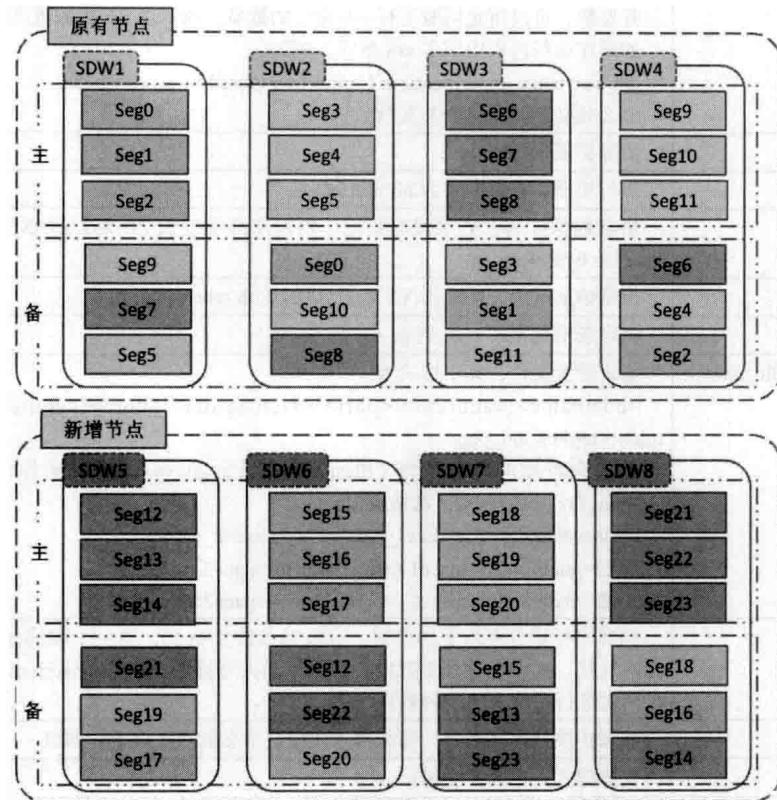


图 13-7 Spread Mirror 的增加节点方案

Segment 初始化阶段包括以下三个操作。

- 1) 创建新的 Segment 实例，该实例拥有所有表的元数据信息，但是没有实际数据。
- 2) 创建一个新的 schema，名为 gpexpand，这个 schema 中保存扩展的所有信息，例如每一个表重分布的详细状态等。
- 3) 将当前数据库中的所有表全部修改成随机分布（Distributed Randomly），这个状态将会在第二阶段——表重分布的时候被修改。

要开始表重分布阶段，用户需要运行 gpexpand 脚本，使用 -d（周期）或 -e（结束时间）参数。在达到指定的结束时间或周期之前，数据库会在新的扩展 schema 中重分布所有的表数据。重分布其实是使用 ALTER TABLE 命令将表恢复至原先的分布字段。

等到所有的表都已经重分布成功后，gpextend 即执行完成了，表 13-1 列举了 gpextend 脚本的参数。

表 13-1 gpextand 脚本的参数说明

-a --analyze	在重分布表之后，重新对表进行 analyze，收集统计信息
-B	并发数，可以指定同时运行 ssh 命令的数量，默认是 8，参数范围是 1 ~ 128。 如果在运行过程中报了 ssh 错误，例如 ssh_exchange_identification: Connection closed by remote host. 那么就必须减少这个并发数
-c --clean	清除扩展 schema
-d   --duration	执行周期，参数格式为 hh:mm:ss。
-D	指定数据库名，如果没有指定，则读取环境变量 PGDATABASE。数据库不能为 template0 或 template1
-e   --end	指定结束时间，格式为 YYYY-MM-DD hh:mm:ss
-f  --hosts-file	指定新增机器的 host 列表
-i   --input <input_file_name>	指定配置文件，其中格式为： <hostname>:<address>:<port>:<fslocation>:<dbid>:<content>:<preferred_role>:<replication_port> 如果有新增的文件系统 (filespace)，那么 gpexpand 需要额外多一个配置文件 (<input_file_name>.fs)，其格式为： filespaceOrder=<filespace1_name>:<filespace2_name>: ... dbid:</path/for/filespace1>:</path/for/filespace2>: ... dbid:</path/for/filespace1>:</path/for/filespace2>: ...
-n	同时进行重分布的表的个数，有效的参数为 1~16，每一个表重分布必须有两个数据库连接。在设置这个参数前，要确保最大连接数 max_connections 这个参数可以满足同时进行重分布的表的数目
-r   --rollback	如果扩展数据库失败，那么可以使用这个参数对扩展进行回滚
-s --slient	安静模式，不会产生提示
-v --verbose	调试模式，打印出详细的执行步骤
-V --novacuum	在创建 schema 副本时，不对数据字典进行 vacuum

使用 gpexpand -f hosts\_expand 可以产生配置文件，此步骤是交互性的，该脚本会咨询如下问题（图 13-8 显示了其中一个问题）。

```

Are you sure you want to continue with this gpexpand session? Yy|Nn (default=N):
> y

You must now specify a mirroring strategy for the new hosts. Spread mirroring places
a given hosts mirrored segments each on a separate host. You must be
adding more hosts than the number of segments per host to use this.
Grouped mirroring places all of a given hosts segments on a single
mirrored host. You must be adding at least 2 hosts in order to use this.

What type of mirroring strategy would you like?
spread|grouped (default=grouped):
> spread

```

图 13-8 使用 gpexpand 生成配置文件

1) 确定是否要初始化一个新的扩展。

2) 选择哪种备份策略 (Spread Mirror 或者 Grouped Mirror)，本例子中采用 Spread

## Mirror。

3) 每个新机器上要新增多少个 Primary Segment, 默认是 0, 如果大于 0, 那么所有的 hosts 节点上都会多出对应的 Primary Segment, 本例子使用默认值。

4) 输入每个 Primary Segment 的数据目录, 这些数据目录需要用户自己手工创建。

`hosts_expand` 文件的内容如下:

```
$ cat hosts_expand
sdw5
sdw6
sdw7
sdw8
```

在回答完上述几个问题之后, 会生成一个名为 `gpexpand_inputfile_20130721_230554` 的配置文件, 这样, 就可以通过这个配置文件来扩展数据库了。

首先, 确保数据库处于启动状态, 并且没有其他的连接

然后运行 `gpexpand` 脚本, 如下:

```
$ gpexpand -i gpexpand_inputfile_20130721_230554
```

在正常情况下, 执行过程如图 13-9 所示。

```
[INFO]:-local Greenplum Version: 'postgres (Greenplum Database) 4.1.1.1 build
[INFO]:-Querying gpexpand schema for current expansion state
[INFO]:-Readyng Greenplum Database for a new expansion
[INFO]:-Checking database testDB for unalterable tables...
[INFO]:-Checking database postgres for unalterable tables...
[INFO]:-Checking database template1 for unalterable tables...
[INFO]:-Checking database testDB for tables with unique indexes...
[INFO]:-Checking database postgres for tables with unique indexes...
[INFO]:-Checking database template1 for tables with unique indexes...
[INFO]:-Creating segment template
[INFO]:-VACUUM FULL on the catalog tables
[INFO]:-Starting copy of segment dbid 1 to location ,data/
[INFO]:-Copying postgresql.conf from existing segment into template
[INFO]:-Copying pg_hba.conf from existing segment into template
[INFO]:-Adding new segments into template pg_hba.conf
[INFO]:-Creating schema tar file
[INFO]:-Distributing template tar file to new hosts
[INFO]:-Configuring new segments (primary)
[INFO]:-Configuring new segments (mirror)
[INFO]:-Backing up pg_hba.conf file on original_segments
[INFO]:-Copying new pg_hba.conf file to original segments
[INFO]:-Configuring original segments
[INFO]:-Cleaning up temporary template files
[INFO]:-Starting Greenplum Database in restricted mode
[INFO]:-Stopping database
[INFO]:-Configuring new segment file spaces
[INFO]:-Cleaning up databases in new segments.
[INFO]:-Starting master in utility mode
[INFO]:-Stopping master in utility mode
[INFO]:-Starting Greenplum Database in restricted mode
[INFO]:-Creating expansion schema
[INFO]:-Populating gpexpand.status_detail with data from database testDB
[INFO]:-Populating gpexpand.status_detail with data from database postgres
[INFO]:-Populating gpexpand.status_detail with data from database template1
[INFO]:-Stopping Greenplum Database
[INFO]:-Starting Greenplum Database
[INFO]:-Starting new mirror segment synchronization
[INFO]:-*****
```

图 13-9 运行 `gpexpand -i` 脚本的结果

在 `gpexpand -i` 脚本运行成功后, 可以看到数据库多了一个 `gpexpand` 的 schema。这个 schema 有两个表和一个视图, 如表 13-2 所示。

表 13-2 gpexpand 模式下的表和视图

名字	类型	说明
gpexpand.status	表	保存扩展时的状态
gpexpand.status_detail	表	保存每个表的详细重分布信息，包括表类型、分布键、优先级等
gpexpand.expansion_progress	视图	显示 Greenplum 扩展的进度，剩余多少表及多少字节，如图 13-10 所示

```
testDB=# select * from gpexpand.expansion_progress;
          name           | value
-----+-----
Tables Left | 10
Bytes Left  | 172517621760
Estimated Time to Completion | ...
Estimated Expansion Rate | ...
(4 rows)
```

图 13-10 视图 expansion\_progress 展示扩展的进度

在运行 `gpexpand -i` 之后，开始执行表重分布，运行下面的命令，执行过程中最消耗时间的是所有数据表的重分布（可以将 `gpextand` 脚本分布在几次数据库比较空闲的时间点执行），开始进行表重分布。

```
$ gpexpand -d 60:00:00
```

之后 `gpexpand` 脚本会开始对每个表进行重分布，并将进度打印在屏幕上。同时，在数据库中，也可以使用下面的命令查看到正在运行的 SQL，结果如图 13-11 所示。

```
testDB=# select * from pg_stat_activity;
```

```
-[ RECORD 3 ]-----+
datid      | 16992
datname    | testDB
procid     | 44376
sess_id    | 59
usesysid   | 10
username   | scutshuxue.chenxf
current_query| ALTER TABLE ONLY "public"."test0" SET WITH(REORGANIZE=TRUE) DISTRIBUTED BY ("a")
waiting    | f
query_start| 2013-07-21 23:31:02.256287+08
backend_start| 2013-07-21 23:31:01.029079+08
client_addr| 127.0.0.1
client_port| 24678
application_name| 
xact_start | 2013-07-21 23:31:01.870912+08
-[ RECORD 4 ]-----+
```

图 13-11 gpexpand 使用 ALTER TABLE 对原表进行重分布

待所有的数据都重分布完成后，使用 `gpexpand -c` 将扩展的 schema 删除。至此，数据库扩容成功。

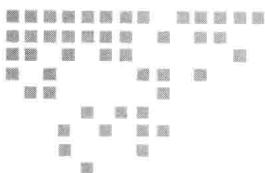
### 13.3 小结

本章重点介绍了如何对 Greenplum 进行扩容、增加存储空间及计算性能，主要讲了两种方法。

- 1) 迁移计算节点：使用 `scp` 或 `gprecoverseg -F`。
- 2) 增加计算节点：使用 `gpextand` 脚本。这个涉及表的重分布，在重分布过程中，数据库的性能会急剧下降。用户在扩展的时候要注意这一点，合理安排扩容时间。

在增加计算节点时，分布的策略要考虑到 Greenplum 的两种备份策略，Spreed Mirror 和 Grouped Mirror，要选择适当的节点分布方式。

扩容是一个影响很大的操作，建议读者在扩容之前，要多在线下环境测试，测试没问题之后才可以在线上数据库中执行。购买了 Greenplum 服务的公司，最好咨询 Greenplum 的技术支持，协商确定好方案后再执行。



## 第 14 章

## 基于 Greenplum 的海量数据实时分析服务平台

互联网领域的实时计算主要都是针对海量数据进行的。实时计算最重要的一个功能是能够实时响应计算结果，一般要求为秒级。互联网行业的实时计算普遍应用于以下两种应用场景。

- 1) 数据源是实时、不间断的（流数据），要求对用户的响应时间也是实时的。
- 2) 数据量大且无法预处理（计算），要求对用户的响应时间也是实时的。

第一种应用场景主要针对互联网流式数据处理，即将数据看做数据流的形式来处理。例如，在精准广告投放应用场景中，需要基于用户的实时点击、实时查询等调整推荐算法。业界较流行的方案如 Storm、S4 等。第二种应用场景主要针对数据服务平台，给线上产品提供计算和查询服务。本章将重点讨论第二种应用场景下结合 Greenplum 的一些实践。

## 14.1 需求概述

从整体纵向需求上来看，主要有如下几个层次的需求，键值访问、筛选查询、聚合计算、关联分析。从横向需求来看，主要有如下几个基本的需求，支持高并发、准实时、海量数据、高可用 / 线性可扩展。如图 14-1 所示。

本章介绍的 Greenplum 实时化方案主要解决以下两个问题。

- 1) 能够对海量数据进行一些报表计算，要求后端数据能做到分钟级别的更新，响应时间要求在分钟以内。
- 2) 能够对海量数据进行一些简单查询，如 KV 查询，或者对上百条数据进行简单的汇总等，响应时间要求在毫秒级。

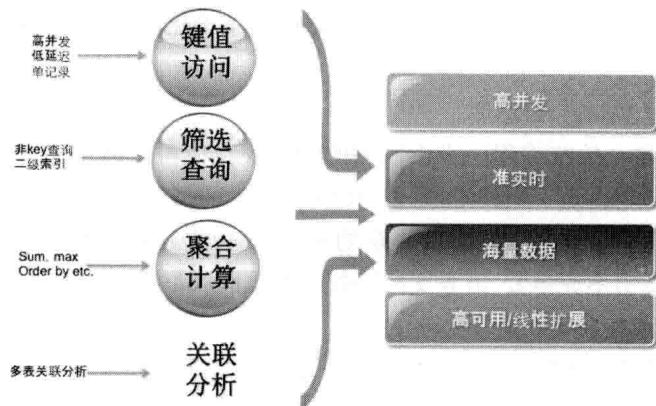


图 14-1 海量数据实时分析服务平台总体需求

一般系统都会将这两者分开，前者对应一些报表数据库，用于分析的，后者一般采用NoSQL，或者对MySQL进行分库分表。本章结合Greenplum探究一种综合方案。

## 14.2 典型方案

目前，在海量数据实时分析服务平台的典型方案可分为如下几类。

- 1) NoSQL 系列。
- 2) 分布式数据库 / 集群。
- 3) 分表分库。

各分类典型的解决方案如图 14-2 所示。

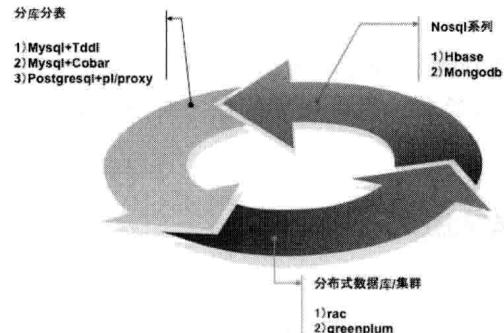


图 14-2 海量数据实时分析服务平台典型架构分类

### 14.2.1 NoSQL

NoSQL 不再使用 SQL 查询语言，而是使用自己特有的查询语言。一般都是开源项目且

为分布式集群而定制开发的，总体基于半结构化或弱结构化的数据模型设计。NoSQL 方案大体上可分如下几类。

#### (1) 键值 (Key-Value) 数据库

键值数据库就像在传统编程语言中使用的哈希表。你可以通过 key 来添加、查询或者删除数据。典型产品如 Redis、Memcached、Riak、Membase 等。比较适用的场景包括存储用户信息（如会话信息、配置文件、参数、购物车等）。但是在基于值查询的场合，键值数据库不是很适合（二级索引不完善），事务支持不好，数据间关联关系难以表达。

#### (2) 文档数据库

文档数据库会将数据以文档的形式储存。每个文档都是自包含的数据单元，是一系列数据项的集合。每个数据项都有一个名称及对应的值，此对应值既可以是简单数据类型，如字符串、数字和日期等，也可以是复杂的类型，如有序列表和关联对象。数据存储的最小单位是文档，同一个表中存储的文档属性可以是不同的，数据可以使用 XML、JSON 或 JSONB 等多种形式存储。典型的文档数据库包括 Mongodb、CouchDB 等。

#### (3) 图数据库

图数据库允许我们将数据以图的方式储存。实体会作为顶点，而实体之间的关系则会作为边。典型的图数据库包括 Neo4J、OrientDB，在任务调度依赖、推荐引擎、社交关系等方面有较为常用的场景。

#### (4) 对象数据库

对象数据库管理系统为面向对象编程语言增加了持久的概念，具备复杂的对象模型、快速键值访问。典型的对象数据库包括 Gemstone、Objectivity。

#### (5) BigTable 类型数据库

BigTable 类型数据库是分布式的、面向列的开源数据库，其中，用户存储数据行在一个表里，一个数据行拥有一个可选择的键和任意数量的列。由于表是疏松的存储的，因此用户可以给行定义各种不同的列。BigTable 类型数据库主要用于需要随机访问，实时读写的场景，具备处理大数据、高负载、高可用等特点。典型的 BigTable 类型数据库包括 Hbase、Cassandra、Hypertable 等。

### 14.2.2 分布式数据库 / 集群

分布式数据库 / 集群基于海量信息并行处理结构 (MPP)，应用程序通过 Master 主机访问数据，每一个存储节点都是独立数据库（无共享），存储节点间、存储节点和 Master 主机间通过高速网络交换数据，读写等数据库操作和传统关系型数据库无异，数据分布在所有的数据节点上，每个节点只需要处理一部分数据。具备高扩展性、高可用性、高性能、易用性等特点。典型的分布式数据库 / 集群包括如下。

- Teradata
- Netezza

- Vertica
- Greenplum
- Aster Data

MPP 是将任务并行地分散到多个服务器和节点上，在每个节点计算完成后，将各自部分的结果汇总在一起得到最终的结果。例如，要查询销售最好的 10 件商品，每个节点都要先计算出自己销售最好的 10 件商品，然后向上汇总。如果是两个表的连接查询，可能会涉及节点之间计算的中间过程如何传递数据的问题：是将大表和小表都平均分布，然后在节点计算的时候将得到的结果汇总（可能要两次汇总），还是将大表平均分布，将小表的数据传输给每个节点，从而只需要一次汇总。不同的产品在处理这些细节方面各有所不同。

### 14.2.3 分表分库

分表分库方案是对数据进行水平拆分以降低单库的压力，并且实现高效且相对透明的来屏蔽掉水平拆分的细节。总体实现原理是借鉴分布式数据库集群的思路，将数据分散以满足高并发、大数据量、高可用的需求。业界比较典型的分表分库方案有 Cobar、TDDL、Ameoba、DDB。这里我们简单介绍一下 Cobar。

Cobar 是阿里巴巴开源的一种关系型数据的分布式处理系统，在分布式的环境下它看上去像传统数据库一样提供海量数据服务。

Cobar 中间件以 Proxy 的形式位于前台应用和实际数据库之间，对前台开放的接口是 MySQL 通信协议。将前台 SQL 语句变更并按照数据分布规则转发到合适的后台数据分库，再合并返回结果。Cobar 除了支持 MySQL、Oracle，还可以很便捷地扩展支持 PostgreSQL 等数据库。Cobar 应用逻辑架构如图 14-3 所示。

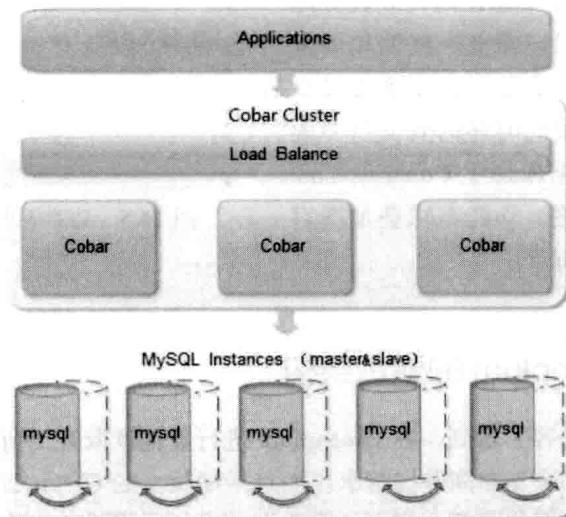


图 14-3 Cobar 应用逻辑架构

应用程序访问 Cobar 集群，通过负载均衡连接至其中一个 Cobar Server（集群中多个 Cobar Server 对等无状态，可根据需要线性扩展），再根据 Cobar Server 中配置的路由规则，定位至其中一个 MySQL 实例，返回期望的结果。Cobar 典型应用方式如图 14-4 所示。

通过 Cobar 提供一个名为 test 的数据库，其中包含 t1, t2 两张表。后台有 3 个 MySQL 实例（ip:port）为其提供服务，分别为 A、B、C。t1 表的数据可放置在实例 A 中，t2 表的数据水平拆成四份并在实例 B 和 C 中各放两份。t2 表的数据具备 HA 功能，即 B 或者 C 实例其中一个出现故障，不影响使用且可提供完整的数据服务。

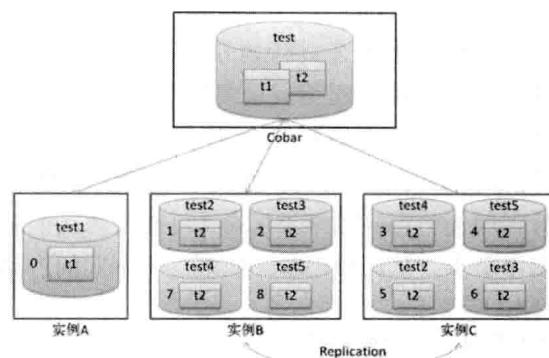


图 14-4 Cobar 应用方式

#### 14.2.4 方案优劣分析

通过前面三小节的分析，我们不难看出每个方案都有各自的侧重点。NoSQL 的优势主要在键值查询、高并发、高可用、高可扩展性。分布式数据库的优势主要在于查询功能强大、技术方案成熟、天然支持聚合计算和关联分析，不过在可用性、可扩展性、并发性及查询性能方面略差。而分表分库是介于 NoSQL 和分布式数据库方案之间的一种折中方案。在满足线性扩展、高可用、海量数据、准实时等需求的基础上，如果业务需求仅需键值查询，NoSQL 方案将会是不错的选择。如果还需要支持大量且复杂的筛选查询，分表分库方案可满足需求。如果还需要一些简单的聚合计算，我们可以在分表分库方案增加接口层计算框架即可。如果同时还需要支持多表关联分析，分布式数据库加上 Cobar 或 Tddl 将是一个可尝试的方案。功能需求叠加图对应解决方案如图 14-5 所示。

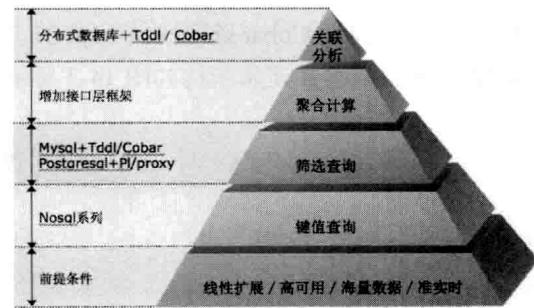


图 14-5 功能需求叠加图对应解决方案

### 14.3 基于 Greenplum 的混合架构

通过前面章节的内容了解到，在 Greenplum 进行数据读取时，所有流量都会经过 Master 节点，扩展性较差，无法支撑高并发请求。而且早期版本的 Greenplum 在处理一个简单的基于主键的查询，所有数据节点都会产生一次查询请求，同样无法支撑高并发请求。因此，结合 Greenplum 及分表分库二者的优势，我们在实际生产中应用了 Greenplum+Cobar 这种架

构，如图 14-6 所示。

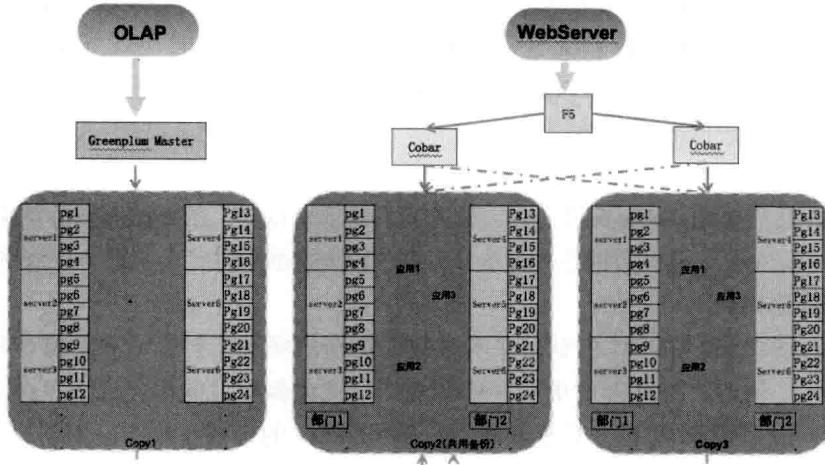


图 14-6 Greenplum+Cobar 海量数据实时分析服务平台



此套架构我们在 Greenplum3.3.7+cobar1.0.9 版本中测试通过，由于 Greenplum4.0 之后，Mirror Segment 无法链接，所以只能采用旧版本的 Greenplum 才能实践。

### 14.3.1 架构分析

如图 14-6 所示，Greenplum 数据节点自身是 Master-Slave 结构，主备（Copy1、Copy2）各 24 个数据节点（PostgreSQL），我们还引入第三组备份（Copy3）。多表关联的请求通过 Greenplum Master 节点，负载压力主要集中在集群 Copy1 上。高并发的简单查询通过 Cobar 来完成，负载压力主要集中在集群 Copy3 上。OLAP 和 OLTP 的请求互相不干扰，保证了集群的稳定及高可用。Copy2 作为 Copy1 和 Copy3 公用的备份集群，当 Copy1 集群有节点死机时，Greenplum 自动切换到备用集群节点，当 Copy3 集群有节点死机时，Cobar 也会自动切换到备用集群节点。

对于一些有实时要求的表，后台有一个数据更新脚本，做到了分钟级别的更新，通过对 MySQL 数据库 binlog 的解析，将 MySQL 的变更数据每 5 分钟往上述的三组备份进行数据合并，保证数据的新鲜度为 5 分钟。对于实时性要求不高的表，则每天更新一次。

### 14.3.2 实施要点

#### (1) 数据切分规则

由于 Cobar 切分规则和 Greenplum 数据分片规则不一致，如果按照 Greenplum 自身的

分片规则，Cobar 将不能正常定位数据节点并返回准确结果。如果按照 Cobar 切分规则，将会导致 Greenplum 数据错误。我们可以修改 Cobar 切分算法，使其和 Greenplum 算法保持一致。但是在实际应用中，我们无法获取 Greenplum 切分算法的实现细节，所以我们在 Greenplum 建表时采用 DISTRIBUTED RANDOMLY，这样在 Master 进行计算的时候，数据会全部重分布，保证数据的准确性。

#### (2) 数据导入

由于切分算法需要和 Cobar 保持一致，因此我们需要将待导入数据按照切分算法预先切割，再导入 3 个 PostgreSQL 集群，数据导入脚本分为 5 分钟级别准实时导入还有 T+1 级别每天导入。

#### (3) 实时数据更新

由于 PostgreSQL 的 MVCC 机制，在实时数据更新的场景，频繁的更新操作将会导致数据查询越来越慢。同时，在高并发更新操作环境下，更新操作性能消耗巨大，严重影响实时数据处理性能。在实际环境下，可考虑采用 Append only 的方式（即不做更新，只做插入），在查询数据时通过通用的分析函数获取数据的最新版本，然后定期做去重操作。

#### (4) 在线 DDL

数据库的在线加字段、在线 Truncate、在线 Rename 等操作，在常规情况下都会锁表，影响实时的数据查询。在实际应用环境下，在线加字段可通过修改 pg\_attribute 数据字典手动插入字段信息，在线 Truncate 表可通过修改 pg\_class 及 pg\_depend 数据字典来交换临时表和正式表对应的底层数据文件。在线 Rename 操作，可通过修改 pg\_class 及 pg\_depend 数据字典来交换临时表和正式表对应的底层数据文件。

#### (5) PostgreSQL 无法强制 Cache 特定表，致使低频率访问时效率较差

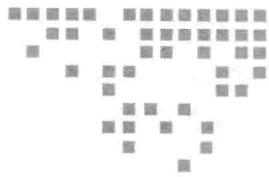
可通过 Mmap 技术将表对应的数据文件在操作系统强制缓存。

#### (6) F5 硬件负载均衡

F5 是一种实现负载均衡的硬件，它通过硬件的方式统一接收全部请求，然后按照设定好的算法将这些请求分配给这个负载均衡组中的所有成员。如果没有硬件负载均衡设备，也可以采用 Nginx、LVS 等软负载均衡设备，它们通过 Keepalived 或 Heartbeat 等方法实现软负载均衡的高可用。

## 14.4 小结

本章首先从需求上简单进行了概述，包括高并发、准实时、海量数据、高可用 / 线性扩展等基础性需求，以及键值访问、筛选查询、聚合计算、关联分析四个层次的高级需求。其次介绍了 NoSQL、分布式数据库 / 集群、分表分库方案在海量数据实时分析服务平台应用场景下的优劣及侧重点。最后介绍了基于 Greenplum 的混合架构方案及实际应用过程中的一些实施要点。实际需求及应用场景千差万别，读者在方案选型及架构设计过程中还需结合自身实际的场景进行调整及优化，适合自己的才是最好的。



## 使用 Greenplum 的常见报错及小技巧

在使用一个工具的时候，我们往往有需要注意很多细节的东西，通过对一些细节的了解，可以更加了解工具的用户以及背后的原理。本章将介绍 Greenplum 的一些常见的数据库报错，以及一些使用小技巧，希望读者在使用 Greenplum 的时候更加得心应手，遇到报错时不再不知所措。

### 15.1 分析常见报错

下面介绍一些 Greenplum 中常见的报错，一般的报错信息从报错原因上就可以清楚地了解，但是有一些报错信息比较难以理解。面对报错，先不要慌，仔细阅读报错的信息，如果报错信息不明显，要结合 SQL 进行分析，是 SQL 本身语法的问题还是理解上有误。

很多报错可能会与分布式数据库有关，第一次使用分布式数据库的用户可能会忽略这个问题，如果用普通的单机数据库的观点来分析，就可能很难理解报错的内容。比如，Master 和 Segment 数据字典不一致，在报错时一直在 Master 上查相关的数据字典都没有发现问题，实际上是 Segment 上的数据字典不一致，如果直接在 Segment 上分析就很容易找到问题，读者在遇到类似的问题的时候要多加注意。

#### 15.1.1 找不到类型 705 对应的操作符

先来看下面报错的 SQL，一个简单的建表语句：

```
testDB=# create table type_error
testDB-#       as select 'a' as id
```

```
testDB-# distributed by (id);
ERROR: no equality operator for typid 705 (cdbmutate.c:1187)
```

初看这个报错有点莫名其妙，typid 为 705 的数据类型没有“等于”按钮操作符。我们先来看下 705 是什么数据类型，通过 `::regtype` 可以很方便地查询类型名（见第 4.1 节介绍）：

```
testDB=# select 705::regtype;
 regtype
-----
 unknown
(1 row)
```

typid 为 705 的数据类型是 `unknown`，难怪没有“等号”按钮操作符。为什么会有 `unknown` 数据类型呢？仔细看下 SQL，就会发现字符串 'a' 并没有制定数据类型，那么 a 的类型可能是 `varchar`，也可能是 `char`，也有可能是 `text` 类。不显示指定数据类型，数据库就无法识别，所以就将 'a' 制定成了 `unknown` 数据类型。

`unkown` 类型怎么来的，我们已经知道了，但是为什么 `unkown` 数据类型需要“等号”按钮操作符呢？仔细看一下会发现 `id` 这个字段是分布键，分布键需要计算 `hash` 值，然后将这个值分发到相应的节点上，在计算 `hash` 值过程中，调用了“等号”按钮操作符，由于 `unkown` 没有这个操作符，所以报这个错。

如果去掉分布键，改成随机分布会怎么样？

```
testDB=# create table type_error
testDB-#      as select '1' as id
testDB-# distributed randomly;
WARNING: column "id" has type "unknown"
DETAIL: Proceeding with relation creation anyway.
SELECT 1
testDB=# \d type_error
Table "public.type_error"
 Column | Type    | Modifiers
-----+-----+-----
 id     | unknown |
Distributed randomly
```

建表成功了，但是会有一个 `WARNING`，告诉用户 `id` 的类型为 `unkown`。在不指定分布键的情况下，这个表默认为随机分布的表，就不需要计算 `hash` 值，也就不用调用“等号”按钮操作符，所以建表成功。

解决方法为：对 `id` 字段指定数据类型。

```
testDB=# create table type_error
testDB-#      as select '1'::varchar(16) as id
testDB-# distributed by (id);
SELECT 1
```

建表成功，没有任何异常。

### 15.1.2 SQL 占用的资源超过了资源队列限制

当前用户执行的 SQL 超过了资源队列限制的大小，会报如下的错：

```
statement requires more resources than resource queue allows
```

此时 SQL 不能执行，出现这个报错，一般是 SQL 写得有问题，检查下是否 SQL 过于复杂，或者产生了笛卡尔积。解决方法如下。

- 1) 增加该用户所在资源队列的大小限制。
- 2) 使用超级用户执行该 SQL，超级用户没有资源队列的限制。
- 3) 改写 SQL，将复杂的 SQL 拆分成多个小 SQL 执行，减少一次 SQL 的消耗。
- 4) 检查 SQL 是否正确，是否出现了笛卡尔积等非常消耗资源的操作。

### 15.1.3 自定义函数不能在 Segment 上执行

当自定义函数从 `tbl_test_func` 表中查询数据时报错：

```
function cannot execute on segment because it accesses relation "public. tbl_test_func"
```

其中，`tbl_test_func` 建表语句如下：

```
create table public.tbl_test_func(
    col1          integer
    ,col2          integer
)Distributed by (col1);
```

这个自定义的函数如下：

```
CREATE OR REPLACE FUNCTION testfunc(pararegistno varchar)
RETURNS varchar AS
$BODY$
declare
    id varchar;
    str varchar;
begin
    str:='';
    for id in select col1 from tbl_test_func limit 10 loop
        str := str||id;
    end loop;
    return str;
end
$BODY$
LANGUAGE plpgsql VOLATILE;
```

单独执行 SQL 的时候，没有问题：

```
testDB=# select testfunc('') ;
testfunc
-----
```

```
35719213941535557
(1 row)
```

当从一张表中查询的时候，报错：

```
testDB=# select testfunc(coll::varchar) from test1;
ERROR:  function cannot execute on segment because it accesses relation
"public. tbl_test_func" (functions.c:150)  (seg2 slice1 sdw2:33000 pid=6966)
(cdbdisp.c:1457)
DETAIL:
        SQL statement " select coll from tbl_test_func limit 10"
PL/pgSQL function "testfunc" line 6 at for over select rows
```

在 Greenplum 中，在 Segment 上，函数中是不能对数据表进行查询的，因为函数会被分发到每个 Segment 上去运行，每个 Segment 只有部分的数据，在函数中操作表，得到的数据本来就不是全部的数据，会造成数据异常，所以 Greenplum 不支持在函数中 Segment 上操作用户数据表。但是数据字典是可以的，因为数据字典在每个 Segment 上都有一份全量数据，不会造成数据异常。把上面函数中的表 `tbl_test_func` 换成 `pg_class`，这个 SQL 就能执行成功。

#### 15.1.4 子查询没有加别名

当 SQL 中的子查询中忘记加别名时会报如下错误：

```
subquery in FROM must have an alias
```

如下面这个 SQL 为：

```
select * from (select * from test1);
```

这个是很普通的数据库报错，在子查询后面加上一个别名就可以解决报错，如下：

```
select * from (select * from test1)t;
```

#### 15.1.5 字段名有歧义

执行下面这个简单的 SQL：

```
select coll from test1 a,test1 b where a.coll=b.coll;
```

此时会有如下报错：

```
column reference "coll" is ambiguous
```

报错的意思是，字段 `coll` 是含糊不清的。这是对两张表进行关联，这两张表都有名为 `coll` 的字段，但是在查询的时候，`coll` 没有指定是哪张表的，产生了歧义，所以报错。

要解决报错只需指定 `coll` 是哪一个表的字段即可，SQL 修改为：

```
select a.coll from test1 a,test1 b where a.coll=b.coll;
```

### 15.1.6 字段重名

执行如下 SQL:

```
create table test3 as
select *
from test1 a,test1 b
where a.col1=b.col1;
```

会有如下报错:

```
column name "col1" is duplicated
```

这是因为在表 test1 和表 test2 中都有名为 col1 的字段，需要在 select 中将同名的字段去掉或重命名:

```
create table test3 as
select a.col1 as a_col1,b.col1 as b_col1
from test1 a,test1 b
where a.col1=b.col1;
```

### 15.1.7 gpfdist 错误：无法读取文件

使用 gpfdist 读取外部表时，报错:

```
gpfdist error - cannot read file
```

这个报错的含义是外部表错误，读取文件失败。这个报错表示一般的数据文件已经损坏了，需要重新初始化数据文件，或者通过其他第三方工具修复该文件。

接下来列举几个外部表相关的报错。

1) gpfdist 中的 URL 的个数超过了 Segment 的个数，报错信息如下:

```
testDB=# CREATE EXTERNAL TABLE gpfdist_error (
testDB(# all_value text
testDB(# )
testDB-# LOCATION ('gpfdist://10.20.151.7:8081/00.dat',
testDB(# 'gpfdist://10.20.151.7:8081/01.dat',
testDB(# 'gpfdist://10.20.151.7:8081/02.dat',
testDB(# 'gpfdist://10.20.151.7:8081/03.dat',
testDB(# 'gpfdist://10.20.151.7:8081/04.dat',
testDB(# 'gpfdist://10.20.151.7:8081/05.dat',
testDB(# 'gpfdist://10.20.151.7:8081/06.dat',
testDB(# 'gpfdist://10.20.151.7:8081/07.dat')
testDB-# FORMAT 'TEXT' (DELIMITER ',' null as '' ESCAPE 'off');
CREATE EXTERNAL TABLE
Time: 444.900 ms
testDB=# select * from gpfdist_error;
ERROR: There are more external files (URLs) than primary segments that can read them. Found 8 URLs and 6 primary segments.
```

URL 的个数不能超过 Segment 的个数，上面的外部表有 8 个 URL，而 Segment 只有 6 个，所以报错。解决方法是：减少 URL 个数（可以在 URL 上使用通配符或正则表达匹配）。

```
CREATE EXTERNAL TABLE gpfdist_correct (
all_value text
)
LOCATION ('gpfdist://10.20.151.7:8081/0[0-7].dat')
FORMAT 'TEXT' (DELIMITER ',' null as '' ESCAPE 'off');
```

2) gpfdist 所在机器的 gpfdist 服务没有启动，需要重新启动下该服务，一般报错信息如下：

```
ERROR: connection with gpfdist failed for gpfdist://10.20.151.5:8080/1.dat...
```

3) 文件不存在的报错信息如下：

```
ERROR: http response code 404 from gpfdist (gpfdist://10.20.151.7:8081/not_exists.dat): HTTP/1.0 404 file not found (url.c:279)
```

### 15.1.8 事务被中止

在执行 SQL 时，偶尔会报如下的错误：

```
current transaction is aborted, commands ignored until end of transaction block
```

该错误表示，当前事务被终止，执行的命令被数据库回滚，一般是事务中有 SQL 执行报错，导致当前事务回滚，如果不是 SQL 的问题，可能是网络问题，或者 Greenplum 太慢导致的系统不稳定，一般重新运行即可。

### 15.1.9 网络异常错误

在执行 SQL 时，偶尔会报如下的错误：

```
Error on receive from seg6 slice1 sdw3-1:30002 pid=1690: server closed the connection unexpectedly
```

子节点执行报错，服务连接异常终止，可以重新运行试试。如果不能解决问题，则可能 Segment 上出现了问题，要检查 Segment 的日志文件进行问题排查，也有可能是因为 SQL 中的某个操作，如自定义函数等触发了 Greenplum 的 Bug，导致在 Segment 上这个 SQL 中断异常。

### 15.1.10 无法删除表

在删除表的时候，报错：

```
cannot drop table v_xxx because other objects depend on it (seg1 sdw1-2:30001 pid=22093)"
```

在子节点上有其他对象依赖于这个表，如果在 Master 上这个表已经删掉，那么可能是

Master 与 Segment 节点数据不一致导致的，可以手工到 SEGMENT 上查询这个表，依赖关系在数据字典 pg\_depend 中可以找到。

在章节 4.7.6 查询表上的视图中，介绍了如何在 pg\_depend 上建立视图，可以快速查询视图依赖的问题，读者遇到该问题，利用这个视图可以很方便地查询，在 Segment 上找出对应的依赖关系，并把对应的视图删除掉。

### 15.1.11 内存不足

内存不足报错：

```
insufficient memory reserved for statement (memquota.c:228)
```

可能是当前系统运行了太多 SQL，或者是某一个 SQL 消耗太大，占用了大量的内存，导致其他 SQL 运行报错。

如何找出使用内存过大的 SQL 呢？由于 Greenplum 没有类似的内存检测工具，一般的做法是看哪个 SQL 执行的时间比较长，一般消耗内存大的，执行时间也相对比较长。或者去一台 Segment 机器上看下哪个进程占用的内存比较大，然后再通过 all\_seg\_sql（在 10.6 节中介绍）找出这个进程号对应的 SQL。

### 15.1.12 文件名在 pg\_class 中已存在

在 truncate 表的时候，报错如下：

```
relfilenode 124451009 already in use in "pg_class" (catalog.c:1130)
```

这个报错在一般情况下不会产生，在系统忙的时候偶尔会出现，这是 Greenplum 还不完善的地方。出现的场景为：在创建或截取表的时候，需要重新生成 pg\_class 的 relfilenode 字段的数值，在生成后提交时会发现 pg\_class 中这个 relfilenode 已经被占用了，所以就报出这个错。一般重新运行就可以解决这个报错。

relfilenode 和 oid 一样，共用同一个序列，这个序列的值可以通过 pg\_highest\_oid 函数来查询。

```
testDB=# select pg_highest_oid();
pg_highest_oid
-----
314186
(1 row)
```

### 15.1.13 不能对分布键执行 Update

对表进行 update 的时候报错：

```
ERROR: Cannot parallelize an UPDATE statement that updates the distribution columns
```

分布键不能被更新，因为这会导致数据重分布，而 Greenplum 4.1 还不支持这个操作。如果分布键一定要更新，解决办法就是新建一张表，然后将修改后的数据导入新表中，从而实现 update 的操作。或者先备份数据，然后删掉要更新的数据，之后再插入修改后的数据。

### 15.1.14 网络错误

网络错误消息解析错误：

```
Interconnect error parsing message (seg25 sdw15-2:30001 pid=9943),"tcSize
25197 > max 8192 header 4 processed 5966/8192 from 1276990"
```

这是 Greenplum 本身某些设计不完善导致的，在系统繁忙或者网络不稳定的时候比较容易发生，一般重运行即可。

### 15.1.15 无法找到数据文件

数据库异常时候会发生这个报错：

```
could not open relation 1663/16384/xxxxx
```

在数据库中，对应的数据文件已经不存在了（其中 1663 是 tablespace 的 OID 在 pg\_tablespace 中通过 OID；而 16384 是数据库的 OID，记录在 pg\_database 中；xxxxx 则是文件的 ID，对应 pg\_class 中的 relfilenode 字段）。

发生这种报错的情况有两种。

1) 数据库中数据文件丢了，在数据库层面无法恢复这个文件，可以尝试利用第三方数据恢复工具恢复数据文件。如果无法恢复，则只能重建表，然后将数据重新加载。

2) 数据字典异常，数据文件存在，但是数据字典中的 relfilenode 与文件不一致，数据库崩溃重启后，在崩溃过程中创建的表可能会出现数据字典不一致的问题，但是修复数据字典很麻烦，建议还是放弃表重建一个吧。

## 15.2 常见问题及解决办法

### 1. 时间分区裁剪

我们来看下这个 SQL：

```
explain select * from public.test_offer where dw_end_date = now()::date;
```

其中 dw\_end\_date 是一个 date 类型。

在 Greenplum 3.x 中，执行计划将对表进行全表扫描，但是在 Greenplum 4.x 中，数据库会算出今天的数据，然后对分区进行裁剪，所以在使用 Greenplum3.x 的时候要注意将这种

SQL 中的时间替换成具体的时间串，然后进行计算。

但是对于下面这个 SQL，无论是 Greenplum 3.x 还是 Greenplum 4.x 都是无法采取分区裁剪：

```
explain select * from public.test_offer where dw_end_date = now();
```

下面介绍在 Greenplum 3.x 中如何定义一个函数来实现 now() 的功能。在 Greenplum 中可以采取分区裁剪。

在 PostgreSQL 的文档中，自定义函数有如下属性，分别代表自定义函数的类型：

- **IMMUTABLE** 表示该函数不能修改数据库，并且在给出同样的参数值时总是返回同样的结果。也就是说，不查询数据库或者只使用那些没有出现在参数列表中的信息。如果给出这个选项，那么任何对该函数的调用都将立即转换成该函数的结果值（如果 func 是一个 IMMUTABLE 的自定义函数，而 func(1) 的值为 100，那么在 SQL 执行前，所有 func(1) 的值都会被替换成 100）。
- **STABLE** 表示该函数能修改数据库，对于相同参数值，在同一次表扫描中，该函数的返回值不变，但是返回值可能在不同 SQL 语句之间变化。这个属性用于那些结果依赖数据库环境（比如当前时区）之类的函数。还要注意 current\_timestamp 函数总是稳定的，因为它们的值在一次事务中不会变化。
- **VOLATILE** 表示该函数值可以在一次表扫描内改变，因此不会做任何优化。只有很少的数据库函数在这个概念上是易变的，比如 random(), currval(), timeofday()。注意任何有副作用的函数都必须列为易变类，即使其结果相当有规律也应该如此，这样才能避免函数被优化，setval() 就是这样的函数。

now 和 current\_timestamp 等函数，都是 STABLE 类型的。对于 Greenplum 3.x 来说，如果使用 STABLE 的函数，数据库在生成执行计划的时候不会去计算这个值的大小。



函数的类型可以在 pg\_proc 数据字典中找到：

```
testDB=# select provolatile from pg_proc where proname='now';
provolatile
-----
s
(1 row)
```

Provolatile 字段的值：i 代表 immutable，s 代表 stable，v 代表 volatile。

那么我们建一个 IMMUTABLE 的函数，数据库是否会采用分区裁剪呢？

```
create or replace function public.curr_date()
RETURNS date
AS
$BODY$
begin
```

```

        return now()::date;
end;
$BODY$
LANGUAGE 'plpgsql' IMMUTABLE ;

```

虽然实际上还是调用 now() 来计算当天的时间，但是数据库并不会分析函数中的内容。将这个函数创建为 IMMUTABLE 类型的，数据库就认为这个函数的返回值无论如何都不会变化，那么在生成执行计划的时候，就会将这个函数的值计算出来，这样分区表查询就会采取分区裁剪了。



**注意** 当函数被创建为 IMMUTABLE 类型的时候，在同一个会话中，这个函数的值是不会变的，除非重新开始一个会话。这样就会有一个问题，例如，对于上面的 SQL，如果是在 1 日 23 : 59 分执行的，当时间到了 2 日凌晨的时候，如果还是在同一个会话中，那么查询到的数据，还是 1 日的数据。

读者可以试试将函数创建为 STABLE 或 VOLATILE 类型来验证数据库是否采取分区裁剪。

## 2. 如何查询当前 SQL 在 Segment 上对应的进程号

在查看当前运行 SQL 的系统视图 (pg\_stat\_activity) 时会发现，Greenplum 比 PostgreSQL 多了一个 sess\_id 的字段，在 Greenplum 中，一个 SQL 会被拆分成子任务在 Segment 上运行，对于同一个 SQL，这些子任务在 Segment 上都拥有同一个 sess\_id (即会话 ID)，这个 sess\_id 就是 Master 和 Segment 之间 SQL 对应的标识。通过 all\_seg\_sql (10.6 节中介绍) 视图，就可以在 Naster 上查询每个 Segment 的进程号了。

还有一种查询 Segment 的进程号的方法，那就是查看进程号。在 Master 上查询好正在运行的 SQL 的 sess\_id，然后使用 ps auxww 看出 Segment 上的进程，细心的读者会发现，进程上有很多关于这个进程的信息：

```
postgres: port 33000, gpadmin testDB 10.20.151.7(51234) con87 seg4 idle
```

其中 con 之后的 87 就是 sess\_id。

进程的其他信息有：这个 Segment 对应的端口，当前 SQL 执行的用户名，数据库名，SQL 是从哪个 IP 发送过来的（括号中的是对应的端口地址），con 后面的数据即 sess\_id，seg 后面的数字则对应该 Segment 对应 gp\_configuration 的 content。



**注意** 通过 netstat -na 就可以查看这个端口：

```
[gpadmin@inc-dw-hadoop-151-7 ~]$ netstat -an |grep 51234
tcp      0      0 10.20.151.7:51234          10.20.151.10:33000      ESTABLISHED
```

### 3. 一个 SQL 在一个 Segment 上只会对应一个进程吗

不是的，一个 SQL 有可能会对应多个进程。前面讲到了，每当有数据广播或重分布时，Segment 都会启动相应的进程来处理数据切片（即执行计划中的 Slice）。所以应该尽量减少一个 SQL 中广播或重分布的次数，减少太多进程给数据库或者操作系统带来的压力。

### 4. 如果有 Segment 死机了，需要将数据迁移到其他的机器上，如何才能让新的机器生效

修改 `gp_segment_configuration` 中对应的 IP 地址的响应的端口号，然后重启数据库，在修改 `gp_segment_configuration` 的时候，可以使用 `gpstart -m` 单独启动 Master，然后通过 utility 模式登录 Master 进行修改（修改前记得将原来的数据备份，以防万一）。

### 5. Segment 数据目录的名字是否可以修改

在 Greenplum 3.x 的版本中是可以随意修改的，在修改之后，只需要将 `gp_configuration` 中对应的 `datadir` 字段改成新的数据位置即可。

在 Greenplum 4.x 版本中，数据目录是记录在 `pg_filespace_entry` 中的。如果没有使用 `filespace`（要使用表空间必须新建一个文件空间），那么在修改目录位置后，可以修改 `pg_filespace_entry` 来完成配置更新。但如果使用了表空间，由于主备之间需要同步数据，在每个子节点的 `gp_persistent_filespace_node` 表中都会记录 `primary` 和 `mirror` 对应的 `filespace` 的目录，这个目录是不能修改的。因此，在 Greenplum 4.x 版本中，数据目录在使用了 `filespace` 之后是不能随意更改的。

### 6. 如何提交 SQL

在提交 SQL 的时候由于有一些参数是动态变化的，比如时间，每天执行任务时，SQL 中操作数据的时间是按天变化的，或者一些内容可能会要求在一个事务中执行，所以提交 SQL 脚本应该对其进行封装。

一般利用脚本语言，例如 Perl、Python，对 SQL 进行封装，这样子还可以在脚本里面处理一些逻辑，使用起来更加灵活，通过封装好统一的接口，可以屏蔽掉一些高危操作，如 `drop` 等。

## 15.3 常用的一些小技巧

本节将介绍一些在 Greenplum 使用过程中经常用到的一些小技巧，利用这些小技巧，可以在操作数据库的过程中更加得心应手。

### 15.3.1 显示 SQL 执行的时间

执行 SQL 之前，设置 \timing 就会打开时间：

```
testDB=# \timing
Timing is on.
testDB# select count(1) from pg_class;
 count
-----
 437
(1 row)

Time: 2.625 ms
```

这个参数是在 psql 客户端设置的，与服务器的设置无关。如果想默认打开这个参数，则要在 psql 的客户端所在机器上配置 ~/.psqlrc 这个文件，在文件中设置默认的参数，这样在此机器上使用 psql 登录 Greenplum 都会默认先应用 ~/.psqlrc 中的所有参数配置，如：

```
[gpadmin@inc-dw-hadoop-151-7 ~]$ cat ~/.psqlrc
\timing
set client_encoding=utf8
```

### 15.3.2 获取某个 schema 下所有的表或视图

要快速将某个 schema 下的所有表或视图查询出来，可以用：

```
testDB=# \dt tmp.*
      List of relations
 Schema |   Name    | Type  | Owner | Storage
-----+-----+-----+-----+-----+
  tmp   | test_1000 | table | gpadmin | heap
(1 row)
```

\dt 为表，\dv 为视图，支持通配符。这一类命令很常用，比如通过 \df+ 函数名可以将函数的定义和源码显示出来。

### 15.3.3 查找分区最多的表

Greenplum 的分区信息记录在 pg\_partition\_rule 和 pg\_partition 这两个数据字典中，通过关联这两个表，即可找出分区最多的表。

```
testDB=# select b.parrelid::regclass,count(1)
testDB-#   from pg_partition_rule a,
testDB-#         pg_partition b
testDB-#   where a.paroid=b.oid
testDB-#   group by b.parrelid order by 2 desc;
          parrelid      | count
-----+-----+
```

```
test_partition      |     8
test_partition2    |     3
(2 rows)
```

### 15.3.4 连接 Segment 节点

在登录的时候设置环境变量 PGOPTIONS=' -c gp\_session\_role=utility'，就可以登录 Segment 节点，如下：

```
[gpadmin@inc-dw-hadoop-151-7 ~]$ PGOPTIONS=' -c gp_session_role=utility' psql -h
sdw2 -p 33000
psql (8.2.15)
Type 'help' for help.

testDB=# show gp_contentid;
 gp_contentid
-----
 2
(1 row)
```

### 15.3.5 psql 默认密码登录

跟 MySQL 和 Oracle 不一样，在 psql 登录数据库时，如果在 pg\_hba.conf 中配置了 md5 模式，则要输入密码。而在 psql 的命令行参数中，不能指定密码。如果要设置默认密码，则要设置环境变量 PGPASSWORD：

```
export PGPASSWORD=123456
```

这样在 psql 登录的时候，就会读取这个环境变量，并使用这个密码登录数据库。

### 15.3.6 查看数据库启动时间

查看数据库从什么时候启动：

```
testDB=# select pg_postmaster_start_time();
 pg_postmaster_start_time
-----
 2012-03-22 16:10:02.415047+08
(1 row)
```

### 15.3.7 查看在 psql 中 \d 到底查询了哪些数据字典

在登录的时候，通过 psql-E 登录数据库，然后再使用 \d 等类型的命令，就会打印出查询的 SQL：

```

testdb> \d
***** QUERY *****
select version()
***** *****

***** QUERY *****
SELECT n.nspname AS "Schema",
       c.relname AS "Name",
       CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'i' THEN 'index' WHEN 's' THEN 'sequence' WHEN 's' THEN 'special' END AS "Type",
       pg_catalog.pg_get_userbyid(c.relowner) AS "Owner", CASE c.relstorage WHEN 'h' THEN 'heap' WHEN 'x' THEN 'external' WHEN 'a' THEN 'append only' WHEN 'v' THEN 'none' WHEN 'c' THEN 'append only columnar' WHEN 'f' THEN 'foreign' END AS "Storage"
FROM pg_catalog.pg_class c
LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r', 'v', 's', 'i')
AND c.relstorage IN ('h', 'a', 'c', 'x', 'f', 'v', '')
AND n.nspname <> 'pg_catalog'
AND n.nspname <> 'information_schema'
AND n.nspname <> 'pg_toast'
AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 1,2;
***** *****

          List of relations
   Schema |      Name      | Type | Owner | Storage
-----+-----+-----+-----+-----+
 public | cn_offer_basic_fdt0 | table | gpadmin | append only columnar
-----+-----+-----+-----+-----+

```

图 15-1 显示使用 \d 等命令查询的 SQL

## 15.4 小结

本章主要介绍了使用 Greenplum 的一些常见的报错、问题及小技巧，这些内容都是笔者平时积累下来的，内容比较散。由于笔者能力有限，对于一些常见报错只介绍一部分，有一些报错无法知道其底层的原因，只能介绍一些“治标”的方法，比如重新运行。

对于分布式数据库而言，Greenplum 的功能非常全面，内容也很多，在使用过程中有很多的技巧，还有一些需要注意的地方。毕竟 Greenplum 发展只有几年的时间，很多地方都不够完善，很多细节还没做好，尤其是在稳定性方面，在数据库压力大的时候容易报出一些奇奇怪怪的错误。还有，Greenplum 的文档相对较少，几乎没有可以借鉴的处理异常报错的参考资料。因此，用户在使用的时候要注意日常的积累，记录下来一些报错的信息并探索一些解决方案，通过不断的积累在增加自身能力的同时提高对 Greenplum 的掌控能力。

Greenplum是一个面向数据仓库应用的关系型数据库，它基于流行的PostgreSQL开发，因为有良好的体系结构，所以在数据存储、高并发、高可用、线性扩展、反应速度、易用性和性价比等方面有非常明显的优势，非常受欢迎。进入大数据时代以后，Greenplum的性能在TB级别数据量的表现非常优秀，单机性能相比Hadoop要快上好几倍；在功能和语法上，要比Hadoop上的SQL引擎Hive好用很多，普通用户更加容易上手；Greenplum有着完善的工具，整个体系都比较完善，不需要像Hive一样花太多的时间和精力进行改造，非常适合作为一些大型数据仓库的解决方案。Greenplum能够方便地与Hadoop进行结合，直接把数据写在Hadoop上，并且能够直接在数据库上写MapReduce任务，同时配置简单。

阿里巴巴是国内最早使用Greenplum作为数据仓库计算中心的企业，本书的两位作者曾经是这个计算中心的亲身实践者和核心技术人员，无论是生产实践还是作者经验，本书都具有一定权威性，而且是接地气的。



上架指导：计算机/数据库

ISBN 978-7-111-48100-3



9 787111 481003 >

定价：69.00元

投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

购书热线：(010) 68326294 88379649 68995259

华章网站：[www.hzbook.com](http://www.hzbook.com)

网上购书：[www.china-pub.com](http://www.china-pub.com)

数字阅读：[www.hzmedia.com.cn](http://www.hzmedia.com.cn)