HashData

目录

1 开发指南 1.1
2 系统概述 2.1
3 访问数据仓库 3.1
4 定义数据库对象 4.1
5 数据的管理 5.1
6 导入导出数据 6.1
7 查询数据 7.1
8 SQL 命令参考 8.1
- 8.1 SQL语法概要 8.1.1
- 8.2 ABORT 8.1.2
- 8.3 ALTER AGGREGATE 8.1.3
- 8.4 ALTER CONVERSION 8.1.4
- 8.5 ALTER DATABASE 8.1.5
- 8.6 ALTER DOMAIN 8.1.6
- 8.7 ALTER EXTENSION 8.1.7
- 8.8 ALTER EXTERNAL TABLE 8.1.8
- 8.9 ALTER FILESPACE 8.1.9
- 8.10 ALTER FUNCTION 8.1.10
- 8.11 ALTER GROUP 8.1.11
- 8.12 ALTER INDEX 8.1.12
- 8.13 ALTER LANGUAGE 8.1.13
- 8.14 ALTER OPERATOR CLASS 8.1.14
- 8.15 ALTER OPERATOR 8.1.15
- 8.16 ALTER PROTOCOL 8.1.16
- 8.17 ALTER RESOURCE QUEUE 8.1.17
- 8.18 ALTER ROLE 8.1.18
- 8.19 ALTER SCHEMA 8.1.19
- 8.20 ALTER SEQUENCE 8.1.20
- 8.21 ALTER TABLE 8.1.21
- 8.22 ALTER TABLESPACE 8.1.22
- 8.23 ALTER TYPE 8.1.23
- 8.24 ALTER USER 8.1.24
- 8.25 ANALYZE 8.1.25
- 8.26 BEGIN 8.1.26
- 8.27 CHECKPOINT 8.1.27
- 8.28 CLOSE 8.1.28
- 8.29 CLUSTER 8.1.29
- 8.30 COMMENT 8.1.30
- 8.31 COMMIT 8.1.31

8.32 COPY 8.1.32
8.33 CREATE AGGREGATE 8.1.33
8.34 CREATE CAST 8.1.34
8.35 CREATE CONVERSION 8.1.35
8.36 CREATE DATABASE 8.1.36
8.37 CREATE DOMAIN 8.1.37
8.38 CREATE EXTENSION(new) 8.1.38
8.39 CREATE EXTERNAL TABLE 8.1.39
8.40 CREATE FUNCTION 8.1.40
8.41 CREATE GROUP 8.1.41
8.42 CREATE INDEX 8.1.42
8.43 CREATE LANGUAGE 8.1.43
8.44 CREATE OPERATOR CLASS 8.1.44
8.45 CREATE OPERATOR FAMILY 8.1.45
8.46 CREATE OPERATOR 8.1.46
8.47 CREATE PROTOCOL 8.1.47
8.48 CREATE RESOURCE QUEUE 8.1.48
8.49 CREATE ROLE 8.1.49
8.50 CREATE RULE 8.1.50
8.51 CREATE SCHEMA 8.1.51
8.52 CREATE SEQUENCE 8.1.52
8.53 CREATE TABLE AS 8.1.53
8.54 CREATE TABLE 8.1.54
8.55 CREATE TABLESPACE 8.1.55
8.56 CREATE TYPE 8.1.56
8.57 CREATE USER 8.1.57
8.58 CREATE VIEW 8.1.58
8.59 DEALLOCATE 8.1.59
8.60 DECLARE 8.1.60
8.61 DELETE 8.1.61
8.62 DISCARD 8.1.62
8.63 DO 8.1.63
8.64 DROP AGGREGATE 8.1.64
8.65 DROP CAST 8.1.65
8.66 DROP CONVERSION 8.1.66
8.67 DROP DATABASE 8.1.67
8.68 DROP DOMAIN 8.1.68
8.69 DROP EXTENSION 8.1.69
8.70 DROP EXTERNAL TABLE 8.1.70
8.71 DROP FILESPACE 8.1.71
8.72 DROP FUNCTION 8.1.72
8.73 DROP GROUP 8.1.73

8.74 DROP INDEX 8.1.74
8.75 DROP LANGUAGE 8.1.75
8.76 DROP OPERATOR CLASS 8.1.76
8.77 DROP OPERATOR FAMILY 8.1.77
8.78 DROP OPERATOR 8.1.78
8.79 DROP OWNED 8.1.79
8.80 DROP PROTOCOL 8.1.80
8.81 DROP RESOURCE QUEUE 8.1.81
8.82 DROP ROLE 8.1.82
8.83 DROP RULE 8.1.83
8.84 DROP SCHEMA 8.1.84
8.85 DROP SEQUENCE 8.1.85
8.86 DROP TABLE 8.1.86
8.87 DROP TABLESPACE 8.1.87
8.88 DROP TYPE 8.1.88
8.89 DROP USER 8.1.89
8.90 DROP VIEW 8.1.90
8.91 END 8.1.91
8.92 EXECUTE 8.1.92
8.93 EXPLAIN 8.1.93
8.94 FETCH 8.1.94
8.95 GRANT 8.1.95
8.96 INSERT 8.1.96
8.97 LOAD 8.1.97
8.98 LOCK 8.1.98
8.99 MOVE 8.1.99
8.100 PREPARE 8.1.100
8.101 REASSIGN OWNED 8.1.101
8.102 REINDEX 8.1.102
8.103 RELEASE SAVEPOINT 8.1.103
8.104 RESET 8.1.104
8.105 REVOKE 8.1.105
8.106 ROLLBACK TO SAVEPOINT 8.1.106
8.107 ROLLBACK 8.1.107
8.108 SAVEPOINT 8.1.108
8.109 SELECT INTO 8.1.109
8.110 SELECT 8.1.110
8.111 SET ROLE 8.1.111
8.112 SET SESSION AUTHORIZATION 8.1.112
8.113 SET TRANSACTION 8.1.113
8.114 SET 8.1.114
8.115 SHOW 8.1.115

8.116 START TRANSACTION 8.1.116
8.117 TRUNCATE 8.1.117
8.118 UPDATE 8.1.118
8.119 VACUUM 8.1.119
8.120 VALUES 8.1.120
9 数据类型 9.1
10 字符集支持 10.1
11 系统配置参数 11.1
12 内置函数摘要 12.1
13 PL/Python 语言扩展 13.1
14 用于分析的 MADlib 扩展 14.1
15 PostGIS 扩展 15.1

开发指南

- 您是第一次使用 *HashData* 数据仓库么？
- 您是数据库开发者么？
- 阅读前需要做的准备工作

您是第一次使用 HashData 数据仓库么？

如果您是第一次使用 HashData 数据仓库，我们强烈建议您先阅读下面的内容。

- 入门指南 - 包含了完整的示例，从创建 HashData 数据仓库集群，到创建数据库中的表，加载数据到最后的测试查询及验证结果。
- 管理指南 - 集群管理手册为您介绍如何创建和管理HashData 数据仓库集群。

如果具有其它关系型数据库或者数据仓库应用的经验，那么您需要注意 HashData 数据仓库与其它产品的设计和实现区别。您可以参考最佳实践章节来了解和学习如何更好地使用 HashData 数据仓库来加载数据和设计表结构。

HashData 数据仓库是基于 greenplum 和 postgres 开发的。

您是数据库开发者么？

如果您是一位数据库用户，数据库设计者，或者数据库开发者，又或是数据库管理员，请参考下面表格，来帮助您快速定位您所需要查找的相关信息。

您需要的信息	推荐参考
快速地创建并使用 HashData 数据仓库	通过 入门指南 介绍，快速的了解部署集群， 连接数据库和尝试一些最简单数据加载和查询。在您熟悉 HashData 数据仓库后，准备搭建您的数据库，将数据导入到表中，在数据仓库中使用查询操作 数据时，再回到本手册了解更多内容。
了解和学习 HashData 数据仓库的内部架构	HashData 数据仓库系统概述 为您介绍 HashData 数据仓库内部架构的宏观概述。

阅读前需要做的准备工作

在正式阅读文档前，您可以完成下面的任务来加速您对 HashData 数据仓库理解和掌握：

- 安装一个 PSQL 客户端程序
- 启动一个 HashData 数据仓库集群
- 使用 PSQL 客户端程序连接到集群的 master 节点

如果您需要关于上面步骤的相关操作帮助，请参阅：入门指南。

您最好了解 SQL 客户端程序的使用方法，并掌握一些基础的 SQL 语言知识。

系统概述

本章节将会介绍 HashData 数据仓库的模块及一些关键特性，让您对本产品拥有更加深刻的认识和理解。

本章节涵盖以下内容:

- 数据仓库架构

HashData 数据仓库架构

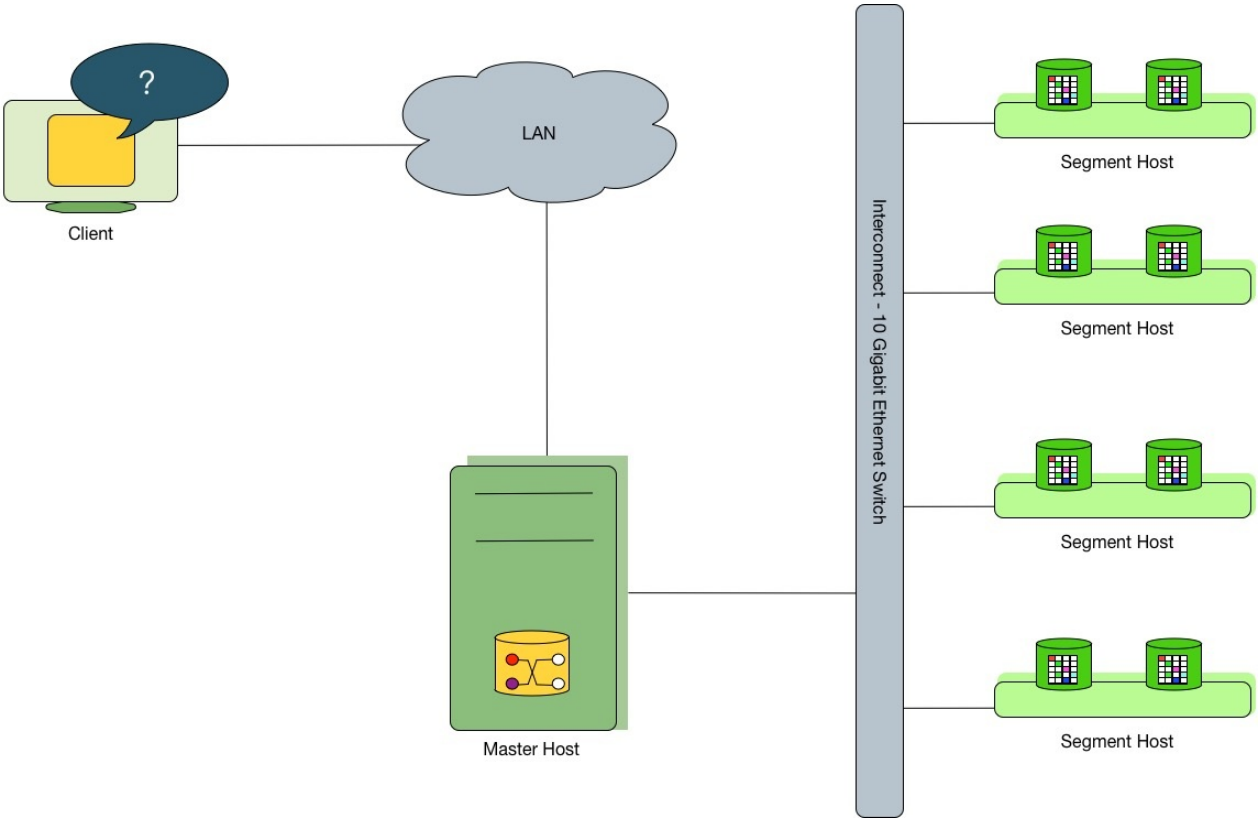
HashData 数据仓库是为了管理大容量分析型数据仓库和商业智能分析业务而设计的大规模并行处理（MPP）数据库服务系统。

MPP（也被称作 Shared Nothing 架构）是指一个系统拥有两个或者两个以上的处理器，相互合作来执行任务。每个处理器都配有独立的内存，操作系统和磁盘。HashData 数据仓库采用高性能的系统架构可以将请求均匀分散到存储 TB 级别的数据仓库上，同时充分利用系统中所有的资源，并行的处理请求。

HashData 数据仓库是基于 Greenplum Database 和 PostgreSQL 开源数据库技术，通过对 PostgreSQL 的修改，得到的并行架构数据库。从系统信息表，优化器，查询执行器，事务管理等各个方面都进行了修改和增强，来满足真正将查询从内部并行运行在多个计算节点上。通过快速的内部软件数据交互模块，满足系统在多个节点间数据的传输和处理要求，使得整个系统在外面看就像是一个计算能力等于上百台机器的单一数据库系统。

HashData 数据仓库的主节点 (Master) 是整个数据库系统的入口节点，用户通过客户端连接主节点来提交 SQL 查询语句。主节点将会协调其它计算节点 (Segment) 来存储和处理用户的数据。

图1. 系统架构



主节点

HashData 数据仓库的主节点是整个数据库系统的入口，用来接受客户端连接，接收 SQL 查询，并将作业分发到计算节点上执行。

HashData 数据仓库的用户可以像使用 PostgreSQL 一样，通过主节点来访问 HashData 数据仓库系统。目前支持客户端程序 psql 或应用编程接口 ODBC 或 JDBC。

主节点存储了描述系统全局结构的系统信息表（Global System Catalog），这些信息表中存储了 HashData 数据仓库自己的元信息（Metadata）。主节点没有存储用户数据的信息，所有的用户数据都存储在计算节点，主节点认证客户端连接，处理客户提交的 SQL 命令，将查询分发到存储数据的计算节点，协调各个计算节点执行，并汇总执行结果，最后返回给客户端程序。

计算节点

HashData 数据仓库的计算节点实际上同样是一个经过修改的 postgres 数据库，每个计算节点都存储了一部分用户数据，并主要负责执行用户的查询。

每当用户连接到主节点，并且发送一个查询时，每个计算节点都会创建一些进程来共同处理改查询。要了解更多关于查询的处理过程，请参考 查询数据。

用户定义的数据表和相应的索引将会自动被分散到 HashData 数据仓库的节点上，每个计算节点
上都存储了一部分用户数据，并且这些数据是不相交的。

软件数据交换模块

软件数据交换模块是 HashData 数据仓库架构中的网络层。此模块负责处理计算节点之间和网络之间的进程间通信。此模块默认使用经过深度调优的带流量控制的 UDP 协议来传输数据。这种算法除了提供 TCP 协议支持的可靠性外，在性能和水平扩展能力都优于 TCP 协议。

访问数据仓库

本小节向您介绍使用不同工具连接 HashData 数据仓库建立会话的方法。

建立会话

用户可以通过兼容 PostgreSQL 的客户端程序来连接 HashData 数据仓库的主节点。你可能需要了解以下连接信息来帮助您配置客户端程序。

链接参数	参数描述	环境变量
应用名称	连接到数据库的应用程序名称。	`$PGAPPNAME`
数据库名称	你想要连接到的数据库的名称。	`$PGDATABASE`
主机名称	HashData 数据仓库主机的主机名称。	`$PGHOST`
端口号	HashData 数据仓库主机的端口号。默认值为 5432。	`$PGPORT`
用户名	要连接的数据库用户 (角色) 名称。	`$PGUSER`

支持客户端列表

用户可以使用不同的客户端应用程序连接到 HashData 数据仓库:

- 兼容标准 PostgreSQL 客户端程序 psql。通过 psql 程序，用户可以使用交互式命令行来访问 HashData 数据仓库。
- pgAdmin III 是一个非常流行的、支持 PostgreSQL 和 HashData 数据仓库扩展特性的图形化管理工具。
- 通过使用标准的数据库应用接口（例如：JDBC 和 ODBC），用户可以创建独立的客户端应用程序来访问 HashData 数据仓库。由于 HashData 数据仓库是基于 PostgreSQL 实现，因此可以直接使用 PostgreSQL 的数据库驱动程序。
- 大部分使用标准数据库访问接口（例如：JDBC 和 ODBC）的第三方客户端程序，通过适当配置就可以连接 HashData 数据仓库。

使用 **psql**

根据您配置的环境变量值，下面的例子展示了如何使用 psql 连接数据库：

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

$ psql gpdatabase

$ psql
```

当您连接到数据库后，psql 将会提示您正在使用的数据库名称，每个数据库名称后面跟随着输入提示字符串 => （如果您使用了超级用户登录，那么提示字符串是 =#）。如下所示：

```
gpdatabase=>
```

在提示字符串后，你可以输入 SQL 命令。每条 SQL 命令必须以分号（；）结束，这样服务器才能接收并执行该命令。如下所示：

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

pgAdmin III

如果您喜欢使用图形化接口，可以使用 pgAdmin III 工具。该图形工具能够支持 PostgreSQL 数据库所有标准特性，也添加了对 HashData 数据仓库特性的支持。pgAdmin III 支持 HashData 数据仓库的特性包括:

- 外部表支持
- Append 表（包括使用压缩特性的 Append 表）
- 分区表信息
- 资源队列
- 图形化 EXPLAIN ANALYZE

数据库应用接口

您也许需要开发专有的客户端程序来连接 HashData 数据仓库系统。 PostgreSQL 为常用的数据库应用接口提供了驱动程序，这些驱动程序可以直接操作 HashData 数据仓库。具体驱动程序及下载链接如下：

API	**PostgreSQL** 驱动程序名称	下载链接
ODBC	pgodbc	源代码、Win64、Win32
JDBC	pgjdbc	JRE6、JRE7、JRE8
Perl DBI	pgperl	http://search.cpan.org/dist/DBD-Pg/
Python DBI	pygresql	http://www.pygresql.org/

使用 HashData 数据仓库应用接口的配置步骤：

1. 根据使用语言和接口，下载相关程序。例如：从 Oracle 官方下载。
2. 根据接口说明，编写您的客户端程序。在编写程序时，请注意 HashData 数据仓库的相关语法，这样可以避免您使用不支持的特性。
3. 下载对应的 PostgreSQL 驱动程序，并配置连接 HashData 数据仓库主节点的信息。

Maven

```
<!-- https://mvnrepository.com/artifact/postgresql/postgresql -->
<dependency>
    <groupId>postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.1-901-1.jdbc4</version>
</dependency>
```

Gradle

```
// https://mvnrepository.com/artifact/postgresql/postgresql
compile group: 'postgresql', name: 'postgresql', version: '9.1-901-1.jdbc4'
```

第三方客户工具

大部分第三方 ETL 和商业智能（BI）工具使用标准数据库接口连接 HashData 数据仓库，例如：ODBC 和 JDBC。目前经过测试支持的应用包括

- Apache Zeppelin
- Tableau Desktop

定义数据库对象

在这个章节中，我们将介绍 HashData 数据仓库支持的数据定义语言 (DDL) 以及如何创建和管理数据库对象。

创建 HashData 数据仓库对象的时候，我们需要考虑很多因素，包括数据分布、存储选项、数据加载以及其它影响数据库系统运行性能的功能。了解可用的选项和数据库内部如何支持这些选项将帮助您做出正确的选择。

通过扩展标准 SQL 的 CREATE DDL 语句，HashData 数据仓库实现了很多高级的功能。

创建和管理数据库

与一些商业数据库（如 Oracle 数据库）不同，HashData 数据仓库支持创建多个数据隔离的独立数据库，但是客户端程序每次只能连接并使用其中一个。

关于数据库模版

您创建的每一个数据库都是基于一个模版得到的。系统中的默认模版数据库叫做：template1。我们建议您不要在 template1 中创建任何数据对象，否则您后续创建的数据库都会包含这些数据。

HashData 数据仓库内部还使用另外两个内置模版：template0 和 postgres。因此请勿删除或修改 template0 和 postgres 数据库。您也可以使用 template0 作为模版，创建一个只含有标准预定义对象的空白数据库，特别是在您已经修改了默认模版数据库 template1 的情况下。

创建数据库

使用 CREATE DATABASE 命令来创建一个新的数据库. 例如:

```
=> CREATE DATABASE new_dbname;
```

若要创建新的数据库,您需要拥有创建数据库的权限或拥有者超级用户权限。如果您没有相应的权限，创建数据库的操作将会失败。可以联系数据管理员来取得创建数据库的权限。

克隆数据库

创建新的数据库时，系统实际上通过克隆一个默认的标准数据库模版 template1 来完成。实际上，您可以指定任意一个数据库作为创建新数据库的模版，这样新的数据库就会自动包含模版数据库中的所有对象和数据。例如：

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

列出所有数据库

如果您使用 psql 客户端程序，您可以使用 \l 命令列出系统中的模版数据库和数据库。如果您使用其他客户端程序并且拥有超级用户权限，您可以通过查询 pg_database 系统表列出所有数据库。例如：

```
=> SELECT datname from pg_database;
```

修改数据库

ALTER DATABASE 命令可以用来修改数据库的属主，名称或者默认参数配置。例如,下面的命令修改了数据库默认模式搜索路径：

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

你需要是数据库的属主或拥有超级用户权限，才可以对数据库信息进行修改。

删除数据库

DROP DATABASE 命令可以删除数据库。该命令将会从系统表中删除数据库相关信息，并在磁盘上删除该数据库相关的所有数据。只有数据库的属主或者超级用户才能够删除数据库。正在被使用的数据库是无法被删除的。例如：

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

> 警告：删除数据库是不可逆的操作，请谨慎使用。

创建和管理模式

模式（Schema）的作用类似于名字空间，实现了数据库对象逻辑上的分类组织。通过使用模式对象，您可以在同一个数据库中，创建同名的对象（例如：表，函数）。

默认模式 "public"

数据库包含一个默认模式：public。如果您没有创建任何模式，新创建的对象会默认使用 public 模式。数据库所有的用户都拥有 public 模式上的 CREATE（创建）和 USAGE（使用）权限。当您创建额外的模式时，您可以对用户授予权限，从而实现访问控制。

创建模式

使用 **CREATE SCHEMA** 命令来创建一个新的模式. 例如:

```
=> CREATE SCHEMA myschema;
```

要在指定的模式下创建对象或访问对象，您需要使用限定名格式来进行。限定名格式是模式名 ”.“ 表名的方式，例如：

```
myschema.table
```

参考 模式的搜索路径 了解更多关于访问模式的说明.通过为用户创建私有的模式，可以更好地限制用户对名称空间的使用。语法如下：

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

模式的搜索路径

通过使用模式限定名，可以指向数据库中特定位置的对象。例如：

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

可以通过设置参数 search_path 来指定模式的搜索顺序。搜索路径中第一个模式就是系统使用的默认模式，当没有引用模式时，对象将会自动创建在默认模式下。

设置模式搜索路径 search_path 配置参数用来设置模式搜索顺序。ALTER DATABASE 命令可以设置数据库内默认搜索路径。例如：

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

还可以通过 ALTER ROLE 命令来为指定的用户修改 search_path 参数。例如：

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

查看当前模式通过 current_schema() 函数，系统可以显示当前模式。例如：

```
=> SELECT current_schema();
```

类似的，使用 SHOW 命令也可以显示当前搜索路径。例如：

```
=> SHOW search_path;
```

删除模式

使用 DROP SCHEMA 命令可以删除一个模式。例如：

```
=> DROP SCHEMA myschema;
```

默认的删除命令只能删除一个空的模式。要删除模式及其内部包含的所有对象（表，数据，函数，等），使用下面的命令：

```
=> DROP SCHEMA myschema CASCADE;
```

系统预定义模式

每个数据库中内置了下列系统模式：

- pg_catalog 包含了系统表，内建数据类型，函数和运算符对象。模式搜索路径时，系统总是会考虑此模式下的所有对象。
- information_schema 模式包含了大量标准化视图来描述数据库内部对象信息。这些视图以标准化方式来展现系统表中的信息。
- pg_toast 存储大对象，例如：记录大小超过页面大小的对象。此模式下的信息是数据库内部使用的。
- pg_bitmapindex 存储 bitmap 所有对象，例如：值列表。此模式下的信息是数据库内部使用的。
- pg_aoseg 存储 append-optimized 表对象。 此模式下的信息是数据库内部使用的。
- gp_toolkit 是一个管理视图，内置一些外部表，视图和函数。可以通过 SQL 语句进行访问。所有数据库用户都能够访问 gp_toolkit 来查看日志文件和其它系统参数。

创建和管理表

HashData 数据仓库中的表和其它关系型数据库十分相似，但由于是 MPP 架构，数据会被打散分发到多个节点存储。每次创建表时，你可以指定数据的分布策略。

创建表

CREATE TABLE 命令用来创建和定义表结构，创建表时，您需要定义下面信息：

- 表中包含的列及其对应数据类型。请参考 选择列数据类型。
- 用于限制表或列存储数据的表约束或列约束。请参考 设置表约束和列约束。
- 数据分布策略，系统根据策略将数据存储到不同节点。请参考 选择数据分布策略。
- 磁盘存储格式。请参考 表存储模型。
- 大表的数据分区策略。请参考 对大表进行分区处理。

选择列数据类型

列数据类型的选择是根据该列存储的数值决定的。选择的数据类型应该尽可能占用空间小，同时能够保证存储所有可能的数值并且最合理地表达数据。例如：使用字符型数据类型保存字符串，日期或者日期时间戳类型保存日期类型，数值类型来保

存数值。

我们建议您使用 VARCHAR 或者 TEXT 来保存文本类数据。我们不推荐使用 CHAR 类型保存文本类型。VARCHAR 或 TEXT 类型对于数据末尾的空白字符将原样保存和处理，但是 CHAR 类型不能满足这个需求。请参考 CREATE TABLE 命令了解更多相关信息。

您应该使用同时满足数值存储和未来扩展需求的最小数据类型。例如：使用 BIGINT 类型存储 INT 或者 SMALLINT 数值会浪费存储空间。如果数据随时间推移需要扩展，并且数据重新加载比较浪费时间，那么在开始的时候就应该考虑使用更大的数据类型。例如：如果当前数值能够用 SMALLINT 存储，但是数值会越来越大，那么长远来看使用 INT 类型可能是更好的选择。

如果您考虑两表连接，那么参与连接的列的数据类型应该保持一致。通常表连接是用一张表的主键和另一张表的外键进行的。当数据类型不一致时，数据库需要进行额外的类型转换，而这开销是完全无意义的。

HashData 数据仓库支持大量原生的数据类型，文档后面会进行详细介绍。

设置表约束和列约束

您可以通过在表或者列上创建约束来限制存储到表中的数据。HashData 数据仓库支持 postgres 所有种类的约束，但是您需要注意一些额外的限制条件：

- CHECK 约束只能引用 CHECK 的目标表。
- UNIQUE 和 PRIMARY KEY 约束必须和数据分布键和分区键兼容。
- FOREIGN KEY 约束能够创建，但是系统不会检查此约束是否满足条件。
- 创建在分区表上的约束将会把整个分区表当成一个整体处理。系统不允许针对表中特定分区定义约束条件。

Check 约束

Check 约束允许你限制某个列值必须满足一个布尔（真值）表达式。例如，要求产品价格必须是一个正数：

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

非空约束

非空约束允许你限制某个列值不能为空，此约束总是以列约束形式使用。例如：

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

唯一约束

唯一约束确保存储在一张表中的一列或多列数据数据一定唯一。要使用唯一约束，表必须使用 Hash 分布策略，并且约束列必须和表的分布键对应的列一致（或者是超集）。例如：

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

主键约束

主键约束是唯一约束和非空约束的组合。要使用主键约束，表必须使用 Hash 分布策略，并且约束列必须和表的分布键对应

的列一致（或者是超集）。如果一张表指定了主键约束，分布键值默认会使用主键约束指定的列。例如：

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

外键约束

HashData 数据仓库不支持外键约束，但是允许您声明外键约束。系统不会进行参照完整性检查。

外键约束指定一列或多列必须与另一张表中的值相匹配，满足两张表之间的参照完整性。HashData 数据仓库不支持数据分布到多个节点的参照完整性检查。

选择数据分布策略

所有 HashData 数据仓库数据表都是分布在多个节点的。当您创建或修改表时，您可以通过使用 DISTRIBUTED BY（基于哈希分布）或者 DISTRIBUTED RANDOMLY(随机分布)来为表指定数据分布规则。

> 注意：gp_create_table_random_default_distribution 参数控制着在 DISTRIBUTED BY 子句缺省情况下的行为。

当您在考虑表的数据分布策略时，您可以从以下三方面来分析并帮助决策：

- 均匀地分布数据 — 为了尽可能获得最佳性能，每个节点应该尽可能获得均匀的数据。如果数据呈现出极度不均匀，那么数据量较大的节点就需要更多资源甚至是时间才能完成相应的工作。选择数据分布键值时尽量保证键值唯一，例如使用主键约束。
- 局部和分布式运算 — 局部运算远远快于分布式运算。如果连接，排序或聚合运算能够在局部进行（计算和数据在一个节点完成），那么查询的整体速度就会更快。如果某些计算需要在整个系统来完成，那么数据需要进行交换，这样的操作就会降低效率。如果参与连接或者排序的表都包含相同的数据分布键，那么这样的操作就可以在局部进行。如果数据采用随机分布策略，系统就无法在局部完成像连接这样的操作。
- 均匀地处理请求 — 为了最优的性能，每个节点应该处理均匀的查询工作。如果表的数据分布策略和查询使用数据不匹配，查询的负载就会产生倾斜。例如：销售交易记录表是按照客户 ID 进行分布的，那么一个查询特定客户 ID 的查询就只会在一个特定的节点进行计算。

声明数据分布

CREATE TABLE 的可选子句 DISTRIBUTED BY 和 DISTRIBUTED RANDOMLY 可以为表指定数据分布策略。表的默认分布策略是使用主键约束（如果有的话）或者使用表的第一列。地理信息类型或者用户自定义数据类型是不能被用来作为表的数据分布列的。如果一张表没有任何合法的数据分布列，系统默认使用随机数据分布策略。

为了尽可能保证数据的均匀分布，尽量选择能够使数据唯一的分布值。如果没有任何值能够满足，可以使用随机分布策略：

```
=> CREATE TABLE products
                       (name varchar(40),
                        prod_id integer,
                        supplier_id integer)
            DISTRIBUTED BY (prod_id);

=> CREATE TABLE random_stuff
                       (things text,
                        doodads text,
                        etc text)
            DISTRIBUTED RANDOMLY;
```

表存储模型

HashData 数据仓库支持多种表存储模型，以及这些模型的混合。当您创建一张表的时候，您可以选择如何存储数据。这个

小节中，我们将解释各种表存储模型，以及如何根据您的工作负载选择性能最优的存储模型。

- 堆存储（Heap Storage）
- 追加优化存储（Append-Optimized Storage）
- 如何选择行存储和列存储
- 数据压缩（只适用于追加优化存储）
- 查看追加优化表的数据压缩和数据分布
- 更改一张表
- 删除一张表

> 提醒：为了简化创建表操作，您可以通过设置配置参数 gp_default_storage_options 来指定默认的表存储模型。

堆存储（Heap Storage）

默认的情况下，HashData 数据仓库会选用和 PostgreSQL 一样的堆存储模型。堆存储的表非常适合这种场景下的 OLTP（在线事务处理）工作负载：数据初次加载后频繁更新。为了确保可靠的数据库事务处理，更新和删除操作需要保存行级别的版本信息。堆存储表最适合小的维度表，尤其是在数据初次加载后频繁更新的场景下。

由于堆存储是默认的存储模型，所以您可以通过如下语句创建堆存储表：

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

追加优化存储（Append-Optimized Storage）

追加优化存储模型适合数据仓库中非规范化的事实表，后者通常是系统中最大的表。事实表通常是批量加载进入数据仓库，并且用于只读的查询中。相对于堆存储模型，追加优化存储省去了行级别版本信息存储开销（每行大约20个字节），并使得存储页结构更加地精简和易于优化。追加优化存储是为数据批量加载而优化的，所以我们并不推荐单行的插入操作。

您可以通过使用 CREATE TABLE 命令的 WITH 子句来指定表存储模型。下面的例子是创建一个没有数据压缩的追加优化存储表：

```
=> CREATE TABLE bar (a int, b text)
          WITH (appendonly=true)
          DISTRIBUTED RANDOMLY;
```

在一个可串行化的事务里面，对追加优化表进行更新和删除操作是不允许的，并且会导致事务中断。另外，追加优化表不支持 CLUSTER， DECLARE...FOR UPDATE，和触发器。

行存储还是列存储

HashData 数据仓库提供了行存储、列存储以及两者组合的存储选项。在这个小节里，我们将讨论如何根据您的数据和工作负载决定存选项，从而取得最优的查询性能。基本原则如下：

- 行存储：适合包含很多迭代事务查询以及需要访问记录中大部分列查询的在线事务处理（OLTP）工作负载。对于这种场景，由于行存储将一条记录所有列的数据放在一起了，读取访问会非常高效。
- 列存储：适合只需要对表中少数列进行聚合操作的数据仓库工作负载，或者定期对表中某一列进行更新（保持其它列不变）的工作负载。

对于大部分通用的或者混合的工作负载，行存储在灵活性和性能方面做了最好的平衡。然而，在某些应用场景中，列存储模型提供了更加高效的 I/O 和存储使用。在选择行存储还是列存储的时候，可以考虑以下几个需求：

- 表数据的更新。如果需要频繁地加载和更新表数据的话，那么您应该选择行导向的堆存储。只有追加优化表支持列存储。
- 频繁的插入操作。如果插入操作很多的话，行导向的存储模型会比较合适。由于每列数据需要写到磁盘上的不同文件，列存储表对写操作不友好。
- 查询中需要访问的列数。如果查询中的 SELECT 列表或者 WHERE 子句包含了表的大多数列，那么您应该考虑使用行存储。列存储表非常适合对表中某一列进行聚合计算并且 WHERE or HAVING 谓词也是针对聚合列。例如：

```
=> SELECT SUM(salary) ...

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

另外一种适合列存储的查询是，WHERE 谓词只正针对某一列并且查询结果返回少数几列表数据。例如：

```
=> SELECT salary, dept, ... WHERE state = 'CA'
```

- 表中列的数量。如果表中的每行数据量比较少，或者表中的大部分列在查询中都会被访问到，这种场景中，相对于列存储，行存储会更高效。对于那些包含很多列的表，如果查询只访问到其中少数一部分列，那么列存储能够提供高好的性能。
- 压缩。由于同一列的数据类型是相同的，所以相对于行式存储，列示存储存在更多数据压缩空间。例如，很多种压缩算法都利用了相邻数据的相似度进行压缩。另一方面，越高的压缩比意味着数据的随机访问越困难，因为数据首先得解压了才能被读取。

创建一个列存储表

您可以通过 CREATE TABLE 语句中的 WITH 子句来指定表的存储选项。默认的情况下，创建的新表都是行式的堆存储表。只有追加优化的表才支持列示存储。下面的例子中，我们创建一个列示存储表：

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

数据压缩（只适用于追加优化存储）

在 HashData 数据仓库中，对于追加优化表，存在两种不同级别的数据压缩：

- 表级别的压缩作用于整张表。
- 列级别的压缩作用于表中的某一列。您可以针对表中的不同列使用不同的压缩算法。

下面的表格总结了现在支持的压缩算法

表导向	支持的压缩类型	支持的压缩算法
行式存储	表压缩	ZLIB
列式存储	列压缩和表压缩	ZLIB 和 RLE_TYPE

在选择压缩算法和压缩级别的时候，您需要考虑下列因素：

- CPU 的使用。您必须确保您的计算节点有足够的 CPU 计算能力去压缩和解压数据。
- 压缩比和磁盘大小。尽可能减少磁盘占用空间是一方面，另一方面我们也需要考虑压缩和扫描数据时所消耗的时间和 CPU 计算能力。在这者之间找到一个合适的平衡点非常关键。
- 压缩速度。zlib 提供了 1-9 的压缩级别。一般来说，级别越高，压缩比越高，但是压缩速度越慢。
- 解压和扫描的速度。压缩数据的查询性能由很多因素决定，包括硬件、查询参数配置和其它因素。为了做出最合适的选择，我们建议您在实际环境中进行性能测试比较。

创建一张压缩表

您可以通过 CREATE TABLE 语句中的 WITH 子句来指定表的存储选项。只有追加优化的表支持压缩。下面的例子演示了如何创建一张使用 zlib 算法、压缩级别为 5 的追加优化表：

```
=> CREATE TABLE bar (a int, b text)
          WITH (appendonly=true, compresstype=zlib, compresslevel=5);
```

每一列单独的压缩算法

下面的例子演示了，对于一张列式存储的表，如何为每一列指定单独的压缩算法：

```
=> CREATE TABLE bar (
      a int ENCODING (compresstype=zlib, compresslevel=5, blocksize=524288),
      b text ENCODING (compresstype=rle_type, compresslevel=3, blocksize=2097152))
      WITH (appendonly=true, orientation=column);
```

查看追加优化表的数据压缩和分布

HashData 数据仓库提供了查看追加优化表数据压缩和分布的内置函数。这些函数接受表的对象 ID 或者名字作为查询参数。您可以给表名加上限定的模式名字。

函数	返回类型	描述	
get_ao_distribution(name \	oid)	(dbid, tuplecount)集合	返回每个 Segment 数据库的元组个数
get_ao_compression_ratio(name \	oid)	float8	返回压缩追加优化表的压缩比；或者 -1

数据压缩比是作为整张表的值返回的。例如，如果返回值是 3.19，或者 3.19:1，那么意味这压缩前的数据大小是压缩后数据大小的 3 倍多一点。

表分布函数返回的是元组的集合，表明每个 Segment 数据库中表元组的数量。例如，在一个包含 4 个 Segment 数据库、数据库 ID 从 0 到 3 的系统中，函数返回如下类似的结果：

```
=> SELECT get_ao_distribution('lineitem_comp');
 get_ao_distribution
---------------------
(0,7500721)
(1,7501365)
(2,7499978)
(3,7497731)
(4 rows)
```

RLE_TYPE 压缩算法

对于列级别的压缩类型，HashData 数据仓库支持 Run-length Encoding（RLE）压缩算法。RLE 算法的原理是将重复出现的值存成一个值和出现的次数。举个例子，一张表包含两列，一列是日期 date，另一列是描述 description。假设这张表中有 200000 行数据中 date 的值是 date1，400000 行数据中 date 的值是 date2，那么使用 RLE 压缩后的数据内容类似 date1 200000，date2
400000。RLE 算法不合适数据中只有很少重复值的表。这种情况下，RLE 反而让需要存储的数据量变大。

RLE 压缩算法分成四中级别。级别越高，压缩比越高，但是压缩速度越慢。

更改一张表

您可以通过使用 ALTER TABLE 语句来更改一张表的定义，包括列的定义、数据分布策略、存储模型和分区结构。例如，给表中的某一列增加非空约束：

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

改变表的数据分布策略

对于一张分区表，表数据分布策略的改变会影响到其所有的子分区表。这个操作将保留包括表的属主在内的其它表属性。例如，下面的命令将表 sales 分布在所有 Segment 数据库中的数据以 customer_id
列为分布键重新分布：

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

当您修改了一个张表的 hash 分布策略后，这张表的数据会自动地重分布。但是，将表的分布策略改成随机分布不会导致数

据的重分布。例如，下面这个例子不会有立即的效果：

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

重分布表数据

我们可以在 ALTER TABLE 的 WITH 子句中指定 REORGANIZE=TRUE 来对表数据进行重分布。当出现数据倾斜问题或者有新的计算资源加入到系统中，重组织数据是一个可性的解决方案。例如，下面这个例子中，表数据将基于现有的数据分布策略（包括随机分布）对数据进行重分布：

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

修改表的存储模型

表的存储模型、压缩和存储导向（行式存储还是列式存储）只能在创建的时候指定。如果需要修改表的存储模型，您需要使用正确的存储选项创建一张新的表，将数据从旧表中加载到新表中，然后删除旧表，将新表重命名为旧表的名字。您当然也需要重新设置所有的表权限。例如：

```
CREATE TABLE sales2 (LIKE sales)
WITH (appendonly=true, orientation=column,
      compresstype=zlib, compresslevel=5);
INSERT INTO sales2 SELECT * FROM sales;
DROP TABLE sales;
ALTER TABLE sales2 RENAME TO sales;
GRANT ALL PRIVILEGES ON sales TO admin;
GRANT SELECT ON sales TO guest;
```

删除一张表

您可以通过 DROP TABLE 语句将一张表从数据库中删除。例如：

```
=> DROP TABLE mytable;
```

如果只是清空表数据而不删除表定义的话，您可以使用 DELETE 或者 TRUNCATE 语句。例如：

```
DELETE FROM mytable;
TRUNCATE mytable;
```

DROP TABLE 语句同时会删除目标表上的所有索引、规则、触发器和约束。通过指定 CASCADE，您还可以一并删除依赖于这张表的视图。

对大表进行分区处理

表分区通过将数据逻辑上划分到多个较小，更容易管理的小表，来支持超大表，例如：事实表（fact table）。HashData 数据仓库查询优化器通过利用分表信息，只检索满足查询要求的数据，从而避免检索大表的所有内容，最终提高查询性能。

- 表分区概述
- 选择分区表策略
- 创建分区表
- 向分区表加载数据
- 验证分区表策略
- 查看分区表设计
- 分区表的维护

表分区概述

表分区不改变计算节点之间数据的分布规则。表分布存储是 HashData 数据仓库是在物理上将分区表和普通表数据存储在多

个 segment 节点上，从而允许并行查询处理。表分区是 HashData 数据仓库在逻辑上将大表数据分开存放，来提升查询性能并且简化数据仓库的维护任务，例如：将旧数据从数据仓库删除。

HashData 数据仓库支持：

- 范围分区：按照数值范围进行分区，例如：日期或价格。
- 列表分区：按照列表包含的数值进行分区，例如：销售地区或产品线。
- 范围分区和列表分区的组合使用。

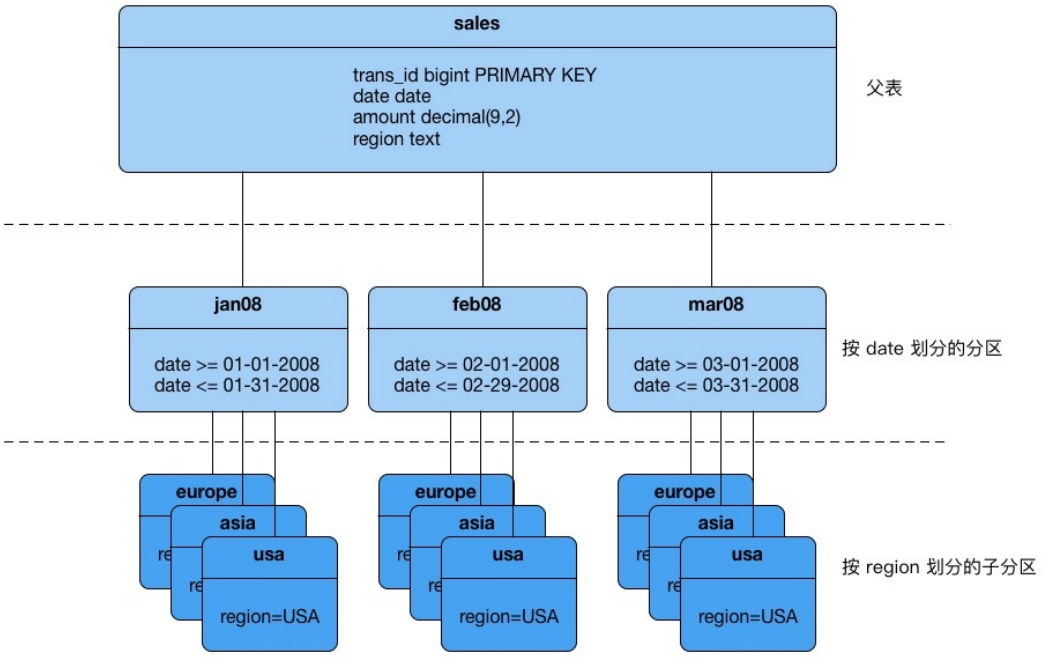


图**1.** 多层分区表结构

HashData 数据仓库的表分区介绍

HashData 数据仓库通过将数据打散成多份来支持并行处理。表分区是在建表语句 CREATE TABLE 中指定 PARTITION BY（以及可选的 SUBPARTITION BY）。分区将会创建一张顶层（父）表以及一层或多层子表。在内部，HashData 数据仓库将会对顶层表和子表创建继承关系，这与 postgres 的 INHERITS 子句十分类似。

HashData 数据仓库将会根据创建表的分区定义为每个分区创建一个 CHECK 约束条件，该约束将会限制该分区能够包含的数据。查询优化器将会利用 CHECK 约束来决定扫描哪些表分区可以满足查询指定的过滤条件。

HashData 数据仓库系统信息表保存分区的结构信息，这样每当有记录插入到顶层的父表时，系统能够正确地将其插入到子分区中。要改变分区结构或表结构，在父表上使用 ALTER TABLE 以及 PARTITION 子句即可。

要插入数据到分区表，你需要指定根分区表（也就是 CREATE TABLE 命令时指定的表）。您还可以在 INSERT 语句中指定分区表的叶子表。如果数据不满足该分区的要求，将会得到错误提示。INSERT 语句不支持指定的非叶子表的分区。执行其他 DML 语句（例如：UPDATE 和 DELETE）时，不支持指定任何子分区。要执行前面的命令，必须指定根分区表（也就是 CREATE TABLE 命令时指定的表）。

选择分区表方案

并不是所有表都是适合使用表分区。如果下面问题的答案大部分或者全部是肯定的，那么表分区将会是一种重要的提升查询性能的数据库设计方案。如果大部分问题回答是否定的，那么采用分区表策略就不大合适了。最后，还需要对设计方案进行测试，以确保查询性能与期望相符。

- 这张表的数据足够多吗？一般来说，数据量很多的事实表（facttable）比较适合采用表分区。如果表中有上百万甚至上亿条记录时，通过逻辑上将数据分散到多个小数据表中，性能将会得到较大提升。对于只有几千条记录甚至更少的表，

管理上的维护开销将会完全抵消带来的性能收益。

- 目前的性能无法满足业务需求？与大部分性能调优的初衷类似，对表进行分区处理应该是在对该表的查询响应时间无法满足需求时进行的。
- 查询过滤条件是否有较固定的访问模型？通过分析查询中 WHERE 子句中涉及的数据列信息，判断是否存在一些列经常作为数据检索的条件。例如：如果大部分的查询趋向使用日志来检索数据，那么一个按月或星期分割的、基于日期的分区设计可能对提升查询性能有较好效果。又或者，查询倾向于按照地区进行数据检索，那么考虑利用列表值进行分区的时候，根据地区信息分区效果可能比较好。
- 数据库仓库是否需要保存一段时间的历史数据？另一个重要的考虑因素就是在机构的商业需求中，对历史数据的维护操作需求。例如，如果数据仓库需要维护过去 12 个月的数据，那么按照月份对数据进行分区，您就可以轻易的将最旧的月份分区直接删除，并将当前数据加载到最近的月份分区中。
- 数据能够根据某些条件分成基本相等的部分吗？选择分区条件时，应该保证数据被分割后，每个分区表的数据量尽可能地均匀。如果每个分区包含的数据量近似相等，查询性能提升与分区表的数量直接相关。例如，将一张大表分成 10 个分区后，如果分区条件能够满足查询检索条件，查询对于分区表的处理能够比对没有分区之前的表快 10 倍。

创建的分区数量不应该超过实际需求数量。创建过多的分区可能会拖慢管理和维护作业，例如：清理工作，节点恢复，集群扩展，查看磁盘使用情况等。

只有当查询优化器能够利用查询过滤条件，来消除一些分区扫描时，表分区才能提高查询性能。查询扫描所有分区的运行时间实际上比扫描没有分区时候的运行时间更长，所以如果没有什么查询能消除一些分区扫描时，请不要使用表分区。可以通过检查查询计划来确认分区是否被消除。

在使用多层表分区时，请注意分区文件数的增长速度可能超出您的预期。例如，一张按照天和城市进行分区的表，当存储 1000 天和 1000 个城市时，需要创建一百万个分区。列存表，将每列独立存储成一个物理表，对于一张有 100 列的表来说，系统管理该表需要管理 1 亿个文件。

在使用多层表分区时，您可以考虑使用单层表分区和位图索引（Bitmap 索引）。由于索引将会降低数据加载速度，因此推荐您使用性能测试来针对您的数据和模式进行评测，选取最优策略。

创建分区表

在使用 CREATE TABLE 命令创建表时，您可以对表进行分区操作。本主题向您展示用于创建不同类型分区表的 SQL 语法。

要将一只表进行分区：

1. 确定分区表的设计：日期范围，数值范围，列表值。
2. 选择用于分区的数据列。
3. 确定分区的层数。例如，你可以首先按照日期范围根据月份进行分区，在按月分区的子分区中，按照销售地区分区。
4. 定义按日期划分的分区表。
5. 定义数值划分的分区表。
6. 定义列表值分区。
7. 定义多层分区。
8. 对已经存在的进行分区。

定义按日期划分的分区表

日期划分的分区表使用一个日期或时间戳列做为分区键值列。如果需要，子分区可以使用与父分区相同的分区键值列。例如：父分区按照月份进行划分，子分区使用日期进行划分。在分区时，应该考虑按照最细的粒度来进行。例如：对于按照日期划分的分区表来说，您应该直接按照天数 创建分区，这样创建 365 个按天存储的分区表即可。应该避免先按照年，再按照月，最后按照天来创建分区表的模式。虽然多层的分区设计可以降低查询计划的时间，但是设计上扁平的分区表在运行时，速度更快。

您可以通过指定起始值（START），终止值（END）和增量子句（EVERY）指出分区的增量值，让 HashData 数据仓库来自动地生成分区。默认情况下，起始值总是包含的（闭区间），而终止值是排除的（开区间）。例如：

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( START (date '2008-01-01') INCLUSIVE
   END (date '2009-01-01') EXCLUSIVE
   EVERY (INTERVAL '1 day') );
```

您还可以为每个分区指定独立的名称，例如：

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( PARTITION Jan08 START (date '2008-01-01') INCLUSIVE ,
  PARTITION Feb08 START (date '2008-02-01') INCLUSIVE ,
  PARTITION Mar08 START (date '2008-03-01') INCLUSIVE ,
  PARTITION Apr08 START (date '2008-04-01') INCLUSIVE ,
  PARTITION May08 START (date '2008-05-01') INCLUSIVE ,
  PARTITION Jun08 START (date '2008-06-01') INCLUSIVE ,
  PARTITION Jul08 START (date '2008-07-01') INCLUSIVE ,
  PARTITION Aug08 START (date '2008-08-01') INCLUSIVE ,
  PARTITION Sep08 START (date '2008-09-01') INCLUSIVE ,
  PARTITION Oct08 START (date '2008-10-01') INCLUSIVE ,
  PARTITION Nov08 START (date '2008-11-01') INCLUSIVE ,
  PARTITION Dec08 START (date '2008-12-01') INCLUSIVE
                END (date '2009-01-01') EXCLUSIVE );
```

除了最后一个分区外，其他分区的终止值可以省略。在上例中，Jan08 的终止值就是 Feb08 的起始值。

定义数值划分的分区表

使用数值范围的分区表，利用单独的数值类型列做为分区键值列。例如：

```
CREATE TABLE rank (id int, rank int, year int, gender
char(1), count int)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
( START (2001) END (2008) EVERY (1),
  DEFAULT PARTITION extra );
```

要了解更多关于默认分区的信息，请参考 增加默认分区。

定义列表值分区

使用列表值进行分区的表可以选用任何支持等值比较的数据类型列做为分区键值列。并且使用列表值进行分区还可以支持多列（复合）分区键值列，与之对应的范围分区只支持单列做为分区键值列。对于列表值分区来说，您必须为每一个要创建的分区指定对应的列表值。例如：

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

要了解更多关于默认分区的信息，请参考 增加默认分区。

定义多层分区

您可以通过在分区下创建子分区来实现多层分区的设计。使用子分区定义模版能够保证每个分区拥有一致的子分区结构，即使在未来添加新分区时，该模版仍然能够保证新的子分区结构。

例如，下面的 SQL 语句可以创建与图 1 一致的两层分区表：

```
CREATE TABLE sales (trans_id int, date date, amount
decimal(9,2), region text)
DISTRIBUTED BY (trans_id)
PARTITION BY RANGE (date)
SUBPARTITION BY LIST (region)
SUBPARTITION TEMPLATE
( SUBPARTITION usa VALUES ('usa'),
  SUBPARTITION asia VALUES ('asia'),
  SUBPARTITION europe VALUES ('europe'),
  DEFAULT SUBPARTITION other_regions)
  (START (date '2011-01-01') INCLUSIVE
   END (date '2012-01-01') EXCLUSIVE
   EVERY (INTERVAL '1 month'),
   DEFAULT PARTITION outlying_dates );
```

下面是一个三层的分区表定义，sales 表分别按照年度，月份，地区进行分区。这里的 SUBPARTITION TEMPLATE 子句保证了每一个按年度的分区表拥有完全一致的子分区表结构。这个例子中，还在每层结构中声明了一个 DEFAULT 分区。

```
CREATE TABLE p3_sales (id int, year int, month int, day int,
region text)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
    SUBPARTITION BY RANGE (month)
       SUBPARTITION TEMPLATE (
        START (1) END (13) EVERY (1),
        DEFAULT SUBPARTITION other_months )
           SUBPARTITION BY LIST (region)
             SUBPARTITION TEMPLATE (
               SUBPARTITION usa VALUES ('usa'),
               SUBPARTITION europe VALUES ('europe'),
               SUBPARTITION asia VALUES ('asia'),
               DEFAULT SUBPARTITION other_regions )
( START (2002) END (2012) EVERY (1),
  DEFAULT PARTITION outlying_years );
```

> 小心：当您基于范围信息创建多层分区表时，很容易导致系统创建大量包含很少甚至没有数据的子分区表。
> 这种情况下，将导致系统信息表中包含大量子分区信息，最终增加优化和执行查询需要的时间和内存。
> 可以通过增加范围间隔或不同的分区策略来减少创建的子分区数量。

对已经存在的进行分区

您只能在创建表时对表进行分区操作。如果您想要对一张进行分区操作，您需要先创建一张分区表，从旧表加载数据到新表，删除旧表，并将新的分区表名称改为旧表。您还需要对表的权限进行重新授予的操作，例如：

```
CREATE TABLE sales2 (LIKE sales)
PARTITION BY RANGE (date)
( START (date '2008-01-01') INCLUSIVE
   END (date '2009-01-01') EXCLUSIVE
   EVERY (INTERVAL '1 month') );
INSERT INTO sales2 SELECT * FROM sales;
DROP TABLE sales;
ALTER TABLE sales2 RENAME TO sales;
GRANT ALL PRIVILEGES ON sales TO admin;
GRANT SELECT ON sales TO guest;
```

分区表的限制

在每层的分区上，分区表都能创建最多 32,767 个子分区。

分区表的主键或者唯一约束并序包含所有分区的键值列。唯一索引可以不包含分区键值列，但是该唯一索引只对分区表部分数据有效，不能对整个分区表生效。

将叶子分区交换为外部表不支持下面情况：被交换的分区是通过 SUBPARTITION 子句创建或者被交换的分区包含子分区。要了解将叶子分区交换为外部表，请参考 将叶子分区交换为外部表。

下面是分区表的子叶分区是外部表时的一些限制：

- 外部表分区必须是一个可读外部表。任何试图反问和修改外部表数据的命令都会返回失败。例如：
  - INSERT，DELETE 和 UPDATE 命令都会试图修改外部表分区的数据，将会返回错误。
  - TRUNCATE 命令返回错误。
  - COPY 命令不能将数据拷贝到分区表，因为该操作可能修改外部表分区。
  - COPY 命令在试图拷贝包含外部表分区的分区表时，将会返回错误。但是可以通过指定 IGNORE EXTERNAL PARTITIONS 子句来避免该错误。如果您使用了上面的子句，外部表分区中的数据将不会被拷贝。要想对包含外部表分区的分区表使用 COPY 命令， 利用 SQL 查询语句来拷贝数据。例如：如果表 my_sales 包含了一个外部表分区， 下面的命令将会把所有数据输出到标准输出：

    ```
    COPY (SELECT * from my_sales ) TO stdout
    ```

  - VACUUM 将会跳过外部表分区。
- 下列命令在数据不发生改变的情况下能够支持外部表分区。否则，将返回一个错误：
  - 添加或删除列。
  - 变更列的类型。
- 下面的 ALTER PARTITION 操作不支持外部表分区：
  - 设置子分区定义模版。
  - 修改分区属性。
  - 创建默认分区。
  - 设置数据分布键值（DISTRIBUTED BY）。
  - 对数据列添加或删除非空约束。
  - 添加或删除约束。
  - 分裂外部表分区。

向分区表加载数据

在创建了分区表结构后，顶层的父表是没有数据的。数据自动地存储到最底层的子分区中。在多层分区结构中，只有在结构最底层的子分区表才会包含数据。

如果记录不满足任何子分区表的要求，插入将会被拒绝，数据加载都会失败。要避免不合要求的记录在加载时被拒绝导致的失败，可以在定义分区结构时，创建一个默认分区（DEFAULT）。任何不满足分区 CHECK 约束记录都会被加载到默认分区。请查看 增加默认分区。

在查询运行时，查询优化器将会扫描整个表的继承结构，使用 CHECK 约束来判断需要扫描哪些子分区来满足查询条件。默认分区每次都会被扫描。包含数据的默认分区将会拖慢整体的扫描时间。

当使用 COPY 或 INSERT 命令向父表加载数据时，数据将会自动的存储到正确的分区中。

向分区表加载数据的最佳实践就是创建一个中间的临时表，向该表加载数据，然后通过交换分区的方式加入到分区表中。请查看 交换分区。

验证分区表策略

当您根据查询条件对表进行分区后，可以通过使用 EXPLAIN 命令检查查询计划的方式，来验证查询优化器只决定扫描和请求相关的分区数据。

例如，假设图 1 中的 sales 表是按照日期范围每个月创建一个分区，并且根据地区创建子分区。对于下面的查询：

```
EXPLAIN SELECT * FROM sales WHERE date='01-07-12' AND region='usa';
```

上面查询的查询计划应该显示出表扫描只考虑如下分区：

- 默认分区，返回 0-1 条结果（如果创建了默认分区）
- 2012 年 1 月分区（sales_1_prt_1）返回 0-1 条
- USA 地区子分区（sales_1_2_prt_usa）返回多条结果

下面是查询计划的部分内容：

```
->  Seq Scan on sales_1_prt_1 sales (cost=0.00..0.00 rows=0 width=0)
      Filter: "date"=01-07-08::date AND region='USA'::text
->  Seq Scan on sales_1_2_prt_usa sales (cost=0.00..9.07 rows=20 width=40)
```

您需要确保优化器没有扫描不必要的分区或子分区（例如：扫描了查询中没有指定的时间或地区），
并且顶层表返回 0-1 行结果。

排查选择性分区扫描

下面的限制可能导致查询计划显示出没有选择部分分区表结构的情况。

- 查询优化器只能在查询使用直接和简单的限制条件和不变运算符（immutable operator），进行选择性分区扫描。例如：=，\<，\<=，>，>= 和 \<>。
- 选择性的扫描只能支持稳定函数（STABLE）和不变函数（IMMUTABLE），不支持易变（VOLATILE）函数。例如，当 WHERE 子句中包含 date > CURRENT_DATE 时，查询优化器可以选择性的扫描分区表，但是 time > TIMEOFDAY 就不会使用选择性分区扫描。

查看分区表设计

通过 pg_partitions 视图，您可以查看分区表设计信息。下面示例可以查看 sales 表的分区设计信息：

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

如下表和视图向您提供分区表相关信息：

- pg_partition - 跟踪分区表及其继承关系信息。
- pg_partition_templates - 创建子分区使用的子分区模版信息。
- pg_partition_columns - 分区表分区键值信息。

分区表的维护

要维护分区表，可以对顶层的分区表（根表）使用 ALTER TABLE 命令。最常见的维护场景就是对于范围分区的设计，通过删除旧分区，创建新分区，来维护一个特定数据窗口。您还可以将旧分区转换（exchange）成追加表格式，并使用压缩方式来节省磁盘的存储空间。如果您的分区表包含了一个默认分区，可以通过分裂默认分区来添加新的分区。

- 增加分区
- 重命名分区
- 增加默认分区
- 删除分区
- 清空分区
- 交换分区
- 分裂分区
- 修改子分区模版
- 将叶子分区交换为外部表

> 重要：在定义或修改分区表时，请指定分区名称（不是表名称）。尽管您可以直接通过 SQL 命令直接对表（或分区表）进行查询或数据加载操作。但是您只能通过 ALTER TABLE ... PARTITION 子句来修改分区表的结构。

分区的名称可以省略。如果一个分区没有名字，使用下面的表达式可以指定分区：

```
PARTITION FOR (value)

PARTITION FOR(RANK(number))
```

增加分区

您可以通过 ALTER TABLE 命令向已有的分区表中添加新的分区。如果原始的分区表包含了用来定义子分区的子分区模版，新增加的分区会根据模版信息创建子分区。例如：

```
ALTER TABLE sales ADD PARTITION
            START (date '2009-02-01') INCLUSIVE
            END (date '2009-03-01') EXCLUSIVE;
```

如果在创建表时没有指定子分区模版，增加新分区时，您需要指定子分区信息：

```
ALTER TABLE sales ADD PARTITION
            START (date '2009-02-01') INCLUSIVE
            END (date '2009-03-01') EXCLUSIVE
      ( SUBPARTITION usa VALUES ('usa'),
        SUBPARTITION asia VALUES ('asia'),
        SUBPARTITION europe VALUES ('europe') );
```

当您向已经存在的分区增加子分区时，需要指定分区来进行操作：

```
ALTER TABLE sales ALTER PARTITION FOR (RANK(12))
      ADD PARTITION africa VALUES ('africa');
```

> 注意：您不能向包含默认分区的分区表中增加新分区。您需要通过将默认分区分裂来增加分区。
> 参考 分裂分区

重命名分区

下面列出分区表命名规则。分区表的子表名称需要保证唯一性且遵守名称长度限制。

```
<parentname>_<level>_prt_<partition_name>
```

一个子表的名称示例：

```
sales_1_prt_jan08
```

对于自动生成的范围分区表，如果您没有指定分区名称，将会自动分配一个数值来生成分区名：

```
sales_1_prt_1
```

要修改分区表的子表名称，需要对顶层父表运行重命名操作。修改将会作用在所有相关的子表
分区之上。下面示例的命令：

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

将修改相关的子表名称为：

```
globalsales_1_prt_1
```

您可以修改指定分区名称，来更加便捷的识别子分区：

```
ALTER TABLE sales RENAME PARTITION FOR ('2008-01-01') TO jan08;
```

操作将修改相关的子表名称为：

```
sales_1_prt_jan08
```

当使用 ALTER TABLE 命令修改分区表时，需要使用分区名称（jan08），不要使用完整的表名（sales_1_prt_jan08）。

> 注意：在 ALTER TABLE 语句时，你不能提供分区名称。例如：ALTER TABLE sales... 是正确的，ALTER TABLE sales_1_part_jan08... 是不允许的。

增加默认分区

您可以使用 ALTER TABLE 来向分区表中增加默认分区。

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

如果您的分区表是多层的，那么每一层结构都需要包含默认分区：

```
ALTER TABLE sales ALTER PARTITION FOR (RANK(1)) ADD DEFAULT PARTITION other;

ALTER TABLE sales ALTER PARTITION FOR (RANK(2)) ADD DEFAULT PARTITION other;

ALTER TABLE sales ALTER PARTITION FOR (RANK(3)) ADD DEFAULT PARTITION other;
```

如果输入的数据不满足分区的 CHECK 约束条件，并且没有创建默认分区，数据将被拒绝插入。默认分区能够保证在输入数据不满足分区时，能够将数据插入到默认分区。

删除分区

您可以通过 ALTER TABLE 命令从分区表中删除分区。如果您要删除的分区包含子分区，删除操作将会自动地将子分区（及其数据）删除。对于范围分区来说，通常是将较老的分区对应范围删除，这样来将数据仓库的旧删除删除掉。例如：

```
ALTER TABLE sales DROP PARTITION FOR (RANK(1));
```

清空分区

您可以通过 ALTER TABLE 命令来清空分区。如果您要清空的分区包含子分区，清空操作将会自动地将子分区清空。

```
ALTER TABLE sales TRUNCATE PARTITION FOR (RANK(1));
```

交换分区

您可以通过 ALTER TABLE 命令来交换分区。交换分区是将一张数据表与已经存在的分区进行数据文件交换。你只能交换分区结构中最底层的分区（只有包含数据的分区才能被交换）。

分区交换对于数据加载非常有帮助。例如：将数据加载到临时表，再食用交换命令将临时表加载到分区表中。您还可以使用交换分区的方式将旧表的存储结构更改为追加表格式。例如：

```
CREATE TABLE jan12 (LIKE sales) WITH (appendonly=true);
INSERT INTO jan12 SELECT * FROM sales_1_prt_1 ;
ALTER TABLE sales EXCHANGE PARTITION FOR (DATE '2012-01-01') WITH TABLE jan12;
```

> 注意：这个例子使用的是单层分区定义的 sales 表，这里的 sales 表是没有运行前面示例操作 时候的状态。
>
> 警告：如果您指定了 WITHOUT VALIDATION 子句，您必须保证用来交换的表中的数据是符合分区约束条件的。否则，当查询涉及到该分区时，返回的查询结果可能不正确。

HashData 数据仓库服务器配置参数 gp_enable_exchange_default_partition 参数用来控制 是否允许使用 EXCHANGE DEFAULT PARTITION 子句。 在 HashData 数据仓库中，此参数默认值 为 off，当您在 ALTER TABLE 命令中使用此子句时，将会得到错误提示。

> 警告：在交换默认分区前，您必须确保将要交换的表中的数据（也就是新的默认分区）是符合默认分区定义的。例如，新默认分区中的已经存在的数据不能是满足分区表中其他分区条件的数据。否则，当查询涉及到使用交换后默认分区的时，查询结果可能不正确。

分裂分区

分裂分区能够将一个分区，分成两个分区。您可以使用 ALTER TABLE 命令来分裂分区。您只能在分区结构的最底层进行分裂操作（只能分裂包含数据的分区）。满足您提供用于分裂值的数据，将会存放到您提供的第二个分区中。

下面示例向您展示，将一个按月划分的分区，分裂成两个独立的分区。第一个分区包含 1 月 1 日到 1 月 15 日的数据，第二个分区包含 1 月 16 日到 1 月 31 日的数据：

```
ALTER TABLE sales SPLIT PARTITION FOR ('2008-01-01')
AT ('2008-01-16')
INTO (PARTITION jan081to15, PARTITION jan0816to31);
```

如果您的分区表中包含默认分区，您必须通过分裂默认分区的方式来增加新的分区。

当您使用 INTO 子句时，您需要将默认分区做为第二个分区名称。下面示例向您展示，将一个
默认按照范围分区的分区表，增加一个专门保存 2009 年 1 月数据的分区的语句：

```
ALTER TABLE sales SPLIT DEFAULT PARTITION
START ('2009-01-01') INCLUSIVE
END ('2009-02-01') EXCLUSIVE
INTO (PARTITION jan09, default partition);
```

修改子分区模版

您可以使用 ALTER TABLE SET SUBPARTITION TEMPLATE 来修改分区表的子分区模版定义。在您修改子分区表模版定义后添加的分区将按照新的分区定义创建。但是对于已经存在分区，新的定义将不会生效。

下面示例介绍修改分区表的子分区表模版定义：

```
CREATE TABLE sales (trans_id int, date date, amount decimal(9,2), region text)
  DISTRIBUTED BY (trans_id)
  PARTITION BY RANGE (date)
  SUBPARTITION BY LIST (region)
  SUBPARTITION TEMPLATE
    ( SUBPARTITION usa VALUES ('usa'),
      SUBPARTITION asia VALUES ('asia'),
      SUBPARTITION europe VALUES ('europe'),
      DEFAULT SUBPARTITION other_regions )
  ( START (date '2014-01-01') INCLUSIVE
    END (date '2014-04-01') EXCLUSIVE
    EVERY (INTERVAL '1 month') );
```

接下来执行下面命令修改上面的子分区模版定义：

```
ALTER TABLE sales SET SUBPARTITION TEMPLATE
( SUBPARTITION usa VALUES ('usa'),
  SUBPARTITION asia VALUES ('asia'),
  SUBPARTITION europe VALUES ('europe'),
  SUBPARTITION africa VALUES ('africa'),
  DEFAULT SUBPARTITION regions );
```

当您向 sales 表中增加一个按日期分隔的范围分区时，它将包含新的地区列表对应的子分区 Africa。例如下面的示例，将会创建子分区：usa，asia，europe，africa 和一个名叫 regions 的默认分区：

```
ALTER TABLE sales ADD PARTITION "4"
  START ('2014-04-01') INCLUSIVE
  END ('2014-05-01') EXCLUSIVE ;
```

要查看分区表 sales 对应创建的子表，你可以使用 psql 命令行工具，并使用命令： \d sales。

要删除子分区模版定义，使用 SET SUBPARTITION TEMPLATE 子句，并使用圆括号提供一个
空定义即可。例如下面示例用来清空子分区模版定义：

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

将叶子分区交换为外部表

您可以将分区表的叶子分区交换为可读外部表（readable external table）。外部表数据可以存放在外部数据源上，例如：青云对象存储。

例如，您有一张按月份分区的分区表，在该表上的查询主要访问较新的数据，您可以将旧数据和访问较少的数据拷贝到外部表，最后将该分区与外部表进行交换分区。对于只访问新数据的查询，您还可以利用分区消除来阻止扫描数据较旧和不必要的分区。

在下面的情况下，您不能交换叶子分区和外部表：

- 被交换的分区是通过 SUBPARTITION 子句创建或者被交换的分区包含子分区。
- 分区表中某个列上带有 CHECK 约束或非空约束。

要了解包含外部表的分区表限制，可以参考分区表的限制。

示例：将分区交换为外部表

这里给出一个简单的例子，该例子将分区表中的叶子分区交换为一张外部表。分区表包含的数据
从 2000 年到 2003 年。

```
CREATE TABLE sales (id int, year int, qtr int, day int, region text)
  DISTRIBUTED BY (id)
  PARTITION BY RANGE (year)
  ( PARTITION yr START (2000) END (2004) EVERY (1) ) ;
```

这张分区表包含了四张叶子分区。每隔叶子分区包含了一年的数据。叶子分区 sales_1_prt_yr_1 包含了 2000 年的数据。下面的步骤将该分区交换为使用 gpfdist 协议的外部表：

1. 确保 HashData 数据仓库系统开启了外部表协议。 下面示例使用 gpfdist 协议。示例命令将会启动 gpfdist 协议服务器：

   ```
   $ gpfdist
   ```

2. 创建可写外部表（writable external table）。 下面的 CREATE WRITABLE EXTENAL TABLE 命令创建一个可写外部表，该表列定义与分区表列定义相同。

   ```
   CREATE WRITABLE EXTERNAL TABLE my_sales_ext ( LIKE sales_1_prt_yr_1 )
     LOCATION ( 'gpfdist://gpdb_test/sales_2000' )
     FORMAT 'csv'
     DISTRIBUTED BY (id) ;
   ```

3. 创建一张可读外部表，该外部表将会从上一步创建的可写外部表的位置读取数据。下面的 CREATE EXTENAL TABLE 命令创建的外部表将会使用与可写外部表相同的数据。

   ```
   CREATE EXTERNAL TABLE sales_2000_ext ( LIKE sales_1_prt_yr_1)
     LOCATION ( 'gpfdist://gpdb_test/sales_2000' )
     FORMAT 'csv' ;
   ```

4. 将数据从叶子分区拷贝到可写外部表。下面的 INSERT 命令将会把分区表中叶子分区的数据拷贝到外部表中。

```
INSERT INTO my_sales_ext SELECT * FROM sales_1_prt_yr_1 ;
```

5. 交换叶子分区和外部表。 下面的 ALTER TABLE 命令指出 EXCHANGE PARTITION 子句，用来将可读外部表和叶子分区交换。

```
ALTER TABLE sales ALTER PARTITION yr_1
   EXCHANGE PARTITION yr_1
   WITH TABLE sales_2000_ext WITHOUT VALIDATION;
```

   外部表将会变成叶子分区，并且名称为 sales_1_prt_yr_1，而原来的叶子分区将会成为表 sales_2000_ext。

   > 警告：为了确保运行在分区表上的查询结果正确，外部表数据必须符合叶子分区的 CHECK 约束条件。在这个例子中，数据是从带有 CHECK 约束条件的叶子分区表中读取的。

6. 删除从分区表中换出的表。

```
DROP TABLE sales_2000_ext ;
```

你可以重命名叶子分区的名称来标识出 sales_1_prt_yr_1 是一张外部表。

下面示例的命令将会把分区名称改为 yr_1_ext 最终的叶子分区表名称为 sales_1_prt_yr_1_ext。

```
ALTER TABLE sales RENAME PARTITION yr_1 TO  yr_1_ext ;
```

创建和使用序列

通过使用序列，系统可以在新的纪录插入表中时，自动地按照自增方式分配一个唯一 ID。使用序列一般就是为插入表中的纪录自动分配一个唯一标识符。您可以通过声明一个 SERIAL 类型的标识符列，该类型将会自动创建一个序列来分配 ID。

创建序列

CREATE SEQUENCE 命令用来创建和初始化一张特殊的单行序列生成器表，该表名称就是指定序列的名称。序列的名称在同一个模式下，不能与其它序列，表，索引或者视图重名。示例：

```
CREATE SEQUENCE myserial START 101;
```

使用序列

在使用 CREATE SEQUENCE 创建系列生成器表后，可以通过 nextval 函数来使用序列。例如下面例子，向表中插入新数据时，自动获得下一个序列值：

```
INSERT INTO vendors VALUES (nextval('myserial'), 'acme');
```

还可以通过使用函数 setval 来重置序列的值。示例：

```
SELECT setval('myserial', 201);
```

请注意 nextval 操作是不会回滚的，数值一旦被获取，即使最终事务回滚，该数据也被认为已经被分配和使用了。这意味着失败的事务会给序列分配的数值中留下空洞。类似地，setval 操作也不支持回滚。

通过下面的查询，可以检查序列的当前值：

```
SELECT * FROM myserial;
```

修改序列

ALTER SEQUENCE 命令可以修改已经存在的序列生成器参数。例如：

```
ALTER SEQUENCE myserial RESTART WITH 105;
```

删除序列

DROP SEQUENCE 命令删除序列生成表。例如：

```
DROP SEQUENCE myserial;
```

使用索引

在绝大部分传统数据中，索引都能够极大地提高数据访问速度。然而，在像 HashData 数据仓库这样的分布式数据库系统中，索引的使用需要更加谨慎。

HashData 数据仓库执行顺序扫描的速度非常快，索引只用来随机访问时，在磁盘上定位特定数据。由于数据是分散在多个节点上的，因此每个节点数据相对更少。再加上使用分区表功能，实际的顺序扫描可能更小。因为商业智能 (BI) 类应用通常返回较大的结果数据，因此索引并不高效。

请尝试在没有索引的情况下，运行查询。一般情况下，对于 OLTP 类型业务，索引对性能的影响更大。因为这类查询一般只返回一条或较少的数据。对于压缩的 append 表来说，对于返回一部分数据的查询来说性能也能得到提高。这是因为优化器可以使用索引访问来避免使用全表的顺序扫描。对于压缩的数据，使用索引访问方法时，只有需要的数据才会被解压缩。

HashData 数据仓库对于包含主键的表自动创建主键约束。要对分区表创建索引，只需要在分区表上创建索引即可。HashData 数据仓库能够自动在分区表下的分区上创建对应索引。HashData 数据仓库不支持对分区表下的分区创建单独的索引。

请注意，唯一约束会隐式地创建唯一索引，唯一索引会包含所有数据分布键和分区键。唯一约束是对整个表范围保证唯一性的（包括所有的分区）。

索引会增加数据库系统的运行开销，它们占用存储空间并且在数据更新时，需要额外的维护工作。请确保查询集合在使用您创建的索引后，性能得到了改善（和全表顺序扫描相比）。您可以使用 EXPLAIN 命令来确认索引是否被使用。

创建索引时，您需要注意下面的问题点：

- 您的查询特点。索引对于查询只返回单条记录或者较少的数据集时，性能提升明显。
- 压缩表。对于压缩的 append 表来说，对于返回一部分数据的查询来说性能也能得到提高。对于压缩的数据，使用索引访问方法时，只有需要的数据才会被解压缩。
- 避免在经常改变的列上创建索引。在经常更新的列上创建索引会导致每次更新数据时写操作大量增加。
- 创建选择率高的 B-树索引。索引选择率是列的唯一值除以记录数的比值。例如，一张表有 1000 条记录，其中有 800 个唯一值，这个列索引的选择率就是 0.8，这个数值就比较好。唯一索引的选择率总是 1.0，也是选择率最好的。HashData 数据仓库只允许创建包含表数据分布键的唯一索引。
- 对于选择率较低的列，使用 Bitmap 索引。
- 对参与连接操作的列创建索引。对经常用于连接的列（例如：外键列）创建索引，可以让查询优化器使用更多的连接算法，进而提高连接效率。
- 对经常出现在 WHERE 条件中的列创建索引。
- 避免创建冗余的索引。如果索引开头几列重复出现在多个索引中，这些索引就是冗余的。
- 在大量数据加载时，删除索引。如果要向表中加载大量数据，考虑加载数据前删除索引，加载后重新建立索引的方法。这样的操作通常比带着索引加载要快。
- 考虑聚簇索引。聚簇索引是指数据在物理上，按照索引顺序存储。如果您访问的数据在磁盘是随机存储，那么数据库就需要在磁盘上不断变更位置读取您需要的数据。如果数据更佳紧密的存储起来，读取数据的操作效率就会更高。例如：在日期列上创建聚簇索引，数据也是按照日期列顺序存储。一个查询如果读取一个日期范围的数据，那么就可以利用磁盘顺序扫描的快速特性。

聚簇索引

对一张非常大的表，使用 CLUSTER 命令来根据索引对表的物理存储进行重新排序可能花费非常长的时间。您可以通过手工将排序的表数据导入一张中间表，来加上面的操作，例如：

```
CREATE TABLE new_table (LIKE old_table)
       AS SELECT * FROM old_table ORDER BY myixcolumn;
DROP old_table;
ALTER TABLE new_table RENAME TO old_table;
CREATE INDEX myixcolumn_ix ON old_table;
VACUUM ANALYZE old_table;
```

索引类型

HashData 数据仓库支持 Postgres 中索引类型 B树 和 GiST。索引类型 Hash 和 GIN 索引不支持。每一种索引都使用不同算法，因此适用的查询也不同。B 树索引适用于大部分常见情况，因此也是默认类型。您可以参考 PostgreSQL 文档中关于索引的相关介绍。

> 注意：唯一索引使用的列必须和表的分布键值一样（或超集）。append-optimized 存储类型的表不支持唯一索引。对于分区表来说，唯一索引不能对整张表（对所有子表）来保证唯一性。唯一索引可以对于一个子分区保证唯一性。

关于 **Bitmap** 索引

HashData 数据仓库提供了 Bitmap 索引类型。Bitmap 索引特别适合大数据量的数据仓库应用和决策支持系统这种查询，临时性查询特别多，数据改动少的业务。

索引提供根据指定键值指向表中记录的指针。一般的索引存储了每个键值对应的所有记录 ID 映射关系。而 Bitmap 索引是将键值存储为位图形式。一般的索引可能会占用实际数据几倍的存储空间，但是 Bitmap 索引在提供相同功能下，需要的存储远远小于实际的数据大小。

位图中的每一位对应一个记录 ID。如果位被设置了，该记录 ID 指向的记录满足键值。一个映射函数负责将比特位置转换为记录 ID。位图使用压缩进行存储。如果键值去重后的数量比较少，bitmap 索引相比普通的索引来说，体积非常小，压缩效果更好，能够更好的节省存储空间。因此 bitmap 索引的大小可以近似通过记录总数乘以索引列去重后的数量得出。

对于在 WHERE 子句中包含多个条件的查询来说，bitmap 索引一般都非常有效。如果在访问数据表之前，就能过滤掉只满足部分条件的记录，那么查询响应时间就会得到巨大的提升。

何时使用 **Bitmap** 索引

Bitmap 索引特别适用数据仓库类型的应用程序，因为数据的更新相对非常少。Bitmap 索引对于去重后列值在 100 到 10,0000 个，并且查询时经常是类似这样的多列参一起使用的查询性能提升非常明显。但是像性别这种只有两个值的类型，实际上索引并不能提供比较好的性能提升。如果去重后的值多余 10,0000 个，bitmap 索引的性能收益和存储效率都会开始下降。

Bitmap 索引对于临时性的查询性能改进比较明显。在 WHERE 子句中的 AND 和 OR 条件来说，可以利用 bitmap 索引信息快速得到满足条件的结果，而不用首先读取记录信息。如果结果集数据很少，查询就不需要使用全表扫描，并且能非常快的返回结果。

不适合使用 **Bitmap** 索引的情况

如果列的数据唯一或者重复非常少，就应该避免使用 bitmap 索引。bitmap 索引的性能优势和存储优势在列的唯一值超过 10,0000 后就会开始下降。与表中的总纪录数没有任何关系。 Bitmap 索引也不适合并发修改数据事务特别多的 OLTP 类型应用。使用 bitmap 索引应该谨慎，仔细对比建立索引前后的查询性能。只添加那些对查询性能有帮助的索引。

创建索引

CREATE INDEX 命令可以给指定的表定义索引。索引的默认类型是：B 树索引。下面例子给表 employee 的 gender 列，添加了一个 B 树索引：

```
CREATE INDEX gender_idx ON employee (gender);
```

为 films 表的 title 列创建 bitmap 索引：

```
CREATE INDEX title_bmp_idx ON films USING bitmap (title);
```

检查索引的使用情况

HashData 数据仓库索引不需要维护和调优。你可以通过真实的查询来检查索引的使用情况。EXPLAIN 命令可以用来检查一个查询使用索引的情况。查询计划用来显示数据库为了回答您的查询所需要的步骤和使用的计划节点类型，并给出每个节点的时间开销评估。要检查索引的使用情况，可以通过检查 EXPLAIN 中包含的查询计划节点来进行输出中下面查询：

- Index Scan - 扫描索引
- Bitmap Heap Scan - 根据 BitmapAnd， BitmapOr，或 BitmapIndexScan 生成位图，从 heap 文件中读取相应的记录。
- Bitmap Index Scan - 通过底层的索引，生成满足多个查询的条件的位图信息。
- BitmapAnd 或 BitmapOr - 根据多个 BitmapIndexScan 生成的位图进行位与和位或运算，生成新的位图。

创建索引前，您需要做一些实验来决定如何创建索引，下面是一些您需要考虑的地方：

- 当你创建或更新索引后，最好运行 ANALYZE 命令。ANALYZE 针对表收集统计信息。查询优化器会利用表的统计信息来评估查询返回的结果数量，并且对每种查询计划估算更真实的时间开销。
- 使用真实数据来进行实验。如果利用测试数据来决定添加索引，那么你的索引只是针对测试数据进行了优化。
- 不要使用可能导致结果不真实或者数据倾斜的小数据集进行测试。
- 设计测试数据时需要非常小心。测试数据如果过于相似，完全随机，按特定顺序导入，都可能导致统计数据与真实数据分布的巨大差异。
- 你可以通过调整运行时参数来禁用某些特定查询类型，这样可以更加针对性对索引使用进行测试。例如：关闭顺序扫描（enable_seqscan）和嵌套连接（enable_nestloop），及其它基础查询计划，可以强制系统选择其它类型的查询计划。通过对查询计时和利用 EXPLAIN ANALYZE 命令来对比使用和不使用索引的查询结果。

索引管理

使用 REINDEX 命令可以对性能不好的索引进行重新创建。REINDEX 重建是对表中数据重建并替换旧索引实现的。

在指定表上重新生成所有索引：

```
REINDEX my_table;
```

对指定索引重新生成：

```
REINDEX my_index;
```

删除索引

DROP INDEX 命令删除一个索引，例如：

```
DROP INDEX title_idx;
```

加载数据时，可以通过首先删除索引，加载数据，再重新建立索引的方式加快数据加载速度。

创建和管理视图

视图能够将您常用或复杂的查询保存起来，并允许您在 SELECT 语句中像访问表一样访问保存的查询。视图并不会导致在磁盘上存储数据，而是在访问视图时，视图定义的查询以自查询的方式被饮用。

如果某个自查询只被某个特定查询使用，考虑使用 SELECT 语句的 WITH 子句来避免创建一张不能被公用的视图。

创建视图

CREATE VIEW 命令根据一个查询定义一个视图，例如：

```
CREATE VIEW comedies AS SELECT * FROM films WHERE kind = 'comedy';
```

视图会忽略视图定义查询中的 ORDER BY 和 SORT 的功能。

删除视图

DROP VIEW 删除一张视图，例如:

```
DROP VIEW topten;
```

数据的管理

本章节想您提供关于在 HashData 数据仓库中操作数据和并发访问的相关信息。本章节包含下面一些子话题：

- HashData 数据仓库中的并发控制
- 插入数据
- 更新存在的数据
- 删除数据
- 使用事务
- 数据库的清理

HashData 数据仓库中的并发控制

在 HashData 数据仓库和 PostgreSQL 中，并发控制并不是通过锁实现的。而是通过使用多个数据版本的多版本并发控制（MVCC）来维护数据一致性的。 MVCC 能够对数据库的每个会话提供事务隔离，并且保证每一个查询事务都能看到一份数据的快照。这种方式保证了事务看到一致的数据，而不会被其他并发运行的事务影响到。

由于 MVCC 不通过显示封锁来控制并发访问，这就使得锁冲突最小化，因此 HashData 数据仓库能够在 多用户环境下依然提供稳定可靠的处理能力。查询数据（读取）使用的锁，不会与写入数据使用的锁产生（冲突）。

HashData 数据仓库提供了多个锁模式来对表中的数据提供并发访问控制。

大多数 HashData 数据仓库的 SQL 命令能够自动地根据执行的命令，在操作的表上获取适当的锁，来 防止删除表或修改表的操作。对于不能简单地修改来兼容 MVCC 行为的应用程序，可以通过使用 LOCK 命令来显示获取指定类型的锁。但是，合理使用 MVCC 能够有效提升性能。

锁模式	相关 **SQL** 命令	冲突
ACCESS SHARE	SELECT	ACCESS EXCLUSIVE
ROW SHARE	SELECT FOR SHARE	EXCLUSIVE, ACCESS EXCLUSIVE
ROW EXCLUSIVE	INSERT, COPY	SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE UPDATE EXCLUSIVE	VACUUM (without FULL), ANALYZE	SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE	CREATE INDEX	ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE ROW EXCLUSIVE		ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
EXCLUSIVE	DELETE, UPDATE, SELECT FOR UPDATE, See Note	ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
ACCESS EXCLUSIVE	ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL	ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE

> 注意: 在 HashData 数据仓库中, UPDATE, DELETE, 和 SELECT FOR UPDATE 命令将会使用更加严格的 EXCLUSIVE 锁，而不是 ROW EXCLUSIVE。

插入数据

使用 INSERT 命令可以在表中创建数据。使用此命令需要提供表名和该表中每列的值，你可以通过显示地指定列名来改变提供列值的顺序。如果您没有指定列名，列值需要按照表中的列顺序指定，并使用逗号进行分隔。

下面是指定按照列名顺序提供列值的示例：

```
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

下面是指提供列值的示例：

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

通常情况下，提供的列值是字面值（常量），但是您也可以使用标量表达式。例如：

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

您还可以在一条命令中插入多行值：

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

您可以通过指定分区表的根表，来向分区表中插入数据。您也可以通过在 INSERT 命令中指定分区表的子表（叶子节点）。如果您提供的数据与该子表存储的数据不匹配，系统将提示错误信息。INSERT 命令不支持指定非叶子节点的子表。

要插入大量数据，请使用外部表命令。外部表数据加载机制在处理大量数据插入方面比 INSERT 命令更加高效。请参考 TODO 了解更多关于海量数据加载问题。

Append 表（追加表）是一种专门为海量数据加载设计的存储模型。HashData 数据仓库不推荐使用单行的 INSERT 命令来操作追加表。目前 HashData 数据仓库允许最多 127 并发事务同时向一张追加表进行插入。

更新存在的数据

UPDATE 命令可以更新一张表中的记录。该命令支持一次更新所有记录，部分记录或者单条记录。该命令还允许针对特定列进行更新，而不影响其他列值。

要执行更新操作，需要如下信息：

- 要更新的表名和相关列名
- 新的列值
- 用来指定需要更新记录的一个或多个选择条件

下面示例展示将所有价格为 5 的产品价格改为 10 的命令：

```
UPDATE products SET price = 10 WHERE price = 5;
```

在 HashData 数据仓库中使用 Update 命令有如下限制：

- HashData 数据仓库数据分布键值不能被更新。
- HashData 数据仓库不支持 RETURNING 子句。
- HashData 数据仓库分区列不能被更新（分区键值）。

删除数据

DELETE 命令可以从表中删除记录。使用 WHERE 子句可以删除符合特定条件的记录。如果没有指定 WHERE 条件，表中所有记录都会被删除。执行的结果就是一张没有记录的空表。

下面示例展示删除所有价格为 10 的产品：

```
DELETE FROM products WHERE price = 10;
```

删除表中所有数据：

```
DELETE FROM products;
```

在 HashData 数据仓库中使用 DELETE 命令有如下限制：

HashData 数据仓库不支持 RETURNING 子句。

清空表

TRUNCATE 命令可以快速地删除表中所有记录。

例如：

```
TRUNCATE mytable;
```

此命令可以将表中所有记录一次清空。由于 TRUNCATE 不对表进行扫描，因此该操作不操作继承表或者 ON DELETE 重写规则。命令只会将指定表中的纪录清空。

使用事务

事务允许您将多个 SQL 语句当做一个原子的操作（要么都做，要么都不做）。下面列出 SQL 的事务命令：

- BEGIN 或 START TRANSACTION 开始一个事务代码块。
- END 或 COMMIT 将会提交事务运行结果。
- ROLLBACK 放弃当前事务对数据库的所有修改。
- SAVEPOINT 保存点可以在当前事务中间保存特定信息来允许事务回滚到指定保存点。通过使用保存点，您可以回滚该保存点之后的所有命令，并且保留该保存点之前执行命令的结果。
- ROLLBACK TO SAVEPOINT 将事务状态恢复到指定保存点。
- RELEASE SAVEPOINT 将事务内的保存点占用资源释放。

事务的隔离级别

HashData 数据仓库支持标准 SQL 事务级别如下：

- 读未提交（read uncommitted）和读提交（read committed）行为与 SQL 标准定义的读已提交一致。
- 可重复读（repeatable read） 不被支持。如果需要使用可重复读行为，可以通过使用串行化级别来兼容。
- 可串行化（serializable）行为与 SQL 标准定义的可串行化一致。

下面向您介绍 HashData 数据仓库各个事务级别的行为介绍：

读提交/读未提交 — 提供快速，简单和部分事务隔离。在读提交和读未提交事务隔离级别下，SELECT, UPDATE, 和 DELETE 操作的数据库快照，是在查询启动时构建的。

SELECT 查询：

- 可以看到查询启动前所有已经提交的数据变动。
- 可以看到事务内部已经执行的操作变动。
- 不可以看到事务外部未提交的数据变动。
- 可能在本事务读取数据后，看到其他并发事务提交的变动。
- 在同一个事务中的后续 SELECT 查询可能会看到其他并发事务在该 SELECT 查询启动前提交的变动。 UPDATE 和 DELETE 命令只操作在命令启动前已经提交的变动。

读提交或读未提交事务隔离级别在进行 UPDATE 或 DELETE 操作时，允许并发事务对记录进行修改或封锁。读提交或读未

提交事务隔离级别可能对于执行复杂查询更新，对数据库系统要求一致性视图的应用程序来说不能胜任。

可串行化 — 提供严格地事务隔离，事务的执行结果类似于事务一个一个地顺序执行，而不是并发执行。 使用可串行化隔离级别的应用程序在设计时，需要考虑串行化失败的情况，进行必要的重试操作。在 HashData 数据仓库中，SERIALIZABLE 隔离级别能够在不使用代价较高的封锁机制下，防止读脏数据， 不可重复读，和读幻象。但是还存在一些其他 SERIALIZABLE 事务之间的交互，使得 HashData 数据仓库不能 提供真正的可串行化结果。并发执行的事务需要进行检查，来识别事务交互时因为紧致并发更新相同的数据导致相互影响。识别的问题可以通过显示地表锁或者事务冲突来避免，例如：通过并发更新一个标志行来引入事务冲突。

SELECT 查询：

- 可以看到事务启动时数据的快照（不是事务中当前查询启动时的快照）
- 只能看到查询启动前已经提交的数据变动。
- 可以看到事务内部已经执行的操作变动。
- 不可以看到事务外部未提交的数据变动。
- 不可以看到并发执行的事务产生的数据变动。
- 在一个事务中的后续 SELECT 命令看到的数据总是一样的。UPDATE，DELETE，SELECT FOR UPDATE， 和 SELECT FOR SHARE 命令只操作在命令启动前本事务已经提交的变动。如果其他并发执行的事务已经更新，删除，或锁住一个要被操作的目标记录，那么可串行化或可重复读的事务将会等待该并发事务完成，再更新该记录，删除该记录，或者回滚事务。

如果并发事务更新或删除记录，那么可串行化或可重复读的事务将会回滚。如果并发事务回滚，那么可串行 化或可重复读的事务将会更新或删除该记录。

HashData 数据仓库中的默认隔离级别是读提交。要改变事务的隔离级别，可以在使用 BEGIN 启动事务时声明，或者在事务启动后，使用 SET TRANSACTION 命令设置。

数据库的清理

虽然被删除或者被更新的记录对于新事务是不可见的，但是它们仍然会占用磁盘的物理空间。需要定期运行 VACUUM 命令来清理这些过期的记录。例如：

```
VACUUM mytable;
```

VACUUM 命令将会收集表相关的统计数据，例如：记录数和页面数。在数据加载后，请使用 Vacuum 对所有的数据表进行处理（包括追加表）。

> 重要: 如果数据更新或删除比较频繁，请使用 VACUUM, VACUUM FULL, 和 VACUUM ANALYZE 命令来维护数据信息。

导入导出数据

本节的主题是描述如何高效地往 HashData 数据仓库导入数据和从 HashData 数据仓库往外导出数据，以及如何格式化导入的文件格式。

HashData 数据仓库支持高效地并发导入和导出数据功能，利用青云的对象存储，可以与不同系统更加简单地进行数据交换操作。

通过使用外部表，HashData 数据仓库让您能够通过 SQL 命令在数据库内直接利用并行机制访问外部数据资源，例如：SELECT，JOIN，ORDER BY 或者在外部表上创建视图。 外部表一般用来将外部数据导入到数据库内部普通表，例如：

```
CREATE TABLE table AS SELECT * FROM ext_table;
```

通过外部表访问对象存储

CREATE EXTERNAL TABLE 可以创建可读外部表。外部表允许 HashData 数据仓库将外部数据源当作数据库普通表来进行处理。

使用青云对象存储

通过在外部表的 LOCATION 子句，您可以指定青云对象存储的位置和访问键值。这样，在每次使用 SQL 命令读取外部表时，HashData 数据仓库会将青云对象存储中该目录下的所有数据文件读取到数据库中进行处理。 如果您需要多次处理该数据，建议您通过 `CREATE TABE table AS SELECT * FROM ext_table` 的方式，将对象存储的数据导入到 HashData 数据仓库的内置表。这样可以更好的利用优化过的存储结构来优化您的数据分析流程。

您可以通过如下的语法来定义访问青云对象存储上面数据的外部表：

```
CREATE READABLE EXTERNAL TABLE foo (name TEXT, id INT)
LOCATION ('qs://<your-bucket-name>.pek3a.qingstor.com/<your-data-path> access_key_id=<access-key-id> secret_access_key=<secret
-access-key>') FORMAT 'csv';
```

您需要将其中的 `<your-bucket-name>` 、 `<your-data-path>` 、 `<access-key-id>` 和 `<secret-access-key>` 替换成您自己相应的值。

下面的示例，向您展示如何从青云对象存储中直接读取 1GB 规模 TPCH 测试数据的例子。由于示例中的 bucket hashdata-public 是公开只读的，因此您不指定 `access_key_id` 和 `secret_access_key` 也能够访问。

```
CREATE READABLE EXTERNAL TABLE e_NATION (LIKE NATION)
LOCATION ('qs://hashdata-public.pek3a.qingstor.com/tpch/1g/nation/') FORMAT 'csv';

CREATE READABLE EXTERNAL TABLE e_REGION (LIKE REGION)
LOCATION ('qs://hashdata-public.pek3a.qingstor.com/tpch/1g/region/') FORMAT 'csv';

CREATE READABLE EXTERNAL TABLE e_PART (LIKE PART)
LOCATION ('qs://hashdata-public.pek3a.qingstor.com/tpch/1g/part/') FORMAT 'csv';

CREATE READABLE EXTERNAL TABLE e_SUPPLIER (LIKE SUPPLIER)
LOCATION ('qs://hashdata-public.pek3a.qingstor.com/tpch/1g/supplier/') FORMAT 'csv';

CREATE READABLE EXTERNAL TABLE e_PARTSUPP (LIKE PARTSUPP)
LOCATION ('qs://hashdata-public.pek3a.qingstor.com/tpch/1g/partsupp/') FORMAT 'csv';

CREATE READABLE EXTERNAL TABLE e_CUSTOMER (LIKE CUSTOMER)
LOCATION ('qs://hashdata-public.pek3a.qingstor.com/tpch/1g/customer/') FORMAT 'csv';

CREATE READABLE EXTERNAL TABLE e_ORDERS (LIKE ORDERS)
LOCATION ('qs://hashdata-public.pek3a.qingstor.com/tpch/1g/orders/') FORMAT 'csv';

CREATE READABLE EXTERNAL TABLE e_LINEITEM (LIKE LINEITEM)
LOCATION ('qs://hashdata-public.pek3a.qingstor.com/tpch/1g/lineitem/') FORMAT 'csv';
```

处理错误

可读外部表最常用于将数据导入到数据库内置表中。您可以通过使用 CREATE TABLE AS SELECT 或 INSERT INTO 命令从外部表读取数据。默认情况下，如果数据中包含格式错误的行，整个命令都会失败，数据不会被成功地加载到目标的数据表中。

SEGMENT REJECT LIMIT 子句允许您将外部表中格式错误的数据进行隔离，继续导入格式正确的数据。使用 SEGMENT REJECT LIMIT 命令设定一个阈值，用 ROWS 指定拒绝的错误记录行数（默认），或者 PERCENT 指定拒绝的错误记录百分比（1-100）。

如果错误的行数到达 SEGMENT REJECT LIMIT 指定的值，整个外部表操作将会终止。错误行数的限制是根据每个加载节点单独计算的。加载操作将会正常处理格式正确的记录，如果错误的记录数没有超过 SEGMENT REJECT LIMIT，跳过格式错误的记录，并将其保存在错误记录中。

LOG ERRORS 子句允许您保留错误记录信息，在命令执行后进一步排查问题。要了解 LOG ERRORS 子句的使用方法，请参考 CREATE EXTERNAL TABLE 命令。

当你使用了 SEGMENT REJECT LIMIT 子句，HashData 数据仓库将会在单行错误隔离模式下扫描外部数据。单行错误隔离模式将会为外部数据记录附加额外的格式错误信息，例如：不一致的列值，错误的列数据类型，非法的客户端编码序列。HashData 数据仓库不会进行约束错误检查，但是您可以通过在使用 SELECT 命令处理外部表时，增加必要的过滤限制条件。例如，下面的例子可以消除重复键值的错误：

```
=# INSERT INTO table_with_pkeys
   SELECT DISTINCT * FROM external_table;
```

> 注意：当使用外部表加载数据时，服务器配置参数 `gp_initial_bad_row_limit` 用来限制在最开始处理时，最多多少行不能遇到错误记录。默认设置为如果在前 1000 行中遇到错误记录，就会停止后续处理。

定义使用单行错误隔离的外部表

下面的例子将会将错误记录保存在 HashData 数据仓库内部，并且设置错误阈值为 10 条记录：

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('qs://hashdata-public.pek3a.qingstor.com/ext_expenses/')
   FORMAT 'TEXT' (DELIMITER '|')
   LOG ERRORS SEGMENT REJECT LIMIT 10
   ROWS;
```

通过使用内置 SQL 函数 `gp_read_error_log('external_table')` 可以读取错误记录数据。下面的示例命令可以显示 `ext_expenses` 的错误记录：

```
SELECT gp_read_error_log('ext_expenses');
```

要了解更多关于错误记录信息，请参考在错误日志中查看错误记录。

内置 SQL 函数 `gp_truncate_error_log('external_table')` 可以删除错误记录。下面的例子用来删除之前访问外部表时记录的错误数据：

```
SELECT gp_truncate_error_log('ext_expenses');
```

捕获记录格式错误和声明拒绝错误记录限制

下面的 SQL 语句片段用来在 HashData 数据仓库中捕获格式错误，并声明最多拒绝 10 条错误记录的限制。

```
LOG ERRORS SEGMENT REJECT LIMIT 10 ROWS
```

在错误日志中查看错误记录

如果您使用单行错误隔离模式（参考 定义使用单行错误隔离的外部表），格式错误的记录信息将会被记录在数据库内部。

通过使用内置 SQL 函数 `gp_read_error_log('external_table')` 可以读取错误记录数据。下面的示例命令可以显示 `ext_expenses` 的错误记录：

```
SELECT gp_read_error_log('ext_expenses');
```

优化数据加载和查询性能

下面的小建议可以帮助您优化数据加载和加载后的查询性能。

如果您向已经存在的数据表加载数据，可以考虑先删除索引（如果该表已经建了索引的话）。在数据上重新创建索引的速度远远快于在已有索引上逐行插入的性能。您还可以临时地增加 maintenance_work_mem 服务器参数来优化 CREATE INDEX 命令的创建参数（该参数可能影响数据库加载性能）。建议您在系统没有活跃操作时删除和重新创建索引。

如果您向新的数据表加载数据，请在数据加载完毕后再创建索引。推荐的步骤是：创建表，加载数据，创建必要的索引。

数据加载完毕后，运行 ANALYZE 命令。如果您操作影响的数据非常多，建议运行 ANALYZE 或者 VACUUM ANALYZE 命令来更新系统的统计信息，这样优化器可以利用最新的信息，更好的优化查询。最新的统计信息能够对查询进行更加精确的优化，从而避免由于统计信息不准确或者不可用时，导致查询性能非常差。

在数据加载出错后，运行 VACUUM 命令。如果数据加载操作没有使用错误记录隔离模式，加载操作将在遇到的第一个错误处停止。这时已经加载的数据虽然不会被访问到，但是他们已经占用了磁盘上的存储空间。请运行 VACUUM 命令来回收这些浪费的空间。

合理使用表分区技术可以简化数据的维护工作。

从 HashData 数据仓库导出数据

通过使用可写外部表，您可以将 HashData 数据仓库中的数据表导出成外部通用文件格式。通过可写外部数据表，您可以简单的实现多系统互联。未来，HashData 数据仓库还会支持更多导出功能，让您能够轻松的利用不同数据处理平台，来优化您的数据处理体验。

本章节向您介绍如何利用可写外部表将数据库内部数据导出到外部存储。

使用青云对象存储

HashData 数据仓库充分考虑云平台优势，因此提供利用高效的青云对象存储来作为数据导出目的地。HashData 数据仓库将会利用其自身并行的架构将数据并行写入到青云对象存储。

下面的例子向您介绍如何利用青云对象存储作为可写外部表的输出目标。

```
CREATE WRITABLE EXTERNAL TABLE test_writable_table (id INT, date DATE, desc TEXT)
    location('qs://<your-bucket-name>.pek3a.qingstor.com/<your-data-path> access_key_id=<access-key-id> secret_access_key=<sec
ret-access-key>') FORMAT 'csv';

INSERT INTO test_writable_table VALUES(1, '2016-01-01', 'qingstor test');
```

在实际使用的时候，您需要将 `<your-bucket-name>`、`<your-data-path>`、`<access-key-id>` 和 `<secret-access-key>` 换成您自己相应的值。

格式化数据文件

在使用青云对象存储来进行数据的导入导出时，需要您指定数据的格式信息。CREATE EXTERNAL TABLE 允许您描述数据的存储格式。数据可以是使用特定分隔符的文本文件或者逗号分隔值的 CSV 格式。只有格式正确的数据才能被 HashData

数据仓库正确读取。本小结向您介绍 HashData 数据仓库期望的数据文件格式。

数据行的格式

HashData 数据仓库期望数据行使用 LF 字符（Line feed，0x0A），CR（Carriage return，0x0D），或者 CR 和 LF（CR+LF，0x0D 0x0A）。LF 是 UNIX 或类 UNIX 操作系统上标准的换行符。像 Windows 或者 Mac OS X 使用 CR 或 CR+LF。HashData 数据仓库能够支持前面提到的三种换行的表示形式。

数据列的格式

对于文本文件和 CSV 文件来说，默认的列分隔符分别是 TAB 字符（0x09）和逗号（0x2C）。您也可以通过 CREATE EXTERNAL TABLE 语句的 DELIMITER 子句为数据文件指定一个字符作为分隔符。分隔符字符需要在两个数据字段之间使用。每行的开头和结尾不需要指定分隔符。下面的示例是一个使用（|）字符的示例输入：

```
data value 1|data value 2|data value 3
```

下面的示例语句介绍如何在语句中指定（|）作为分隔符：

```
=# CREATE EXTERNAL TABLE ext_table (name text, date date)
LOCATION ('qs://<your-bucket-name>.pek3a.qingstor.com/filename.txt')
FORMAT 'TEXT' (DELIMITER '|');
```

在数据中表示空值

数据库中的空值（NULL）表示列值未知。在您提供的数据文件中，可以指定一个字符串来表示空值。对于文本文件来说，默认的字符串是 N，对于 CSV 文件来说，使用没有双引号保护的空值。您还可以在 CREATE EXTERNAL TABLE 的 NULL 子句中指定其他的字符串来表示空值。例如，如果您不需要区分空值和空字符串的话，可以使用空字符串来表示空值。在数据加载时，任何数据值和指定的空值字符串相同时，就会被解释为空值。

转义

在 HashData 数据仓库中有两个保留字符具有特殊含义：

- 被用来在数据文件中作为列的分隔符的字符。
- 被用来在数据文件中作为换行的字符。

如果您提供的数据中包含上面的字符，需要将该字符进行转义，这样 HashData 数据仓库就会将其解释为数据，而不会解释为列分隔符或换行。默认情况下，文本文件的转义字符是 （反斜线），CSV 文件是 “（双引号）。

文本文件中的转义操作

默认情况下，文本格式文件的转义字符是 （反斜线）。您可以在 CREATE EXTERNAL TABLE 的 ESCAPE 语句中指定其他的转义字符。如果您的数据中包含了转义字符本身，可以使用转义字符来转义它自己。

让我们来看一下例子，假设您有一张包含三个列的表，您希望将下面三列作为列值加载到表中：

```
backslash =
vertical bar = |
exclamation point = !
```

您指定 | 作为列分隔符，作为转义字符。您数据文件中，格式化后的数据行应该类似下面的样子：

```
backslash = \\ | vertical bar = \| | exclamation point = !
```

这里请您注意反斜线是一部分数据，因此需要使用另一个反斜线进行转义。另外 | 字符也是数据的一部分，因此需要使用反斜线将其转义。

您还可以使用转义字符来转义八进制和十六进制序列。在 HashData 数据仓库加载数据时，转义的值将会被转换为对应的字符。例如，要加载 & 字符，可以用转义字符来转义该字符的十六进制（0x26）或者八进制（046）表示。

您也可以在文件格式文件中，利用 ESCAPE 语句来禁用转义功能，例如：

```
ESCAPE 'OFF'
```

如果您要加载包含大量反斜线的数据时（例如，互联网类日志），禁用转义功能将会非常方便。

CSV 文件中的转义操作

默认情况下，CSV 格式文件的转义字符是 “（双引号）。您可以在 CREATE EXTERNAL TABLE 的 ESCAPE 语句中指定其他的转义字符。如果您的数据中包含了转义字符本身，可以使用转义字符来转义它自己。

让我们来看一下例子，假设您有一张包含三个列的表，您希望将下面三列作为列值加载到表中：

```
Free trip to A,B
5.89
Special rate “1.79”
```

您指定，（逗号）作为列分隔符，”（双引号） 作为转义字符。您数据文件中，格式化后的数据行应该类似下面的样子：

```
"Free trip to A,B","5.89","Special rate ""1.79"""
```

如果逗号是数据的一部分，需要使用双引号来保护。如果双引号是数据的一部分，即使整个数据是用双引号来保护的，还是需要用双引号来转义双引号的。

使用双引号保护整个数据字段可以保证数据开头和结尾的空白字符被正确解释。

```
"Free trip to A,B ","5.89 ","Special rate ""1.79"" "
```

> 注意：在 CSV 格式中，字符解释的处理非常严格。例如：被引号保护的空白字符或者其他非分隔字符。由于上面的原因，如果向系统导入的数据在末尾补充了保持每行相同长度的空白字符，将会引起错误。因此，在使用 HashData 数据仓库读取数据时，需要先将 CSV 文件进行预处理，移除不必要的末尾空白字符。

字符编码的处理

字符编码系统是通过一种编码方法，将字符集中的字符和特定数值组成一个固定映射关系，来允许对数据传输和存储。HashData 数据仓库支持多种不同的字符集，例如：单字节的字符集 ISO 8859 及其衍生字符集，多字节字符集 EUC（扩展 UNIX 编码），UTF-8 等。虽然有一小部分字符集不支持作为服务器端存储编码，但是客户端可以使用所有的字符集。

数据文件使用的编码必须是 HashData 数据仓库能够识别的。 在数据加载时，如果数据文件包含无效或不支持的编码序列时，将会导致错误。

> 注意：如果数据文件是在微软的 Windows 操作系统上生成的，请在进行数据加载前，运行 dos2unix 命令来删除原始文件中 Windows 平台的特殊字符。

查询数据

本章节向您介绍使用 SQL 的相关信息。

您可以通过交互式 SQL 客户端（例如：psql）或者其他客户端工具向指定数据库输入 SQL 语言，来查阅，修改和进行数据分析。

- 数据处理简介
- 查询的定义
- 使用函数和运算符

数据处理简介

本主题为您介绍 HashData 数据仓库是如何处理查询请求的。理解查询处理的过程，对您编写和优化查询有非常巨大的帮助。

用户向 HashData 数据仓库发送查询命令和使用其它数据库管理系统完全一样。 通过使用客户端应用程序（例如：psql）连接 HashData 数据仓库主节点，您可以提供 SQL 语句命令。

理解查询优化和查询分发

主节点负责接收，分析和优化用户查询。最终的执行计划可以是完全并行的，也可以是运行在特定节点的。 如 图1 对于并行查询计划，主节点将其发送到所有的计算节点上。如 图2 所示， 对于运行在特定节点的执行计划，主节点将会发送查询计划到一个单独的节点运行。 每个计算节点只负责在自己对应的数据上进行相应的数据操作。

大多数的数据库操作是在所有计算节点并行进行的，例如：扫描数据表，连接运算，聚合运算和排序操作。 每个计算节点的操作都不依赖存储在其它计算节点上的数据。

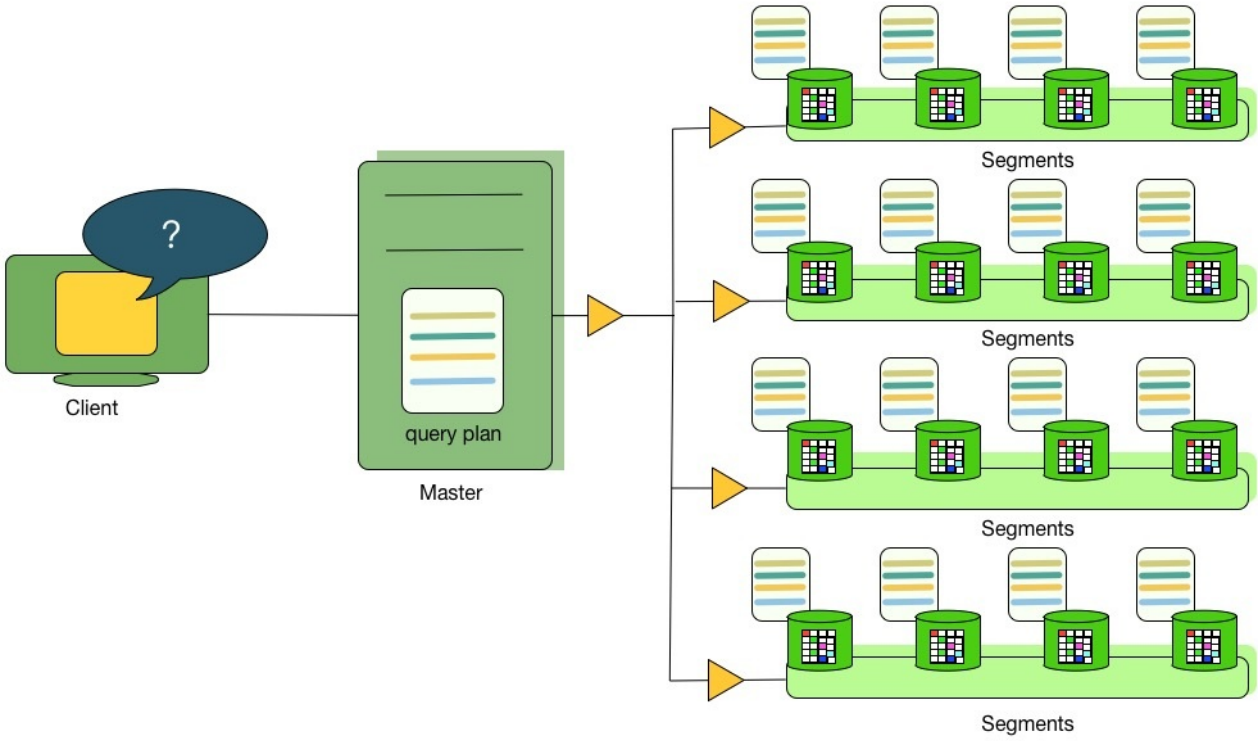


图1：分发并行查询计划

一些查询可能只访问特定计算节点的数据，例如：单行插入，更新，删除或者是查询操作只涉及表中特定数据（过滤条件正好是表的数据分布键值）。对于上述的查询，查询计划不会发送给所有的计算节点， 而是将查询计划发送给该查询影响的节

点。

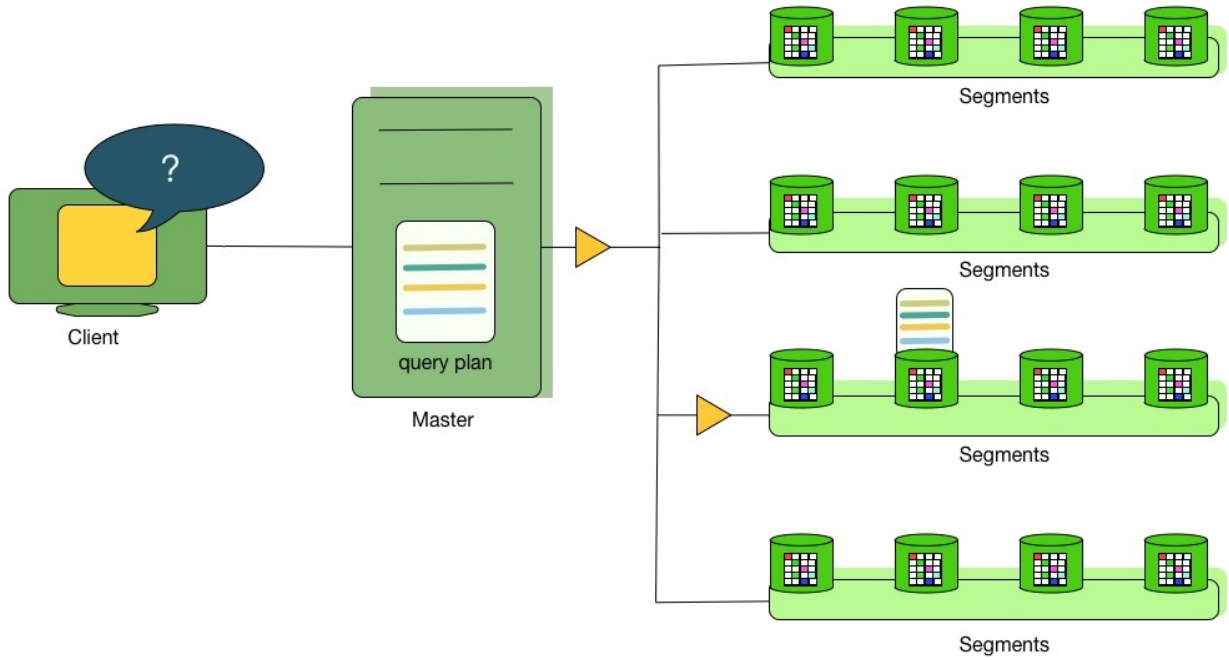


图**2**：分发特定节点查询计划

理解查询计划

查询计划就是 HashData 数据仓库为了计算查询结果的一系列操作的步骤。查询计划中的每个步骤（节点）代表了一种数据库操作，例如：表扫描，连接运算，聚合运算或者排序操作。查询计划的读取和执行都是自底向上的。

除了常见的操作外，HashData 数据仓库还支持一些特殊的操作：motion 节点（移动）。移动节点就是查询处理过程中，在不同计算节点直接移动数据。需要注意的是，不是所有的查询都需要数据移动的。例如：运行在特定节点的查询是不需要任何数据移动的。

为了能让查询执行获得最大的并行粒度，HashData 数据仓库通过将查询计划进行切片来进一步分解任务。每个切片都是可以被一个计算节点独立执行的查询计划子集。当查询计划中包含了数据移动节点时，查询计划就是被分片的。数据移动节点的上下两部分各自是一个独立的分片。

让我们来看下面这个例子：这是一个简单的两张表连接的运算：

```
SELECT customer, amount
FROM sales JOIN customer USING (cust_id)
WHERE dateCol = '04-30-2008';
```

图3 展示了上面查询的查询计划。每个计算节点都会收到一份查询计划的拷贝，并且并行进行处理。

这个示例的查询计划包含了一个数据重分布的数据移动节点，该节点用来在不同计算节点直接移动记录使得连接运算得意完成。这里之所有需要数据重分布的节点，是因为 customer 表（用户表）的数据分布是通过 cust_id 来进行的，而 sales 表（销售表）是根据 sale_id 进行的。为了进行连接操作，sales 表的数据需要重新根据 cust_id 来分布。因此查询计划在数据重分布节点两侧被切片，分别是 slice 1 和 slice 2。

这个查询计划还包括了另一种数据移动节点：数据聚合节点。数据聚合节点是为了让所有的计算节点将结果发送给主节点，最后从主节点发送给用户引入的。由于查询计划总是在数据移动节点出现时被切片，这个查询计划还包括了一个隐藏的切片，该切片位于查询的最顶层（slice3）。并不是所有的查询都包含数据聚合移动节点，例如：

```
CREATE TABLE x AS
SELECT ...
```

语句不需要使用数据聚合移动节点，这是因为数据将会移动到新创建的数据表中，而非主节点。

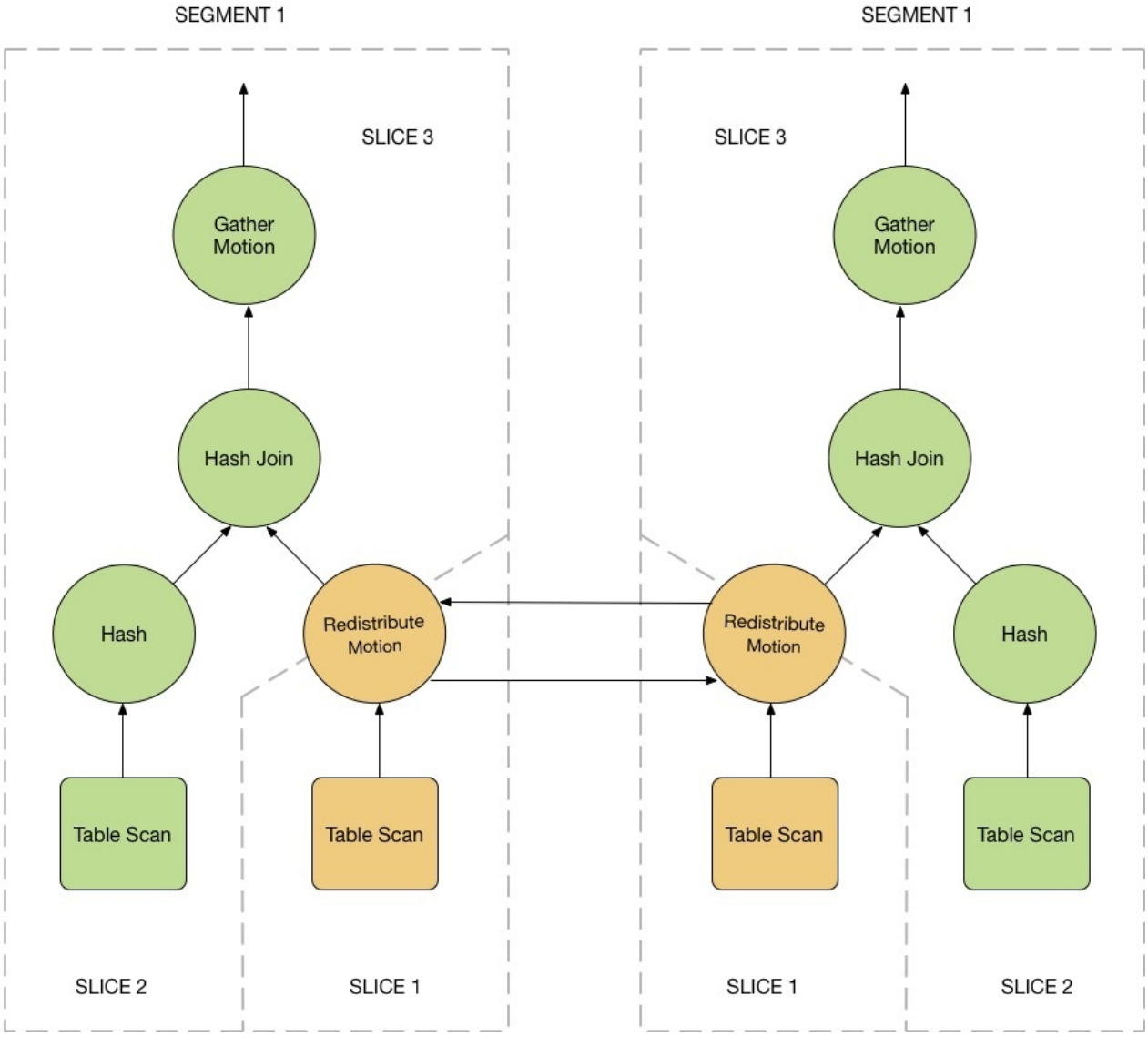


图**3**：查询计划切片

理解并行查询计划的执行

HashData 数据仓库将会创建多个数据库进程来处理查询的相关工作。在主节点上，查询工作进程被称为查询分派器（QD）。QD 负责创建和分派查询计划。它同时负责收集和展示最终查询结果。 在计算节点上，查询工作进程被称作查询执行器（QE）。QE 负责执行分配给该进程的查询计划并通过通信模块将中间结果发送给其它工作进程。

每个查询计划的切片都至少会有一个工作进程与之对应负责执行。工作进程会被赋予不会互相依赖的查询计划片段。查询执行的过程中，每个计算节点都会有多个进程并行地参与查询的处理工作。

在不同计算节点上执行相同切片查询计划的工作进程被称为进程组。随着一部分工作的完成，数据记录将会从一个进程组流向其它进程组。这种在数据节点之间的进程间通信被称为互联组件。

图4 向您展示对于 图3 中查询计划在主节点和两个计算节点上的进程分布情况。

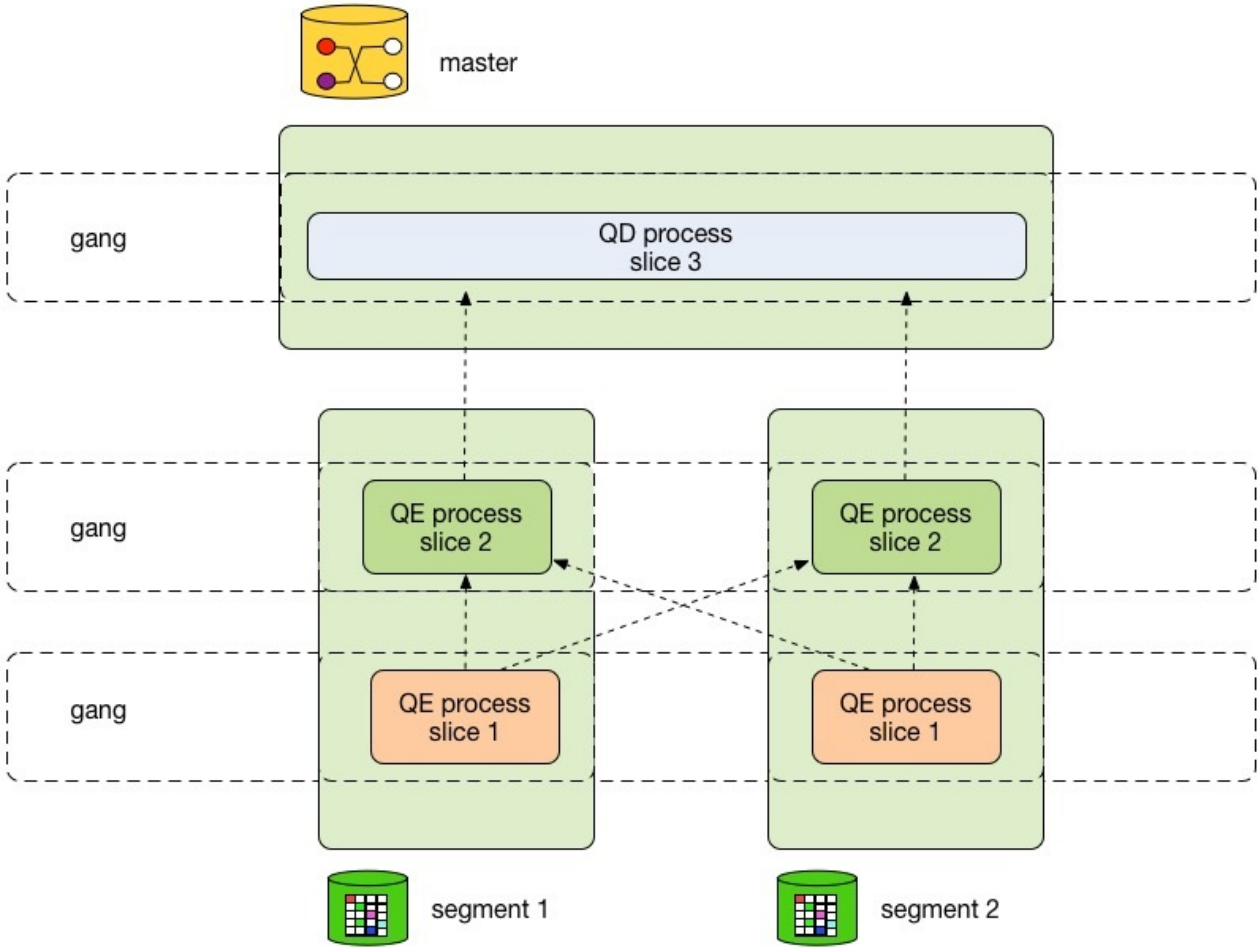


图**4**：查询执行器处理请求

查询的定义

HashData 数据仓库查询命令是基于 PostgreSQL 开发，而 PostgreSQL 是实现了 SQL 标准。

本小结介绍如何在 HashData 数据仓库中编写 SQL 查询。

- SQL 词法元素
- SQL 值表达式

SQL 词法元素

SQL 是一种标准化的数据库访问语言。不同元素组构成的语言允许控制数据存储，数据获取，数据分析， 数据变换和数据修改等。你需要通过使用 SQL 命令来编写 HashData 数据仓库理解的查询和命令。SQL 查询由一条或多条命令顺序组成。每一条命令是由多个词法元素组成正确的语法结构构成的， 每条命令使用分号（;）分隔。

HashData 数据仓库在 PostgreSQL 的语法结构上进行了一些扩展，并根据分布式环境增加了部分的限制。如果您希望了解更多关于 PostgreSQL 中的 SQL 语法规则和概念，您可以参考 PostgreSQL8.2英文手册中SQL语法章节 或者 PostgreSQL9.3中文手册中 SQL 语法章节。由于中文网站没有 8.2 手册，请您注意相关资料中的语法变动。

SQL 值表达式

SQL 值表达式由一个或多个值，符号，运算符，SQL 函数和数据组成。表达式通过比较数据，执行计算病返回一个结果。表达式计算包括：逻辑运算，算数运算和集合运算。

下面列出值表达式的类别：

- 聚合表达式
- 数组构造表达式
- 列引用
- 常亮或字面值
- 相关自查询
- 成员选择表达式
- 函数调用
- INSERT 或 UPDATE 语句中，为列提供的值
- 运算符调用
- 列引用
- 在函数体内或 Prepared 语句中引用位置参数
- 记录构造表达式
- 标量子查询
- WHERE 子句中的搜索条件
- SELECT 语句中的返回列表
- 类型转换
- 括号保护的子表达式
- 窗口表达式

像函数和运算符这样的 SQL 结构虽然属于表达式，但是与普通的语法规则不相同。请参考使用函数和运算符了解更多信息。

列引用

列引用的格式如下：

```
correlation.columnname
```

上面的示例中，correlation 是表的名称（也可以使用限定名格式：在表名前面添加模式名）或者定义在 FROM 子句中的表的别名。如果列名在查询访问的表中是唯一的，那么 “correlation.” 部分是可以被省略的。

位置参数

位置参数是指通过指定传递给 SQL 语句或函数参数的位置信息来引用的参数。例如：`$1` 引用第一个参数，`$2` 引用第二个参数，依此类推。位置参数的值是通过在 SQL 语句的外部参数传递或者通过函数调用方式传递。 一些客户端接口库函数支持在 SQL 命令之外指定数值，在这种情况下参数引用的是 SQL 之外的实际值。

引用位置参数的格式如下：

```
$number
```

示例:

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

这里，`$1` 引用的是在函数调用时，传递给函数的第一个参数值。

下标表达式

如果一个表达式产生了一个数组类型值，那么你可以通过下面的方法获取数组中一个指定的元素值：

```
expression[subscript]
```

您还可以获取多个相连的元素值，称为数组分片，示例如下：

```
expression[lower_subscript:upper_subscript]
```

下标可以是一个表达式，该表达式必须返回整数值类型。

大部分时候，数组表达式必须在括号中使用。如果下标表达式访问列引用或者位置参数，括号是可以省略的。 对于多维数组，可以直接连接多个下标表达式来进行访问，示例如下：

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

成员选择表达式

如果表达式的值是一个复合类型（例如：记录类型），你可以通过下面的表达式来选择该复合类型中的特定成员值：

```
expression.fieldname
```

记录表达式通常需要在括号中使用，如果被访问的表达式是表引用或者位置参数，括号是可以省略的。示例：

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

一个限定的列引用是成员选择表达式的特例。

运算符调用

运算符调用支持下面的几种语法：

```
expression operator expression(binary infix operator)
operator expression(unary prefix operator)
expression operator(unary postfix operator)
```

示例中的 operator 实际是运算符符号，例如：AND，OR，+ 等。运算符也有限定名格式，例如：

```
OPERATOR(schema.operatorname)
```

可以使用的运算符以及他们究竟是一元运算符还是二元运算符，取决于系统和用户的定义。可以参考内建函数和运算符，了解更多信息。

函数调用

函数调用的语法是函数名（限定名格式：在函数名开头添加模式名）跟随着使用括号保护的参数列表：

```
function ([expression [, expression ... ]])
```

下面示例是通过函数调用计算2的平方根：

```
sqrt(2)
```

参考内置函数和运算符，了解更多信息。

聚集表达式

聚合表达式是指对于查询选择的所有数据记录上应用一个聚合函数。聚合函数在一组值上进行运算，并返回一个结果。例如：对一组值进行求和运算或者计算平均值。下面列出聚合表达式的语法结构：

- aggregate_name(expression [, …]) — 处理所有值为非空的输入记录值。
- aggregate_name(ALL expression [, …]) — 和上一个表达式行为一致，因为 ALL 是默认参数。
- aggregate_name(DISTINCT expression [, …]) — 处理所有去除重复后的非空输入记录值。
- aggregate_name(*) — 处理所有输入记录值，非空值和空值都会被处理。通常这个表达式都是为count(*)服务的。

上面表达式中的 aggregate_name 是一个预定义的聚合函数名称（可以使用模式限定名格式）。上面表达式中的 expression 可以是除聚合表达式自身外的任何值合法表达式。

例如，count(*) 返回输入记录的总数量，count(f1) 返回 f1 值中非空的总数量，count(distinct f1) 返回的是 f1 值中非空并去除重复值后的总数量。

要了解预定义的聚合函数，请参考内置函数和运算符。除了预定义聚合函数外，您还可以创建自定义的聚合函数。

HashData 数据仓库提供 MEDIAN 聚合函数，该函数返回 PERCENTILE_CONT 的 50 分位数结果。 下面是逆分布函数支持的特殊聚合表达式：

```
PERCENTILE_CONT(percentage) WITHIN GROUP (ORDER BY expression)
PERCENTILE_DISC(percentage) WITHIN GROUP (ORDER BY expression)
```

目前只有上面两个表达式可以使用关键字 WITHIN GROUP。

聚合表达式的限制

下面列出了目前聚合表达式的限制：

HashData 数据仓库不支持下面关键字：ALL，DISTINCT，FILTER 和 OVER。请参考 表5 了解更多信息。

聚合表达式只能出现在 SELECT 命令的结果列表或者 HAVING 子句中。在其它位置紧致访问聚合表达式，例如：WHERE。这是因为在其它其它位置的计算早于聚合数据的操作。此限制特指聚合表达式所属的查询层次。

当一个聚合表达式出现在子查询中，聚合操作相当于作用在子查询的返回结果上。如果聚合函数的参数只包含外层变量，该聚合表达式属于最近一层的外部表查询，并且也在该查询结 果上进行聚合运算。该聚合表达式对于出现的子查询来说，将会当成一个外部引用，并以常量值处理。请参考 表2 了解标量子查询。

HashData 数据仓库不支持在多个输入表达式上使用 DISTINCT。

窗口表达式

窗口表达式允许应用开发人员更加简单地通过标准SQL语言，来构建复杂的在线分析处理（OLAP）。 例如，通过使用窗口表达式，用户可以计算移动平均值，某个范围内的总和，根据某些列值的变化重置聚合表达式或排名，还可以用简单的表达式表述复杂的比例关系。

窗口表达式表示在窗口帧上应用窗口函数，窗口帧是通过非常特别的 OVER() 子句定义的。窗口分区是分组后的应用于窗口函数的记录集合。与聚合函数针对每个分组的记录返回一个结果不同，窗口函数真对每行都返回结果，但是该值的计算是完全真对根据记录对应的窗口分区进行的。如果不指定分区，窗口函数就会在整个结果集赏进行计算。

窗口表达式的语法如下：

```
window_function ( [expression [, ...]] ) OVER ( window_specification )
```

这里的 window_function 是表 3 列出的函数之一，表达式是任何不包含窗口表达式的合法值。window_specification 定义如下：

```
[window_name]
[PARTITION BY expression [, ...]]
[[ORDER BY expression [ASC | DESC | USING operator] [, ...]
    [{RANGE | ROWS}
       { UNBOUNDED PRECEDING
       | expression PRECEDING
       | CURRENT ROW
       | BETWEEN window_frame_bound AND window_frame_bound }]]
```

上面的 window_frame_bound 定义如下:

```
UNBOUNDED PRECEDING
expression PRECEDING
CURRENT ROW
expression FOLLOWING
UNBOUNDED FOLLOWING
```

窗口表达式只能在 SELECT 的返回里表中出现。例如：

```
SELECT count(*) OVER(PARTITION BY customer_id), * FROM sales;
```

OVER 子句是窗口函数与其他聚合函数或报表函数最大的区别。OVER 子句定义的 window_specification 确定了窗口函数应用的范围。窗口说明包含下面特征：

PARTITION BY 子句定义应用于窗口函数上的窗口分区。如果省略此参数，整个结果集将会作为一个分区使用。

ORDER BY 子句定义在窗口分区中用于排序的表达式。窗口说明中的 ORDER BY 子句和主查询中的 ORDER BY 子句是相互独立的。ORDER BY 子句对于计算排名的窗口函数来说是必需的，这是因为排序后才能获得排名值。 对于在线分析处理聚合操作，窗口帧（ROWS 或 RANGE 子句）需要 ORDER BY 子句才能使用。 ROWS/RANGE 子句为聚合窗口函数（非排名操作）定义一个窗口帧。窗口帧是在一个分区内的一组记录。 定义了窗口帧之后，窗口函数将会在移动窗口帧上进行计算，而不是固定的在整个窗口分区上进行。 窗口帧可以是基于记录分隔的也可以是基于值分隔的。

类型转换

类型转换表达式可以将一个数据类型的数据转换为另一个数据类型。HashData 数据仓库支持下面两种等价的类型转换语法：

```
CAST ( expression AS type )
expression::type
```

CAST 的语法是符合 SQL 标准的的；而语法 :: 是 PostgreSQL 历史遗留的习惯。

对于已知类型值表达式的类型转换操作是运行时类型转换。只有当系统中适用的类型转换函数，类型转换才可能成功。 这与直接在常量上应用类型转换并不相同。在字符串字面值上应用的类型转换代表了用字面值常量对一个类型进行初始赋值。 因此，该字符串字面值只要是该类型接收的合法输入，该类型转换都会成功。

在一些位置上，表达式的值类型如果不会产生歧义时，显示类型转换是可以被省略的。 例如，当为一张表的某个列赋值时，系统能够自动应用正确的类型转换。 系统要应用自动类型转换规则的前提是，当且仅当系统表中定义隐式地类型转换是合法的。 其他的类型转换，必需通过类型转换语法显示地进行调用。这样做可以阻止一部分用户意料之外的非期望类型转换的发生。

标量子查询

标量子查询是指一个括号中的 SELECT 查询语句，并且该语句返回值是一行一列（一个值）。标量子查询不支持使用返回多行或多列的 SELECT 查询语句。外部查询运行并使用相关自查询的返回结果。相关标量子查询是指标量子查询中引用了外部查询变量的查询。

相关子查询

相关子查询是指一个 SELECT 查询位于 返回列表或 WHERE 条件语句中，并引用了外部查询参数的查询语句。相关子查询允许更高效的表示出引用其他查询的返回结果。HashData 数据仓库能够支持相关子查询特性，此特性能够允许兼容很多已经存在的应用程序。 相关子查询可以根据返回记录是一条还是多条，返回结果可以是标量或者表表达式， 这取决于它返回的记录是一条还是多条。HashData 数据仓库目前不支持引用跨层的变量（不支持间接相关子查询）。

相关子查询示例

示例 ***1 –*** 标量相关子查询

```
SELECT * FROM t1 WHERE t1.x
           > (SELECT MAX(t2.x) FROM t2 WHERE t2.y = t1.y);
```

示例 ***2 –*** 相关 ***EXISTS*** 子查询

```
SELECT * FROM t1 WHERE
EXISTS (SELECT 1 FROM t2 WHERE t2.x = t1.x);
```

HashData 数据仓库利用下面两个算法来运行相关子查询：

将相关子查询展开成为连接运算：这种算是是最高效的方法，这也是 HashData 数据仓库对于大部分相关子查询使用的方法。一些 TPC-H 测试集中的查询都可以通过此方法进行优化。 对于引用的查询的每一条记录，都执行一次相关子查询：这是一种相对来说低效的算法。HashData 数据仓库对于位于 SELECT 返回列表中的相关子查和 WHERE 条件中 OR 连接表达式中的相关子查询使用这种算法。

下面的例子，向您展示对于不同类型的查询，如何通过查询重写来改进性能。

示例 ***3 - Select*** 返回列表中的相关子查询

原始查询

```
SELECT T1.a,
     (SELECT COUNT(DISTINCT T2.z) FROM t2 WHERE t1.x = t2.y) dt2
FROM t1;
```

重写后的查询首先与表 t1 执行内连接，再执行左外连接。查询重写只能对等值连接中的相关条件进行处理。

重写后的查询

```
SELECT t1.a, dt2 FROM t1
       LEFT JOIN
        (SELECT t2.y AS csq_y, COUNT(DISTINCT t2.z) AS dt2
              FROM t1, t2 WHERE t1.x = t2.y
              GROUP BY t1.x)
       ON (t1.x = csq_y);
```

示例 ***4 - OR*** 子句中的相关子查询 原始查询

```
SELECT * FROM t1
WHERE
x > (SELECT COUNT(*) FROM t2 WHERE t1.x = t2.x)
OR x < (SELECT COUNT(*) FROM t3 WHERE t1.y = t3.y)
```

重写后的查询是根据 OR 条件，将原来查询分成两个部分，并使用 UNION 进行连接。

重写后的查询

```
SELECT * FROM t1
WHERE x > (SELECT count(*) FROM t2 WHERE t1.x = t2.x)
UNION
SELECT * FROM t1
WHERE x < (SELECT count(*) FROM t3 WHERE t1.y = t3.y)
```

要查看查询计划，可以使用 EXPLAIN SELECT 或者 EXPLAIN ANALYZE SELECT。查询计划中的 Subplan 节点代表查询

将会对外部查询的每一条记录都处理一次，因此暗示着查询可能可以被重写和优化。

高级“表”表达式

HashData 数据仓库支持能够将“表”表达式作为参数的函数。您可以对输入高级“表”函数的记录使用 ORDER BY 进行排序。您可以使用 SCATTER BY 子句并指定一列或多列（或表达式）对输入记录进行重新分布。这种使用方式与创建表的时候，DISTRIBUTED BY 子句十分类似，但是此处重新分布的操作是在查询运行时发生的。

> 注意：根据数据的分布，HashData 数据仓库能够自动地在计算节点并行的运行 “表” 表达式。

数组构造表达式

数据构造表达式是通过提供成员值的方式构造数组值的表达式。一个简单的数组构造表达式由：关键字 ARRAY，左方括号（[），用来组成数组元素值的通过逗号分隔的一个多个表达式，和一个右方括号（]）。例如：

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
```

数组元素的类型就是其成员表达式的公共类型，确定的方式和 UNION，CASE 构造器规则相同。

通过嵌套数组构造表达式，您还可以创建多维数组值。内部的数组构造器，可以省略关键字 ARRAY。例如，下面两个 SELECT 语句返回的结果完全相同：

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
SELECT ARRAY[[1,2],[3,4]];
     array
--------------
 {{1,2},{3,4}}
```

由于多维数组一定是矩形（长方形），在同一层的内部构造表达式产生的子数组必须拥有相同的维度。

多维数组构造表达器中的元素不一定是子数组构造表达式，它们可以是任何一个产生适当类型数组的表达式。例如：

```
CREATE TABLE arr(f1 int[], f2 int[]);
INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]],
ARRAY[[5,6],[7,8]]);
SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                    array
------------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
```

你可以使用子查询的结果来构造数组。这里的数组构造表达式是关键字 ARRAY 开头，后面跟着在圆括号中的子查询。例如：

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                          ?column?
-----------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31}
```

这里的子查询只能返回单列。生成的一维数组中的每个元素对应着子查询每一条记录，数组元素的类型是子查询输出列的类型。通过数组构造表达式得到的数组，下标总是从 1 开始编号。

记录构造表达式

记录构造器是一种用来从成员值构建记录值的表达式（记录表达式也被称为复合类型）。例如：

```
SELECT ROW(1,2.5,'this is a test');
```

记录构造表达式还支持语法 rowvalue.* ，该表达式能够将记录值的成员展开成列表，这个操作类似于当你在 SELECT 目标列表时使用的 .* 语法。例如，如果表 t 有两列 f1 和 f2，下面的查询是等价的：

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

记录构造表达式默认创建的记录值具有匿名记录类型。根据需要，您可以将该值通过类型转换表达式，转换成一个命名复合类型：数据表的记录类型或者是通过 CREATE TYPE AS 命令创建的复合类型。您可以显示地提供类型转来避免出现歧义。例如：

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);
CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1'
LANGUAGE SQL;
```

下面的查询语句中，因为全局只有一个 getf1() 函数，所以这里不产生任何的歧义，您也就不需要进行类型转换的处理：

```
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);
CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT
$1.f1' LANGUAGE SQL;
```

下面的例子需要通过类型转换来指定具体调用的函数：

```
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique
SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS
myrowtype));
 getf1
-------
    11
```

你可以使用记录构造器来构建复合值，将其存储在复合类型的列中或者将其传给接受复合类型参数的函数。

表达式求值规则

自表达式的求值顺序是未定义的。运算符或者函数的求值不一定遵守从左到右的规则，也不保证按照任何特定顺序进行。

如果表达式的值能够由表达式中的一部分子表达式确定，那么其他部分的子表达式可能不会被求值。例如，下面的表达式：

```
SELECT true OR somefunc();
```

somefunc() 函数可能不会被调用。类似的情况对下面例子也适用：

```
SELECT somefunc() OR true;
```

这个特点和大多数编程语言中，布尔运算符求值顺序总是从左到右不太一样。

不要在复杂表达式中使用带有副作用的函数，特别是像 WHERE 或者 HAVING 子句。因为这里语句在生成查询计划过程中会被多次处理。在上面语句中的布尔表达式（AND/OR/NOT 组合）将会根据布尔代数规则重新排列成为任何最优合法结构。

您可以使用 CASE 结构来保证求值顺序。下面的示例就是一个不能保证避免除0错误的情况：

```
SELECT ... WHERE x <> 0 AND y/x > 1.5;
```

下面的示例能够保证避免除0错误：

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

这种 CASE 结构将会阻止查询优化，因此请小心使用。

使用函数和运算符

HashData 数据仓库在 SQL 表达式中对函数和运算符进行求值。一些函数和运算符只能运行在主节点上，如果在计算节点运行，会导致结果出现不一致状态。

如何使用函数

表**1 HashData** 数据仓库中的函数

函数类型	**HashData** 数据仓库支持情况	说明	备注
IMMUTABLE	完全支持	函数只直接依赖参数列表中提供的信息。对于相同的参数值，返回结果不变。	
STABLE	大部分情况支持	在一次的表扫描过程中，其返回结果对相同输入参数保持不变，但是结果在不同 SQL 语句之间会发生改变。	其返回结果取决于数据库查询或参数值。例如： current_timestamp 家族的函数都是 STABLE 的。在一次执行中，该函数值保持不变。
VOLATILE	限制性的使用	在一次表扫描过程中，函数值也会发生变化。 例如：random(), currval(), timeofday()。	即使结果可以预测，任何带有副作用（side effects）的函数仍然属于易变函数。例如：setval()。

HashData 数据仓库不支持函数返回表引用（rangeFuncs）或者函数使用 refCursor 数据类型。

用户自定义函数

此功能正在开发中，未来版本将会开放。

内置函数和运算符

下表列出了 PostgreSQL 支持的内置函数和运算符的种类。

表**2** 内置函数和运算符

Operator/Function Category	**Volatile** 函数
Logical Operators	
Comparison Operators	
Mathematical Functions and Operators	random、setseed
String Functions and Operators	All built-in conversion functions
Binary String Functions and Operators	
Bit String	

Functions and Operators	
Pattern Matching	
Data Type Formatting Functions	
Date/Time Functions and Operators	timeofday
Geometric Functions and Operators	
Network Address Functions and Operators	
Sequence Manipulation Functions	currval、lastval、nextval、setval
Conditional Expressions	
Array Functions and Operators	
Aggregate Functions	
Subquery Expressions	
Row and Array Comparisons	
Set Returning Functions	generate_series
System Information Functions	
System Administration Functions	set_config、pg_cancel_backend、pg_reload_conf、pg_rotate_logfile、pg_start_backup、pg
XML Functions	

窗口函数

下面列出的内置窗口函数是 HashData Database 对 PostgreSQL 的扩展。 所有的窗口函数都是 immutable 的。要了解更多关于窗口函数的信息，请参考 窗口表达式 。

函数	返回类型	语法	说明
cume_dist()	double precision	CUME_DIST() OVER ([PARTITION BY expr] ORDER BY expr)	计算数值在一个组数值的累积分布。相同值的记录求值结果总会得到相同的累积分布值。
dense_rank()	bigint	DENSE_RANK() OVER ([PARTITION BY expr] ORDER BY expr)	计算一个有序组中，记录的排名，排名值连续分配。记录 值相同的情况下，分配相同的排名。
first_value(expr)	与输入表达式 类型相同	`FIRST_VALUE( expr ) OVER ( [PARTITION BY expr ] ORDER BY expr [ROWS RANGE frame_expr ] )`	返回一个有序的值集合中的第一个值。
lag(expr [,offset] [,default])	与输入表达式类型相同	`LAG( expr [, offset ] [, default ]) OVER ( [PARTITION BY expr ] ORDER BY expr )`	提供一种不使用自连接访问一张表中多行记录。查询返回 时每个记录有一个物理位置，LAG 允许访问从当前物理位 置向前 offset 的记录。offset 的默认值是 1 。 default 值会在 offset 值超出窗口范围后被返回。 如果 default 没有指定，默认返回 NULL。
last_value(expr)	与输入表达式 类型相同	`LAST_VALUE(expr) OVER ( [PARTITION BY expr] ORDER BY expr [ROWS RANGE frame_expr] )`	返回一个有序的值集合中的最后一个值。
lead(expr [,offset] [,default])	与输入表达式 类型相同	`LEAD(expr [,offset] [,default]) OVER ( [PARTITION BY expr] ORDER BY expr )`	提供一种不使用自连接访问一张表中多行记录。查询返回 时每个记录有一个物理位置，LAG 允许访问从当前物理位 置向后 offset 的记录。offset 的默认值是 1 。 default 值会在 offset 值超出窗口范围后被返回。 如果 default 没有指定，默认返回 NULL。
ntile(expr)	bigint	NTILE(expr) OVER ([PARTITION BY expr] ORDER BY expr)	将一个有序数据集划分到多个桶（buckets）中，桶的数量 由参数决定，并为每条记录分配一个桶编号。
percent_rank()	double precision	PERCENT_RANK () OVER ([PARTITION BY expr] ORDER BY expr)	通过如下公式计算排名：记录排名 - 1 除以总排名记录 数 - 1 。
rank()	bigint	RANK () OVER ([PARTITION BY expr] ORDER BY expr)	计算一个有序组中，记录的排名。记录值相同的情况下， 分配相同的排名。分配到相同排名的每组记录的数量将会 被用来计算下个分配的排名。这种情况下，排名的分配可 能不是连续。
row_number()	bigint	ROW_NUMBER () OVER ([PARTITION BY expr] ORDER BY expr) 为每一个记录分配一个唯一的编号。（可以是整个查询的 结果记录也可以是窗口分区中的记录）。	\

高级分析函数

下面列出的内置窗口函数是 HashData Database 对 PostgreSQL 的扩展。 分析函数是 immutable 。

表4 高级分析函数

函数	返回类型	语法	说明
matrix_add(array[], array[])	smallint[], int[], bigint[], float[]	matrix_add(array[[1,1],[2,2]], array[[3,4],[5,6]])	将两个矩阵相加。两个矩阵必须一致。
matrix_multiply(array[], array[])	smallint[]int[], bigint[], float[]	matrix_multiply(array[[2,0,0], [0,2,0],[0,0,2]], array[[3,0,3], [0,3,0],[0,0,3]])	将两个矩阵相乘。两个矩阵必须一致。
matrix_multiply(array[], expr)	int[], float[]	matrix_multiply(array[[1,1,1], [2,2,2], [3,3,3]], 2)	将一个矩阵与一个标量数值相乘。
matrix_transpose(array[])	与输入表达式 类型相同	matrix_transpose(array [[1,1,1], [2,2,2]])	将一个矩阵转置。
pinv(array [])	smallint[]int[], bigint[], float[]	pinv(array[[2.5,0,0], [0,1,0], [0,0,.5]])	计算矩阵的 Moore-Penrose pseudoinverse 。
unnest(array[])	set of anyelement	unnest(array[‘one’, ‘row’, ‘per’, ‘item’])	将一维数组转化为多行。返回 anyelement 的集合。该类型是 PostgreSQL 中的多态伪类型。

表5 高级聚合函数

函数	返回类型	语法	说明
MEDIAN (expr)	timestamp, timestampz, interval, float	MEDIAN (expression) Example: `SELECT department_id MEDIAN(salary) FROM employees GROUP BY department_id;`	计算中位数。
PERCENTILE_CONT (expr) WITHIN GROUP (ORDER BY expr [DESC/ASC])	timestamp, timestampz, interval, float	PERCENTILE_CONT(percentage) WITHIN GROUP (ORDER BY expression) Example: `SELECT department_id,PERCENTILE_CONT (0.5) WITHIN GROUP (ORDER BY salary DESC) "Median_cont" FROM employees GROUP BY department_id;`	在连续分布模型下，进行逆分布函数运算。函数输入 一个分位比例和排序信息，返回类型是计算数据的类 型。计算结果是进行线性插值后的结果。 计算过程中 NULL 值将被忽略。
PERCENTILE_DISC (expr) WITHIN GROUP (ORDER BY expr [DESC/ASC])	timestamp, timestampz, interval, float	PERCENTILE_DISC(percentage) WITHIN GROUP (ORDER BY expression) Example: `SELECT department_id,PERCENTILE_DISC (0.5) WITHIN GROUP (ORDER BY salary DESC) "Median_cont" FROM employees GROUP BY department_id;`	在连续分布模型下，进行逆分布函数运算。函数输入 一个分位比例和排序信息。返回的结果是输入集合中 的值。计算过程中 NULL 值将被忽略。
sum(array[])	smallint[], int[], bigint[], float[]	sum(array[[1,2],[3,4]]) Example: `CREATE TABLE mymatrix (myvalue int[]); INSERT INTO mymatrix VALUES ( array[[1,2],[3,4]]); INSERT INTO mymatrix VALUES ( array[[0,1],[1,0]]); SELECT sum(myvalue) FROM mymatrix;`	执行矩阵加法运算。将输入的二维数组当作矩阵处理。
pivot_sum (label[], label, expr)	int[], bigint[], float[]	pivot_sum(array['A1','A2'], attr, value)	透视聚合函数，通过聚合求来消除重复项。
mregr_coef (expr, array[])	float[]	mregr_coef(y, array[1, x1, x2])	四个 mregr_* 开头的聚合函数使用最小二乘法进行线 性回归计算。mregr_coef 用于计算回归系数。 mregr_coef 返回的数组，包含每个自变量的回归系数， 因此大小与输入的自变量数组大小相等。
mregr_r2 (expr, array[])	float	mregr_r2(y, array[1, x1, x2])	四个 mregr_* 开头的聚合函数使用最小二乘法进行线 性回归计算。mregr_r2 计算回归的 r 平方值。
mregr_pvalues (expr, array[])	float[]	mregr_pvalues(y, array[1, x1, x2])	四个 mregr_* 开头的聚合函数使用最小二乘法进行线 性回归计算。mregr_pvalues 计算回归的 p 值。
mregr_tstats (expr, array[])	float[]	mregr_tstats(y, array[1, x1, x2])	四个 mregr_* 开头的聚合函数使用最小二乘法进行线 性回归计算。mregr_tstats 计算回归的 t 统计量值。
nb_classify (text[], bigint, bigint[], biggint[])	text	nb_classify(classes, attr_count, class_count, class_total)	使用朴素贝叶斯分类器对记录进行分类。此聚合函数使 用训练数据作为基线，对输入记录进行分类预测，返回 该记录最有可能出线的分类名称。
nb_probabilities (text[], bigint, bigint[], biggint[])	text	nb_probabilities(classes, attr_count, class_count, class_total)	使用朴素贝叶斯分类器计算每个分类的概率。此聚合函数使训练数据作为基线，对输入记录进行分类预测，返回该记录出现在各个分类中的概率。

高级分析函数示例

本章节向您展示在简化的示例数据上应用上面部分高级分析函数的操作过程。 示例包括：多元线性回归聚合函数和使用 nb_classify 的朴素贝叶斯分类。

线性回归聚合函数示例

下面示例使用四个线性回归聚合函数：mregr_coef，mregr_r2，mregr_pvalues 和 mregr_tstats 在示例表 regr_example 进行计算。 在下面的示例中，所有聚合函数第一个参数是因变量（dependent variable），第二个参数是自变量数组（independent variable）。

```
SELECT mregr_coef(y, array[1, x1, x2]),
       mregr_r2(y, array[1, x1, x2]),
       mregr_pvalues(y, array[1, x1, x2]),
       mregr_tstats(y, array[1, x1, x2])
from regr_example;
```

表 regr_example 中的数据:

```
 id | y  | x1 | x2
----+----+----+----
  1 |  5 |  2 |  1
  2 | 10 |  4 |  2
  3 |  6 |  3 |  1
  4 |  8 |  3 |  1
```

在表上运行前面的示例查询，返回一行数据，包含下面列出的值：

```
mregr_coef:

{-7.105427357601e-15,2.00000000000003,0.999999999999943}
mregr_r2:

0.86440677966103
mregr_pvalues:

{0.999999999999999,0.454371051656992,0.783653104061216}
mregr_tstats:

{-2.24693341988919e-15,1.15470053837932,0.35355339059327}
```

如果上面的聚合函数返回值未定义，HashData 数据仓库将会返回 NaN（不是一个数值）。当数据量太少时，可能遇到这种情况。

> 注意: 如上面的示例所示，变量参数估计值（intercept）是通过将一个自变量设置为 1 计算得到的。

朴素贝叶斯分类示例

使用 nb_classify 和 nb_probabilities 聚合函数涉及到四步的分类过程，包含了为训练数据创建的表和视图。 下面的两个示例展示了这四个步骤。第一个例子是在一个小的随意构造的数据集上展示。第二个例子是 HashData Database 根据天气条件使用非常受欢迎的贝叶斯分类的示例。

总览

下面向你介绍朴素贝叶斯分类的过程。在下面的示例中，值的名称（列名）将会做为属性值（field attr）使用：

- 将数据逆透视（unpivot）

如果数据是范式化存储的，需要对所有数据进行逆透视，连同标识字段和分类字段创建视图。 如果数据已经是非范式化的，请跳过此步。

- 创建训练表

训练表将数据视图变化为属性值（field attr）。

- 基于训练数据创建摘要视图

使用 nb_classify，nb_probabilities 或将 nb_classify 和 nb_probabilities 结合起来处理数据。

朴素贝叶斯分类示例**1 -** 小规模数据

例子将从包含范式化数据的示例表 class_example 开始，通过四个独立的步骤完成：

表 class_example 中的数据:

```
 id | class | a1 | a2 | a3
----+-------+----+----+----
  1 | C1    |  1 |  2 |  3
  2 | C1    |  1 |  4 |  3
  3 | C2    |  0 |  2 |  2
  4 | C1    |  1 |  2 |  1
  5 | C2    |  1 |  2 |  2
  6 | C2    |  0 |  1 |  3
```

将数据逆透视（unpivot）

为了能够用于训练数据，需要对 class_example 中范式化的数据进行逆透视操作。

在下面语句中，单引号引起的项将会作为新增的属性值（field attr）的值使用。习惯上，这些值与范式化的表列名相同。 在这个例子中，为了更容易的从命令中找到这些值，这些值以大写方式书写。

```
CREATE view class_example_unpivot AS
SELECT id, class,
  unnest(array['A1', 'A2', 'A3']) as attr,
  unnest(array[a1,a2,a3]) as value
FROM class_example;
```

使用后面的 SQL 语句可以查看非范式化的数据 `SELECT * from class_example_unpivot` ：

```
 id | class | attr | value
----+-------+------+-------
  2 | C1    | A1   |     1
  2 | C1    | A2   |     2
  2 | C1    | A3   |     1
  4 | C2    | A1   |     1
  4 | C2    | A2   |     2
  4 | C2    | A3   |     2
  6 | C2    | A1   |     0
  6 | C2    | A2   |     1
  6 | C2    | A3   |     3
  1 | C1    | A1   |     1
  1 | C1    | A2   |     2
  1 | C1    | A3   |     3
  3 | C1    | A1   |     1
  3 | C1    | A2   |     4
  3 | C1    | A3   |     3
  5 | C2    | A1   |     0
  5 | C2    | A2   |     2
  5 | C2    | A3   |     2
(18 rows)
```

创建训练表

下面查询中，单引号引起的项用来定义聚合的项。通过数组形式传递给 pivto_sum 函数的项必须和原始数据的分类名称和数量相符。 本例中，分类为 C1 和 C2：

```
CREATE table class_example_nb_training AS
SELECT attr, value,
  pivot_sum(array['C1', 'C2'], class, 1) as class_count
FROM   class_example_unpivot
GROUP BY attr, value
DISTRIBUTED by (attr);
```

下面是训练表的结果：

```
 attr | value | class_count
------+-------+-------------
 A3   |     1 | {1,0}
 A3   |     3 | {2,1}
 A1   |     1 | {3,1}
 A1   |     0 | {0,2}
 A3   |     2 | {0,2}
 A2   |     2 | {2,2}
 A2   |     4 | {1,0}
 A2   |     1 | {0,1}
(8 rows)
```

基于训练数据创建摘要视图

```
CREATE VIEW class_example_nb_classify_functions AS
SELECT attr, value, class_count,
  array['C1', 'C2'] AS classes,
  SUM(class_count) OVER (wa)::integer[] AS class_total,
  COUNT(DISTINCT value) OVER (wa) AS attr_count
FROM class_example_nb_training
WINDOW wa AS (partition by attr);
```

下面是训练数据的最终结果：

```
 attr| value | class_count| classes | class_total |attr_count
-----+-------+------------+---------+-------------+---------
 A2  |     2 | {2,2}      | {C1,C2} | {3,3}       |         3
 A2  |     4 | {1,0}      | {C1,C2} | {3,3}       |         3
 A2  |     1 | {0,1}      | {C1,C2} | {3,3}       |         3
 A1  |     0 | {0,2}      | {C1,C2} | {3,3}       |         2
 A1  |     1 | {3,1}      | {C1,C2} | {3,3}       |         2
 A3  |     2 | {0,2}      | {C1,C2} | {3,3}       |         3
 A3  |     3 | {2,1}      | {C1,C2} | {3,3}       |         3
 A3  |     1 | {1,0}      | {C1,C2} | {3,3}       |         3
(8 rows)
```

使用 nb_classify 对记录进行归类以及使用 nb_probabilities 显示归类的概率分布。

在准备好摘要视图后，训练的数据已经可以做为对新记录分类的基线了。下面的查询将会通过 nb_classify 聚合函数来预测新的记录是属于 C1 还是 C2。

```
SELECT nb_classify(classes, attr_count, class_count, class_total) AS class
FROM class_example_nb_classify_functions
WHERE (attr = 'A1' AND value = 0)
  OR (attr = 'A2' AND value = 2)
  OR (attr = 'A3' AND value = 1);
```

使用前面的训练数据后，运行示例查询，返回下面符合期望的单行结果：

```
class
-------
C2
(1 row)
```

可以使用 nb_probabilities 显示各个列别的概率。在准备好视图后，训练的数据已经可以做为对新记录分类的基线了。 下面的查询将会通过 nb_probabilities 聚合函数来预测新的记录是属于 C1 还是 C2。

```
SELECT nb_probabilities(classes, attr_count, class_count, class_total) AS probability
FROM class_example_nb_classify_functions
WHERE (attr = 'A1' AND value = 0)
  OR (attr = 'A2' AND value = 2)
  OR (attr = 'A3' AND value = 1);
```

使用前面的训练数据后，运行示例查询，返回每个分类的概率情况，第一值是 C1 的概率，第二个值是 C2 的概率：

```
probability
------------
 {0.4,0.6}
(1 row)
```

您可以通过下面的查询同时显示分类结果和概率分布。

```
SELECT nb_classify(classes, attr_count, class_count, class_total) AS class,
       nb_probabilities(classes, attr_count, class_count, class_total) AS probability
FROM class_example_nb_classify
WHERE (attr = 'A1' AND value = 0)
  OR (attr = 'A2' AND value = 2)
  OR (attr = 'A3' AND value = 1);
```

查询返回如下结果：

```
 class | probability
-------+-------------
    C2 | {0.4,0.6}
 (1 row)
```

在生产环境中的真实数据相比示例数据更加全面，因此预测效果更好。 当训练数据集较大时，使用 nb_classify 和 nb_probabilities 的归类准确度将会显著提高。

朴素贝叶斯分类示例**2 –** 天气和户外运动

在这个示例中，将会根据天情况来计算是否适宜用户进行户外运动，例如：高尔夫球或者网球。 表 weather_example 包含了一些示例数据。表的标示字段是 day。 用于分类的字段 play 包含两个值：Yes 或 No。天气包含四种属性：状况，温度，湿度，风力。 数据按照范式化存储。

```
 day | play | outlook  | temperature | humidity | wind
-----+------+----------+-------------+----------+--------
 2   | No   | Sunny    | Hot         | High     | Strong
 4   | Yes  | Rain     | Mild        | High     | Weak
 6   | No   | Rain     | Cool        | Normal   | Strong
 8   | No   | Sunny    | Mild        | High     | Weak
10   | Yes  | Rain     | Mild        | Normal   | Weak
12   | Yes  | Overcast | Mild        | High     | Strong
14   | No   | Rain     | Mild        | High     | Strong
 1   | No   | Sunny    | Hot         | High     | Weak
 3   | Yes  | Overcast | Hot         | High     | Weak
 5   | Yes  | Rain     | Cool        | Normal   | Weak
 7   | Yes  | Overcast | Cool        | Normal   | Strong
 9   | Yes  | Sunny    | Cool        | Normal   | Weak
11   | Yes  | Sunny    | Mild        | Normal   | Strong
13   | Yes  | Overcast | Hot         | Normal   | Weak
 (14 rows)
```

由于这里的数据是按照范式化存储的，因此贝叶斯分类的四个步骤都是需要的。

逆透视化数据

```
CREATE view weather_example_unpivot AS
SELECT day, play,
  unnest(array['outlook','temperature', 'humidity','wind']) AS attr,
  unnest(array[outlook,temperature,humidity,wind]) AS value
FROM weather_example;
```

请注意上面单引号的使用。

语句 `SELECT * from weather_example_unpivot` 将会显示经过逆透视化后的非范式的数据，数据一共 56 行。

```
 day | play | attr        | value
-----+------+-------------+----------
   2 | No   | outlook     | Sunny
   2 | No   | temperature | Hot
   2 | No   | humidity    | High
   2 | No   | wind        | Strong
   4 | Yes  | outlook     | Rain
   4 | Yes  | temperature | Mild
   4 | Yes  | humidity    | High
   4 | Yes  | wind        | Weak
   6 | No   | outlook     | Rain
   6 | No   | temperature | Cool
   6 | No   | humidity    | Normal
   6 | No   | wind        | Strong
   8 | No   | outlook     | Sunny
   8 | No   | temperature | Mild
   8 | No   | humidity    | High
   8 | No   | wind        | Weak
  10 | Yes  | outlook     | Rain
  10 | Yes  | temperature | Mild
  10 | Yes  | humidity    | Normal
  10 | Yes  | wind        | Weak
  12 | Yes  | outlook     | Overcast
  12 | Yes  | temperature | Mild
  12 | Yes  | humidity    | High
  12 | Yes  | wind        | Strong
  14 | No   | outlook     | Rain
  14 | No   | temperature | Mild
  14 | No   | humidity    | High
  14 | No   | wind        | Strong
   1 | No   | outlook     | Sunny
   1 | No   | temperature | Hot
   1 | No   | humidity    | High
   1 | No   | wind        | Weak
   3 | Yes  | outlook     | Overcast
   3 | Yes  | temperature | Hot
   3 | Yes  | humidity    | High
   3 | Yes  | wind        | Weak
   5 | Yes  | outlook     | Rain
   5 | Yes  | temperature | Cool
   5 | Yes  | humidity    | Normal
   5 | Yes  | wind        | Weak
   7 | Yes  | outlook     | Overcast
   7 | Yes  | temperature | Cool
   7 | Yes  | humidity    | Normal
   7 | Yes  | wind        | Strong
   9 | Yes  | outlook     | Sunny
   9 | Yes  | temperature | Cool
   9 | Yes  | humidity    | Normal
   9 | Yes  | wind        | Weak
  11 | Yes  | outlook     | Sunny
  11 | Yes  | temperature | Mild
  11 | Yes  | humidity    | Normal
  11 | Yes  | wind        | Strong
  13 | Yes  | outlook     | Overcast
  13 | Yes  | temperature | Hot
  13 | Yes  | humidity    | Normal
  13 | Yes  | wind        | Weak
 (56 rows)
```

创建训练表

```
CREATE table weather_example_nb_training AS
SELECT attr, value, pivot_sum(array['Yes','No'], play, 1) AS class_count
FROM weather_example_unpivot
GROUP BY attr, value
DISTRIBUTED BY (attr);
```

语句 `SELECT * from weather_example_nb_training` 显示训练数据，一共 10 行：

```
 attr        | value    | class_count
-------------+----------+-------------
outlook      | Rain     | {3,2}
humidity     | High     | {3,4}
outlook      | Overcast | {4,0}
humidity     | Normal   | {6,1}
outlook      | Sunny    | {2,3}
wind         | Strong   | {3,3}
temperature  | Hot      | {2,2}
temperature  | Cool     | {3,1}
temperature  | Mild     | {4,2}
wind         | Weak     | {6,2}
(10 rows)
```

基于训练数据创建摘要视图

```
CREATE VIEW weather_example_nb_classify_functions AS
SELECT attr, value, class_count,
  array['Yes','No'] as classes,
  sum(class_count) over (wa)::integer[] as class_total,
  count(distinct value) over (wa) as attr_count
FROM weather_example_nb_training
WINDOW wa as (partition by attr);
```

语句 `SELECT * from weather_example_nb_classify_function` 将会返回 10 行训练数据。

```
attr        |  value  | class_count| classes | class_total| attr_count
------------+-------- +------------+---------+------------+-----------
temperature | Mild    | {4,2}      | {Yes,No}| {9,5}      |         3
temperature | Cool    | {3,1}      | {Yes,No}| {9,5}      |         3
temperature | Hot     | {2,2}      | {Yes,No}| {9,5}      |         3
wind        | Weak    | {6,2}      | {Yes,No}| {9,5}      |         2
wind        | Strong  | {3,3}      | {Yes,No}| {9,5}      |         2
humidity    | High    | {3,4}      | {Yes,No}| {9,5}      |         2
humidity    | Normal  | {6,1}      | {Yes,No}| {9,5}      |         2
outlook     | Sunny   | {2,3}      | {Yes,No}| {9,5}      |         3
outlook     | Overcast| {4,0}      | {Yes,No}| {9,5}      |         3
outlook     | Rain    | {3,2}      | {Yes,No}| {9,5}      |         3
(10 rows)
```

使用 nb_classify，nb_probabilities 或将 nb_classify 和 nb_probabilities 结合起来处理数据。

首先要决定对什么样的信息进行归类。例如对下面一条记录进行归类。

```
temperature | wind | humidity | outlook
------------+------+----------+---------
Cool        | Weak | High     | Overcast
```

下面的查询用来计算分类结果。结果将会给出判断是否适宜户外运动，并给出 Yes 和 No 的概率。

```
SELECT nb_classify(classes, attr_count, class_count, class_total) AS class,
       nb_probabilities(classes, attr_count, class_count, class_total) AS probability
FROM weather_example_nb_classify_functions
WHERE
  (attr = 'temperature' AND value = 'Cool') OR
  (attr = 'wind'        AND value = 'Weak') OR
  (attr = 'humidity'    AND value = 'High') OR
  (attr = 'outlook'     AND value = 'Overcast');
```

结果是一条记录：

```
class  |             probability
-------+-------------------------------------
 Yes   | {0.858103353920726,0.141896646079274}
(1 row)
```

要对一组记录进行分类，可以将他们存储到表中，再进行归类。例如下面的表 t1 包含了要分类的记录：

```
 day | outlook  | temperature | humidity |  wind
-----+----------+-------------+----------+--------
  15 | Sunny    | Mild        | High     | Strong
  16 | Rain     | Cool        | Normal   | Strong
  17 | Overcast | Hot         | Normal   | Weak
  18 | Rain     | Hot         | High     | Weak
(4 rows)
```

下面的查询用来对整张表计算分类结果。计算结果是对表中每条记录判断是否适宜户外运动，并给出 Yes 和 No 的概率。 这个例子中 nb_classify 和 nb_probabilities 两个聚合函数都参与了运算。

```
SELECT t1.day,
       t1.temperature, t1.wind, t1.humidity, t1.outlook,
       nb_classify(classes, attr_count, class_count, class_total) AS class,
       nb_probabilities(classes, attr_count, class_count, class_total) AS probability
FROM t1, weather_example_nb_classify_functions
WHERE
  (attr = 'temperature' AND value = t1.temperature) OR
  (attr = 'wind'        AND value = t1.wind) OR
  (attr = 'humidity'    AND value = t1.humidity) OR
  (attr = 'outlook'     AND value = t1.outlook)
GROUP BY t1.day, t1.temperature, t1.wind, t1.humidity, t1.outlook;
```

结果返回四条记录，分别对应 t1 中的记录。

```
day| temp| wind   | humidity | outlook  | class | probability
---+-----+--------+----------+----------+-------+-------------
15 | Mild| Strong | High     | Sunny    | No    | {0.244694132334582,0.755305867665418}
16 | Cool| Strong | Normal   | Rain     | Yes   | {0.751471997809119,0.248528002190881}
18 | Hot | Weak   | High     | Rain     | No    | {0.446387538890131,0.553612461109869}
17 | Hot | Weak   | Normal   | Overcast | Yes   | {0.9297192642788,0.0702807357212004}
(4 rows)
```

SQL 命令参考

下面是 HashData 中支持的SQL命令：

- SQL语法概要
- ABORT
- ALTER AGGREGATE
- ALTER CONVERSION
- ALTER DATABASE
- ALTER DOMAIN
- ALTER EXTENSION
- ALTER EXTERNAL TABLE
- ALTER FILESPACE
- ALTER FUNCTION
- ALTER GROUP
- ALTER INDEX
- ALTER LANGUAGE
- ALTER OPERATOR
- ALTER OPERATOR CLASS
- ALTER PROTOCOL
- ALTER RESOURCE QUEUE
- ALTER ROLE
- ALTER SCHEMA
- ALTER SEQUENCE
- ALTER TABLE
- ALTER TABLESPACE
- ALTER TYPE
- ALTER USER
- ANALYZE
- BEGIN
- CHECKPOINT
- CLOSE
- CLUSTER
- COMMENT
- COPY
- CREATE AGGREGATE

- CREATE CAST
- CREATE CONVERSION
- CREATE DATABASE
- CREATE DOMAIN
- CREATE EXTENSION
- CREATE EXTERNAL TABLE
- CREATE FUNCTION
- CREATE GROUP
- CREATE INDEX
- CREATE LANGUAGE
- CREATE OPERATOR
- CREATE OPERATOR CLASS
- CREATE OPERATOR FAMILY
- CREATE PROTOCOL
- CREATE RESOURCE QUEUE
- CREATE ROLE
- CREATE RULE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TABLE AS
- CREATE TABLESPACE
- CREATE TYPE
- CREATE USER
- CREATE VIEW
- DEALLOCATE
- DECLARE
- DELETE
- DISCARD
- DO
- DROP AGGREGATE
- DROP CAST
- DROP CONVERSION
- DROP DATABASE
- DROP DOMAIN
- DROP EXTENSION

- DROP EXTERNAL TABLE
- DROP FILESPACE
- DROP FUNCTION
- DROP GROUP
- DROP INDEX
- DROP LANGUAGE
- DROP OPERATOR
- DROP OPERATOR CLASS
- DROP OPERATOR FAMILY
- DROP OWNED
- DROP PROTOCOL
- DROP RESOURCE QUEUE
- DROP ROLE
- DROP RULE
- DROP SCHEMA
- DROP SEQUENCE
- DROP TABLE
- DROP TABLESPACE
- DROP TYPE
- DROP USER
- DROP VIEW
- END
- EXECUTE
- EXPLAIN
- FETCH
- GRANT
- INSERT
- LOAD
- LOCK
- MOVE
- PREPARE
- REASSIGN OWNED
- REINDEX
- RELEASE SAVEPOINT
- RESET
- REVOKE

- ROLLBACK
- ROLLBACK TO SAVEPOINT
- SAVEPOINT
- SELECT
- SELECT INTO
- SET
- SET ROLE
- SET SESSION AUTHORIZATION
- SET TRANSACTION
- SHOW
- START TRANSACTION
- TRUNCATE
- UPDATE
- VACUUM
- VALUES

上级主题：HashData 数据库参考指南

SQL语法概要

终止

终止当前事务

```
ABORT [WORK | TRANSACTION]
```

更多信息参阅 ABORT。

修改聚集函数

改变聚集函数的定义

```
ALTER AGGREGATE name ( type [ , ... ] ) RENAME TO new_name

ALTER AGGREGATE name ( type [ , ... ] ) OWNER TO new_owner

ALTER AGGREGATE name ( type [ , ... ] ) SET SCHEMA new_schema
```

更多信息参阅 ALTER AGGREGATE。

修改转换

修改转换的定义。

```
ALTER CONVERSION name RENAME TO newname

ALTER CONVERSION name OWNER TO newowner
```

更多信息参阅 ALTER CONVERSION。

修改数据库

修改数据库属性

```
ALTER DATABASE name [ WITH CONNECTION LIMIT connlimit ]

ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }

ALTER DATABASE name RESET parameter

ALTER DATABASE name RENAME TO newname

ALTER DATABASE name OWNER TO new_owner
```

更多信息参阅 ALTER DATABASE。

修改域

改变域的定义

```
ALTER DOMAIN name { SET DEFAULT expression | DROP DEFAULT }

ALTER DOMAIN name { SET | DROP } NOT NULL

ALTER DOMAIN name ADD domain_constraint

ALTER DOMAIN name DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]

ALTER DOMAIN name OWNER TO new_owner

ALTER DOMAIN name SET SCHEMA new_schema
```

更多信息请参阅 ALTER DOMAIN。

修改扩展

改变在 HashData 数据库中注册的扩展的定义。

```
ALTER EXTENSION name UPDATE [ TO new_version ]
ALTER EXTENSION name SET SCHEMA new_schema
ALTER EXTENSION name ADD member_object
ALTER EXTENSION name DROP member_object
```

其中 member_object 是：

```
  ACCESS METHOD object_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  CONVERSION object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  [ PROCEDURAL ] LANGUAGE object_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  TABLE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TYPE object_name |
  VIEW object_name
```

aggregate_signature 是：

```
* | [ argmode ] [ argname ] argtype [ , ... ] |
  [ [ argmode ] [ argname ] argtype [ , ... ] ]
    ORDER BY [ argmode ] [ argname ] argtype [ , ... ]
```

更多信息参阅 ALTER EXTENSION。

修改外部表

改变外部表的定义。

```
ALTER EXTERNAL TABLE name RENAME [COLUMN] column TO new_column

ALTER EXTERNAL TABLE name RENAME TO new_name

ALTER EXTERNAL TABLE name SET SCHEMA new_schema

ALTER EXTERNAL TABLE name action [, ... ]
```

更多信息参阅 ALTER EXTERNAL TABLE。

修改文件空间

改变文件空间的定义

```
ALTER FILESPACE name RENAME TO newname

ALTER FILESPACE name OWNER TO newowner
```

更多信息参阅 ALTER FILESPACE。

修改函数

改变函数的定义

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   action [, ... ] [RESTRICT]

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   RENAME TO new_name

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   OWNER TO new_owner

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   SET SCHEMA new_schema
```

更多信息参阅 ALTER FUNCTION。

修改组

改变角色名字或者成员信息

```
ALTER GROUP groupname ADD USER username [, ... ]

ALTER GROUP groupname DROP USER username [, ... ]

ALTER GROUP groupname RENAME TO newname
```

更多信息参阅 ALTER GROUP。

修改索引

改变索引的定义

```
ALTER INDEX name RENAME TO new_name

ALTER INDEX name SET TABLESPACE tablespace_name

ALTER INDEX name SET ( FILLFACTOR = value )

ALTER INDEX name RESET ( FILLFACTOR )
```

更多信息参阅 ALTER INDEX。

修改语言

改变程序语言的名字

```
ALTER LANGUAGE name RENAME TO newname
ALTER LANGUAGE name OWNER TO new_owner
```

更多信息参阅 ALTER LANGUAGE。

修改操作符

改变操作符的定义

```
ALTER OPERATOR name ( {lefttype | NONE} , {righttype | NONE} )
   OWNER TO newowner
```

更多信息参阅 ALTER OPERATOR。

修改操作符类

改变操作符类的定义

```
ALTER OPERATOR CLASS name USING index_method RENAME TO newname

ALTER OPERATOR CLASS name USING index_method OWNER TO newowner
```

更多信息参阅 ALTER OPERATOR CLASS。

修改操作符族

修改操作符族的定义

```
ALTER OPERATOR FAMILY name USING index_method ADD
  {  OPERATOR strategy_number operator_name ( op_type, op_type ) [ RECHECK ]
    | FUNCTION support_number [ ( op_type [ , op_type ] ) ] funcname ( argument_type [, ...] )
  } [, ... ]
ALTER OPERATOR FAMILY name USING index_method DROP
  {  OPERATOR strategy_number ( op_type, op_type )
    | FUNCTION support_number [ ( op_type [ , op_type ] )
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method RENAME TO newname

ALTER OPERATOR FAMILY name USING index_method OWNER TO newowner
```

修改协议

修改协议的定义

```
ALTER PROTOCOL name RENAME TO newname

ALTER PROTOCOL name OWNER TO newowner
```

更多信息参阅 ALTER PROTOCOL。

修改资源队列

修改资源队列的限制

```
ALTER RESOURCE QUEUE name WITH ( queue_attribute=value [, ... ] )
```

更新信息参阅 ALTER RESOURCE QUEUE。

修改角色

修改数据库角色（用户或组）。

```
ALTER ROLE name RENAME TO newname

ALTER ROLE name SET config_parameter {TO | =} {value | DEFAULT}

ALTER ROLE name RESET config_parameter

ALTER ROLE name RESOURCE QUEUE {queue_name | NONE}

ALTER ROLE name [ [WITH] option [ ... ] ]
```

更多信息参阅 ALTER ROLE。

修改模式

改变模式的定义

```
ALTER SCHEMA name RENAME TO newname

ALTER SCHEMA name OWNER TO newowner
```

更多信息参阅 ALTER SCHEMA。

修改序列

改变序列生成器的定义

```
ALTER SEQUENCE name [INCREMENT [ BY ] increment]
     [MINVALUE minvalue | NO MINVALUE]
     [MAXVALUE maxvalue | NO MAXVALUE]
     [RESTART [ WITH ] start]
     [CACHE cache] [[ NO ] CYCLE]
     [OWNED BY {table.column | NONE}]

ALTER SEQUENCE name RENAME TO new_name
ALTER SEQUENCE name SET SCHEMA new_schema
```

更多信息参阅 ALTER SEQUENCE。

修改表

改变的表的定义

```
ALTER TABLE [ONLY] name RENAME [COLUMN] column TO new_column

ALTER TABLE name RENAME TO new_name

ALTER TABLE name SET SCHEMA new_schema

ALTER TABLE [ONLY] name SET
     DISTRIBUTED BY (column, [ ... ] )
   | DISTRIBUTED RANDOMLY
   | WITH (REORGANIZE=true|false)

ALTER TABLE [ONLY] name action [, ... ]

ALTER TABLE name
   [ ALTER PARTITION { partition_name | FOR (RANK(number))
   | FOR (value) } partition_action [...] ]
   partition_action
```

更多信息参阅 ALTER TABLE。

修改表空间

改变表空间的定义。

```
ALTER TABLESPACE name RENAME TO newname

ALTER TABLESPACE name OWNER TO newowner
```

更多信息参阅 ALTER TABLESPACE。

修改类型

改变数据类型的定义

```
ALTER TYPE name
   OWNER TO new_owner | SET SCHEMA new_schema
```

更多信息参阅 ALTER TYPE。

修改用户

修改数据库角色（用户）的定义。

```
ALTER USER name RENAME TO newname

ALTER USER name SET config_parameter {TO | =} {value | DEFAULT}

ALTER USER name RESET config_parameter

ALTER USER name [ [WITH] option [ ... ] ]
```

更新信息参阅 ALTER USER。

修改视图

改变视图的定义

```
ALTER VIEW name RENAME TO newname
```

分析

收集关于数据库的数据

```
ANALYZE [VERBOSE] [ROOTPARTITION [ALL] ]
   [table [ (column [, ...] ) ]]
```

更新信息参阅 ANALYZE。

开始

启动事务块

```
BEGIN [WORK | TRANSACTION] [transaction_mode]
      [READ ONLY | READ WRITE]
```

更多信息参阅 BEGIN 。

检查点

强制事务记录检查点

```
CHECKPOINT
```

更多信息参阅 CHECKPOINT。

关闭

关闭游标

```
CLOSE cursor_name
```

更多信息参阅 CLOSE 。

集簇

根据索引对磁盘上的堆存储表进行物理重新排序。不是 HashData 数据库的推荐操作。

```
CLUSTER indexname ON tablename

CLUSTER tablename

CLUSTER
```

更多信息参阅 CLUSTER。

注释

定义或者修改对一个对象的注释。

```
COMMENT ON
{ TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type [, ...]) |
  CAST (sourcetype AS targettype) |
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  FILESPACE object_name |
  FUNCTION func_name ([[argmode] [argname] argtype [, ...]]) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  OPERATOR op (leftoperand_type, rightoperand_type) |
  OPERATOR CLASS object_name USING index_method |
  [PROCEDURAL] LANGUAGE object_name |
  RESOURCE QUEUE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TABLESPACE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name }
IS 'text'
```

更多信息参阅 COMMENT。

提交

提交当前事务

```
COMMIT [WORK | TRANSACTION]
```

更多信息参阅 COMMIT。

复制

在文件和表之间拷贝数据。

```
COPY table [(column [, ...])] FROM {'file' | STDIN}
     [ [WITH]
       [BINARY]
       [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]
       SEGMENT REJECT LIMIT count [ROWS | PERCENT] ]

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

更多信息参阅 COPY 。

创建聚集函数

定义一个新的聚集函数

```
CREATE [ORDERED] AGGREGATE name (input_data_type [ , ... ])
      ( SFUNC = sfunc,
        STYPE = state_data_type
        [, PREFUNC = prefunc]
        [, FINALFUNC = ffunc]
        [, INITCOND = initial_condition]
        [, SORTOP = sort_operator] )
```

更多信息参阅 CREATE AGGREGATE。

创建投影

定义一个新的投影。

```
CREATE CAST (sourcetype AS targettype)
       WITH FUNCTION funcname (argtypes)
       [AS ASSIGNMENT | AS IMPLICIT]

CREATE CAST (sourcetype AS targettype) WITHOUT FUNCTION
       [AS ASSIGNMENT | AS IMPLICIT]
```

更多信息参阅 CREATE CAST。

创建转换

定义一个新的编码转换。

```
CREATE [DEFAULT] CONVERSION name FOR source_encoding TO
     dest_encoding FROM funcname
```

更多信息参阅 CREATE CONVERSION。

创建数据库

创建一个信息的数据库。

```
CREATE DATABASE name [ [WITH] [OWNER [=] dbowner]
                     [TEMPLATE [=] template]
                     [ENCODING [=] encoding]
                     [TABLESPACE [=] tablespace]
                     [CONNECTION LIMIT [=] connlimit ] ]
```

更多信息参阅 CREATE DATABASE。

创建域

定义一个新的域。

```
CREATE DOMAIN name [AS] data_type [DEFAULT expression]
       [CONSTRAINT constraint_name
       | NOT NULL | NULL
       | CHECK (expression) [...]]
```

更多信息参阅 CREATE DOMAIN。

创建扩展

在 HashData 数据库中注册一个扩展。

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
  [ WITH ] [ SCHEMA schema_name ]
           [ VERSION version ]
           [ FROM old_version ]
           [ CASCADE ]
```

更多信息参阅 CREATE EXTENSION。

创建外部表

定义一张外部表

```
CREATE [READABLE] EXTERNAL TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION ('file://seghost[:port]/path/file' [, ...])
       | ('gpfdist://filehost[:port]/file_pattern[#transform=trans_name]'
          [, ...]
       | ('gpfdists://filehost[:port]/file_pattern[#transform=trans_name]'
          [, ...])
       | ('gphdfs://hdfs_host[:port]/path/file')
       | ('s3://S3_endpoint[:port]/bucket_name/[S3_prefix]
            [region=S3-region]
           [config=config_file]')
     [ON MASTER]
     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] )]
          | 'AVRO'
          | 'PARQUET'
          | 'CUSTOM' (Formatter=<formatter_specifications>)
    [ ENCODING 'encoding' ]
      [ [LOG ERRORS] SEGMENT REJECT LIMIT count
      [ROWS | PERCENT] ]

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] )]
          | 'CUSTOM' (Formatter=<formatter specifications>)
     [ ENCODING 'encoding' ]
     [ [LOG ERRORS] SEGMENT REJECT LIMIT count
       [ROWS | PERCENT] ]

CREATE WRITABLE EXTERNAL TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION('gpfdist://outputhost[:port]/filename[#transform=trans_name]'
          [, ...])
      | ('gpfdists://outputhost[:port]/file_pattern[#transform=trans_name]'
          [, ...])
      | ('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'] )]
          | 'AVRO'
          | 'PARQUET'

          | 'CUSTOM' (Formatter=<formatter specifications>)
    [ ENCODING 'write_encoding' ]
    [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]

CREATE WRITABLE EXTERNAL TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION('s3://S3_endpoint[:port]/bucket_name/[S3_prefix]
            [region=S3-region]
            [config=config_file]')
```

```
      [ON MASTER]
      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'] )]

CREATE WRITABLE EXTERNAL WEB TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
    EXECUTE 'command' [ON ALL]
    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'] )]
           | 'CUSTOM' (Formatter=<formatter specifications>)
    [ ENCODING 'write_encoding' ]
    [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
```

更多信息参阅 CREATE EXTERNAL TABLE。

创建函数

定义一个新的函数

```
CREATE [OR REPLACE] FUNCTION name
    ( [ [argmode] [argname] argtype [ { DEFAULT | = } defexpr ] [, ...] ] )
      [ RETURNS { [ SETOF ] rettype
        | TABLE ([{ argname argtype | LIKE other table }
          [, ...]])
        } ]
    { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINE
    | COST execution_cost
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol' } ...
    [ WITH ({ DESCRIBE = describe_function
           } [, ...] ) ]
```

更多信息参阅 CREATE FUNCTION。

创建组

定义一个新的数据库角色。

```
CREATE GROUP name [ [WITH] option [ ... ] ]
```

更多信息参阅 CREATE GROUP。

创建索引

定义一个新的索引。

```
CREATE [UNIQUE] INDEX name ON table
       [USING btree|bitmap|gist]
       ( {column | (expression)} [opclass] [, ...] )
       [ WITH ( FILLFACTOR = value ) ]
       [TABLESPACE tablespace]
       [WHERE predicate]
```

更多信息参阅 CREATE INDEX。

创建语言

定义一个新的程序语言。

```
CREATE [PROCEDURAL] LANGUAGE name

CREATE [TRUSTED] [PROCEDURAL] LANGUAGE name
       HANDLER call_handler [ INLINE inline_handler ] [VALIDATOR valfunction]
```

更多信息参阅 CREATE LANGUAGE。

创建操作符

定义一个新的操作符。

```
CREATE OPERATOR name (
       PROCEDURE = funcname
       [, LEFTARG = lefttype] [, RIGHTARG = righttype]
       [, COMMUTATOR = com_op] [, NEGATOR = neg_op]
       [, RESTRICT = res_proc] [, JOIN = join_proc]
       [, HASHES] [, MERGES]
       [, SORT1 = left_sort_op] [, SORT2 = right_sort_op]
       [, LTCMP = less_than_op] [, GTCMP = greater_than_op] )
```

更多信息参阅 CREATE OPERATOR。

创建操作符类

定义一个新的操作符类

```
CREATE OPERATOR CLASS name [DEFAULT] FOR TYPE data_type
  USING index_method AS
  {
  OPERATOR strategy_number op_name [(op_type, op_type)] [RECHECK]
  | FUNCTION support_number funcname (argument_type [, ...] )
  | STORAGE storage_type
  } [, ... ]
```

更多信息参阅 CREATE OPERATOR CLASS。

创建操作符族

定义一个新的操作符族

```
CREATE OPERATOR FAMILY name  USING index_method
```

更多信息参阅 CREATE OPERATOR FAMILY。

创建协议

注册自定义数据访问协议，当定义 HashData 数据库外部表时可以指定。

```
CREATE [TRUSTED] PROTOCOL name (
   [readfunc='read_call_handler'] [, writefunc='write_call_handler']
   [, validatorfunc='validate_handler' ])
```

更多信息参阅 CREATE PROTOCOL。

创建资源队列

定义一个新的资源队列。

```
CREATE RESOURCE QUEUE name WITH (queue_attribute=value [, ... ])
```

更多信息参阅 CREATE RESOURCE QUEUE。

创建角色

定义一个新的数据库角色（用户或组）。

```
CREATE ROLE name [[WITH] option [ ... ]]
```

更多信息参阅 CREATE ROLE。

创建规则

定义一个新的重写规则

```
CREATE [OR REPLACE] RULE name AS ON event
  TO table [WHERE condition]
  DO [ALSO | INSTEAD] { NOTHING | command | (command; command
  ...) }
```

更多信息参阅 CREATE RULE。

创建模式

定义一个新的模式。

```
CREATE SCHEMA schema_name [AUTHORIZATION username]
   [schema_element [ ... ]]

CREATE SCHEMA AUTHORIZATION rolename [schema_element [ ... ]]
```

更多信息参阅 CREATE SCHEMA。

创建序列

定义一个新的序列生成器。

```
CREATE [TEMPORARY | TEMP] SEQUENCE name
       [INCREMENT [BY] value]
       [MINVALUE minvalue | NO MINVALUE]
       [MAXVALUE maxvalue | NO MAXVALUE]
       [START [ WITH ] start]
       [CACHE cache]
       [[NO] CYCLE]
       [OWNED BY { table.column | NONE }]
```

更多信息参阅 CREATE SEQUENCE。

创建表

定义一个新的表。

```
CREATE [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name (
[ { column_name data_type [ DEFAULT default_expr ]
   [column_constraint [ ... ]
[ ENCODING ( storage_directive [,...] ) ]
]
   | table_constraint
   | LIKE other_table [{INCLUDING | EXCLUDING}
                      {DEFAULTS | CONSTRAINTS}] ...}
   [, ... ] ]
   )
   [ 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
           [(...)]
        ) ]
    )
```

更多信息参阅 CREATE TABLE。

创建表如（AS）

从查询的结果中定义一个新的表。

```
CREATE [ [GLOBAL | LOCAL] {TEMPORARY | TEMP} ] TABLE table_name
   [(column_name [, ...] )]
   [ WITH ( storage_parameter=value [, ... ] ) ]
   [ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP}]
   [TABLESPACE tablespace]
   AS query
   [DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY]
```

更多信息参阅 CREATE TABLE AS。

创建表空间

定义一个新的表空间。

```
CREATE TABLESPACE tablespace_name [OWNER username]
       FILESPACE filespace_name
```

更多信息参阅 CREATE TABLESPACE。

创建类型

定义一个新的类型。

```
CREATE TYPE name AS ( attribute_name data_type [, ... ] )

CREATE TYPE name AS ENUM ( 'label' [, ... ] )

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [, RECEIVE = receive_function]
    [, SEND = send_function]
    [, TYPMOD_IN = type_modifier_input_function ]
    [, TYPMOD_OUT = type_modifier_output_function ]
    [, INTERNALLENGTH = {internallength | VARIABLE}]
    [, PASSEDBYVALUE]
    [, ALIGNMENT = alignment]
    [, STORAGE = storage]
    [, DEFAULT = default]
    [, ELEMENT = element]
    [, DELIMITER = delimiter] )

CREATE TYPE name
```

更多信息参阅 CREATE TYPE。

创建用户

定义一个默认带有 LOGIN 权限的数据库角色。

```
CREATE USER name [ [WITH] option [ ... ] ]
```

更多信息参阅 CREATE USER。

创建视图

定义一个新的视图

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

更多信息参阅 CREATE VIEW。

取消分配

取消分配一个已经准备（预编译）的语句

```
DEALLOCATE [PREPARE] name
```

更多信息参阅 DEALLOCATE。

声明

定义一个游标。

```
DECLARE name [BINARY] [INSENSITIVE] [NO SCROLL] CURSOR
    [{WITH | WITHOUT} HOLD]
    FOR query [FOR READ ONLY]
```

更多信息参阅 DECLARE。

函数

从表中删除行。

```
DELETE FROM [ONLY] table [[AS] alias]
     [USING usinglist]
     [WHERE condition | WHERE CURRENT OF cursor_name ]
```

更多信息参阅 DELETE。

丢弃

丢弃会话的状态。

```
DISCARD { ALL | PLANS | TEMPORARY | TEMP }
```

更多信息参阅 DISCARD。

删除聚集函数

删除聚集函数。

```
DROP AGGREGATE [IF EXISTS] name ( type [, ...] ) [CASCADE | RESTRICT]
```

更多信息参阅 DROP AGGREGATE。

（做）DO

执行匿名代码块作为暂时匿名函数。

```
DO [ LANGUAGE lang_name ] code
```

更多信息参阅 DO。

删除投影

删除一个投影。

```
DROP CAST [IF EXISTS] (sourcetype AS targettype) [CASCADE | RESTRICT]
```

更多信息参阅 DROP CAST。

删除转换

删除一个转换。

```
DROP CONVERSION [IF EXISTS] name [CASCADE | RESTRICT]
```

更多信息参阅 DROP CONVERSION。

删除数据库

删除一个数据库。

```
DROP DATABASE [IF EXISTS] name
```

更多信息参阅 DROP DATABASE。

删除域

删除一个域。

```
DROP DOMAIN [IF EXISTS] name [, ...]  [CASCADE | RESTRICT]
```

更多信息参阅 DROP DOMAIN。

删除扩展

从 HashData 数据库中删除一个扩展。

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

更多信息参阅 DROP EXTENSION。

删除外部表

删除一个外部表定义。

```
DROP EXTERNAL [WEB] TABLE [IF EXISTS] name [CASCADE | RESTRICT]
```

更多信息参阅 DROP EXTERNAL TABLE。

删除文件空间

删除一个文件空间。

```
DROP FILESPACE [IF EXISTS] filespacename
```

更多信息参阅 DROP FILESPACE。

删除函数

删除一个函数。

```
DROP FUNCTION [IF EXISTS] name ( [ [argmode] [argname] argtype
    [, ...] ] ) [CASCADE | RESTRICT]
```

更多信息参阅 DROP FUNCTION。

删除组

删除一个数据库角色。

```
DROP GROUP [IF EXISTS] name [, ...]
```

更多信息参阅 DROP GROUP。

删除索引

删除一个索引。

```
DROP INDEX [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息参阅 DROP INDEX。

删除语言

删除一个程序语言。

```
DROP [PROCEDURAL] LANGUAGE [IF EXISTS] name [CASCADE | RESTRICT]
```

更多信息参阅 DROP LANGUAGE。

删除操作符

删除一个操作符。

```
DROP OPERATOR [IF EXISTS] name ( {lefttype | NONE} ,
    {righttype | NONE} ) [CASCADE | RESTRICT]
```

更多信息参阅 DROP OPERATOR。

删除操作符类

删除一个操作符类。

```
DROP OPERATOR CLASS [IF EXISTS] name USING index_method [CASCADE | RESTRICT]
```

更多信息参阅 DROP OPERATOR CLASS。

删除操作符族

删除一个操作符族。

```
DROP OPERATOR FAMILY [IF EXISTS] name USING index_method [CASCADE | RESTRICT]
```

更多信息参阅 DROP OPERATOR FAMILY。

删除拥有（owned）

珊瑚数据库角色所拥有的数据库对象。

```
DROP OWNED BY name [, ...] [CASCADE | RESTRICT]
```

更多信息参阅 DROP OWNED。

删除协议

从数据库中删除外部表的访问协议。

```
DROP PROTOCOL [IF EXISTS] name
```

更多信息参阅 DROP PROTOCOL。

删除资源队列

删除一个资源队列。

```
DROP RESOURCE QUEUE queue_name
```

更多信息参阅 DROP RESOURCE QUEUE。

删除角色

删除一个数据库角色。

```
DROP ROLE [IF EXISTS] name [, ...]
```

更多信息参阅 DROP ROLE。

删除规则

删除一个重写规则。

```
DROP RULE [IF EXISTS] name ON relation [CASCADE | RESTRICT]
```

更多信息参阅 DROP RULE。

删除模式

删除一个模式。

```
DROP SCHEMA [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息参阅 DROP SCHEMA。

删除序列

删除一个序列

```
DROP SEQUENCE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息参阅 DROP SEQUENCE。

删除表

删除一个表。

```
DROP TABLE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息参阅 DROP TABLE。

删除表空间

删除一个表空间

```
DROP TABLESPACE [IF EXISTS] tablespacename
```

更多信息参阅 DROP TABLESPACE。

删除类型

删除一个数据类型

```
DROP TYPE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息参阅 DROP TYPE。

删除用户

删除一个数据库角色

```
DROP USER [IF EXISTS] name [, ...]
```

更多信息参阅 DROP USER。

删除视图

删除一个视图

```
DROP VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

更多信息参阅 DROP VIEW。

结束（END）

提交当前事务

```
END [WORK | TRANSACTION]
```

更多信息参阅 END。

执行

执行一个已经准备好的SQL语句。

```
EXECUTE name [ (parameter [, ...] ) ]
```

更多信息参阅 EXECUTE。

解释

展示语句的查询计划。

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

更多信息参阅 EXPLAIN。

提取

使用游标获取查询结果的行。

```
FETCH [ forward_direction { FROM | IN } ] cursorname
```

更多信息参阅 FETCH 。

授权

定义一个访问权限

```
GRANT { {SELECT | INSERT | UPDATE | DELETE | REFERENCES |
TRIGGER | TRUNCATE } [,...] | ALL [PRIVILEGES] }
    ON [TABLE] tablename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {USAGE | SELECT | UPDATE} [,...] | ALL [PRIVILEGES] }
    ON SEQUENCE sequencename [, ...]
    TO { rolename | PUBLIC } [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | CONNECT | TEMPORARY | TEMP} [,...] | ALL
[PRIVILEGES] }
    ON DATABASE dbname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { EXECUTE | ALL [PRIVILEGES] }
    ON FUNCTION funcname ( [ [argmode] [argname] argtype [, ...]
] ) [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { USAGE | ALL [PRIVILEGES] }
    ON LANGUAGE langname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | USAGE} [,...] | ALL [PRIVILEGES] }
    ON SCHEMA schemaname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { CREATE | ALL [PRIVILEGES] }
    ON TABLESPACE tablespacename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT parent_role [, ...]
    TO member_role [, ...] [WITH ADMIN OPTION]

GRANT { SELECT | INSERT | ALL [PRIVILEGES] }
    ON PROTOCOL protocolname
    TO username
```

更多信息参阅 GRANT 。

插入

在表中创建新的行

```
INSERT INTO table [( column [, ...] )]
   {DEFAULT VALUES | VALUES ( {expression | DEFAULT} [, ...] )
   [, ...] | query}
```

更多信息参阅 INSERT。

加载

加载或重新加载共享库文件

```
LOAD 'filename'
```

更多信息参阅 LOAD 。

锁

-

锁住一张表

```
LOCK [TABLE] name [, ...] [IN lockmode MODE] [NOWAIT]
```

更多信息参阅 LOCK 。

移动

放置一个游标

```
MOVE [ forward_direction {FROM | IN} ] cursorname
```

更多信息参阅 MOVE 。

准备

准备一个执行的语句

```
PREPARE name [ (datatype [, ...] ) ] AS statement
```

更多信息参阅 PREPARE。

重新分配拥有

改变数据库角色所拥有的数据库对象的所有权。

```
REASSIGN OWNED BY old_role [, ...] TO new_role
```

更多信息参阅 REASSIGN OWNED。

重新索引

重新构建索引

```
REINDEX {INDEX | TABLE | DATABASE | SYSTEM} name
```

更多信息参阅 REINDEX。

释放 SAVEPOINT

销毁一个之前定义过的 savepoint。

```
RELEASE [SAVEPOINT] savepoint_name
```

更多信息参阅 RELEASE SAVEPOINT。

重置

恢复系统配置参数的值为默认值。

```
RESET configuration_parameter

RESET ALL
```

更多信息参阅 RESET 。

撤销

撤销访问权限

```
REVOKE [GRANT OPTION FOR] { {SELECT | INSERT | UPDATE | DELETE
       | REFERENCES | TRIGGER | TRUNCATE } [,...] | ALL [PRIVILEGES] }
       ON [TABLE] tablename [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {USAGE | SELECT | UPDATE} [,...]
       | ALL [PRIVILEGES] }
       ON SEQUENCE sequencename [, ...]
       FROM { rolename | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {CREATE | CONNECT
       | TEMPORARY | TEMP} [,...] | ALL [PRIVILEGES] }
       ON DATABASE dbname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {EXECUTE | ALL [PRIVILEGES]}
       ON FUNCTION funcname ( [[argmode] [argname] argtype
                              [, ...]] ) [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {USAGE | ALL [PRIVILEGES]}
       ON LANGUAGE langname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [GRANT OPTION FOR] { {CREATE | USAGE} [,...]
       | ALL [PRIVILEGES] }
       ON SCHEMA schemaname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { CREATE | ALL [PRIVILEGES] }
       ON TABLESPACE tablespacename [, ...]
       FROM { rolename | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [ADMIN OPTION FOR] parent_role [, ...]
       FROM member_role [, ...]
       [CASCADE | RESTRICT]
```

更多信息参阅 REVOKE。

回滚

中止当前事务

```
ROLLBACK [WORK | TRANSACTION]
```

更多信息参阅 ROLLBACK。

回滚到 SAVEPOINT

将当前事务回滚到某个 savepoint。

```
ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] savepoint_name
```

更多信息参阅 ROLLBACK TO SAVEPOINT。

SAVEPOINT

在当前事务定义一个新的 savepoint。

```
SAVEPOINT savepoint_name
```

更多信息参阅 SAVEPOINT。

选择（SELECT）

从表或者视图中检索行。

```
[ WITH with_query [, ...] ]
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] [NULLS {FIRST | LAST}] [, ...]]
  [LIMIT {count | ALL}]
  [OFFSET start]
  [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT] [...]]
```

更多信息参阅 SELECT。

选择到（SELECT INTO）

从查询结果中定义一个新的表。

```
[ WITH with_query [, ...] ]
SELECT [ALL | DISTINCT [ON ( expression [, ...] )]]
    * | expression [AS output_name] [, ...]
    INTO [TEMPORARY | TEMP] [TABLE] new_table
    [FROM from_item [, ...]]
    [WHERE condition]
    [GROUP BY expression [, ...]]
    [HAVING condition [, ...]]
    [{UNION | INTERSECT | EXCEPT} [ALL] select]
    [ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}] [, ...]]
    [LIMIT {count | ALL}]
    [OFFSET start]
    [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT]
    [...]]
```

更多信息参阅 SELECT INTO。

设置

改变 HashData 数据库配置参数的值。

```
SET [SESSION | LOCAL] configuration_parameter {TO | =} value |
    'value' | DEFAULT}

SET [SESSION | LOCAL] TIME ZONE {timezone | LOCAL | DEFAULT}
```

更多信息参阅 SET。

设置角色

设置当前会话当前角色的标识符。

```
SET [SESSION | LOCAL] ROLE rolename

SET [SESSION | LOCAL] ROLE NONE

RESET ROLE
```

更多信息参阅 SET ROLE。

设置会话授权

设置会话角色标识符和当前会话当前角色的标识符。

```
SET [SESSION | LOCAL] SESSION AUTHORIZATION rolename

SET [SESSION | LOCAL] SESSION AUTHORIZATION DEFAULT

RESET SESSION AUTHORIZATION
```

更多信息参阅 SET SESSION AUTHORIZATION。

设置事务（SET TRANSACTION）

设置当前事务的特征。

```
SET TRANSACTION [transaction_mode] [READ ONLY | READ WRITE]

SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode
     [READ ONLY | READ WRITE]
```

更多信息参阅 SET TRANSACTION。

显示（SHOW）

显示当前系统配置参数的值。

```
SHOW configuration_parameter

SHOW ALL
```

更多信息参阅 SHOW 。

开始事务

开始一个事务块。

```
START TRANSACTION [SERIALIZABLE | READ COMMITTED | READ UNCOMMITTED]
                  [READ WRITE | READ ONLY]
```

更多信息参阅 START TRANSACTION。

截断（TRUNCATE）

清空表的所有行。

```
TRUNCATE [TABLE] name [, ...] [CASCADE | RESTRICT]
```

更多信息参阅 TRUNCATE。

更新

更新表的行。

```
UPDATE [ONLY] table [[AS] alias]
   SET {column = {expression | DEFAULT} |
   (column [, ...]) = ({expression | DEFAULT} [, ...])} [, ...]
   [FROM fromlist]
   [WHERE condition | WHERE CURRENT OF cursor_name ]
```

更多信息参阅 UPDATE。

清理

垃圾收集和选择性分析数据库。

```
VACUUM [FULL] [FREEZE] [VERBOSE] [table]

VACUUM [FULL] [FREEZE] [VERBOSE] ANALYZE
              [table [(column [, ...] )]]
```

更多信息参阅 VACUUM。

值

计算一组行。

```
VALUES ( expression [, ...] ) [, ...]
   [ORDER BY sort_expression [ASC | DESC | USING operator] [, ...]]
   [LIMIT {count | ALL}] [OFFSET start]
```

更多信息参阅 VALUES。

上级话题： SQL命令参考

ABORT

终止当前事务。TEST

概要

```
ABORT [WORK | TRANSACTION]
```

描述

ABORT 当前事务并导致事务所做的所有更新都被丢弃。出于历史原因，此命令与标准 SQL 命令的行为相同。

参数

WORK

TRANSACTION

可选关键字，它们没有效果。

注解

使用 COMMIT 成功终止一个事务。

在一个事务块之外发出 ABORT 会发出一个警告信息并且不会产生效果。

兼容性

由于历史原因，此命令是 HashData 数据库扩展名。 ROLLBACK 是等效的标准 SQL 命令。ROLLBACK 是等效的标准 SQL 命令。

另见

BEGIN，COMMIT，ROLLBACK

上级主题：SQL命令参考

ALTER AGGREGATE

更改聚集函数的定义

概要

```
ALTER AGGREGATE name ( type [ , ... ] ) RENAME TO new_name
ALTER AGGREGATE name ( type [ , ... ] ) OWNER TO new_owner
ALTER AGGREGATE name ( type [ , ... ] ) SET SCHEMA new_schema
```

描述

ALTER AGGREGATE 更改聚集函数的定义。

用户必须拥有聚集函数才能去使用 ALTER AGGREGATE。 要更改聚合函数的模式，还必须对新模式具有CREATE 特权。要更改所有者，用户还必须是新任命的直接或间接成员，并且该角色必须对聚合函数的模式具有 CREATE 特权。(这些限制强制要求拥有者不能通过丢弃并重建该聚集函数来做任何不能做的事情。然而，超级用户可以改变任何聚合函数的所有权。)

参数

name

一个现有聚集函数的名称（可以是限定模式）

type

聚集函数能运行的输入数据类型。要引用零参数聚合函数，写入*代替输入数据类型的列表。

new_name

聚集函数的新名称。

new_owner

聚集函数新的所有者。

new_schema

聚集函数的新模式。

示例

要把 integer 类型的聚合函数 myavg 重命名为 my_average:

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

更改 integer 类型的聚合函数 myavg 的所有者为 joe:

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

将 integer 类型的聚合函数 myavg 移动到模式 myschema 中:

```
ALTER AGGREGATE myavg(integer) SET SCHEMA myschema;
```

兼容性

在 SQL 标准中没有 ALTER AGGREGATE 语句。

另见

CREATE AGGREGATE，DROP AGGREGATE

上级主题： SQL命令参考

ALTER CONVERSION

更改一个转换的定义。

概要

```
ALTER CONVERSION name RENAME TO newname
ALTER CONVERSION name OWNER TO newowner
```

描述

ALTER CONVERSION 更改一个转换的定义。

用户必须拥有使用 ALTER CONVERSION 的转换权限。 要更改所有者，用户还必须是新任命的直接或间接成员，并且该角色必须对转换的模式具有 CREATE 权限。 (这些限制强制要求拥有者不能通过删除和重新创建转换来组做任何用户不能做的事情，但超级用户可以改变任何转换的所有权。)

参数

name

现有转换的名称（可选方案限定）。

newname

新的转换名称。

newowner

转换的新所有者。

示例

要把转换 iso_8859_1_to_utf8 重命名为 latin1_to_unicode：

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

更改转换iso_8859_1_to_utf8 的所有者为 joe:

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

兼容性

在 SQL 标准中没有 ALTER CONVERSION 语句。

另见

CREATE CONVERSION，DROP CONVERSION

上级主题： SQL命令参考

ALTER DATABASE

更改数据库的属性

概要

```
ALTER DATABASE name [ WITH CONNECTION LIMIT connlimit ]

ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }

ALTER DATABASE name RESET parameter

ALTER DATABASE name RENAME TO newname

ALTER DATABASE name OWNER TO new_owner
```

描述

ALTER DATABASE 更改一个数据库的属性。

第一种形式更改数据库的允许连接限制。只有数据库所有者或超级用户可以更改此设置。

第二种和第三种形式更改了 HashData 数据库的配置参数的会话默认值。每当随后在该数据库中启动新会话时，指定的值将成为会话默认值。数据库中指定的参数会默认覆盖在服务器配置文件 postgresql.conf 中的参数值。只有数据库所有者或超级用户可以更改数据库的会话默认值。某些参数不能以这种方式设置，或只能由超级用户设置。

第四种形式更改数据库的名称。只有数据库所有者或超级用户可以重命名数据库; 非超级用户也必须具有 CREATEDB 特权。用户不能重命名当前登录的数据库。重命名之前需要首先连接到不同的数据库。

第五种形式更改数据库的所有者。要更改所有者，用户必须拥有数据库，并且也是新拥有角色的直接或间接成员， 并且必须具有 CREATEDB 特权. (请注意，超级用户自动拥有所有这些权限。)

参数

name

要更改其属性的数据库的名称。

connlimit

最大可能的并发连接数。 缺省值 -1 表示没有限制。

parameter value

将指定配置参数的数据库的会话默认值设置为给定值。如果值是 DEFAULT 或者, 等效使用 RESET 则删除数据库特定的设置，因此系统范围的默认设置将在新会话中继承。 使用 RESET ALL 清除所有特定于数据库的设置。

newname

数据库的新名称

new_owner

数据库的新所有者.

注意

还可以为特定角色（用户）而不是数据库设置配置参数会话默认值。如果存在冲突，角色特定的设置将覆盖数据库特定的设置。 参见 ALTER ROLE.

示例

设置默认方案的搜索路径为 mydatabase 数据库:

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

兼容性

ALTER DATABASE 语句是一个 HashData 数据库的扩展

另见

CREATE DATABASE，DROP DATABASE，SET

上级主题： SQL命令参考

ALTER DOMAIN

更改现有域的定义。

概要

```
ALTER DOMAIN name { SET DEFAULT expression | DROP DEFAULT }

ALTER DOMAIN name { SET | DROP } NOT NULL

ALTER DOMAIN name ADD domain_constraint

ALTER DOMAIN name DROP CONSTRAINT constraint_name \[RESTRICT | CASCADE\]

ALTER DOMAIN name OWNER TO new_owner

ALTER DOMAIN name SET SCHEMA new_schema
```

描述

ALTER DOMAIN 更改一个现有域的定义。 有几种形式：

- **SET/DROP DEFAULT** — 这些形式设置或删除域的默认值。请注意，默认值仅适用于后续的INSERT 命令。 它们不影响使用域的表中已经存在的行。
- **SET/DROP NOT NULL** — 这些形式会改变域是否被标记为允许 NULL 值或者拒绝 NULL 值。 用户只能 SET NOT NULL 当使用域的列不包含空值时。
- **ADD domain_constraint** — 这种形式使用和 CREATE DOMAIN 相同的语法为域增加一个新的约束。如果所有使用域的列满足新的约束，则添加成功。
- **DROP CONSTRAINT** — 此形式删除域的约束。
- **OWNER** — 此形式将域的所有者更改为指定的用户。
- **SET SCHEMA** — 此形式更改域的模式。与域相关联的任何约束也被移动到新的模式中。

用户必须拥有域才能 ALTER DOMAIN. 要更改域的模式，用户还必须对新模式具有 CREATE 特权 要更改所有者，用户还必须是新拥有角色的直接或间接成员，并且该角色必须对该域的模式具有 CREATE 特权。 这些限制强制修改拥有者不能做一些通过删除和重建域做不到的事情。不过，一个超级用户怎么都能更改任何域的所有权。

参数

name

要更改的现有域的名称（可选方案限定）。

domain_constraint

域的新域约束。

constraint_name

要删除的现有约束的名称。

CASCADE

自动删除依赖约束的对象。

RESTRICT

如果有任何依赖对象，拒绝删除约束。这是默认行为。

new_owner

域的新所有者的用户名

new_schema

域的新模式。

示例

添加 NOT NULL 约束到一个域:

```
ALTER DOMAIN zipcode SET NOT NULL;
```

删除NOT NULL 约束从一个域中:

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

向域添加检查约束：

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

从域中删除检查约束：

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

将域移动到不同的模式：

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

兼容性

ALTER DOMAIN 符合 SQL 标准，除了OWNER 和 SET SCHEMA 变型, 它们是 HashData 数据库扩展。

另见

CREATE DOMAIN，DROP DOMAIN

上级主题： SQL命令参考

ALTER EXTENSION

更改在 HashData 数据库中注册的扩展的定义

概要

```
ALTER EXTENSION name UPDATE [ TO new_version ]
ALTER EXTENSION name SET SCHEMA new_schema
ALTER EXTENSION name ADD member_object
ALTER EXTENSION name DROP member_object
```

其中 member_object是：

```
  ACCESS METHOD object_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  CONVERSION object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name ( \[ \[ argmode \] \[ argname \] argtype \[, ...\] \] ) |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  \[ PROCEDURAL \] LANGUAGE object_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  TABLE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TYPE object_name |
  VIEW object_name
```

aggregate_signature 是：

```
\* | \[ argmode \] \[ argname \] argtype \[ , ... \] |
  \[ \[ argmode \] \[ argname \] argtype \[ , ... \] \]
    ORDER BY \[ argmode \] \[ argname \] argtype \[ , ... \]
```

描述

ALTER EXTENSION 更改已安装扩展的定义。 有几种子形式:

UPDATE

此形式将扩展更新到一个较新版本。该扩展必须提供一个合适的更新脚本（或一系列脚本），可以将当前安装的版本修改为所要求的版本。

SET SCHEMA

这种形式将扩展对象移动到另一个模式中。 扩展名必须 是 可重定位。

ADD member_object

这种形式将一个现有对象添加到该扩展中。这在扩展更新脚本中很有用。 该对象后续将被当作该扩展的一个成员。尤其是该

对象只有通过删除扩展 才能删除。

DROP member_object

这种形式从扩展中删除一个成员对象。这这主要对扩展更新脚本有用。只有 撤销该对象与其扩展之间的关联后才能删除该对象。

参阅把相关对象打包到扩展中获取更多关于这些操作的信息。

用户必须拥有扩展功能对于 ALTER EXTENSION. ADD 和 DROP 形式还要求被增加/删除对象的所有权。

参数

name

T一个已安装扩展的名称

new_version

新版本的扩展。 new_version 可以是标识符或字符串文字。如果未指定，该命令将尝试更新到扩展控制文件中的默认版本。

new_schema

扩展的新模式

object_name

aggregate_name

function_name

operator_name

要从该扩展增加或者移除的对象的名称。表、聚集、域、外部表、函数、 操作符、操作符类、操作符族、序列、文本搜索对象、类型和视图的名称 可以被方案限定。

source_type

该转换的源数据类型的名称。

target_type

cast的目标数据类型的名称。

argmode

函数或聚集参数的模式：IN, OUT, INOUT, or VARIADIC. 默认为 IN.

该命令忽略了 OUT 参数。 只需要输入参数来确定函数的身份。列出 IN, INOUT, and VARIADIC 参数就足够了。

argname

函数或聚集参数的名称。

该命令忽略参数名称，因为只需要参数数据类型来确定函数标识。

argtype

函数或聚集参数的数据类型。

left_type

right_type

操作符参数的数据类型（可以是方案限定）。 指定 NONE 对于一个前缀或后缀操作符的缺失的参数.

PROCEDURAL

这是一个噪声词。

type_name

该转换的数据类型的名称。

lang_name

该转换的语言的名称。

示例

要将hstore扩展更新为2.0版本：

ALTER EXTENSION hstore UPDATE TO '2.0';

更改 hstore 的扩展模式为 utils:

ALTER EXTENSION hstore SET SCHEMA utils;

要将现有函数添加到 hstore 扩展中:

ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);

兼容性

ALTER EXTENSION 是一个 HashData 数据库扩展。

参考

CREATE EXTENSION, DROP EXTENSION

上级主题： SQL命令参考

ALTER EXTERNAL TABLE

更改外部表的定义。

概要

```
ALTER EXTERNAL TABLE name RENAME [COLUMN] column TO new_column

ALTER EXTERNAL TABLE name RENAME TO new_name

ALTER EXTERNAL TABLE name SET SCHEMA new_schema

ALTER EXTERNAL TABLE name action [, ... ]
```

其中 action 是下列之一：

```
  ADD [COLUMN] new_column type
  DROP [COLUMN] column [RESTRICT|CASCADE]
  ALTER [COLUMN] column TYPE type [USING expression]
  OWNER TO new_owner
```

描述

ALTER EXTERNAL TABLE 更改一个现有外部表的定义。有几种子形式：

- **ADD COLUMN** —向外部表定义添加一个新列。
- **DROP COLUMN** — 从外部表定义中删除一列。请注意，如果用户删除可读的外部表列，它只会更改 HashData 数据库中的表定义。外部数据文件不会更改。如果需要删除任何表外部的依赖于此列的数据，比如一个指向此列的视图，则需要指定 CASCADE 关键字。
- **ALTER COLUMN TYPE** — 更改表的列的数据类型。
- **OWNER** — 将外部表的所有者更改为指定的用户。
- **RENAME** — 更改外部表的名称或表中单个列的名称。对外部数据没有影响。
- **SET SCHEMA** —将外部表移动到另一个模式。

用户必须拥有外部表才能使用 ALTER EXTERNAL TABLE。 要更改外部表的模式，用户还必须对新模式具有 CREATE 权限。要更改所有者，用户还必须是新拥有角色的直接或间接成员，该角色必须对外部表的模式具有 CREATE 特权。超级用户自动拥有这些权限。

在此版本中， ALTER EXTERNAL TABLE 不能修改外部表类型（read, write, web），数据格式或外部数据的位置。 要修改此信息，用户必须删除并重新创建外部表定义。

参数

name

要修改的现有外部表定义的名称（可以是方案限定）。

column

新列或现有列的名称。

new_column

现有列的新名称。

new_name

外部表的新名称。

type

新列的数据类型或现有列的新数据类型。

new_owner

外部表的新所有者的角色名称。

new_schema

该表要被移动到其中的模式的名称。

示例

向外部表定义添加新列：

```
ALTER EXTERNAL TABLE ext_expenses ADD COLUMN manager text;
```

更改外部表的名称：

```
ALTER EXTERNAL TABLE ext_data RENAME TO ext_sales_data;
```

更改外部表的所有者：

```
ALTER EXTERNAL TABLE ext_data OWNER TO jojo;
```

更改外部表的模式：

```
ALTER EXTERNAL TABLE ext_leads SET SCHEMA marketing;
```

兼容性

ALTER EXTERNAL TABLE 是 HashData 数据库的一个扩展。在标准 SQL 或常规 PostgreSQL 中没有 ALTER EXTERNAL TABLE 语句。

另见

CREATE EXTERNAL TABLE，DROP EXTERNAL TABLE

上级主题： SQL命令参考

ALTER FILESPACE

更改文件空间的定义。

概要

```
ALTER FILESPACE name RENAME TO newname

ALTER FILESPACE name OWNER TO newowner
```

描述

ALTER FILESPACE 更改一个文件空间的定义。

用户必须拥有文件空间才能使用 ALTER FILESPACE 命令。修改文件空间的所有者，用户还必须是新的角色的直接或间接成员(请注意，超级用户自动拥有这些权限）。

参数

name

现有文件空间的名称。

newname

文件空间的新名称。新名称不能以 pg_ 和 gp_ 开头。(为系统文件空间保留）。

newowner

文件空间的新所有者。

示例

重命名文件空间 myfs 为 fast_ssd:

```
ALTER FILESPACE myfs RENAME TO fast_ssd;
```

更改表空间 myfs 的拥有者：

```
ALTER FILESPACE myfs OWNER TO dba;
```

兼容性

在标准 SQL 或 PostgreSQL 中没有 ALTER FILESPACE 语句

另见

DROP FILESPACE、gpfilespace

上级主题： SQL命令参考

ALTER FUNCTION

更改一个函数的定义。

概要

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] ) action [, ... ] [RESTRICT]
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] ) RENAME TO new_name
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] ) OWNER TO new_owner
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] ) SET SCHEMA new_schema
```

其中 action 是下列之一：

```
{CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
{IMMUTABLE | STABLE | VOLATILE}
{[EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER}
```

描述

ALTER FUNCTION 更改一个函数的定义。

用户必须拥有该函数以使用 ALTER FUNCTION。要更改函数的模式，用户还必须对新模式具有 CREATE 特权。要更改所有者，用户还必须是新拥有角色的直接或间接成员，并且该角色必须对该函数的模式具有 CREATE 特权。(这些限制强制修改拥有者不能做一些通过删除和重建该函数做不到的事情。但是，超级用户可以改变任何函数的所有权。)

参数

name

现有函数的名称（可选方案限定）。

argmode

参数的模式： IN， OUT，INOUT。 如果省略，默认值为 IN。 请注意 ALTER FUNCTION 实际上并不关注 OUT 参数，因为只需要输入参数来确定函数的身份。 因此对于只列出 IN, INOUT 参数已经足够了。

argname

参数的名称。请注意，ALTER FUNCTION 实际上并不关心参数名称，因为只需要参数数据类型来确定函数的身份。

argtype

函数参数（如果有）的数据类型（可以是方案限定）。

new_name

函数的新名称。

new_owner

函数的新拥有者。请注意，如果函数被标记为 SECURITY DEFINER, 随后它将作为新的所有者执行。

new_schema

该函数的新模式。

CALLED ON NULL INPUT

RETURNS NULL ON NULL INPUT

STRICT

CALLED ON NULL INPUT 将该函数改为在某些或者全部参数为空值时可以被调用。 RETURNS NULL ON NULL INPUT 或者 STRICT 更改函数，以便如果其任何参数为空，则不会调用该函数; 而是自动假设一个空的结果。 参阅 CREATE FUNCTION 获取更多信息。

IMMUTABLE

STABLE

VOLATILE

将函数的波动性改为指定的设置。参阅 CREATE FUNCTION 以便问题得以解决。

[EXTERNAL] SECURITY INVOKER

[EXTERNAL] SECURITY DEFINER

更改该函数是否为一个安全性定义者。 关键词 EXTERNAL 为了 SQL 的一致性而被忽略。 参阅 CREATE FUNCTION 获取更多有关此功能的信息。

RESTRICT

忽略 SQL 标准。

注意

HashData 数据库对某些定义的函数有 STABLE 或者 VOLATILE 这样的限制。参阅 CREATE FUNCTION 获取更多信息。

示例

将 integer 类型的函数 sqrt 重命名为 square_root:

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

更改 integer 类型的 sqrt 函数的所有者为 joe:

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

更改 integer 类型的函数 sqrt 的模式为math:

```
ALTER FUNCTION sqrt(integer) SET SCHEMA math;
```

要调整一个函数的自动搜索路径:

```
ALTER FUNCTION check_password(text) RESET search_path;
```

兼容性

这个语句部分兼容 SQL 标准中的 ALTER FUNCTION 语句。该标准允许修改一个函数的更多属性，但是不提供重命名一个函数、标记一个函数为安全性定义者、为一个函数附加配置参数值或者更改一个函数的拥有者、模式或者稳定性等功能。 该标准还需要 RESTRICT 关键字, 它在 HashData 数据库中是可选的。

另见

CREATE FUNCTION，DROP FUNCTION

上级主题： SQL命令参考

ALTER GROUP

更改角色名称或成员关系。

概要

```
ALTER GROUP groupname ADD USER username [, ... ]

ALTER GROUP groupname DROP USER username [, ... ]

ALTER GROUP groupname RENAME TO newname
```

描述

ALTER GROUP 该表用户组的属性。这是一个被废弃的命令，不过为了向后兼容还是被接受。 用户组（和用户）已被角色的更一般概念所取代。 参阅 ALTER ROLE 获取更多信息

参数

groupname

要修改的组（角色）的名称

username

要添加到组或从组中删除的用户（角色）。用户（角色）必须已经存在。

newname

组（角色）的新名称。

示例

要将用户添加到组中：

```
ALTER GROUP staff ADD USER karl, john;
```

从组中删除用户:

```
ALTER GROUP workers DROP USER beth;
```

兼容性

在 SQL 标准中没有 ALTER GROUP 语句。

另见

ALTER ROLE ，GRANT，REVOKE

上级主题： SQL命令参考

ALTER INDEX

更改一个索引的定义。

概要

```
ALTER INDEX name RENAME TO new_name

ALTER INDEX name SET TABLESPACE tablespace_name

ALTER INDEX name SET ( FILLFACTOR = value )

ALTER INDEX name RESET ( FILLFACTOR )
```

描述

ALTER INDEX 更改一个现有索引的定义。有几种子形式：

- **RENAME** — 更改索引的名称。对存储的数据没有影响
- **SET TABLESPACE** — 将索引的表空间更改为指定的表空间，并将与索引关联的数据文件移动到新的表空间。 另见 CREATE TABLESPACE。
- **SET FILLFACTOR** — 更改索引的索引方法特定的存储参数。内置索引方法都要接受一个独有参数： FILLFACTOR。 索引的 fillfactor 是一个百分比，用于确定索引方法将如何填充索引页。 此命令不会立即修改索引内容。 使用 REINDEX 重建索引以获得所需的效果。
- **RESET FILLFACTOR** — 将FILLFACTOR 重置为默认值。 与 SET 一样，可能需要 REINDEX 来完全更新索引。

参数

name

要修改的现有索引的名称（可选方案限定）。

new_name

索引的新名称。

tablespace_name

要移动索引的表空间。

FILLFACTOR

T 索引的填充因子是一个百分比，用于确定索引方法将如何填充索引页。对于 B 树，在初始索引编译期间，还会在右侧扩展索引（最大键值）时将页面填充到此百分比。如果页面随后变得完全充满，它们将被分割，导致索引效率逐渐下降。

B 树使用默认填充因子 90，但是可以选择从 10 到 100 的任何值。如果表是静态的，那么 fillfactor 为 100 对最小化索引的物理大小是最好的，但是对于需要大量更新的表，较小的填充因子对最小化页分割来说是有利的。 其他索引方法使用不同但大致相似的填充因子; 默认填充因子因方法而异。

注意

这些操作也可以使用 ALTER TABLE。

不允许更改系统目录索引的任何部分。

示例

重命名现有索引:

```
ALTER INDEX distributors RENAME TO suppliers;
```

将索引移动到不同的表空间:

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

要更改索引的填充因子（假设 index 方法支持它）:

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

兼容性

ALTER INDEX 是一个 HashData 数据库扩展。

另见

CREATE INDEX ，REINDEX，ALTER TABLE

上级主题： SQL命令参考

ALTER LANGUAGE

更改一种过程语言的定义。

概要

```
ALTER LANGUAGE name RENAME TO newname
```

描述

ALTER LANGUAGE 更改特定数据库的过程语言的定义。 用户必须是超级用户或语言的所有者才能使用 ALTER LANGUAGE。

参数

name

一种语言的名称。

newname

该语言的新名称。

兼容性

在 SQL 标准中没有 ALTER LANGUAGE 语句。

另见

CREATE LANGUAGE ，DROP LANGUAGE

上级主题： SQL命令参考

ALTER OPERATOR CLASS

更改一个操作符类的定义。

概要

```
ALTER OPERATOR CLASS name USING index_method RENAME TO newname

ALTER OPERATOR CLASS name USING index_method OWNER TO newowner
```

描述

ALTER OPERATOR CLASS 更改操作符类的定义。

用户必须拥有操作符类才能够使用 ALTER OPERATOR CLASS。要改变所有者，用户必须是新拥有角色的直接或间接成员，并且新角色在操作符类模式上必须有 CREATE 特权。（这些限制强制要求修改所有者不能做那些通过删除或重建操作符类都不能做到的事情。然而，超级用户可以随意修改操作符类的所有者）。

参数

name

一个存在的操作符的类的名称。

index_method

操作符类索引方法的名称。

newname

操作符类的新名称。

newowner

操作符类的新的拥有者。

兼容性

在 SQL 标准中没有 ALTER OPERATOR CLASS 语句。

另见

CREATE OPERATOR CLASS ，DROP OPERATOR CLASS

上级主题： SQL命令参考

ALTER OPERATOR

更改操作符的定义。

概要

```
ALTER OPERATOR name ( {lefttype | NONE} , {righttype | NONE} ) OWNER TO newowner
```

描述

ALTER OPERATOR 更改一个操作符的定义。 目前唯一可用的功能是更改操作符的所有者。

用户必须拥有操作符才能使用 ALTER OPERATOR。 要更改所有者，用户还必须是新拥有角色的一个直接或间接的成员，并且该角色必须具有该操作符所在模式上的 CREATE 特权。（（这种限制强制要求即使更改所有者也不能做那些通过删除或重建操作符所不能做到的事情。然而，超级用户可以任意修改操作符的所有权）

参数

name

现有操作符的名称（可选方案限定）

lefttype

操作符左操作数的数据类型；如果没有左操作数记为 NONE

righttype

操作符右操作数的数据类型；如果操作符没有右操作数记为 NONE

newowner

操作符新的所有者。

示例

更改 text 类型的一个自定义操作符 a @@ b 的所有者:

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

兼容性

在 SQL 标准中没有 ALTER OPERATOR 语句。

另见

CREATE OPERATOR ，DROP OPERATOR

上级主题： SQL命令参考

ALTER PROTOCOL

更改一个协议的定义

概要

```
ALTER PROTOCOL name RENAME TO newname

ALTER PROTOCOL name OWNER TO newowner
```

描述

ALTER PROTOCOL 更改一个协议的定义。只有协议的名称或拥有者可被更改。

用户必须拥有一个协议才能去使用 ALTER PROTOCOL。要更改所有者，用户必须是新拥有角色的直接或间接的成员，并且该角色必须在转换的模式是上具有 CREATE 特权。

这些限制适当的确保修改所有者只能通过删除或重建协议。注意一个超级用户可以修改任何协议的所属关系。

参数

name

现有协议的名称（可选方案限定）。

newname

协议的新名称。

newowner

协议的新的所有者。

示例

重命名转换 GPDBauth 为 GPDB_authentication:

```
ALTER PROTOCOL GPDBauth RENAME TO GPDB_authentication;
```

更改转换 GPDB_authentication 的所有者为 joe:

```
ALTER PROTOCOL GPDB_authentication OWNER TO joe;
```

兼容性

在 SQL 标准中没有 ALTER PROTOCOL 语句

另见

CREATE EXTERNAL TABLE ，CREATE PROTOCOL

上级主题： SQL命令参考

ALTER RESOURCE QUEUE

更改资源队列的限制。

概要

```
ALTER RESOURCE QUEUE name WITH ( queue_attribute=value [, ... ] )
```

其中 *queue_attribute* 是：

```
   ACTIVE_STATEMENTS=integer
   MEMORY_LIMIT='memory_units'
   MAX_COST=float
   COST_OVERCOMMIT={TRUE|FALSE}
   MIN_COST=float
   PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX}
```

```
ALTER RESOURCE QUEUE name WITHOUT ( queue_attribute [, ... ] )
```

其中 *queue_attribute* 是：

```
   ACTIVE_STATEMENTS
   MEMORY_LIMIT
   MAX_COST
   COST_OVERCOMMIT
   MIN_COST
```

> 注意： 资源队列必须有一个 ACTIVE_STATEMENTS 或一个 MAX_COST 值。不要从资源队列中删除这两个 queue_attributes 。

描述

ALTER RESOURCE QUEUE 更改资源队列的限制。只有超级用户可以更改资源队列。 资源队列必须有一个 ACTIVE_STATEMENTS 或一个 MAX_COST 值 (或者可以同时使用)。用户还可以设置或重置资源队列的优先级，以控制与队列相关联的查询使用的可用 CPU 资源的相对份额，或资源队列的内存限制，以控制通过队列提交的所有在段主机上查询消耗的内存量。

ALTER RESOURCE QUEUE WITHOUT 删除先前设置的资源的指定限制。资源队列必须有一个 ACTIVE_STATEMENTS 或一个 MAX_COST 值。 不要从资源队列中删除这两个 queue_attributes。

参数

name

要更改其限制的资源队列的名称。

ACTIVE_STATEMENTS *integer*

任何时刻系统中允许在该资源队列中的用户提交的活动语句的数量。 ACTIVE_STATEMENTS 的值应该是一个大于 0 的整数。要把 ACTIVE_STATEMENTS 重置为没有限制，输入一个 -1.0 值。

MEMORY_LIMIT *'memory_units'*

设置从此资源队列中的用户提交的所有语句的总内存配额。内存单位可以用 kB，MB 或 GB 指定。资源队列的最小内存配额

为 10MB。没有最大值。然而，查询执行时间的上边界受到段主机的物理内存的限制。默认值为无限制 (-1)。

MAX_COST *float*

任何时刻系统中允许在该资源队列中的用户提交的语句的查询优化器总代价。MAX_COST 的值被指定为一个浮点数（例如 100.00）或者还可以被指定为一个指数（例如 1e+2）。要把 MAX_COST 重置为没有限制，输入一个 -1.0 值。

COST_OVERCOMMIT *boolean*

如果资源队列受到基于查询代价的限制，那么管理员可以允许代价过量使用（COST_OVERCOMMIT=TRUE，默认）。这意味着一个超过允许的代价阈值的查询将被允许运行，但只能在系统空闲时运行。如果指定COST_OVERCOMMIT=FALSE，超过代价限制的查询将总是被拒绝并且绝不会被允许运行。

MIN_COST *float*

代价低于此限制的查询将不会排队而是立即运行。代价是以取得的磁盘页为单位来衡量的。1.0 等于一次顺序磁盘页面读取。MIN_COST 的值被指定为浮点数（例如 100.00）或者还能被指定为一个指数（例如 1e+2）。要把 MIN_COST 重置为没有限制，输入一个 -1.0 值。

PRIORITY = {MIN | LOW | MEDIUM | HIGH | MAX }

设置与资源队列关联的查询的优先级。具有较高优先级的队列中的查询或语句将在竞争中获得更多的可用 CPU 资源份额。低优先级队列中的查询可能会被延迟，同时执行更高优先级的查询。

注解

使用CREATE ROLE 或ALTER ROLE 将角色（用户）添加到资源队列中。

示例

更改资源队列的活动查询限制：

```
ALTER RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20);
```

更改资源队列的内存限制：

```
ALTER RESOURCE QUEUE myqueue WITH (MEMORY_LIMIT='2GB');
```

将资源队列的最大和最小查询代价限制重置为无限制：

```
ALTER RESOURCE QUEUE myqueue WITH (MAX_COST=-1.0, MIN_COST= -1.0);
```

将资源队列的查询代价限制重置为 3e+10 (or 30000000000.0) 不允许过量使用：

```
ALTER RESOURCE QUEUE myqueue WITH (MAX_COST=3e+10, COST_OVERCOMMIT=FALSE);
```

将与资源队列关联的查询的优先级重置为最小级别：

```
ALTER RESOURCE QUEUE myqueue WITH (PRIORITY=MIN);
```

去除 MAX_COST 和 MEMORY_LIMIT 资源队列限制：

```
ALTER RESOURCE QUEUE myqueue WITHOUT (MAX_COST, MEMORY_LIMIT);
```

兼容性

ALTER RESOURCE QUEUE 语句是一个 HashData 数据库扩展。此命令在标准 PostgreSQL 中不存在。

另见

CREATE RESOURCE QUEUE ，DROP RESOURCE QUEUE，CREATE ROLE ，ALTER ROLE

上级主题： SQL命令参考

ALTER ROLE

更改一个数据库角色（用户或组）。

概要

```
ALTER ROLE name RENAME TO newname

ALTER ROLE name SET config_parameter {TO | =} {value | DEFAULT}

ALTER ROLE name RESET config_parameter

ALTER ROLE name RESOURCE QUEUE {queue_name | NONE}

ALTER ROLE name [ [WITH] option [ ... ] ]
```

其中 option 可以是：

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEEXTTABLE | NOCREATEEXTTABLE
      [ ( attribute='value'[, ...] ) ]
           where attributes and value are:
           type='readable'|'writable'
           protocol='gpfdist'|'http'
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | [ENCRYPTED | UNENCRYPTED] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | [ DENY deny_point ]
    | [ DENY BETWEEN deny_point AND deny_point]
    | [ DROP DENY FOR deny_point ]
```

描述

ALTER ROLE 更改 HashData 数据库角色的属性。此命令有几种变体：

- **RENAME** — 更改角色的名称。数据库超级用户可以重命名任何角色。角色有 CREATE ROLE 特权 可以重命名非超级用户角色。无法重命名当前会话用户（以其他用户身份连接重命名角色）。因为 MD5 加密的密码使用角色名称作为密钥，如果密码为 MD5 加密，则重命名角色将清除其密码。
- **SET | RESET** — 更改指定配置参数的角色的会话默认值。每当角色随后启动新会话时，指定的值将成为会话默认值，覆盖服务器配置文件中存在的任何设置(postgresql.conf). 对于没有 LOGIN 权限的角色, 会话默认值没有任何效果。普通角色可以更改自己的会话默认值。超级用户可以更改任何人的会话默认值。 有 CREATE ROLE 特权的角色可以更改非超级用户角色的默认值。
- **RESOURCE QUEUE** — 将角色分配给工作负载管理资源队列。 在发出查询时，角色将受到分配资源队列的限制。指定 NONE 将角色分配给默认资源队列。一个角色只能属于一个资源队列。对于没有 LOGIN 特权的角色，会话默认值没有任何作用。 参考 CREATE RESOURCE QUEUE 获取更多信息。
- **WITH** 选项 — 更改在 CREATE ROLE 中指定的角色的许多属性。命令中未提及的属性保留其以前的设置。 数据库超级用户可以更改任何角色的任何这些设置。角色有 CREATE ROLE 权限可以更改任何这些设置，但只能用于非超级用户角色。普通角色只能改变自己的密码。

参数

name

角色的名称，其属性将被更改。

newname

角色的新名称。

config_parameter=value

将指定配置参数的此角色的会话默认值设置为给定值。如果值为 DEFAULT 或者如果 RESET 被使用时，角色特定的变量设置被删除，因此该角色将在新会话中继承系统范围的默认设置。使用RESET ALL 清除所有特定于角色的设置。 参考 SET 以获取有关用户可设置的配置参数的信息。

queue_name

要分配用户级角色的资源队列的名称。 只有有 LOGIN 特权的角色 可以分配给资源队列。要从资源队列中取消分配角色并将其置于默认资源队列中，请指定 NONE。 角色只能属于一个资源队列。

SUPERUSER | NOSUPERUSER

CREATEDB | NOCREATEDB

CREATEROLE | NOCREATEROLE

CREATEEXTTABLE | NOCREATEEXTTABLE [(attribute='value')]

如果 CREATEEXTTABLE 被指定， 允许定义的角色创建外部表。默认类型是可读并且默认协议是 gpfdist。NOCREATEEXTTABLE（默认）拒绝角色有创建外部表的能力。 注意使用 file 或 execute 协议的外部表只能由超级用户创建。

INHERIT | NOINHERIT

LOGIN | NOLOGIN

CONNECTION LIMIT connlimit

PASSWORD password

ENCRYPTED | UNENCRYPTED

VALID UNTIL 'timestamp'

这些子句通过 CREATE ROLE 改变了原来设置的角色属性。

DENY deny_point

DENY BETWEEN deny_point AND deny_point

DENY 和 DENY BETWEEN 关键字设置了在登录时强制执行的基于时间的约束。DENY 设置一天或一天的时间来拒绝访问。DENY BETWEEN 设置访问被拒绝的间隔。 两者都使用以下格式的参数 deny_point ：

```
DAY day [ TIME 'time' ]
```

deny_point两部分参数使用以下格式：

对于 day:

```
{'Sunday' | 'Monday' | 'Tuesday' |'Wednesday' | 'Thursday' | 'Friday' |
'Saturday' | 0-6 }
```

对于 time:

```
{ 00-23 : 00-59 | 01-12 : 00-59 { AM | PM }}
```

DENY BETWEEN 子句使用两个 deny_point 参数。

```
DENY BETWEEN deny_point AND deny_point
```

DROP DENY FOR deny_point

该 DROP DENY FOR 子句从角色中删除基于时间的约束。它使用上述的 deny_point 参数。

注解

使用 GRANT 和 REVOKE 用于添加和删除角色成员资格。

使用此命令指定未加密的密码时，必须小心。密码将以明文形式发送到服务器，也可能会记录在客户端的命令历史记录或服务器日志中。 该 psql 命令行客户端包含一个元命令 password 可用于安全地更改角色的密码。

还可以将会话默认值与特定数据库绑定而不是角色。如果存在冲突，则特定于角色的设置将覆盖数据库特定的设置。参阅 ALTER DATABASE。

示例

更改角色的密码：

```
ALTER ROLE daria WITH PASSWORD 'passwd123';
```

更改密码失效日期：

```
ALTER ROLE scott VALID UNTIL 'May 4 12:00:00 2015 +1';
```

使密码永久有效：

```
ALTER ROLE luke VALID UNTIL 'infinity';
```

赋予角色创建其他角色和新数据库的能力：

```
ALTER ROLE joelle CREATEROLE CREATEDB;
```

给角色一个非默认设置 maintenance_work_mem 参数：

```
ALTER ROLE admin SET maintenance_work_mem = 100000;
```

将角色分配给资源队列：

```
ALTER ROLE sammy RESOURCE QUEUE poweruser;
```

授予创建可写外部表的角色权限：

```
ALTER ROLE load CREATEEXTTABLE (type='writable');
```

更改角色在星期日不允许登录访问：

```
ALTER ROLE user3 DENY DAY 'Sunday';
```

改变角色以消除星期日不允许登录访问的约束：

```
ALTER ROLE user3 DROP DENY FOR DAY 'Sunday';
```

兼容性

ALTER ROLE 语句是一个 HashData 数据库扩展。

另见

CREATE ROLE，DROP ROLE，SET，CREATE RESOURCE QUEUE，GRANT，REVOKE

上级主题： SQL命令参考

ALTER SCHEMA

更改一个模式定义。

概要

```
ALTER SCHEMA name RENAME TO newname

ALTER SCHEMA name OWNER TO newowner
```

描述

ALTER SCHEMA 更改一个模式定义。

用户必须拥有该模式才能使用 ALTER SCHEMA。 要重命名一个模式， 用户还必须拥有该数据库的 CREATE 特权。 要更改拥有者，用户还必须是新拥有角色的一个直接或者间接成员，并且该角色必须具有该数据库上的 CREATE 特权。 注意超级用户自动拥有所有这些特权。

参数

name

现有模式的名称。

newname

该模式的新名称。新名称不能以 pg_ 开头,因为这些名称被保留用于系统模式。

newowner

该模式的新的拥有者。

兼容性

在 SQL 标准中没有 ALTER SCHEMA 语句。

另见

CREATE SCHEMA, DROP SCHEMA

上级主题: SQL命令参考

ALTER SEQUENCE

更改一个序列的定义。

概要

```
ALTER SEQUENCE name [INCREMENT [ BY ] increment]
     [MINVALUE minvalue | NO MINVALUE]
     [MAXVALUE maxvalue | NO MAXVALUE]
     [RESTART [ WITH ] start]
     [CACHE cache] [[ NO ] CYCLE]
     [OWNED BY {table.column | NONE}]

ALTER SEQUENCE name SET SCHEMA new_schema
```

描述

ALTER SEQUENCE 更改一个现有序列发生器的参数。任何没有被明确在 ALTER SEQUENCE 命令中设置的参数，都要维持他们之前的设置。

用户必须拥有该序列才能使用 ALTER SEQUENCE。要更改一个序列的模式，用户还必须拥有新模式上的 CREATE 特权。注意超级用户自动拥有所有的特权。

参数

name

要修改的序列的名称（可选方案限定）。

increment

子句 INCREMENT BY increment 是可选的。一个正值将产生一个上升序列，一个负值会产生一个下降序列。如果未被指定，则旧的增量值将被保持。

minvalue

NO MINVALUE

可选的子句 MINVALUE minvalue 决定一个序列能产生的最小值。 如果 NO MINVALUE 被指定，上升序列和下降序列的默认值分别是 1 和 -263-1。如果这些选项都没有被指定，将保持当前的最小值。

maxvalue

NO MAXVALUE

可选子句 MAXVALUE maxvalue 决定一个序列能产生的最大值。 如果 NO MAXVALUE 被指定, 上升序列和下降序列的默认值分别是 263-1 和 -1。如果这些选项都没有被指定，将保持当前的最大值。

start

可选子句 RESTART WITH 更改该序列被记录的开始值。

cache

子句 CACHE cache 使得序列数字被预先分配并且保存在内存中以便更快的访问。最小值是 1（每次只产生一个值，即无缓存）。如果没有指定，旧的缓冲值将被保持。

CYCLE

当一个上升或者下降序列已经达到 maxvalue 或者 minvalue 时，可选的 CYCLE 关键词可以被用来允许该序列绕回。如果达到限制，下一个被生成的数字将分别是 minvalue 或者 maxvalue。

NO CYCLE

如果可选的 NO CYCLE 关键字被指定，在该序列达到其最大值后对 nextval 的任何调用将会返回错误。如果 CYCLE 或 NO CYCLE 都没有被指定, 将维持旧的循环行为。

OWNED BY *table.column*

OWNED BY NONE

The OWNED BY 选项导致该序列与一个特定的表列相关联，这样如果该列（或者整个表）被删除，该序列也会被自动删除。 如果指定了关联， 这种关联会替代之前为该序列指定的任何关联。被指定的表必须具有相同的 拥有者并且与该序列在同一个模式中。 OWNED BY NONE 可以移除任何现有的关联。

new_schema

该序列的新模式。

注解

为了避免从同一个序列获得数字的并发事务阻塞， ALTER SEQUENCE 在该序列生成参数上的结果永远不会被回滚，那些更改立刻生效并且无法逆转。不过， OWNED BY、RENAME TO 以及 SET SCHEMA 子句会导致普通目录被更新并且无法被回滚。

ALTER SEQUENCE 不会立即影响会话中的 nextval() 值，也不会影响当前会话中预先分配的序列值。在注意到序列生成参数被更改之前它们将用尽所有缓存的值。当前会话将被立刻影响。

一些 ALTER TABLE 的变体可以很好的用于序列。 例如, 重命名一个序列利用 ALTER TABLE RENAME。

示例

重启一个被称为 serial 的序列在 105:

```
ALTER SEQUENCE serial RESTART WITH 105;
```

兼容性

ALTER SEQUENCE 符合 SQL 标准，OWNED BY、RENAME TO 和 SET SCHEMA 子句除外，它们是 HashData 数据库的扩展。

另见

CREATE SEQUENCE，DROP SEQUENCE，ALTER TABLE

上级主题： SQL命令参考

ALTER TABLE

更改一个表的定义。

概要

```
ALTER TABLE [ONLY] name RENAME [COLUMN] column TO new_column
ALTER TABLE name RENAME TO new_name
ALTER TABLE name SET SCHEMA new_schema
ALTER TABLE [ONLY] name SET
     DISTRIBUTED BY (column, [ ... ] )
   | DISTRIBUTED RANDOMLY
   | WITH (REORGANIZE=true|false)
ALTER TABLE [ONLY] name action [, ... ]
ALTER TABLE name
   [ ALTER PARTITION { partition_name | FOR (RANK(number))
   | FOR (value) } partition_action [...] ]
   partition_action
```

其中 action 是下列之一：

```
  ADD [COLUMN] column_name type
      [column_constraint [ ... ]]
  DROP [COLUMN] column [RESTRICT | CASCADE]
  ALTER [COLUMN] column TYPE type [USING expression]
  ALTER [COLUMN] column SET DEFAULT expression
  ALTER [COLUMN] column DROP DEFAULT
  ALTER [COLUMN] column { SET | DROP } NOT NULL
  ALTER [COLUMN] column SET STATISTICS integer
  ADD table_constraint
  DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]
  DISABLE TRIGGER [trigger_name | ALL | USER]
  ENABLE TRIGGER [trigger_name | ALL | USER]
  CLUSTER ON index_name
  SET WITHOUT CLUSTER
  SET WITHOUT OIDS
  SET (FILLFACTOR = value)
  RESET (FILLFACTOR)
  INHERIT parent_table
  NO INHERIT parent_table
  OWNER TO new_owner
  SET TABLESPACE new_tablespace
```

其中 partition_action 是下列之一：

```
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 [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)]
```

其中partition_element是：

```
    VALUES (list_value [,...] )
  | START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
    [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
  | END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]
```

其中 subpartition_spec 是：

```
subpartition_element [, ...]
```

subpartition_element 是：

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

其中 storage_parameter 是：

```
  APPENDONLY={TRUE|FALSE}
  BLOCKSIZE={8192-2097152}
  ORIENTATION={COLUMN|ROW}
  COMPRESSTYPE={ZLIB|QUICKLZ|RLE_TYPE|NONE}
  COMPRESSLEVEL={0-9}
  FILLFACTOR={10-100}
  OIDS[=TRUE|FALSE]
```

描述

ALTER TABLE 更改一个表的定义。下文描述了几种形式:

- **ADD COLUMN** — 向表中增加一个新列，使用和 CREATE TABLE 相同的语义。 将列添加到append 优化的表时，需要一个 DEFAULT 子句。
- **DROP COLUMN** — 这种形式从表中删除一个列，请注意，如果将用作 HashData 数据库分发密钥的表列删除，表的分发策略将更改为随机分配。涉及列的索引和表约束也将自动删除。 用户将使用级联删除如果任何外部表依赖于此列（例如视图）。
- **ALTER COLUMN TYPE** — 更改表中一列的数据类型。请注意用户不能修改正在用于分配密钥或分区键的列的数据类型。 涉及到该列的索引和简单表约束将通过重新解析最初提供的表达式被自动转换为使用新的列类型。 可选的 USING 子句指定如何从旧的列值计算新列值，如果被省略，默认的转换和从旧类型 到新类型的赋值造型一样。 如果没有从旧类型到新类型的隐式或者赋值造型， 则必须提供一个 USING 子句。
- **SET / DROP DEFAULT** — 这些形式为一列设置或者移除默认值。 默认的值只能在随后的 INSERT 语句中生效。它们不会导致已经在表中的行改变。也会为视图创建默认值，在视图的规则被应用之前。在这种情况下，他们的 ON INSERT 规则被应用之前被插入到视图。
- **SET/DROP NOT NULL** — 允许被标记的列为空值。更改列是否标记为允许空值或拒绝空值。当列不包含空值时，用户只能使用 SET NOT NULL 。
- **SET STATISTICS** — 设置后续的每列统计收集目标 ANALYZE 操作。目标可以设置在 0 到 1000 之间， 或者设置为 -1，以恢复为使用系统默认的统计目标 (default_statistics_target)。
- **ADD table_constraint** — 使用与 CREATE TABLE 相同的语法将新的约束添加到表（而不仅仅是分区）。
- **DROP CONSTRAINT** — 在表上删除指定的约束。
- **DISABLE/ENABLE TRIGGER** — 启用或禁用触发器属于表。系统仍然知道被禁用触发器的存在，但是即使它的触发事件发生也不会执行它。对于一个延迟触发器， 会在事件发生时而不是触发器函数真正被执行时检查其启用状态。 可以禁用或者启用用名称指定的单个触发器或者表上的所有触发器，又或者用户拥有的触发器。禁用或者启用内部生成的约束触发器要求超级用户特权。

  > 注意： HashData 数据库不支持触发器。由于 HashData 数据库的并行性，触发器通常具有非常有限的功能。

- **CLUSTER ON / SET WITHOUT CLUSTER** — 这种形式为未来的 CLUSTER 操作选择或者移除默认的索引。它不会真正地对表进行聚簇。注意 CLUSTER 不建议在 HashData 数据库中重新排序表的方式，因为它需要很长时间。最好是用 CREATE TABLE AS 重新创建表并由索引列排序。

  > 注意： CLUSTER ON 在 append 优化的表上不支持。

- **SET WITHOUT OIDS** — 从该表移除 oid 系统列。注意没有 ALTER TABLE 变体情况下，一旦它们被删除，就可以将 OID 恢复到表中。
- **SET (FILLFACTOR = value) / RESET (FILLFACTOR)** — 更改表的填充因子。表的填充因子是 10 到 100 之间的百分比。默认值为 100（完全打包）。当指定较小的填充因子时， INSERT 操作打包表页面仅指示百分比; 保留每个页面上的剩余空间用于更新该页面上的行。这给了 UPDATE 有机会将更新的副本放在与原始页面相同的页面上，这比将其放置在不同页面上更有效。对于其表项永远不会更新的表格，完全打包是最佳选择，但是在大量更新的表中，较小的填充因子是适当的。请注意，此命令不会立即修改表内容。用户将需要重写表以获得所需的效果。
- **SET DISTRIBUTED** — 设置表的分配策略。对哈希分发策略的更改将导致表数据在物理上重新分布在磁盘上，这可能是资源密集型的。
- **INHERIT parent_table / NO INHERIT parent_table** — 将目标表添加或删除为指定父表的子级。对父母的查询将包括其子表的记录。要作为子级添加，目标表必须已经包含与父级相同的所有列（也可能有其他列）。列必须具有匹配的数据类型， 如果他们父母的约束有 NOTNULL ，他们孩子的约束也必须有 NOT NULL 。 所有人也必须有匹配父母的约束的子表约束 CHECK 。
- **OWNER** — 将表，序列或视图的所有者更改为指定的用户。
- **SET TABLESPACE** — 将表的表空间更改为指定的表空间，并将与表关联的数据文件移动到新的表空间。表上的索引（如果有的话）不移动; 但他们可以单独用 SET TABLESPACE 命令移动。 参阅 CREATE TABLESPACE。如果更改分区表的表空间，则所有子表分区也将被移动到新的表空间。
- **RENAME** — 更改表（或索引，序列或视图）的名称或表中单个列的名称。对存储的数据没有影响。请注意，HashData 数据库分发密钥列不能重命名。
- **SET SCHEMA** — 将表移动到另一个模式。关联的索引，约束和表列所有的序列也被移动。
- **ALTER PARTITION | DROP PARTITION | RENAME PARTITION | TRUNCATE PARTITION | ADD PARTITION | SPLIT PARTITION | EXCHANGE PARTITION | SET SUBPARTITION TEMPLATE** — 更改分区表的结构。在大多数情

况下，用户必须通过父表来更改其子表分区之一。

> 注解： 如果将分区添加到具有子分区编码的表中，则新分区将继承子分区的存储指令。

要使用 ALTER TABLE 用户必须拥有该表。要更改一个表的模式或者表空间，用户还必须拥有新模式或表空间上的 CREATE 特权。 要把一个表作为一个父表的新子表加入， 用户必须也拥有该父表。要更改拥有者，用户还必须是新拥有角色的一个直接或者间接成员，并且该角色必须具有该表的模式上的 CREATE 特权。 超级用户自动拥有这些权限。

> 注意： 如果表具有多个分区，如果表具有压缩，或者表的 blocksize 大，则内存使用量会显着增加。如果与表关联的关系数量较大，则此条件可能会迫使表上的操作使用更多内存。例如，如果表是一个 CO 表并且具有大量列，则每一列都是一个关系。 一个操作像 ALTER TABLE ALTER COLUMN 打开表中的所有列分配关联的缓冲区。如果 CO 表有 40 列和 100 分区，列被压缩，并且块大小为 2 MB（系统系数为 3 ），则系统尝试分配 24 GB，即（40×100）×（2×3）MB 或 24 GB。

参数

ONLY

只对指定的表名进行操作。如果 ONLY 关键字不被使用，将对命名表和与该表相关联的任何子表分区执行操作。

name

要更改的现有表的名称（可能是方案限定的）。 如果 ONLY 被指定，只有该表被更改。如果 ONLY 未指定，该表及其所有后代表（如果有）都会被修改。

> 注意：限制只能添加到整个表中，而不能添加到分区。由于这个限制，该 name 参数只能包含一个表名，而不能是一个分区名。

column

新列或现有列的名称。请注意， HashData 数据库分发密钥列必须特别小心处理。更改或删除这些列可以更改表的分发策略。

new_column

现有列的新名称。

new_name

表的新名称。

type

新列的数据类型，或现有列的新数据类型。如果更改 HashData 分发密钥列的数据类型，则只允许将其更改为兼容类型 (例如， text 到 varchar 是可以的,但是 text 到 int 不行)。

table_constraint

表的新表约束。请注意， HashData Database 目前不支持外键限制。还有一个表只允许一个唯一的约束，唯一性必须在 HashData 数据库分配密钥中。

constraint_name

要删除的现有约束的名称。

CASCADE

自动删除依赖于已删除列或约束的对象（例如，引用该列的视图）。

RESTRICT

如果有任何依赖对象，拒绝删除列或约束。这是默认行为

trigger_name

要禁用或启用的单个触发器的名称。请注意， HashData 数据库不支持触发器。

ALL

禁用或启用属于表的所有触发器，包括与约束相关的触发器。这需要超级用户权限

USER

禁用或启用属于该表的所有用户创建的触发器。

index_name

表应标记为聚类的索引名称。注意 CLUSTER 不是建议在 HashData 数据库中重新排序表的方式，因为它需要很长时间。最好用 CREATE TABLE AS 重新创建表并由索引列排序。

FILLFACTOR

设置表的 fillfactor 百分比

value

FILLFACTOR 参数的新值, 在 10 到 100 之间的百分比。 默认值为 100。

DISTRIBUTED BY (column) | DISTRIBUTED RANDOMLY

指定表的分发策略。更改哈希分发策略将导致表数据在物理上重新分配到磁盘上，这可能是资源密集型的。如果用户声明相同的哈希散布策略或从哈希更改为随机分布， 除非用户声明了 SET WITH (REORGANIZE=true) 否则不会重新分发数据.。

REORGANIZE=true|false

使用 REORGANIZE=true 当哈希分发策略没有改变或者当用户从一个哈希更改为一个随机分布，并且用户想重新分发数据。

parent_table

与该表关联或取消关联的父表

new_owner

表的新所有者的角色名称。

new_tablespace

要移动表的表空间的名称。

new_schema

要移动表的模式的名称。

parent_table_name

更改分区表时，该顶级父表的名称。

ALTER [DEFAULT] PARTITION

如果更改分区比第一级分区更深 那么 ALTER PARTITION 子句用于指定要更改的层次结构中的哪个子分区。

DROP [DEFAULT] PARTITION

删除指定的分区。如果分区具有子分区，子分区也将自动删除。

TRUNCATE [DEFAULT] PARTITION

截断指定的分区。如果分区具有子分区，则子分区也将自动截断。

RENAME [DEFAULT] PARTITION

更改分区的分区名称（而不是关系名称）。使用命名约定创建分区表 `<parentname>_<level>_prt_<partition_name>` 。

ADD DEFAULT PARTITION

将默认分区添加到现有分区设计。当数据与现有分区不匹配时，它将被插入到默认分区中。没有默认分区的分区设计将拒绝

与现有分区不匹配的传入行。必须给默认分区一个名称。

ADD PARTITION

partition_element - 使用表（范围或列表）的现有分区类型，定义要添加的新分区的边界。

name - 此新分区的名称。

VALUES - 对于列表分区，定义分区将包含的值。

START - 对于范围分区，定义分区的起始范围值。默认情况下，起始值为 INCLUSIVE。例如，如果用户宣布开始日期为 '2016-01-01', 那么分区将包含大于或等于 '2016-01-01'。 通常数据类型为 START 表达式与分区键列类型相同。如果不是这样，那么用户必须显式地转换为预期的数据类型。

END - 对于范围分区，定义分区的结束范围值。默认情况下，结束值为 EXCLUSIVE. 例如，如果用户宣布结束日期为 '2016-02-01', t那么分区将包含所有日期小于但不等于 '2016-02-01'. 通常数据类型为END 表达式与分区键列类型相同。如果不是这样，那么用户必须显式地转换为预期的数据类型。

WITH - 设置分区的表存储选项。例如，用户可能希望较旧的分区是追加优化的表和较新的分区为常规堆表。参考 CREATE TABLE 有关存储选项的说明。

TABLESPACE - 要在其中创建分区的表空间的名称。

subpartition_spec - 只允许在没有子分区模板的情况下创建的分区设计。声明要添加的新分区的子分区规范。如果最初使用子分区模板定义了分区表，那么将使用该模板自动生成子分区。

EXCHANGE [DEFAULT] PARTITION

将另一个表交换到分区层次结构到现有分区的位置。在多级别分区设计中，只能交换最低级别的分区（包含数据的分区）。

HashData 数据库服务器配置参数 gp_enable_exchange_default_partition 控制可用 EXCHANGE DEFAULT PARTITION 子句。 参数的默认值为 off。 该子句不可用并且如果在 ALTER TABLE命令中指定了子句 HashData 数据库将返回错误。

> 警告: 在交换默认分区之前，用户必须确保要被交换的表（新的默认分区）中的数据对于默认分区是有效的。例如，新默认分区中的数据不能含有对该分区表其他叶子子分区中有效的数据。否则，针对被交换默认分区所在分区表的查询在被 GPORCA 执行时可能返回不正确的结果。

WITH TABLE table_name - 用户正在换入到分区设计中的表名。用户可以交换一个表数据被存储在数据库中的表。例如，该表由CREATE TABLE命令创建。

通过 EXCHANGE PARTITION 子句，用户还可以交换一个可读外部表（由 CREATE EXTERNAL TABLE 命令创建）到分区层次中替换一个现有的叶子子分区。如果用户指定一个可读外部表，用户还必须指定 WITHOUT VALIDATION 子句以跳过针对正在交换的分区上 CHECK 约束的表验证。

这些情况中不支持用外部表交换叶子子分区：

- 分区表通过 SUBPARTITION 子句创建，或者如果分区具有子分区。
- 分区表含有一个带有检查约束或者 NOT NULL 约束的列。

WITH | WITHOUT VALIDATION - 验证表中的数据匹配正在交换的分区的 CHECK 约束。默认是针对 CHECK 约束验证数据。

> 警告：如果用户指定 WITHOUT VALIDATION 子句，用户必须确保正在交换一个现有叶子子分区的表中的数据对于该分区上的CHECK约束有效。否则针对分区表的查询可能返回不正确的结果。

SET SUBPARTITION TEMPLATE

为一个现有分区修改子分区模板。在设置了一个新的子分区模板后，所有增加的新分区都将具有新的子分区设计（现有分区不会被修改）。

SPLIT DEFAULT PARTITION

分裂一个默认分区。在一种多级分区设计中，用户只能分裂最低层的默认分区（它们包含数据）。分裂默认分区会创建包含指定值的新分区并且让默认分区包含不匹配现有分区的任何值。

AT - 对于列表分区表，指定应该被用作分裂条件的单个列表值。

START - 对于范围分区表，为新分区指定一个开始值。

END - 对于范围分区表，为新分区指定一个结束值。

INTO - 允许用户为新分区指定一个名字。在使用 INTO 子句分裂默认分区时，被指定的第二个分区名应该总是现有的默认分区。如果用户不知道默认分区的名称，用户可以使用 pg_partitions 视图找到它。

SPLIT PARTITION

将一个现有分区分裂成两个分区。在多层分区设计中，用户只能分裂最低层的分区（包含着数据）。

AT - 指定应该被用作分裂条件的单个值。分区将被划分成两个新分区，指定的分裂值会成为后一个分区的开始范围。

INTO - 允许用户为分裂创建两个新分区指定名字。

partition_name

给定的分区名称。

FOR (RANK(number))

对于范围分区，分区在范围中的排名。

FOR ('value')

通过声明一个落在分区边界说明中的值来指定一个分区。如果用FOR声明的值匹配一个分区和它的一个子分区（例如，如果值是一个日期并且表先按月分区然后按日分区），那么FOR将在第一个找到匹配的层次上操作（例如，每月的分区）。如果用户的目的是在子分区上操作，则必须按如下的方式声明： `ALTER TABLE name ALTER PARTITION FOR ('2016-10-01') DROP PARTITION FOR ('2016-10-01');`

注解

ALTER TABLE 命令中指定的表名不能是一个表中的分区名。

在修改或者删除作为 HashData 数据库分布键一部分的列时要特别小心，因为这可能会改变表的分布策略。

HashData 数据库当前不支持外键约束。对于要在 HashData 数据库中实施的唯一约束，表必须被哈希分布（不能用 DISTRIBUTED RANDOMLY ），并且所有的分布键列必须和唯一约束列中前部的列相同。

增加 CHECK 或者 NOT NULL 约束要求扫描表以验证现有的行是否符合约束。

在用 ADD COLUMN 增加列时，表中所有的现有行都用该列的默认值初始化，如果没有指定 DEFAULT 子句则初始化为 NULL（注意不允许在追加优化表中增加不指定默认值的列）。增加一个非空默认值的列或者更改现有列的类型要求重写整个表。对于大型表这可能需要大量的时间，并且会临时要求两倍的磁盘空间。

用户可以在单个 ALTER TABLE 命令中指定多个更改，它们将在对表的一趟扫描中完成。

DROP COLUMN 形式不会物理上移除列，而是简单地将它变成对 SQL 操作不可见。后续在表中的插入和更新操作将为该列存储一个空值。因此，删除一列很快，但是不会立即减少表的磁盘尺寸，因为被删除列所占据的空间没有被回收。这些空间将随着现有行被更新而逐渐被回收。

ALTER TYPE 要求重写整个表的做法有时候是一种优点，因为重写过程会消除表中的任何死亡空间。例如，要立即回收被删除列占据的空间，最快的方法是： `ALTER TABLE table ALTER COLUMN anycol TYPE sametype;` ，其中 anycol 是任何现有表列而 sametype 是该列现有的数据类型。这不会对表产生任何语义可见的变化，但是该命令强制进行了重写，这就会去除无用的数据。

如果表被分区或者由任意后代表，就不允许对父表增加列、重命名列或者更改列类型而不对其后代表做同样的事情。这确保了后代总是具有和父辈相匹配的列。

要查看一个分区表的结构，用户可以使用视图 pg_partitions。这个视图可以帮助确定用户想要修改的特定分区。

只有当后代表没有从任何其他父表继承某一列并且也没有对该列的独立定义时，递归的 DROP COLUMN 操作才会移除后代表的这个列。非递归的 DROP COLUMN（ALTER TABLE ONLY ... DROP COLUMN）不会移除任何后代列，而是将它们标记为独立定义而不是继承而来。

TRIGGER、CLUSTER、OWNER 以及 TABLESPACE 动作不会递归到后代表，也就是说它们的动作总是好像在指定了 ONLY 的情况下执行。增加约束的动作中只有 CHECK 约束可以递归。

如果在包含已经被交换为使用外部表的叶子子分区的分区表上没有数据被更改，则支持这些 ALTER PARTITION 操作。否则会返回错误。

- 添加或删除列。
- 更改列的数据类型。

这些 ALTER PARTITION 对于已经交换为使用外部表的叶子分区的分区表，不支持以下操作：

- 设置子分区模板。
- 更改分区属性。
- 创建一个默认分区。
- 制定分销政策。
- 设置或删除 NOT NULL 列的约束。
- 添加或删除约束。
- 拆分外部分区。

不允许更改系统目录表的任何部分。

示例

向列中添加列：

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

重命名现有列：

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

重命名现有表：

```
ALTER TABLE distributors RENAME TO suppliers;
```

向列添加非空约束：

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

向表中添加检查约束

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK
(char_length(zipcode) = 5);
```

将表移动到不同的模式：

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

将新分区添加到分区表中：

```
ALTER TABLE sales ADD PARTITION
            START (date '2017-02-01') INCLUSIVE
            END (date '2017-03-01') EXCLUSIVE;
```

将默认分区添加到现有分区设计中：

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

重命名分区：

```
ALTER TABLE sales RENAME PARTITION FOR ('2016-01-01') TO jan08;
```

删除范围序列中的第一个（最旧的）分区：

```
ALTER TABLE sales DROP PARTITION FOR (RANK(1));
```

将表交换到用户的分区设计中：

```
ALTER TABLE sales EXCHANGE PARTITION FOR ('2016-01-01') WITH TABLE jan08;
```

拆分默认分区(现有的默认分区名称other)为2017年1月添加新的每月分区：

```
ALTER TABLE sales SPLIT DEFAULT PARTITION
START ('2017-01-01') INCLUSIVE
END ('2017-02-01') EXCLUSIVE
INTO (PARTITION jan09, PARTITION other);
```

第一个分区包含日期为1月1日至15日，第二个分区包含日期1月16日至31日：

```
ALTER TABLE sales SPLIT PARTITION FOR ('2016-01-01')
AT ('2016-01-16')
INTO (PARTITION jan081to15, PARTITION jan0816to31);
```

兼容性

ADD， DROP， SET DEFAULT 项符合 SQL 标准。 其他形式是 SQL 标准的 HashData 数据库扩展。 另外，在一个单一的指定多个操作的能力 ALTER TABLE 命令是一个扩展。

ALTER TABLE DROP COLUMN 可用于删除表的唯一列，留下零列表。这是 SQL 的扩展，它不允许零列表。

另见

CREATE TABLE， DROP TABLE

上级主题： SQL命令参考

ALTER TABLESPACE

更改一个表空间的定义。

概要

```
ALTER TABLESPACE name RENAME TO newname

ALTER TABLESPACE name OWNER TO newowner
```

描述

ALTER TABLESPACE 可用于更改一个表空间的定义。

要使用 ALTER TABLESPACE，用户必须拥有该表空间。要修改拥有者，用户还必须是新拥有角色的直接或者间接成员（注意超级用户自动拥有这些特权）。

参数

name

一个现有表空间的名称。

newname

新的表空间的名称。新的表空间名称不能以 pg_ 或 gp_ 开头。(这类名称保留用于系统表空间).

newowner

该表空间的新拥有者。

示例

重命名表空间 index_space 为 fast_raid:

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

更改表空间 index_space的所有者:

```
ALTER TABLESPACE index_space OWNER TO mary;
```

兼容性

在 SQL 标准中没有 ALTER TABLESPACE 语句。

另见

CREATE TABLESPACE，DROP TABLESPACE

上级主题： SQL命令参考

ALTER TYPE

更改一个数据类型的定义。

概要

```
ALTER TYPE name
   OWNER TO new_owner | SET SCHEMA new_schema
```

描述

ALTER TYPE 更改一种现有类型的定义。用户可以更改类型的模式和所有者。

用户必须拥有此类型才能使用 ALTER TYPE. 要更改类型的模式，用户还必须对新模式具有 CREATE 特权。 要更改类型拥有者用户必须是新角色的直接或间接成员，并且该角色在类型的模式上有 CREATE 特权(这些限制强制修改拥有者不能做一些通过删除和重建类型做不到的事情。不过，一个超级用户怎么都能更改任何类型的所有权)。

参数

name

要修改的一个现有类型的名称（可选限定模式）。

new_owner

该类型新的拥有者的类型名。

new_schema

该类型的新模式。

示例

更改用户自定义的 email 拥有者为 joe:

```
ALTER TYPE email OWNER TO joe;
```

更改用户自定义的 email 类型模式为 customers:

```
ALTER TYPE email SET SCHEMA customers;
```

兼容性

在 SQL 标准中没有 ALTER TYPE 语句。

另见

CREATE TYPE， DROP TYPE

上级主题： SQL命令参考

ALTER USER

更改数据库用户（角色）的定义。

概要

```
ALTER USER name RENAME TO newname

ALTER USER name SET config_parameter {TO | =} {value | DEFAULT}

ALTER USER name RESET config_parameter

ALTER USER name [ [WITH] option [ ... ] ]
```

其中 option 可能是:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
```

描述

ALTER USER 是一个已弃用的命令，但由于历史原因仍然被接受，它是 ALTER ROLE 的别名。参阅 ALTER ROLE 获取更多信息。

兼容性

ALTER USER 语句是一个 HashData 数据库扩展。 SQL 标准中使用用户的定义来实现。

另见

ALTER ROLE

上级主题： SQL命令参考

ANALYZE

收集有关一个数据库的统计信息。

概要

```
ANALYZE [VERBOSE] [ROOTPARTITION [ALL] ]
   [table [ (column [, ...] ) ]]
```

描述

ANALYZE 收集一个数据库中的表的内容的统计信息， 并且将结果存储在 pg_statistic 系统目录中。接下来，查询规划器会使用这些统计信息来帮助确定查询最有效的执行计划。

如果不带参数， ANALYZE 会检查当前数据库中的所有表。用户可以指定表名来收集单个表的统计信息。用户可以指定一组列名称，在这种情况下，仅收集这些列的统计信息。

ANALYZE 不收集外部表上的统计信息。

> 重要：如果用户想要在启用 GPORCA（默认启用）时在分区表上执行查询，那么用户必须在分区表的根分区上用 ANALYZE ROOTPARTITION 命令收集统计信息。
>
> 注意： 用户还可以使用 HashData 的数据库工具 analyzedb 更新表统计信息。analyzedb 工具可以并发地为多个表更新统计信息。该工具还能检查表统计信息并且只在统计信息不是当前最新或者不存在时更新之。

参数

ROOTPARTITION [ALL]

只在分区表的根分区上根据分区表中的数据收集统计信息。不会在叶子子分区上收集统计信息，其中的数据只会被采样。当用户指定 ROOTPARTITION 时，必须指定 ALL 或者一个分区表的名称。

如果用户在 ROOTPARTITION 中指定 ALL， HashData 数据库会为该数据库中所有分区表的根分区收集统计信息。如果在数据库中没有分区表，会返回一个消息说明没有分区表。对于非分区表不会收集统计信息。

如果用户在 ROOTPARTITION 中指定一个表名并且该表不是分区表，则不会为该表收集统计信息并且会返回一个警告消息。

ROOTPARTITION 子句对 VACUUM ANALYZE 不合法。命令 VACUUM ANALYZE ROOTPARTITION 会返回一个错误。

运行 ANALYZE ROOTPARTITION 所需的时间类似于分析一个有着相同数据的非分区表所需的时间，因为 ANALYZE ROOTPARTITION 仅采样叶子子分区数据。

对于分区表 *sales_curr_yr*，这个命令例子只在分区表的根分区上收集统计信息。ANALYZE ROOTPARTITION sales_curr_yr;

这个 ANALYZE 命令的例子在数据库中所有分区表的根分区上收集统计信息。

```
ANALYZE ROOTPARTITION ALL;
```

VERBOSE

启用进度消息的显示。被指定时，ANALYZE 会发出这些信息

- The table that is being processed.
- The query that is executed to generate the sample table.
- The column for which statistics is being computed.

- The queries that are issued to collect the different statistics for a single column.
- The statistics that are generated.

table

要分析的特定表的名称（可能被方案限定）。默认为当前数据库中的所有表。

column

要分析的特定列的名称。默认为所有列。

注解

定期运行 ANALYZE 是个好主意，或者只在表内容发生大量变化时才运行。准确的统计信息会帮助 HashData 数据库选择最合适的查询计划，并且因此改进查询处理的速度。一种常用的策略是每天在系统负载低的时候运行一次 VACUUM 和 ANALYZE。

ANALYZE 要求目标表上的 SHARE UPDATE EXCLUSIVE 锁。这个锁会与这些锁冲突：SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、 ACCESS EXCLUSIVE。

对于分区表，指定要分析该表的哪一部分，如果分区表有大量已经分析过的分区并且只有少数叶子子表改变，根分区或者子分区（叶子子表）会很有用。

- 当用户在根分区表上运行 ANALYZE 时，会为所有的叶子子表（ HashData 数据库为分区表创建的子表层次中最低层的表）收集统计信息。
- 当用户在一个叶子子表上运行 ANALYZE 时，只会为该叶子子表收集统计信息。当用户在非叶子子表的子表上运行 ANALYZE 时，不会有统计信息被收集。

  例如，用户可以创建一个分区从 2006 年到 2016 年的分区表，每年中的每个月是一个子分区。如果用户在 2013 年的子表上运行 ANALYZE，则不会有统计信息被收集。如果用户在 2013 年三月的叶子子表上运行 ANALYZE，则只会为该叶子子表收集统计信息。

  > 注意： 当用户用 CREATE TABLE 命令创建一个分区表时， HashData 数据库会创建用户指定的表（根分区或者父表），还会基于用户指定的分区层次创建表的层次（子表）。分区表、子表和它们的继承层次关系通过系统视图 *pg_partitions* 跟踪。

对于一个含有被交换为使用外部表的叶子子分区的分区表，ANALYZE 不会为外部表分区收集统计信息：

- 如果 ANALYZE [ROOTPARTITION] 被运行，外部表不会被采样并且根表统计信息不包括外部表分区。
- 如果在外部表分区上运行 ANALYZE，该分区不会被分析。
- 如果指定了 VERBOSE 子句，会显示一个报告消息：skipping external table.

  HashData 数据库服务器配置参数 optimizer_analyze_root_partition 影响何时在分区表的根分区上收集统计信息。如果该参数被启用，当用户运行 ANALYZE（没有 ROOTPARTITON 关键词）并且指定根分区时，会在根分区上收集统计信息。

ANALYZE 收集的统计信息通常包括每个列中一些最常见值构成的列表以及显示每列中近似数据分布的柱状图。如果 ANALYZE 觉得对两者中的一个或者全部不感兴趣（例如，在一个唯一键列中，其中没有常见值）或者该列的数据类型不支持合适的操作符，它可能会忽略它们。

对于大型表，ANALYZE 会取得该表内容的一份随机采样，而不是检查每一行。这允许非常大的表在很短的时间内被分析完。不过要注意，这样的统计信息只是近似的，并且即使实际表内容没有改变，统计信息也将会在每次运行 ANALYZE 后发生少许变化。这可能会导致 EXPLAIN 所显示的规划器估计代价出现少许变化。在很少见的情况下，这无疑将导致在两次 ANALYZE 运行之间查询优化器选择不同的查询计划。为了避免这种情况，可以通过调整 default_statistics_target 配置参数增加 ANALYZE 收集的统计信息量，或者逐列用 ALTER TABLE ... ALTER COLUMN ... SET STATISTICS（见 ALTER TABLE）设置每列的统计信息目标。该目标值设置最常见值列表中的最大项数以及柱状图中的最大箱数。默认的目标值是 10，但可以改变它以平衡规划器估计精度以及 ANALYZE 所花的时间和在 pg_statistic 中所占用的空间。特别地，将统计信息目标设置为零会禁用该列的统计信息收集。对那些从不出现在查询的 WHERE、GROUP BY 或者 ORDER BY 子句中的列来说，这可能会很有用，因为规划器将不会使用这类列上的统计信息。

被分析列中最大的统计信息目标决定了为准备统计信息要采样的表行数。增加该目标会导致执行 ANALYZE 所需的时间空间成比例增加。

当 HashData 数据库执行一次 ANALYZE 操作为一个表收集统计信息并且检测到所有被采样表的数据页为空（不含有效数据）时， HashData 数据库会显示应该执行一次 VACUUM FULL 操作的消息。如果被采样页为空，该表的统计信息将不准确。页面会在对表的大量更新后变为空，例如删除了大量行。一次 VACUUM FULL 操作会移除空页并且允许 ANALYZE 操作收集准确的统计信息。

如果表没有统计信息，服务器配置参数 gp_enable_relsize_collection 控制传统查询优化器是否使用默认统计信息文件或者使用 pg_relation_size 函数估算表的尺寸。默认情况下，如果统计信息不可用，传统优化器使用默认统计信息文件来估算行数。

示例

为表mytable收集统计信息：

```
ANALYZE mytable;
```

兼容性

SQL 标准中没有 ANALYZE 语句。

另见

ALTER TABLE、 EXPLAIN、 VACUUM

上级主题： SQL命令参考

BEGIN

开始一个事务块。

概要

```
BEGIN [WORK | TRANSACTION] [transaction_mode]
      [READ ONLY | READ WRITE]
```

其中 transaction_mode 是下列之一：

```
ISOLATION LEVEL | {SERIALIZABLE | READ COMMITTED | READ UNCOMMITTED}
```

描述

BEGIN 开始一个事务块，也就是说所有 BEGIN 命令之后的所有语句将被在一个 事务中执行，直到给出一个显式的 COMMIT 或 ROLLBACK。 默认情况下 (没有 BEGIN)， HashData 数据库在 "自动提交" 模式中执行事务，也就是说每个语句都在自己的事务中执行并且在语句结束时隐式地执行一次提交（如果执行成功，否则会完成一次回滚）。

在一个事务块内的语句会执行得更快，因为事务的开始/提交也要求可观的 CPU 和磁盘活动。在进行多个相关更改时，在一个事务内执行多个语句也有助于保证一致性：在所有相关更新还没有完成之前，其他会话将不 能看到中间状态。

如果指定了隔离级别、读/写模式或者延迟模式，新事务也会有那些特性， 就像执行了SET TRANSACTION 一样。

参数

WORK

TRANSACTION

可选的关键词。它们没有效果。

SERIALIZABLE

READ COMMITTED

READ UNCOMMITTED

SQL 标准定义了四个事务隔离级别： READ COMMITTED, READ UNCOMMITTED, SERIALIZABLE 和 REPEATABLE READ。默认行为是一个语句只能看到在开始之前提交的行 (READ COMMITTED)。在 HashData 数据库中 READ UNCOMMITTED 被视为与 READ COMMITTED 相同。不支持 REPEATABLE READ； 如果需要此行为请使用 SERIALIZABLE 。 SERIALIZABLE 是最严格的事务隔离。该级别模拟串行事务执行，就好像事务已经连续执行，而不是同时执行。使用此级别的应用程序必须准备好重试因序列化失败而导致的事务。

READ WRITE

READ ONLY

确定事务是读/写还是只读。读/写是默认值。当事务为只读时，不允许以下 SQL 命令：INSERT, UPDATE, DELETE, 和 COPY FROM 如果他们的表不是临时表的话； 所有的 CREATE, ALTER, 和 DROP 命令还有 GRANT, REVOKE, TRUNCATE;和 EXPLAIN ANALYZE 以及 EXECUTE 如果他们将执行的命令列在列表的话。

注解

START TRANSACTION 具有 BEGIN 相同的功能。

使用 COMMIT 或 ROLLBACK 来终止一个事务块。

在一个事务块中时发出 BEGIN 将惹出一个警告消息。事务状态不会被影响。要在一个事务块中嵌套事务，可以使用保存点(见 SAVEPOINT)。

示例

开始一个事务块：

```
BEGIN;
```

要使用可序列化隔离级别开始事务块：

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

兼容性

BEGIN 是 HashData 数据库语言扩展。它相当于 SQL 标准的命令START TRANSACTION。

顺便一提， BEGIN 关键词用于嵌入式 SQL 中的不同用途。建议在移植数据库应用程序时注意事务语义。

另见

COMMIT， ROLLBACK，START TRANSACTION，SAVEPOINT

上级主题： SQL命令参考

CHECKPOINT

强制执行事务日志检查点。

概要

```
CHECKPOINT
```

描述

预写式日志记录（WAL）每隔一段时间将一个检查点放在事务日志中。根据服务器配置参数 checkpoint_segments 和 checkpoint_timeout， 每个 HashData 数据库段实例设置自动检查点间隔。CHECKPOINT 命令在发出命令时强制立即检查点，而不等待计划的检查点。

检查点是事务日志序列中的所有数据文件已更新以反映日志中的信息的一点。所有数据文件将刷新到磁盘。

只有超级用户可以调用 CHECKPOINT。该命令不适用于正常操作。

兼容性

CHECKPOINT 是 HashData 数据库语言扩展。

上级主题： SQL命令参考

CLOSE

关闭一个游标。

概要

```
CLOSE cursor_name
```

描述

CLOSE 释放与一个已打开游标相关的资源。在游标被关闭后，不允许在其上做后续的操作。当不再需要使用一个游标时应该关闭它。

当一个事务被 COMMIT 或 ROLLBACK 终止时，每一个非可保持的已打开游标会被隐式地关闭。 当创建一个可保持游标的事务通过 ROLLBACK 中止时，该可保持游标会被隐式地关闭。如果该创建事务成功地提交，可保持游标会保持打开，直至执行一个显式的 CLOSE 或者客户端连接断开。

参数

cursor_name

要关闭的已打开游标的名称。

注解

HashData 没有一个显式的 OPEN 游标语句，一个游标在被声明时就被认为是打开的。使用 DECLARE 语句可以声明游标。

通过查询 pg_cursors 系统视图可以看到所有可用的游标。

示例

关闭游标 portala:

```
CLOSE portala;
```

兼容性

CLOSE 完全服从 SQL 标准。

另见

DECLARE，FETCH，MOVE

上级主题：SQL命令参考

CLUSTER

根据索引对磁盘上的堆存储表进行物理重新排序。不是在 HashData 数据库中推荐的操作。

概要

```
CLUSTER indexname ON tablename

CLUSTER tablename

CLUSTER
```

描述

CLUSTER 根据索引订购堆存储表。 CLUSTER 在追加优化的存储表中不支持。 聚类索引意味着记录根据索引信息在磁盘上进行物理排序。如果用户需要的记录随机分布在磁盘上，那么数据库必须在磁盘上查找以获取请求的记录。如果这些记录更紧密地存储在一起，那么从磁盘读取更顺序。聚集索引的一个很好的例子是在日期列中，其中数据按日期顺序排列。针对特定日期范围的查询将导致从磁盘进行有序的提取，从而利用更快的顺序访问。

聚类是一次性操作：当表随后更新时，更改不会聚类。也就是说，没有尝试根据其索引顺序存储新的或更新的行。如果有人愿意，可以通过再次发出命令来定期重新排队。

当一个表被更新时， HashData 会记住它是按照哪个索引聚簇的。CLUSTER tablename 会使用前面所用的同一个索引对表重新聚簇。不带任何参数的 CLUSTER 会重新聚集调用用户拥有的当前数据库中所有以前的群集表，或者超级用户调用的所有表。 这种形式的 CLUSTER 不能在一个事务块内执行。

当一个表被聚簇时，会在其上要求一个 ACCESS EXCLUSIVE 锁。这会阻止任何其他数据库操作（包括读和写）在 CLUSTER 结束前在该表上操作。

参数

indexname

一个索引的名称。

tablename

一个表的名称（可能是方案限定的）。

注解

在随机访问一个表中的行时，表中数据的实际顺序是无关紧要的。 不过，如果用户想要更多地访问其中一些数据，并且有一个索引把它们分组在一起，使用 CLUSTER 就会带来好处。 如果用户从一个表中要求一个范围的被索引值或者多行都匹配的一个单一值，CLUSTER 就会有所帮助，因为一旦该索引标识出了第一个匹配行所在的表页，所有其他匹配行很可能就在同一个表页中，并且因此节省了磁盘访问并且提高了查询速度。

在集群操作期间，创建表的临时副本，其中包含索引顺序中的表数据。也创建表上每个索引的临时副本。因此，用户需要至少等于表大小和索引大小之和的磁盘上的可用空间。

因为规划器会记录有关表顺序的统计信息，建议在新近被聚簇的表上运行 ANALYZE 。 否则，规划器可能会产生很差的查询计划。

还有另一种聚类数据的方式。CLUSTER comm 命令通过使用用户指定的索引扫描原始表来重新排序。这在大型表上可能很慢，因为以索引顺序从表中提取行，如果表无序，则条目在随机页面上，因此每个移动的行都检索到一个磁盘页面。（

HashData 数据库有一个缓存，但大部分的大表不适合缓存。）集群表的另一种方法是使用如下语句：

```
CREATE TABLE newtable AS SELECT * FROM table ORDER BY column;
```

这使用 HashData 数据库排序代码来产生所需的顺序，通常要比无序数据的索引扫描快得多。然后用户删除旧表， 使用 ALTER TABLE ... RENAME 将 newtable 重命名为旧名称，并重新创建表的索引。这种方法的最大缺点是它不保留表的 OID，约束，授予权限和其他辅助属性 - 所有这些项必须手动重新创建。另一个缺点是这样需要与表本身大小相同的排序临时文件，因此峰值磁盘使用量大约是表大小的三倍，而不是表大小的两倍。

> 注意: CLUSTER 不支持追加优化的表。

示例

基于索引 employees 的聚集表 emp_ind:

```
CLUSTER emp_ind ON emp;
```

通过重新创建并以正确的索引顺序加载它来集群大型表： order:

```
CREATE TABLE newtable AS SELECT * FROM table ORDER BY column;
DROP table;
ALTER TABLE newtable RENAME TO table;
CREATE INDEX column_ix ON table (column);
VACUUM ANALYZE table;
```

兼容性

在 SQL 标准中没有 CLUSTER 语句。

另见

CREATE TABLE AS, CREATE INDEX

上级主题： SQL命令参考

COMMENT

定义或更改对象的注释。

概要

```
COMMENT ON
{ TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type [, ...]) |
  CAST (sourcetype AS targettype) |
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  FILESPACE object_name |
  FUNCTION func_name ([[argmode] [argname] argtype [, ...]]) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  OPERATOR op (leftoperand_type, rightoperand_type) |
  OPERATOR CLASS object_name USING index_method |
  [PROCEDURAL] LANGUAGE object_name |
  RESOURCE QUEUE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TABLESPACE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name }
IS 'text'
```

描述

COMMENT 存储关于数据库对象的注释。因此为了修改一段注释，对同一个对象发出一个新的 COMMENT 命令。 要移除一段注释，可在文本字符串的位置上写上 NULL。当对象被删除时，其注释也会被自动删除。

可以使用 psql 元命令轻松检索注释 dd, d+, l+。其他检索注释的用户接口可以构建在 psql 使用的内建函数之上即 obj_description, col_description, shobj_description.

参数

object_name

table_name.column_name

agg_name

constraint_name

func_name

op

rule_name

trigger_name

要注释的对象的名称。表，聚合，域，函数，索引，操作符，操作符类，序列，类型和视图的名称可能是方案限定的。

> 注意：HashData 数据库不支持触发器。

agg_type

聚合功能运行的输入数据类型。要引用零参数聚合函数，写 * 代替输入数据类型的列表。

sourcetype

造型的源数据类型的名称。

targettype

造型的目标数据类型的名称。

argmode

一个函数或者聚集的参数的模式：IN, OUT, INOUT,或者 VARIADIC。如果被省略，默认值是 IN。 注意 COMMENT ON FUNCTION 并不真正关心 OUT 参数， 因为决定函数的身份只需要输入参数。 因此，列出 IN, INOUT, 和 VARIADIC 参数就足够了。

argname

一个函数或者聚集参数的名称。 注意 COMMENT ON FUNCTION 并不真正关心参数名称， 因为决定函数的身份只需要参数数据类型。

argtype

一个函数或者聚集参数的数据类型。

large_object_oid

大对象的 OID。

PROCEDURAL

这是一个噪声词。

text

写成一个字符串的新注释。如果要删除注释，写成 NULL。

注解

当前对查看注释没有安全机制：任何连接到一个数据库的用户能够看到该数据库中所有对象的注释。对于数据库、角色、表空间这类共享对象，注释被全局存储，因此连接到集簇中任何数据库的任何用户可以看到共享对象的所有注释。因此，不要在注释中放置有安全性风险的信息。

示例

为表 mytable 附加一段注释：

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

移除它：

```
COMMENT ON TABLE mytable IS NULL;
```

兼容性

SQL 标准中没有 COMMENT 语句。

上级主题： SQL命令参考

COMMIT

提交当前事务。

概要

```
COMMIT [WORK | TRANSACTION]
```

描述

COMMIT 提交当前事务。所有由该事务所作的更改会变得对他人可见并且被保证在崩溃发生时仍能持久。

参数

WORK

TRANSACTION

可选的关键词。它们没有效果。

注解

使用 ROLLBACK 中止一个事务。

当不在一个事务内发出 COMMIT 时不会产生危害，但是它会产生一个警告消息。

示例

要提交当前事务并且让所有更改持久化：

```
COMMIT;
```

兼容性

SQL 标准仅指定了两种形式： COMMIT 和 COMMIT WORK。除此之外，这个命令完全符合。

另见

BEGIN， END， START TRANSACTION， ROLLBACK

上级主题： SQL命令参考

COPY

在一个文件和一个表之间复制数据。

概要

```
COPY table [(column [, ...])] FROM {'file' | STDIN}
     [ [WITH]
       [BINARY]
       [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]
       SEGMENT REJECT LIMIT count [ROWS | PERCENT] ]

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

描述

COPY 在 HashData 数据库表和标准文件系统文件之间移动数据。 COPY TO 把一个表的内容复制到另一个文件 (如果复制存在 ON SEGMENT 条件，则根据 Segment 的 ID 复制多个文件), 而 COPY FROM 则从一个文件复制数据到一个表 (把数据追加到表中原有数据)。 COPY TO 也能复制一个 SELECT 查询的结果。

> 注意: COPY FROM 当前不支持带 ON SEGMENT 选项的 COPY TO 产生的 Segment 文件拷贝数据，但是其他工具可以被用来恢复数据。

如果指定了一个列列表，COPY 将只把指定列的数据复制到文件或者从文件复制数据到指定列。如果表中有列不在列列表中， COPY FROM 将会为那些列插入默认值。

带一个文件名的 COPY 指示 HashData 数据库服务器直接从一个文件读取或者写入到一个文件。该 master 主机必须可访问该文件，并且必须从 master 主机的角度指定该名称。当指定 STDIN 或 STDOUT 时，数据通过客户端和 master 主机之间的连接来传输。

当 COPY 和 ON SEGMENT 选项一起被使用时， COPY TO 导致 Segment 创建面向 Segment 的个体文件，这些文件保留在 Segment 主机上。ON SEGMENT 的文件参数是用字符串文字 `<SEGID>` (必须) 并使用绝对路径或 `<SEG_DATA_DIR>` 字符串文字。 当运行 COPY 操作时，Segment 的 ID 和 Segment 数据目录的路径被替换为字符串文字值。

ON SEGMENT 选项允许用户将表数据复制到 Segment 主机上的文件，以用于诸如在集群之间迁移数据或执行备份等操作。通过 ON SEGMENT 选项创建的 Segment 数据可以通过 gpfdist 等工具进行恢复，这对于高速数据加载是有用的。

COPY FROM 当前不支持从带 ON SEGMENT 选项的 COPY TO 产生的 Segment 文件拷贝数据，但是其他工具可以被用来恢复数据。

当指定 STDIN 或 STDOUT 时，通过客户端和主机之间的连接传输数据。 STDOUT 不能与 ON SEGMENT 选项一起使用。

如果使用了 SEGMENT REJECT LIMIT ，则 COPY FROM 操作将在单行错误隔离模式下运行。在此版本中，单行错误隔离模式仅适用于具有格式错误的输入文件中的行 - 例如，额外或缺少属性，错误数据类型的属性或无效的客户端编码序列。诸如违反了 NOT NULL, CHECK, 或 UNIQUE 约束的约束错误仍将以 'all-or-nothing' 输入模式进行处理。用户可以指定可接受的错误行数（以每个 Segment 为单位），之后整个 COPY FROM 操作将被终止， 并且不再加载任何行。 错误行的计数以每个 Segment 而不是整个加载操作为基础。如果未达到每 Segment 的拒绝限制，则将加载不包含错误的所有行，并丢弃任何错误行。要保留错误行进一步检查，请指定 LOG ERRORS 子句以捕获错误日志信息。错误信息和行存储在 HashData 数据库中。

输出

成功完成后， COPY 命令将返回表单的命令标签，其中 count 是复制的行数：

```
COPY count
```

如果单行隔离模式下运行 COPY FROM 命令， 如果由于格式错误而未加载任何行，则将返回以下通知消息， 其中 count 是拒绝的行数：

```
NOTICE: Rejected count badly formatted rows.
```

参数

table

一个现有表的名称（可以是方案限定的）。

column

可选的要被复制的列列表。如果没有指定列列表，则该表的所有列都会被复制。

当以文本格式复制时，默认为一列数据类型 bytea 最高可达 256MB.

query

一个SELECT 或 VALUES 其结果将被复制。请注意，查询需要括号。

file

输入或者输出文件的路径名。

PROGRAM '*command*'

指定一个命令行去执行。该命令行必须从 HashData 数据库的 Master 主机系统指定，并且对 HashData 数据库管理员用户（gpadmin）可执行。COPY FROM 命令行从该命令行的标准输出读取数据作为输入，对于 COPY TO 命令的数据被写入该命令行的标准输入中。

该命令行是被 shell 唤起的。当传输参数到 shell 时，要避免或去除任何对于 shell 存在特殊意义的字符。为了安全考虑，最好使用固定的命令行，或者至少应该避免传输任何用户输入到字符串中。

当指定 ON SEGMENT 选项时，该命令必须在所有 HashData 数据库 primary segment 主机上，对于 HashData 数据库管理员用户（gpadmin）是可执行的。每个 HashData Segment Instance 都会执行此命令。此命令需要指定 `<SEGID>` 。

STDIN

指定输入来自客户端应用。

STDOUT

指定输出会去到客户端应用。

ON SEGMENT

单独指定 Segment 主机上面的 Segment 数据文件。每个文件都包含 primary Segment Instance 管理的表的数据。例如，当使用 COPY TO...ON SEGMENT 命令从一个表中拷贝数据到文件时，该命令会为在 segment 主机上面的每个 segment

Instance 创建一个文件。每个文件都包含由 segment instance 管理的表的数据。COPY 命令不会从 mirror segment instances 和 segment 数据文件拷贝数据或者写入数据。

STDIN 和 STDOUT 关键字不能与 ON SEGMENT 一起使用。

使用 `<SEG_DATA_DIR>` 和 `<SEGID>` 字符串，通过以下语法指定一个绝对路径和文件名称：

```
COPY table [TO|FROM] '<SEG_DATA_DIR>/gpdumpname<SEGID>_suffix' ON SEGMENT;
```

镜像 Segment 不会将其数据拷贝到 Segment 文件中。

`<SEG_DATA_DIR>`

表示 ON SEGMENT 拷贝的 Segment 数据目录的完整路径的字符串文字。'<' 和 '>' 是用于指定路径的字符串文字的一部分。COPY 运行时，用字符串文字替换段路径。 可以用绝对路径代替使用 `<SEG_DATA_DIR>` 字符串文字。

`<SEGID>`

表示复制 ON SEGMENT 时要复制的 Segment 的内容 ID 号的字符串文字。 `<SEGID>` 是 ON SEGMENT 选项的必需部分。'<' 和 '>' 是用于指定路径的字符串文字的一部分。 COPY 运行时， COPY 将使用内容 ID 替换字符串文字。

BINARY

使所有数据以二进制格式存储或读取，而不是文本。用户不能指定 DELIMITER, NULL,或 CSV 二进制模式下的选项。

当以二进制格式复制时，一行数据最多可达 1GB。

OIDS

指定复制每行的 OID。（如果为没有 OID 的表指定了 OIDS，或者在复制查询的情况下，则会引发错误。）

delimiter

单个 ASCII 字符，用于分隔文件每行（行）中的列。默认是文本模式下的制表符，逗号在 CSV 模式。

null string

表示空值的字符串。 文本模式中的默认值为 \N (反斜杠 - N) ， CSV 模式中不含引号的空值。 在不想将空值与空字符串区分开的情况下，即使在文本模式下，也可能更喜欢空字符串。 当使用 COPY FROM 时，与字符串匹配的任何数据项将被转储为空值，因为用户应该确保使用与 COPY TO 中使用的字符串相同。

escape

指定用于 C 转义序列的单个字符(如 \n,\t,\100, 等) 和引用可能被视为行或列分隔符的数据字符。确保选择在实际列数据中的任何地方都不使用的转义字符。默认转义符为 (反斜杠)用于文本文件或 " (双引号) 用于 CSV 文件, 但是可以指定任何其他字符来表示转义。还可以通过指定值 'OFF' 作为转义值。这对于诸如 Web 日志数据之类的数据非常有用，该数据具有许多嵌入式反斜杠，这些反斜杠不是要转义的。

NEWLINE

指定数据文件中使用的换行符 — LF（换行，0x0A）、CR（回车，0x0D）或者 CRLF（回车加换行，0x0D 0x0A）。如果未指定， HashData 数据库的 Segment 将通过查看其接收的第一行数据并使用遇到的第一个换行符来检测换行类型。

CSV

选择逗号分隔值（CSV）模式。

HEADER

指定一个文件包含一个标题行和文件中每列的名称。在输出时，第一行包含表中的列名，在输入时，第一行将被忽略。

引用

以 CSV 模式指定引用字符。默认是双引号。

FORCE QUOTE

在 CSV COPY TO 模式下，强制引用用于每个指定列中的所有非 NULL 值。 NULL 值输出从不引用。

FORCE NOT NULL

在 CSV COPY FROM 模式下，处理每一个指定的列，就好像它被引用一样， 因此不是 NULL 值。对于 CSV 模式默认的字符串(两个分割符之间不存在)，这会导致将值作为零长度字符串计算。

FILL MISSING FIELDS

在 TEXT 和 CSV 中的 COPY FROM 中，指定 FILL MISSING FIELDS 时，当一行数据在行或行的末尾缺少数据字段时，将丢失尾字段值设置为 NULL (而不是报告错误)。 空行，具有 NOT NULL 在 TEXT 和 CSV 中的 COPY FROM 中，指定 FILL MISSING FIELDS 时，当一行数据在行或行的末尾缺少数据字段时，将丢失尾字段值设置为 NULL（而不是报告错误）。 空行，具有 NOT NULL 约束的字段和行上的尾随分隔符仍然会报告错误。

LOG ERRORS

这是一个可选子句，可以在 SEGMENT REJECT LIMIT 之前捕获有关格式错误的行的错误日志信息。

错误日志信息在内部存储，并使用数据库内置 SQL 函数 `gp_read_error_log()` 访问。

这是一个可选的子句，可以在 SEGMENT REJECT LIMIT 子句之前捕获有关格式错误的行的错误日志信息。错误日志信息在内部存储， 并使用 HashData 数据库内置 SQL 函数 `gp_read_error_log( )` 访问。

SEGMENT REJECT LIMIT count [ROWS | PERCENT]

在单行模式下运行 COPY FROM 操作。 如果输入行具有格式错误，则它们将被丢弃，前提是在加载操作期间在任何 HashData 数据库 Segment 实例上未达到拒绝限制计数。拒绝限制计数可以指定为行数（默认值）或总行数百分比（1-100）。 如果 PERCENT 被使用，每个 Segment 只有在处理参数 `gp_reject_percent_threshold` 所指定的行数之后才开始计算行百分比。 `gp_reject_percent_threshold` 默认值为 300 行。 诸如违反 NOT NULL， CHECK, 或 UNIQUE的约束错误仍将以 'all-or-nothing' 输入模式进行处理。如果没有达到限制，所有好的行将被加载，任何错误将被丢弃。

> 注意：HashData 数据库限制了可能包含格式错误的初始行数 SEGMENT REJECT LIMIT 不是首先被触发或没有被指定。 如果前 1000 行被拒绝，COPY 操作停止并回滚。

可以使用 HashData Database 服务器配置参数更改初始拒绝行数的限制 `gp_initial_bad_row_limit` 。

IGNORE EXTERNAL PARTITIONS

从分区表复制数据时，数据不会从作为外部表的叶子分区复制。当不复制数据时，会将消息添加到日志文件中。

如果未指定此子句，并且 HashData 数据库尝试从作为外部表的叶子分区复制数据，则会返回错误。

注解

COPY 只能与表一起使用，而不能与外部表或视图一起使用。 但是，用户可以写 COPY (SELECT * FROM viewname) TO ...

要从具有作为外部表的叶子分区的分区表复制数据，请使用 SQL 查询来复制数据。例如，如果表 my_sales 包含一个叶子子分区，这是一个外部表，这个命令 COPY my_sales TO stdout 返回错误。 此命令将数据发送到 stdout：

```
COPY (SELECT * from my_sales ) TO stdout
```

该 BINARY 关键字将所有数据存储/读取为二进制格式而不是文本。它比正常的文本模式要快一点，但二进制格式的文件在机器架构和 HashData 数据库版本之间的移植性更低。 另外，如果数据是二进制格式, 用户不能运行 COPY FROM 在单行错误隔离模式下。

用户必须对其值由 COPY TO 读取的表具有 SELECT 权限，并在 COPY FROM 插入的值上插入特权。

在 COPY 命令中命名的文件由数据库服务器直接读取或写入，而不是由客户端应用程序读取或写入。 因此，它们必须驻留在 HashData 数据库 master 主机上或可访问，而不是客户端。 它们必须由 HashData 数据库系统用户（服务器运行的用户 ID）而不是客户机可访问和可读写。 COPY 命名文件只允许数据库超级用户使用，因为它允许读取或写入服务器具有访问权限的任何文件。

COPY FROM 将调用目标表上的任何触发器和检查约束。 但是，它不会调用重写规则。 请注意，在此版本中，不对单行错误隔离模式评估对约束的违规。

COPY 的输入输出受 DateStyle 影响。为了确保对可能使用非默认 DateStyle 设置的其他 HashData 数据库安装的可移植性，在使用 COPY TO 之前， 应将 DateStyle 设置为 ISO。

在文本模式下从文件复制 XML 数据时，服务器配置参数 xmloption 影响复制的 XML 数据的验证。 如果值为 content (默认)， XML 数据被验证为 XML 内容片段。如果参数值为 document, XML 数据被验证为 XML 文档。如果 XML 数据无效，COPY 返回错误。

默认情况下， COPY 在第一个错误时停止运行。 在 COPY TO 的情况下，这不应该导致问题但是目标表已经在 COPY FROM 中接受了较早的行。这些行将不可见或可访问，但他们仍占用磁盘空间，如果故障发生在大的 COPY FROM 操作中这可能会相当大量的浪费磁盘空间。用户可能希望 VACUUM 来恢复浪费的空间。 另一个选择是使用单行错误隔离模式来过滤错误行，同时仍然加载好的行。

当用户指定 LOG ERRORS 子句时， HashData 数据库捕获读取外部表数据时发生的错误。用户可以查看和管理捕获的错误日志数据。

- 使用内置的 SQL 函数 `gp_read_error_log('table_name')` 。需要在 table_name 上有 SELECT 特权。 此示例显示使用 COPY 命令加载到表 ext_expenses 中的数据错误日志信息：

  ```
  SELECT * from gp_read_error_log('ext_expenses');
  ```

  如果 table_name 表不存在该函数返回 FALSE 。

- 如果指定的表存在错误日志数据，新的错误日志数据将附加到现有的错误日志数据。错误日志信息不会复制到镜像 Segment。

- 使用内置的 SQL 函数 `gp_truncate_error_log('table_name')` 删除 table_name中的错误日志数据。它需要表所有者权限。此示例删除将数据移动到表中时捕获的错误日志信息 ext_expenses:

  ```
  SELECT gp_truncate_error_log('ext_expenses');
  ```

  如果 table_name 表不存在函数返回 FALSE 。

  指定 * 通配符删除当前数据库中现有表的错误日志信息。指定字符串 *.* 删除所有数据库错误日志信息，包括由于以前的数据库问题而未被删除的错误日志信息。如果 * 没被指定， 则需要数据库所有者权限。如果 *.* 被指定，需要操作系统超级用户权限。

当不是超级用户的 HashData 数据库用户运行 COPY 命令时， 命令可以由资源队列控制。必须配置资源队列 ACTIVE_STATEMENTS 参数，指定分配给该队列的角色可以执行的查询数量的最大限制。 HashData 数据库不会将消耗值或内存值应用于 COPY 命令， 只有消耗或内存限制的资源队列不影响运行 COPY 命令。

非超级用户只能运行这些类型 COPY 命令：

- COPY FROM 命令，其中源为 stdin
- COPY TO 命令，其中源为 stdout

文件格式

文件格式支持 COPY。

文本格式

当使用的 COPY 不带 BINARY 或 CSV 选项时，读取或写入的数据是每个表行一行的文本文件。一行中的列由分隔符字符(默认选项卡)分割。列值本身是由每个属性的数据类型的输出函数生成的或输入函数可接受的字符串。使用指定的空字符串代替为空的列。如果输入文件的任何行包含比预期的更多或更少的列， COPY FROM 将引发错误。如果 OIDS 被指定，OID 作为被读取或写入第一列，位于用户数据列之前。

数据文件有两个具有 COPY 特殊含义的保留字符:

- 指定的分隔符（默认为 tab），用于分隔数据文件中的字段。
- UNIX 样式换行符 (n 或 0x0a), 用于指定数据文件中的新行。强烈建议应用程序生成 COPY 数据的应用程序将数据换行

符转换为 UNIX 样式的换行符，而不是 Microsoft Windows 样式回车换行 (rn 或0x0a 0x0d)。

如果用户的数据包含这些字符，用户必须转义该字符，因此 COPY 将其视为数据而不是字段分隔符或新行。

默认情况下，转义字符是文本格式文件的（反斜杠）和 " (双引号) 为 csv 格式的文件。如果要使用不同的转义字符，可以使用 ESCAPE AS 子句。 确保选择一个在数据文件中的任何位置不被用作实际数据值的转义字符。用户还可以通过使用禁用文本格式文件中的转义 ESCAPE 'OFF'。

例如，假设用户有一个具有三列的表，并且用户想使用 COPY 加载以下三个字段。

- 百分比 = %
- 垂直条 = |
- 反斜杠 =

指定的分隔符是 | (管道字符), 用户指定的转义字符是 * (星号)。数据文件中格式化的行将如下所示：

```
percentage sign = % | vertical bar = ***|** | backslash =
```

请注意，使用星号（*）转义数据的一部分的管道字符。 还要注意，由于我们使用替代转义字符，我们不需要转义反斜杠。

以下字符必须在转义字符前面，如果它们显示为列值的一部分：转义字符本身，换行符，回车符和当前分隔符字符。用户可以使用 ESCAPE AS 子句指定其他转义符。

CSV Format

此格式用于导入和导出许多其他程序（如电子表格）使用的逗号分隔值（CSV）文件格式。 而不是 HashData Database 标准文本模式使用的转义，它会生成并识别常用的 CSV 转义机制。

每个记录的值由 DELIMITER 字符分隔。 如果值包含分隔符字符，则 QUOTE 字符， ESCAPE 字符(默认为双引号), NULL 字符串， 回车符或换行字符，则整个值为前缀，后缀为 QUOTE 字符。用户可以使用 FORCE QUOTE 强制引用在特定列中输出非NULL 。

CSV 格式没有标准的方法来区分 NULL 值和空字符串。 HashData 数据库通过 COPY 引用来处理 。一个 NULL 作为 NULL 字符串输出，不引用， 而与 NULL 字符串匹配的数据值被引用。 因此，使用默认设置，NULL 写为无引号的空字符串，而空字符串用双引号（“”）写入。阅读值遵循相似的规则。用户可以使用 FORCE NOT NULL 来阻止特定列的 NULL 输出比较。

因为反斜杠不是一个特殊的字符 CSV 格式，出现在行上的单个条目的数据值在输出上自动引用，并且在输入时（如果引用）不会被解释为数据结尾标记。 如果用户正在加载另一个应用程序创建的文件，该应用程序具有单个未引用的列， 并且值可能为 . ， 用户可能需要在输入文件中引用该值。

> 注解： 在 CSV 模式，所有字符都很重要。由空格或 DELIMITER 以外的任何字符包围的引用值将包含这些字符。如果用户从系统中将数据从白色空间填充到某些固定宽度的系统中，则可能会导致错误。如果出现这种情况，则在将数据导入到 HashData 数据库之前，用户可能需要预处理 CSV 文件以删除尾随的空格。

CSV 模式将会识别并生成包含嵌入回车符和换行符的引用值的 CSV 文件。因此，文件不是每个表行严格一行，如文本模式文件。

注解： 许多程序产生奇怪且偶然的 CSV 文件，因此文件格式比标准更为常规。因此，用户可能会遇到一些无法使用此机制导入的文件，COPY 可能会生成 其他程序无法处理的文件。

二进制格式

该 BINARY 格式由文件头，包含行数据的零个或多个元组和文件预告片组成。标题和数据是网络字节顺序

- 文件头 — 文件头由 15 个字节的固定字段组成，后面是可变长度的标题扩展区。固定字段是：
  - 签名 — 11 字节序列 PGCOPY n 377 r n 0 - 请注意，零字节是签名的必需部分。（签名被设计为容易地识别由非8位清理传输所掩盖的文件，该签名将通过行尾转换过滤器，丢弃的零字节，丢弃的高位或奇偶变化。）
  - 标志字段 — 32 位整数位掩码，用于表示文件格式的重要方面。位从 0（LSB）到31（MSB）编号。请注意，该字段以网络字节顺序（最高有效字节优先）存储，以及文件格式中使用的所有整数字段。位 16-31 保留以表示关键文件格式问题; 如果发现在此范围内设置了意外的位，读取器将中止。bit 0-15 被保留以表示向后兼容的格式问题; 读者应该简单地忽略在此范围内设置的任何意外的位。目前只定义了一个标志，其余的标志位必须为零（如果数据有 OID，则为16：1，否则为 0）。

    - 标题扩展区长度 — 32 位整数，标题剩余字节长度，不包括自身。目前，这是零，第一个元组立即跟随。格式的未来更改可能允许在标题中存在附加数据。读者应该默默地跳过任何不知道该怎么做的标题扩展名数据。标题扩展区域被设想为包含一系列自识别块。 标志字段不是要告诉读者扩展区域是什么。标题扩展内容的具体设计留待以后发布。
- 元组 — 每个元组从元组中的字段数的 16 位整数开始。（目前，表中的所有元组都将具有相同的计数，但可能并不总是如此）。然后，对于元组中的每个字段重复，都有一个 32 位长度的字， 后跟多个字段的字段数据。（长度字不包括本身，可以为零）。作为特殊情况，-1 表示 NULL 字段值。在 NULL 的情况下没有值字节。

  字段之间没有对齐填充或任何其他额外的数据。

  目前，COPY BINARY 文件中的所有数据值都被假定为二进制格式（格式代码一）。预计未来的扩展可能会添加一个头域，允许指定每列格式代码。

  如果 OID 包含在文件中，则 OID 字段紧跟在字段计数字之后。 这是一个正常的字段，除了它不包括在字段计数中。 特别是它有一个长度字 - 这将允许处理 4 字节与 8 字节 OID 没有太多的痛苦，并将允许 OID 显示为 null，如果有证明是可取的。

- **File Trailer** — file trailer 由包含 -1 的 16 位整数组成。 这很容易与元组的字段计数字区分开。 如果字段计数字不是 -1 -1也不是预期的列数，读者应该报告错误。 这提供额外的检查，以防止与数据不同步。

示例

使用垂直条（|）作为字段分隔符将表复制到客户端：

```
COPY country TO STDOUT WITH DELIMITER '|';
```

从文件中复制数据到 country 表中:

```
COPY country FROM '/home/usr1/sql/country_data';
```

复制到名称以 'A' 开头的国家/地区的文件:

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO
'/home/usr1/sql/a_list_countries.copy';
```

将数据从文件复制到 sales 表使用单行错误隔离模式和日志错误：

```
COPY sales FROM '/home/usr1/sql/sales_data' LOG ERRORS
   SEGMENT REJECT LIMIT 10 ROWS;
```

要拷贝 Segment 数据供以后使用，使用 ON SEGMENT 语句。 使用 ON SEGMENT 语句的形式为:

```
COPY table TO '<SEG_DATA_DIR>/gpdumpname<SEGID>_suffix' ON SEGMENT;
```

该 `<SEGID>` 是必须的。 但是，用户可以替换绝对路径 `<SEG_DATA_DIR>` 字符串文字在路径中。

当用户传递字符串文字 `<SEG_DATA_DIR>` 和 `<SEGID>` 到 COPY, COPY 将在运行操作时填写适当的值。

例如, 如果用户有表 mytable 以及如下所示的 Segment 和镜像 Segment：

```
contentid | dbid | file segment location
    0     |  1   |/home/usr1/data1/gpsegdir0

    0     |  3   | /home/usr1/data_mirror1/gpsegdir0

    1     |  4   | /home/usr1/data2/gpsegdir1

    1     |  2   | /home/usr1/data_mirror2/gpsegdir1
```

运行命令:

```
COPY mytable TO '<SEG_DATA_DIR>/gpbackup<SEGID>.txt' ON SEGMENT;
```

将导致以下文件:

```
/home/usr1/data1/gpsegdir0/gpbackup0.txt
/home/usr1/data2/gpsegdir1/gpbackup1.txt
```

第一列中的内容 ID 是插入到文件路径中的标识符 (例如， gpsegdir0/gpbackup0.txt above) 文件是在 Segment 主机上创建的，而不是在主服务器上，因为它们将在标准中 COPY 操作。 使用时，不会为镜像 Segment 创建任何数据文件 ON SEGMENT 复制。

如果指定绝对路径，而不是 `<SEG_DATA_DIR>` ,如在声明中

```
COPY mytable TO '/tmp/gpdir/gpbackup_<SEGID>.txt' ON SEGMENT;
```

文件将被放置在每个段上的 /tmp/gpdir 中。

> 注解： 该 COPY FROM 操作目前还不支持 ON SEGMENT 语句。像 gpfdist 这样的工具可以恢复数据。

兼容性

在 SQL 标准中没有 COPY 语句。

另见

CREATE EXTERNAL TABLE

上级主题： SQL命令参考

CREATE AGGREGATE

定义一个新的聚集函数。

概要

```
CREATE [ORDERED] AGGREGATE name (input_data_type [ , ... ])
      ( SFUNC = sfunc,
        STYPE = state_data_type
        [, PREFUNC = prefunc]
        [, FINALFUNC = ffunc]
        [, INITCOND = initial_condition]
        [, SORTOP = sort_operator] )
```

描述

CREATE AGGREGATE 定义一个新的聚集函数。在发布中已经包括了一些基本的和常用的聚集函数，像 count、min、max、sum、avg 等在 HashData 数据库中已被提供。如果要定义一个新类型或者需要一个还没有被提供的聚集函数，那么 CREATE AGGREGATE 就可以被用来提供想要的特性。

一个聚集函数需要用它的名称和输入数据类型标识。同一个方案中的两个聚集可以具有相同的名称，只要它们在不同的输入类型上操作即可一个聚集的名称和输入数据类型必须与同一方案中的每一个普通函数区分开。

聚集函数由一个，两个或三个常规函数组成（所有这些函数都必须是 IMMUTABLE 函数）：

- 一个状态转移函数 sfunc
- 一个可选的 Segment 级预备计算函数 prefunc
- 一个可选的最终计算函数 ffunc

这些函数被使用如下：

```
sfunc( internal-state, next-data-values ) ---> next-internal-state
prefunc( internal-state, internal-state ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value
```

用户可以指定 PREFUNC 作为优化聚集执行的方法。通过指定 PREFUNC，聚集可以先在 Segment 上并行执行然后再在 Master 上执行。在执行双层聚集时，SFUNC 在 Segment 上被执行以产生部分聚集结果，而 PREFUNC 在 Master 上被执行以聚集来自 Segment 的部分结果。如果执行单层聚集，所有的行都会被发送到 Master，然后在行上应用 sfunc。

单层聚集和双层聚集是等效的执行策略。任何一种聚集类型都可以在查询计划中实现。当用户实现函数 prefunc 和 sfunc 时，必须确保在 Segment 实例上调用 sfunc 接着在 Master 上调用 prefunc 产生的结果和单层聚集（将所有的行发送到 Master 然后只在行上应用 sfunc）相同。

HashData 数据库会创建一个数据类型为 stype 的临时变量来保存聚集函数的内部状态。在每一个输入行，聚集参数值会被计算并且状态转移函数在当前状态值和新的参数值上被调用以计算一个新的内部状态值。在所有的行被处理之后，最终函数被调用一次以计算该聚集的返回值。如果没有最终函数，则最终状态会被原样返回。

聚集函数可以提供一个可选的初始条件，即一个内部状态值的初始值。在数据库中这被指定并且存储为文本类型的值，但是它必须是内部状态之数据类型的一个合法外部表达。如果不支持这种初始值，则状态值会开始于 NULL。

如果状态转移函数被声明为 STRICT，那么就不能用 NULL 输入调用它。对于这样一个转移函数，聚集执行会按下面的方式执行。带有任何空输入值的行会被忽略（该函数不会被调用并且保持之前的状态值）。如果初始状态值为 NULL，那么在第一个具有所有非空输入值的行那里，第一个参数值会替换状态值，并且对后续具有所有非空输入值的行调用转移函数。这对于实现 max 之类的聚集很有用。注意，只有当 state_data_type 与第一个 input_data_type 相同时，这种行为才可用。当这些类型不同时，用户必须提供一个非空初始条件或者使用一个非严格的转移函数。

如果状态转移函数没有被声明为 STRICT，那么对每一个输入行将无条件调用它，并且它必须自行处理 NULL 输入和 NULL 转移状态。这允许聚集的作者完全控制聚集对 NULL 值的处理。

如果最终函数被声明为 STRICT，那么当结束状态值为 NULL 时不会调用它，而是自动返回一个 NULL 结果（这是 STRICT 函数的通常行为）。在任何情况下，最终函数都有返回一个 NULL 值的选项。例如，avg 的最终函数会在它看到零个输入行时返回 NULL。

单参数聚集函数（例如 min 和 max）有时候可以通过查看索引而不是扫描每个输入行来优化。如果可以这样优化聚集，可通过指定一个排序操作符来指明。基本的要求是该聚集必须得到该操作符产生的排序顺序中的第一个元素，换句话说：

```
SELECT agg(col) FROM tab;
```

必须等效于：

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

进一步的假定是聚集函数会忽略 NULL 输入，并且它只在没有非空输入时才给出一个 NULL 结果。通常，一种数据类型的 < 操作符是用于 MIN 的正确排序操作符，而 > 是用于 MAX 的正确排序操作符。注意除非指定的操作符是 B-树 索引操作符类中的 “小于” 或者 “大于” 策略成员，那么这种优化将不会实际生效。

有序聚集

如果出现 ORDERED，则创建的聚集函数是一种 有序聚集 。这种情况下，不能指定预备聚集函数 prefunc。

有序聚集使用下面的语法调用。

```
name ( arg [ , ... ] [ORDER BY sortspec [ , ...]] )
```

如果 ORDER BY 被省略，则会使用一种系统定义的排序。有序聚集函数的转移函数 sfunc 会在其输入参数上以指定顺序在单个 Segment 上被调用。在 pg_aggregate 表中有一个新列 aggordered 来指定聚集函数是否被定义为有序聚集。

参数

name

要创建的聚集函数的名称（可以被方案限定）。

input_data_type

这个聚集函数在其上进行操作的输入数据类型。要创建一个零参数的聚集函数，在输入数据类型的列表位置写上 * 。这类聚集的例子是 count(*)。

sfunc

为每一个输入行调用的状态转移函数的名称。对于一个 N- 参数的聚集函数，sfunc 必须接收 N+1 个参数，第一个参数类型为 state_data _type 而其他的匹配该聚集声明的输入数据类型。该函数必须返回一个类型为 state_data_type 的值。这个函数会拿到当前状态值和当前的输入数据值，并且返回下一个状态值。

state_data_type

聚集状态值的数据类型。

prefunc

预备聚集函数的名称。这是一个有两个参数的函数，参数的类型都是 state_data_type。它必须返回一个 state_data_type 类型的值。预备函数会拿到两个转移状态值并且返回一个新的转移状态值来表示组合的聚集。在 HashData 数据库中，如果聚集函数的结果是以分 Segment 的方式计算，预备聚集函数就会在一个个内部状态上调用以便把它们组合成一个结束内部状态。

注意在 Segment 内部的哈希聚集模式中也会调用这个函数。因此，如果用户调用这个没有预备函数的聚集函数，则绝不会选

中哈希聚集。由于哈希聚集很有效，因此只要可能，还是应该考虑定义预备函数。

ffunc

在所有的输入行都已经被遍历过后，要调用来计算聚集结果的最终函数的名称。这个函数必须接收类型为 state_data_type 的单个参数。该聚集的返回数据被定义为这个函数的返回类型。如果没有指定 ffunc，那么结束状态值会被用作聚集结果并且该返回类型为 state_data_type。

initial_condition

状态值的初始设置。这必须是一个数据类型 state_data_type 可接受形式的字符串常量。如果没有指定，状态值会开始于 NULL。

sort_operator

用于 MIN- 或者 MAX 类聚集函数的相关排序操作符。这只是一个操作符名称（可能被方案限定）。该操作符被假定为具有与聚集函数（必须是一个单参数聚集函数）相同的输入数据类型。

注解

用于定义新聚集函数的普通函数必须先被定义。注意在这个 HashData 数据库的发行中，要求用于创建聚集的 sfunc、ffunc 以及 prefunc 函数被定义为 IMMUTABLE。

如果在窗口表达式中使用用户定义的聚集，必须为该聚集定义 prefunc 函数。

如果 HashData 数据库服务器配置参数 gp_enable_multiphase_agg 的值为 off，只有单层聚集会被执行。

任何用于自定义函数的已编译代码（共享库文件）在用户的 HashData 数据库阵列（Master 和所有 Segment）中的每一台主机上都必须被放置在相同的位置。这个位置还必须在 LD_LIBRARY_PATH 中，这样服务器可以定位到文件。

示例

下面的简单例子创建一个计算两列总和的聚集函数。

在创建聚集函数之前，创建两个被用作该函数的 SFUNC 和 PREFUNC 函数的函数。

这个函数被指定为该聚集函数中的 SFUNC 函数。

```
CREATE FUNCTION mysfunc_accum(numeric, numeric, numeric)
  RETURNS numeric
   AS 'select $1 + $2 + $3'
   LANGUAGE SQL
   IMMUTABLE
   RETURNS NULL ON NULL INPUT;
```

这个函数被指定为该聚集函数中的 PREFUNC 函数。

```
CREATE FUNCTION mypre_accum(numeric, numeric )
  RETURNS numeric
   AS 'select $1 + $2'
   LANGUAGE SQL
   IMMUTABLE
   RETURNS NULL ON NULL INPUT;
```

这个 CREATE AGGREGATE 命令会创建将两列相加的聚集函数。

```
CREATE AGGREGATE agg_prefunc(numeric, numeric) (
   SFUNC = mysfunc_accum,
   STYPE = numeric,
   PREFUNC = mypre_accum,
   INITCOND = 0 );
```

下列命令创建一个表，增加一些行并且运行聚集函数。

```
create table t1 (a int, b int) DISTRIBUTED BY (a);
insert into t1 values
   (10, 1),
   (20, 2),
   (30, 3);
select agg_prefunc(a, b) from t1;
```

这个 EXPLAIN 命令显示两阶段聚集。

```
explain select agg_prefunc(a, b) from t1;

QUERY PLAN
------------------------------------------------------------------------
Aggregate (cost=1.10..1.11 rows=1 width=32)
 -> Gather Motion 2:1 (slice1; segments: 2) (cost=1.04..1.08 rows=1
      width=32)
     -> Aggregate (cost=1.04..1.05 rows=1 width=32)
       -> Seq Scan on t1 (cost=0.00..1.03 rows=2 width=8)
 (4 rows)
```

兼容性

CREATE AGGREGATE 是一种 HashData 数据库的语言扩展。SQL 标准不提供用户定义的聚集函数。

另见

ALTER AGGREGATE，DROP AGGREGATE，CREATE FUNCTION

上级主题： SQL命令参考

CREATE CAST

定义一种新的造型。

概要

```
CREATE CAST (sourcetype AS targettype)
       WITH FUNCTION funcname (argtypes)
       [AS ASSIGNMENT | AS IMPLICIT]

CREATE CAST (sourcetype AS targettype) WITHOUT FUNCTION
       [AS ASSIGNMENT | AS IMPLICIT]
```

描述

CREATE CAST 定义一种新的造型。 一种造型指定如何在两种数据类型之间执行转换。例如，

```
SELECT CAST(42 AS text);
```

通过调用一个之前指定的函数将整数常量 42 转换成类型 text ，这种情况下是 text(int4)。 如果没有定义合适的造型， 该转换会失败。

两种类型可以是二进制可兼容，这表示他们可以不调用任何函数而转化为另外一个。这要求相应的值使用同样的内部表示，例如,类型 text 和 varchar 二进制可兼容。

默认情况下，只有一次显式造型请求才会调用造型， 形式是 `CAST(x AS typename)` 或 `x:: typename` 构造。

如果造型被标记为 AS ASSIGNMENT 那么在为一个目标数据类型的列赋值时会隐式地调用它。例如，假设 foo.f1 是一类型为 text 的列, 则:

```
INSERT INTO foo (f1) VALUES (42);
```

将被允许如果从类型 integer 到类型 text 的造型被标记为 AS ASSIGNMENT, 否则不会允许。 我们通常使用 赋值造型 来描述此类造型。

如果造型被标记为 AS IMPLICIT 那么可以在任何上下文中隐式地调用它，无论是赋值还是在一个表达式内部。 我们通常用术语 隐式造型 来描述这类造型。例如，因为 || 需要 text 操作数,

```
SELECT 'The time is ' || now();
```

将被允许如果从类型 timestamp 到类型 text 的造型被标记为 AS IMPLICIT. 否则，有必要明确地写出转换，例如

```
SELECT 'The time is ' || CAST(now() AS text);
```

对标记造型为隐式持保守态度是明智的。过多的隐式造型路径可能导致 HashData 数据库以令人吃惊的方式解释命令，或者由于有多种可能解释而根本无法解析命令。 一种好的经验是让一种造型只对于同一种一般类型分类中的类型间的信息保持转换隐式可调用。 例如， 从 int2 到 int4 的造型可以被合理地标记为隐式，, 但是从 float8 到 int4 的造型可能应该只能在赋值时使用。 跨类型分类的造型如 text 到 int4, 最好只被用于显式使用。

为了能够创建一个造型，用户必须拥有源或目标数据类型。要创建二进制兼容的造型，用户必须是超级用户。

参数

sourcetype

该造型的源数据类型的名称。

targettype

该造型的目标数据类型的名称。

funcname(argtypes)

被用于执行该造型的函数。函数名称可以用方案限定。如果没有被限定， 将在模式搜索路径中查找该函数。 函数的结果数据类型必须是该造型的目标数据类型。

造型实现函数可以具有 1 到 3 个参数。 第一个参数类型必须等于源类型或者能从源类型二进制强制得到。 第二个参数（如果存在）必须是类型 integer; 它接收与目标类型相关联的类型修饰符，如果没有类型修饰符，它会收到 -1 。第三个参数,（如果存在）必须是类型 boolean; 如果该造型是一种显式造型，它会收到 true 否则会收 false 。SQL 标准在某些情况中对显式和隐式造型要求不同的行为。这个参数被提供给必须实现这 类造型的函数。不推荐在设计自己的数据类型时用它）

通常，造型必须具有不同的源和目标数据类型。 但是，如果具有多个参数的造型实现函数，则允许声明具有相同源和目标类型的造型。 这用于表示系统目录中的特定类型的长度强制功能。 named函数用于将类型的值强制为由其第二个参数给出的类型修饰符值。 (由于语法目前仅允许某些内置数据类型具有类型修饰符，因此此功能对用户定义的目标类型无效。)

当一个造型具有不同的源和目标类型和一个需要多个参数的函数时，它支持从一个类型转换为另一个类型，并在一个步骤中应用长度强制。 当没有这样的条目可用时， 强制使用类型修饰符的类型涉及两个转换步骤，一个用于在数据类型之间转换，另一个用于应用修饰符。

WITHOUT FUNCTION

指示源类型可以二进制强制到目标类型，因此执行该造型不需要函数。

AS ASSIGNMENT

指示该造型可以在赋值的情况下被隐式调用。

AS IMPLICIT

指示该造型可以在任何上下文中被隐式调用。

注解

请注意，在此版本的 HashData 数据库中，用户定义的造型中使用的用户定义函数必须定义为IMMUTABLE 。必须将用于自定义函数的任何编译代码（共享库文件）放置在 HashData Database 数组（主数据库和所有段）中每个主机上的相同位置。 这个位置也必须在 LD_LIBRARY_PATH 以便服务器可以找到文件。

记住如果用户想要能够双向转换类型，用户需要在两个方向上都显式声明造型。

建议用户遵循在目标数据类型之后命名操行实现函数的惯例，因为内置的操行实现函数能被命名。 许多用户习惯于使用函数式符号来造型数据类型，也就是说 `typename(x)` 。

示例

要使用函数 `int4(text)` 创建一种从类型 text 到类型 int4 的赋值造型(在系统中这种造型已经被预定义。):

```
CREATE CAST (text AS int4) WITH FUNCTION int4(text);
```

兼容性

CREATE CAST 命令符合 SQL 标准， 不过 SQL 没有为二进制可兼容类型或者实现函数的额外参数做好准备。 AS IMPLICIT 也是一种 HashData 数据库的扩展。

另见

CREATE FUNCTION，CREATE TYPE， DROP CAST

上级主题： SQL命令参考

CREATE CONVERSION

定义一种新的编码转换。

概要

```
CREATE [DEFAULT] CONVERSION name FOR source_encoding TO
     dest_encoding FROM funcname
```

描述

CREATE CONVERSION 定义一种字符集编码间新的转换。转换名称可能用于转换功能来指定特定的编码转换。另外, 被标记为 DEFAULT 的转换将被自动地用于客户端和服务器之间的编码转换。为了这个目的，必须定义两个转换（从编码 A 到 B 以及从编码 B 到 A）。

要创建一个转换，用户必须拥有该函数上的 EXECUTE 特权以及目标模式上的 CREATE 特权。

参数

DEFAULT

表示这个转换是从源编码到目标编码的默认转换。在一个模式中对于每一个编码对，只应该有一个默认转换。

name

转换的名称，可以被方案限定。如果没有被方案限定，该转换被定义在当前模式中。在一个模式中，转换名称必须唯一。

source_encoding

源编码名称。

dest_encoding

目标编码名称。

funcname

被用来执行转换的函数。函数名可以被方案限定。如果没有，将在路径中查找该函数。该函数必须具有一下的特征：

```
conv_proc(
    integer,  -- source encoding ID
    integer,  -- destination encoding ID
    cstring,  -- source string (null terminated C string)
    internal, -- destination (fill with a null terminated C string)
    integer   -- source string length
) RETURNS void;
```

注解

请注意，在本版本的 HashData 数据库中，用户定义的转换中使用的用户定义函数必须定义为 IMMUTABLE。必须将用于自定义函数的任何编译代码（共享库文件）放置在 HashData 数据库数组（Master 和所有 Segment）中每个主机上的相同位置。 该位置也必须位于 LD_LIBRARY_PATH 中， 以便服务器可以找到文件。

示例

使用 myfunc 创建一个从编码 UTF8 到 LATIN1 的转换:

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

兼容性

在 SQL 标准中没有 CREATE CONVERSION 语句。

另见

ALTER CONVERSION，CREATE FUNCTION，DROP CONVERSION

上级主题： SQL命令参考

CREATE DATABASE

创建一个新的数据库。

概要

```
CREATE DATABASE name [ [WITH] [OWNER [=] dbowner]
                     [TEMPLATE [=] template]
                     [ENCODING [=] encoding]
                     [TABLESPACE [=] tablespace]
                     [CONNECTION LIMIT [=] connlimit ] ]
```

描述

CREATE DATABASE 创建一个新的数据库。 要创建一个新的数据库, 用户必须是超级用户或者有 CREATEDB 特权。

默认情况下，创建者即为新数据库的所有者。超级用户可以通过 OWNER 子句创建数据库归属其它用户。 他们甚至可以创建没有特殊权限的用户拥有的数据库。 有 CREATEDB 特权的非超级用户只能创建自己拥有的数据库。

默认情况下，将通过克隆标准系统数据库 template1 创建新数据库。 可以通过 TEMPLATE 名称指定不同的模板。 特别是, 通过写 TEMPLATE template0, 用户可以创建一个仅包含由 HashData 数据库预定义的标准对象的干净数据库。 这对用户希望避免复制添加到 template1 的任何安装对象中的内容非常有用。

参数

name

要创建的数据库的名称。

dbowner

将拥有新数据库的数据库用户的名称，或 DEFAULT 使用默认的所有者（执行该命令的用户）。

template

要创建新数据库的模板的名称，或 DEFAULT 使用默认模板 (template1)。

encoding

在新数据库中使用的字符集编码。指定一个字符串常量(例如 'SQL_ASCII')， 一个整数编码号，或 DEFAULT 使用默认编码。

tablespace

将与新数据库相关联的表空间的名称， 或 DEFAULT 使用模板数据库的表空间。 此表空间将是用于在此数据库中创建的对象的默认表空间。

connlimit

最大并发连接数可以使用。缺省值 -1 表示没有限制。

注解

CREATE DATABASE 不能在事务块内执行。

当用户通过指定其名称作为模板来复制数据库时，在复制数据库时，不会将其他会话连接到模板数据库。到模板数据库的新

连接被锁定，直到 CREATE DATABASE 完成。

CONNECTION LIMIT 不会对超级用户产生限制。

示例

要创建一个新的数据库：

```
CREATE DATABASE gpdb;
```

创建数据库 sales 由用户 salesapp 拥有，默认表空间为 salesspace:

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

创建数据库 music 支持 ISO-8859-1 字符集：

```
CREATE DATABASE music ENCODING 'LATIN1';
```

兼容性

SQL 标准中没有 CREATE DATABASE 语句。 数据库等同于目录，其创建是由实践定义的。

另见

ALTER DATABASE，DROP DATABASE

上级主题： SQL命令参考

CREATE DOMAIN

定义一个新的域。

概要

```
CREATE DOMAIN name [AS] data_type [DEFAULT expression]
       [CONSTRAINT constraint_name
       | NOT NULL | NULL
       | CHECK (expression) [...]]
```

描述

CREATE DOMAIN 创建一个新的域。域本质上是一种带有可选约束（在允许的值集合上的限制）的数据类型。 定义域的用户将成为它的拥有者。域的名称在其模式中的类型和域之间必须保持唯一。

域主要被用于把字段上的常用约束抽象到一个单一的位置以便维护。例如，几个表可能都包含电子邮件地址列，而且都要求相同的 CHECK 约束来验证地址的语法。可以为此定义一个域，而不是在每个表上都单独设置一个约束。

参数

name

要被创建的域的名称（可以被方案限定）。

data_type

域的底层数据类型。可以包括数组指示符。

DEFAULT *expression*

为该域数据类型的列指定一个默认值。该值是任何没有变量的表达式（但不允许子查询）。默认值表达式的数据类型必须匹配域的数据类型。如果没有指定默认值，那么默认值就是空值。默认值表达式将被用在任何没有指定列值的插入操作中。如果为一个特定列定义了默认值，它会覆盖与域相关的默认值。继而，域默认值会覆盖任何与底层数据类型相关的默认值。

CONSTRAINT *constraint_name*

一个约束的名称（可选）。如果没有指定，系统会生成一个名称。

NOT NULL

这个域的值通常不能为空值。

NULL

这个域的值允许为空值，这是默认值。这个子句只是为了与非标准 SQL 数据库相兼容而设计。在新的应用中不鼓励使用它。

CHECK (*expression*)

CHECK 子句指定该域的值必须满足的完整性约束或者测试。每一个约束必须是一个产生布尔结果的表达式。 它应该使用关键词 VALUE 来引用要被测试的值。目前, CHECK 表达式不能包含子查询，也不能引用 VALUE 之外的变量。

示例

创建 `us_zip_code` 数据类型。使用正则表达式测试来验证该值是否为有效的美国邮政编码。

```
CREATE DOMAIN us_zip_code AS TEXT CHECK
       ( VALUE ~ '^d{5}$' OR VALUE ~ '^d{5}-d{4}$' );
```

兼容性

CREATE DOMAIN 符合 SQL 标准。

另见

ALTER DOMAIN，DROP DOMAIN

上级主题： SQL命令参考

CREATE EXTENSION(new)

在 HashData 数据库中注册一个扩展名。

概要

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
  [ WITH ] [ SCHEMA schema_name ]
           [ VERSION version ]
           [ FROM old_version ]
           [ CASCADE ]
```

描述

CREATE EXTENSION 将新的扩展加载到当前数据库中。 不得加载的同名的扩展名。

加载扩展本质上等于运行扩展脚本文件。脚本通常创建新的 SQL 对象，例如函数，数据类型，操作符和索引支持方法。CREATE EXTENSION 命令还记录所有创建的对象的身份，如果 DROP EXTENSION 执行时可以再次删除它们。

加载扩展需要创建组件扩展对象所需的相同权限。对于大多数扩展，这意味着需要超级用户或数据库所有者权限。运行 CREATE EXTENSION 的用户成为后续特权检查目的的扩展的所有者，以及由扩展脚本创建的任何对象的所有者。

参数

IF NOT EXISTS

如果具有相同名称的扩展名已经存在，请勿抛出错误。 在这种情况下发出通知， 现在的扩展程序不能保证与已安装的扩展名相似。

extension_name

要安装的扩展名。该名称在数据库中必须是唯一的。 扩展名是根据扩展控制文件中的细节创建 `SHAREDIR/extension/extension_name.control` .

SHAREDIR 是安装共享数据目录, 例如 `/usr/local/ HashData -db/share/postgresql` 。 命令 `pg_config --sharedir` 显示目录。

SCHEMA *schema_name*

要在其中安装扩展对象的方案名称。这假设扩展允许其内容被重新定位。指定的方案必须已经存在。如果未指定，并且扩展控制文件未指定方案，则使用当前的默认对象创建方案。

如果扩展名在其控制文件中指定了方案参数，则该方案无法用 SCHEMA 子句覆盖。通常, 如果给出了SCHEMA 子句与扩展参数冲突，则会引发错误。但是, 如果还给出了 CASCADE 子句, 那么当发生冲突的时候 schema_name 会被忽略。给定的 schema_name 用于安装任何必要的但未在其控制文件中指定方案的扩展。

扩展本身不在任何方案中：扩展的未限定名称在数据库中必须是唯一的。 但属于该扩展的对象可以在一个方案中。

VERSION *version*

要安装的扩展版本。这可以写成标识符或字符串文字。默认版本是在扩展控制文件中指定的值。

FROM *old_version*

指定 FROM old_version 当用户尝试安装一个扩展名来替换 旧样式 模板， 该模块是未打包到扩展中的对象的集合。如果指定, CREATE EXTENSION 运行一个替代安装脚本， 将现有对象吸收到扩展中，而不是创建新对象。确保 SCHEMA 子句指定包含这些预先存在的对象的模式。

用于旧版本的值由扩展名作者确定， 如果有多个版本的旧式模块可以升级到扩展名，则可能会有所不同。对于 9.1 之前的

PostgreSQL 提供的标准附加模块，指定 unpackaged 为 old_version 将模块更新为扩展样式时。

CASCADE

自动安装依赖尚未安装的扩展名。递归检查依赖扩展名，并且这些依赖关系也自动安装。如果指定了 SCHEMA 子句，则该模式适用于所安装的扩展名和所有依赖扩展。指定的其他选项不适用于自动安装的从属扩展。特别是，安装从属扩展时，始终选择默认版本。

注解

目前可用于加载的扩展可以从 *pg_available_extensions* 或 *pg_available_extension_versions* 系统视图中识别。

在用户使用 CREATE EXTENSION 将扩展加载到数据库中之前，必须安装支持扩展文件，其中包括扩展控制文件和至少一个至少一个 SQL 脚本文件。 支持文件必须安装在所有 HashData 数据库主机的相同位置。

兼容性

CREATE EXTENSION 是 HashData 数据库扩展。

另见

ALTER EXTENSION，DROP EXTENSION

上级主题： SQL命令参考

CREATE EXTERNAL TABLE

定义一个新的外部表。

概要

```
CREATE [READABLE] EXTERNAL TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION ('file://seghost[:port]/path/file' [, ...])
       | ('gpfdist://filehost[:port]/file_pattern[#transform=trans_name]'
           [, ...]
       | ('gpfdists://filehost[:port]/file_pattern[#transform=trans_name]'
           [, ...])
       | ('gphdfs://hdfs_host[:port]/path/file')
       | ('s3://S3_endpoint[:port]/bucket_name/[S3_prefix]
            [region=S3-region]
           [config=config_file]')
     [ON MASTER]
     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] )]
         | 'AVRO'
         | 'PARQUET'
         | 'CUSTOM' (Formatter=<formatter_specifications>)
    [ ENCODING 'encoding' ]
      [ [LOG ERRORS] SEGMENT REJECT LIMIT count
      [ROWS | PERCENT] ]

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] )]
          | 'CUSTOM' (Formatter=<formatter specifications>)
     [ ENCODING 'encoding' ]
     [ [LOG ERRORS] SEGMENT REJECT LIMIT count
       [ROWS | PERCENT] ]
```

```
CREATE WRITABLE EXTERNAL TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION('gpfdist://outputhost[:port]/filename[#transform=trans_name]'
          [, ...])
      | ('gpfdists://outputhost[:port]/file_pattern[#transform=trans_name]'
          [, ...])
      | ('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'] )]
           | 'AVRO'
           | 'PARQUET'

           | 'CUSTOM' (Formatter=<formatter specifications>)
    [ ENCODING 'write_encoding' ]
    [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]

CREATE WRITABLE EXTERNAL TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION('s3://S3_endpoint[:port]/bucket_name/[S3_prefix]
            [region=S3-region]
            [config=config_file]')
      [ON MASTER]
      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'] )]

CREATE WRITABLE EXTERNAL WEB TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
    EXECUTE 'command' [ON ALL]
    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'] )]
           | 'CUSTOM' (Formatter=<formatter specifications>)
    [ ENCODING 'write_encoding' ]
    [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
```

描述

CREATE EXTERNAL TABLE 或 CREATE EXTERNAL WEB TABLE 在 HashData 数据库中创建一个新的可读外部表定义。可读外部表通常用于快速并行数据加载。定义外部表后，可以使用 SQL 命令直接（并行）查询其数据。例如，用户可以 select, join, sort 外部表数据。用户还可以为外部表创建视图。 DML 操作 (UPDATE, INSERT, DELETE, TRUNCATE) 在可读外部表上不可操作，用户不能在可读外部表上创建索引。

CREATE WRITABLE EXTERNAL TABLE 或 CREATE WRITABLE EXTERNAL WEB TABLE 在 HashData 数据库中创建一个新的可写外部表定义。可写外部表通常用于将数据从数据库卸载到一组文件或命名管道中。 可写外部 Web 表也可用于将数据输出到可执行程序。可写外部表也可以用作 HashData 并行 MapReduce 计算的输出目标。一旦可写外部表被定义，可以从数据库表中选择数据并将其插入到可写外部表中。可写外部表仅允许 INSERT 操作 – SELECT， UPDATE， DELETE 或 TRUNCATE 不被允许。

常规外部表和外部 Web 表之间的主要区别是它们的数据源。常规可读外部表访问静态平面文件，而外部 Web 表访问动态数据源 - 无论是在 Web 服务器上还是通过执行 OS 命令或脚本。

FORMAT 子句用于描述外部表文件的格式。有效的文件格式是分隔文本 (TEXT) 逗号分隔值(CSV) 格式, 与 PostgreSQL 可用的格式化选项类似 COPY 命令。 如果文件中的数据不使用默认列分隔符，转义字符，空字符串等，则必须指定其他格式选项，以便外部文件中的数据被 HashData 数据库正确读取。

为了 gphdfs 协议, 用户可以在 FORMAT 子句中指定 AVRO 或 PARQUET 读取或写入 Avro 或 Parquet 格式文件。

在创建写入或从 Amazon Web Services（AWS）S3 存储区中读取的外部表之前，必须配置 HashData 数据库以支持协议。S3 外部表可以使用 CSV 或文本格式的文件可写的 S3 外部表仅支持插入操作。

参数

READABLE | WRITABLE

指定外部表的类型，默认可读。可读外部表用于将数据加载到 HashData 数据库中。可写外部表用于卸载数据。

WEB

在 HashData 数据库中创建可读或可写的外部 Web 表定义。 有两种形式的可读外部 Web 表 - 那些访问文件的形式 `http://` 协议或通过执行 OS 命令访问数据的协议。 可写外部 Web 表将数据输出到可接受数据输入流的可执行程序。在执行查询期间，外部 Web 表不能重新计算。

s3 协议不支持外部 Web 表。但是，用户可以创建一个外部 Web 表，执行第三方工具直接从 S3 读取数据或向 S3 写入数据。

table_name

新外部表的名称。

column_name

在外部表定义中创建的列的名称。与常规表不同，外部表不具有列约束或默认值，因此不要指定。

LIKE *other_table*

该 LIKE 子句指定新的外部表自动复制所有列名，数据类型和 HashData 分发策略的表。 如果原始表指定了任何列约束或默认列值，那么它们将不会被复制到新的外部表定义中。

data_type

列的数据类型。

LOCATION *('protocol://host[:port]/path/file' [, ...])*

对于可读外部表，指定用于填充外部表或 Web 表的外部数据源的 URI。常规可读外部表允许 gpfdist 或 文件协议。 外部 Web 表允许 http 协议。 如果端口被省略, 端口 8080 被假定为 http 和 gpfdist 协议端口, 端口 9000 为 gphdfs 协议端口。 如果使用 gpfdist 协议, 路径 是相对于 gpfdist 服务文件的目录（启动 gpfdist 程序时，gpfdist 指定的目录)。此外， gpfdist 使用通配符或其他 C-style 模式匹配(例如，空格符为[[:space:]])表示目录中的多个文件。例如：

```
'gpfdist://filehost:8081/*'
'gpfdist://masterhost/my_load_file'
'file://seghost1/dbfast1/external/myfile.txt'
'http://intranet.example.com/finance/expenses.csv'
```

对于 gphdfs 协议, URI 的 LOCATION 不能包含以下四个字符：, ', <, >。如果 URI 包含任何此类字符，该 CREATE EXTERNAL TABLE 命令将返回错误。

如果用户正在使用 MapR 集群的 gphdfs 协议,用户需要指定一个特定的集群和文件：

- 要指定默认群集，MapR 配置文件中的第一个目录 `/opt/mapr/conf/mapr-clusters.conf` ，使用以下语法指定表的位置：

```
LOCATION ('gphdfs:///file_path')
```

该 file_path 是文件路径。

- 要指定配置文件中列出的另一个 MapR 集群，请使用以下语法指定文件：

```
LOCATION ('gphdfs:///mapr/cluster_name/file_path')
```

cluster_name 是在配置文件中指定的集群的名称并且 file_path 是文件的路径。

对于可写外部表,指定 gpfdist 进程或 S3 协议将会从 HashData 的 Segment 收集输出的数据并将其写入一个或多个命名文件。对于 gpfdist ，该路径是相对于 gpfdist 服务文件的目录 (启动 gpfdist 程序时指定的目录)。如果多个 gpfdist 列出了位置,发送数据的 Segment 将在可用的输出位置间均匀分配例如:

```
'gpfdist://outputhost:8081/data1.out',
'gpfdist://outputhost:8081/data2.out'
```

有两个 gpfdist 如上述示例中列出的位置，一半的 Segment 将其输出数据发送到 `data1.out` 文件，另一半发送到 `data2.out` 文件中。

在可选择的 *#transform=trans_name* 中, 用户可以指定要在加载或提取数据时应用的转换。 trans_name 是用户运行 gpfdist 实用程序指定的 YAML 配置文件的转换名称。

如果用户指定 gphdfs 协议去读取或写入文件到 Hadoop 文件系统（HDFS）中，你可以通过指定 FORMAT 子句的 AVRO 或 PARQUET 选项，来读写一个 AVRO 或 PARQUET 格式的文件。

有关指定Avro或Parquet文件位置的信息， 参阅 gphdfs协议的HDFS文件格式支持。

当用户用 s3 协议创建一个外部表时,只支持 TEXT 和 CSV 格式。 这些文件可以是 gzip 压缩格式。该S3协议识别 gzip 格式并解压缩文件。只支持 gzip 压缩格式。

用户可以指定 s3 协议，访问 Amazon S3 上的数据或 Amazon S3 兼容服务上的数据。在用户用 s3 协议创建外部表之前, 用户必须配置 HashData 数据库。有关配置 HashData 数据库的信息，参阅 s3 协议配置。

对于 s3 协议来说, LOCATION 子句指定数据文件为表上传的 S3 端点和存储桶名称。对于可写外部表，用户可以为插入到表中的数据创建新文件时指定一个可选的 S3 文件前缀。

如果为只读 S3 表指定了 S3_prefix ，则 s3 选择那些具有指定的 S3 文件前缀的文件。

> 注解: 尽管 S3_prefix 是语法的可选部分，但应始终为可写和只读 S3 表包含一个 S3 前缀，以分隔数据集作为 CREATE EXTERNAL TABLE 的语法。

LOCATION 子句中的 config 参数，指定包含 Amazon Web Services (AWS) 连接凭据和通信参数所需的 s3 协议配置参数文件的位置。 有关 s3 配置文件参数的信息， 参见 s3 协议配置文件。

这是一个使用 s3 协议定义的只读外部表的示例。

```
CREATE READABLE EXTERNAL TABLE S3TBL (date text, time text, amt int)
   LOCATION('s3://s3-us-west-2.amazonaws.com/s3test.example.com/dataset1/normal/
      config=/home/gpadmin/aws_s3/s3.conf')
   FORMAT 'csv';
```

S3 桶位于 S3 端点 `s3-us-west-2.amazonaws.com` ， s3 的桶名称为 `s3test.example.com` 。 桶中文件的 s3 前缀为 `/dataset1/normal/` 。配置文件位于所有 HashData 数据库的 Segment 的 `/home/gpadmin/s3/s3.conf` 中。

这个只读外部表示例指定相同的位置，并使用 region 参数指定 Amazon s3 区域 `us-west-2` 。

```
CREATE READABLE EXTERNAL TABLE S3TBL (date text, time text, amt int)
   LOCATION('s3://amazonaws.com/s3test.example.com/dataset1/normal/
      region=s3-us-west-2 config=/home/gpadmin/aws_s3/s3.conf')
   FORMAT 'csv';
```

该端口在 LOCATION 子句中的 URL 中是可选的。如果在 LOCATION 的子句的 URL 中未指定端口,则 s3 配置文件参数 encryption 会影响 s3 协议使用的端口 (端口 80 for HTTP 或端口 443 for HTTPS)。如果指定端口，则使用该端口，而不管 encryption 设置如何。例如，如果在 s3 配置文件中的LOCATION 子句中指定了端口 80，并且 encryption=true，则 HTTPS 请求将发送到端口 80 而不是 443，并记录警告。

LOCATION 子句中的 config 参数指定包含 Amazon Web Services（AWS）连接凭据和通信参数的所需 s3 协议配置文件的位置。 将 s3 的配置文件参数版本设置为 2 并在 LOCATION 子句中指定 region 参数。 (如果版本是 2， 则 LOCATION 子句中需要 region 参数。)在 LOCATION 子句中定义服务时，可以在 URL 中指定服务端点，并在 region 参数中指定服务器。这是一个 LOCATION 子句示例， 包含了一个 region 参数并指定了一个 Amazon S3 兼容服务。

```
LOCATION ('s3://test.company.com/s3test.company/dataset1/normal/ **region=local-test**
      config=/home/gpadmin/aws_s3/s3.conf')
```

当版本参数是 2 时， 还可以指定一个 Amazon S3 端点。 此示例指定了 Amazon S3 端点。

```
LOCATION ('s3://s3-us-west-2.amazonaws.com/s3test.example.com/dataset1/normal/ **region=us-west-2**
      config=/home/gpadmin/aws_s3/s3.conf')
```

更多关于s3协议的信息，参阅 HashData 数据库管理员指南中的 `s3://协议` 。

ON MASTER

将所有与表相关的操作限制在 HashData 主 Segment。 仅允许使用 s3 或自定义协议创建的可读写的外部表。gpfdist, gpfdists, gphdfs,和文件协议不支持 ON MASTER。

> 注意： 在使用 ON MASTER 子句创建的外部表进行阅读或写入时，请注意潜在的资源影响。将表操作仅限于 HashData 主分区时，可能会遇到性能问题。

EXECUTE 'command' [ON ...]

允许只读可读外部 Web 表或可写外部表。 对于可读取的外部 Web 表，要指定由 Segment 实例执行的 OS 命令。该命令可以是单个 OS 命令或脚本。 ON 子句用于指定哪些 Segment 实例将执行给定的命令。

- ON ALL 是默认值。该命令将由 HashData 数据库系统中所有 Segment 主机上的每个活动（主） Segment 实例执行。如果命令执行一个脚本，该脚本必须位于所有 Segment 主机上的相同位置，并且可由 HashData 超级用户 (gpadmin)执行。
- ON MASTER 仅在 master 主机上运行命令。

  注意： 当指定ON MASTER子句时，外部web表不支持日志记录 。
- ON number 表示命令将由指定数量的 Segment 执行。 HashData 数据库系统在运行时随机选择特定的 Segment。如果命令执行脚本，该脚本必须位于所有 Segment 主机上的相同位置，并且可由 HashData 超级用户(gpadmin)执行。
- HOST 意味着命令将由每个 Segment 主机上的一个 Segment 执行（每个 Segment 主机一次），而不管每个主机的活动 Segment 实例数如何。
- HOST segment_hostname 表示该命令将由指定 Segment 主机上的所有活动（主）Segment 实例执行。
- SEGMENT segment_id 表示命令只能由指定的 Segment 执行一次。 用户可以通过查看系统目录表 `gp_segment_configuration` 中的内容编号来确定 Segment 实例的 ID。 HashData 数据库的 Master的内容 ID始终为 -1。

对于可写外部表， EXECUTE 子句中指定的命令必须准备好将数据管道传输到其中。由于具有发送数据的所有 Segment 都将其输出写入指定的命令或程序，因此 ON 的唯一可选项为 ON ALL。

FORMAT 'TEXT | CSV | AVRO | PARQUET' (options)

指定外部或 Web 表数据的格式 - 纯文本 (TEXT)或或逗号分隔值(CSV) 格式。

仅使用 gphdfs 协议支持 AVRO 和 PARQUET 格式。

FORMAT 'CUSTOM' (formatter=formatter_specification)

指定自定义数据格式。 formatter_specification 指定用于格式化数据的函数，后跟格式化函数的逗号分隔参数。 格式化程序规范的长度，包括 Formatter= 的字符串的长度可以高达约 50K 字节。

有关使用自定义格式的信息，请参阅 HashData 数据库管理员指南中的 “装载和卸载数据”。

DELIMITER

指定单个 ASCII 字符，用于分隔每行（行）数据中的列。 默认值为 TEXT 模式下的制表符， CSV 格式为逗号。在可读外部表的 TEXT 模式下，对于将非结构化数据加载到单列表中的特殊用例，可以将分隔符设置为 OFF

对于 s3 协议，分隔符不能是换行符 (n) 或回车字符 (r)。

NULL

指定表示 NULL 值的字符串。在 TEXT 模式下，默认值是 \N（反斜杠-N）， CSV 模式中不含引号的空值。在 TEXT 模式下用户可能更希望不想将 NULL 值与空字符串区分开的情况下，也能使用 NULL 字符串。使用外部和 Web 表时，与此字符串匹配的任何数据项将被视为 NULL 值。使用外部和 Web 表时，与此字符串匹配的任何数据项将被视为 NULL 值。

作为 text 格式的示例，此 FORMAT 子句可用于指定两个单引号 ('') 的字符串为 NULL 值。

```
FORMAT 'text' (delimiter ',' null '''''' )
```

ESCAPE

指定用于 C 转义序列的单个字符 (例如 \n,\t,\100, 等等) 以及用于转义可能被视为行或列分隔符的数据字符。确保选择在实际列数据中的任何地方都不使用的转义字符。默认转义字符是文本格式文件的（反斜杠）和 csv 格式文件的 " (双引号)，但是可以指定其他字符来表示转义，也可以禁用文本转义通过指定值'OFF'作为转义值，格式化的文件对于诸如文本格式的 Web 日志数据之类的数据非常有用，这些数据具有许多不希望转义的嵌入式反斜杠。

NEWLINE

指定数据文件中使用的换行符 – LF (换行符, 0x0A)， CR (回车符号, 0x0D), 或 CRLF (回车加换行， 0x0D 0x0A)。 如果未指定， HashData 数据库的 Segment 将通过查看其接收的第一行数据并使用遇到的第一个换行符来检测换行类型。

HEADER

对于可读外部表，指定数据文件中的第一行是标题行（包含表列的名称），不应作为表的数据包含。如果使用多个数据源文件，则所有文件必须有标题行。

对于 s3 协议，标题行中的列名不能包含换行符 (n) 或回车符 (r)。

QUOTE

指定 CSV 模式的报价字符。 默认值为双引号 (")。

FORCE NOT NULL

在 CSV 模式下，处理每个指定的列，就像它被引用一样，因此不是一个 NULL 值。对于 CSV 模式中的默认空字符串（两个分隔符之间不存在），这将导致将缺少的值作为零长度字符串计算。

FORCE QUOTE

在可写外部表的 CSV 模式下，强制引用用于每个指定列中的所有非 NULL 值。 NULL 输出从不引用。

FILL MISSING FIELDS

在可读外部表的 TEXT 和 CSV 模式下， 指定 FILL MISSING FIELDS 时，当一行数据在行或行的末尾缺少数据字段时，将丢失尾字段值设置为 NULL （而不是报告错误）。空行，具有 NOT NULL约束的字段和行上的尾随分隔符仍然会报告错误。

ENCODING *'encoding'*

字符集编码用于外部表。 指定一个字符串常量 (如 'SQL_ASCII'), 一个整数编码号或者 DEFAULT 来使用默认的客户端编码。参见字符集支持。

LOG ERRORS

这是一个可选的子句，可以在 SEGMENT REJECT LIMIT 子句之前记录有关具有格式错误的行的信息。 错误日志信息在内部存储，并使用 HashData 数据库内置 SQL 函数 `gp_read_error_log()` 访问。

SEGMENT REJECT LIMIT *count* [ROWS | PERCENT]

在单行错误隔离模式下运行 COPY FROM 操作。如果输入行具有格式错误，则它们将被丢弃，前提是在加载操作期间在任何 HashData 的 Segment 实例上未达到拒绝限制计数。拒绝限制计数可以指定为行数（默认值）或总行数百分比（1-100）。如果使用 PERCENT，则每个 Segment 只有在处理了参数 `gp_reject_percent_threshold` 所指定的行数之后才开始计算坏行百分比。 `gp_reject_percent_threshold` 的默认值为 300 行。诸如违反 NOT NULL, CHECK， 或 UNIQUE 约束的约束错误仍将以 “all-or-nothing” 输入模式进行处理。 如果没有达到限制，所有好的行将被加载，任何错误行被丢弃。

> 注解：读取外部表时，如果未首先触发 SEGMENT REJECT LIMIT 或未指定 SEGMENT REJECT LIMIT，则 HashData 数据库将限制可能包含格式错误的初始行数。如果前 1000 行被拒绝，则 COPY 操作将被停止并回滚。

可以使用 HashData 数据库服务器配置参数 `gp_initial_bad_row_limit` 更改初始拒绝行的数量限制。有关参数的信息，请参阅服务器配置参数。

DISTRIBUTED BY (column, [...])

DISTRIBUTED RANDOMLY

用于为可写外部表声明 HashData 数据库分发策略。默认情况下，可写外部表是随机分布的。如果要从中导出数据的源表具有散列分发策略，则为可写外部表定义相同的分发密钥列可以通过消除在互连上移动行的需要来改善卸载性能。 当用户发出诸如 `INSERT INTO wex_table SELECT * FROM source_table` ，的卸载命令时，如果两个表具有相同的散列分布策略，则可以将卸载的行直接从 Segment 发送到输出位置。

示例

在端口 8081 的后台启动 gpfdist 文件服务器程序，从目录 `/var/data/staging` 提供文件：

```
gpfdist -p 8081 -d /var/data/staging -l /home/gpadmin/log &
```

创建一个名为 `ext_customer` 的可读外部表使用的 gpfdist 协议和 gpfdist 目录中找到的任何文本格式的文件（*.txt）文件格式化为管道 (|)作为列分隔符，空格为 NULL。 也可以在单行错误隔离模式下访问外部表：

```
CREATE EXTERNAL TABLE ext_customer
   (id int, name text, sponsor text)
   LOCATION ( 'gpfdist://filehost:8081/*.txt' )
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
   LOG ERRORS SEGMENT REJECT LIMIT 5;
```

创建与上述相同的可读外部表定义，但使用 CSV 格式的文件：

```
CREATE EXTERNAL TABLE ext_customer
   (id int, name text, sponsor text)
   LOCATION ( 'gpfdist://filehost:8081/*.csv' )
   FORMAT 'CSV' ( DELIMITER ',' );
```

使用文件协议和具有标题行的几个 CSV 格式的文件创建名为 *ext_expenses* 的可读外部表：

```
CREATE EXTERNAL TABLE ext_expenses (name text, date date,
amount float4, category text, description text)
LOCATION (
'file://seghost1/dbfast/external/expenses1.csv',
'file://seghost1/dbfast/external/expenses2.csv',
'file://seghost2/dbfast/external/expenses3.csv',
'file://seghost2/dbfast/external/expenses4.csv',
'file://seghost3/dbfast/external/expenses5.csv',
'file://seghost3/dbfast/external/expenses6.csv'
)
FORMAT 'CSV' ( HEADER );
```

创建一个可读的外部 Web 表，每个 Segment 主机执行一次脚本：

```
CREATE EXTERNAL WEB TABLE log_output (linenum int, message
text)  EXECUTE '/var/load_scripts/get_log_data.sh' ON HOST
 FORMAT 'TEXT' (DELIMITER '|');
```

创建一个名为 sales_out 的可写外部表，它使用 gpfdist 将输出数据写入名为 sales.out 的文件。 文件格式化为管道 (|) 作为列分隔符，空格空。

```
CREATE WRITABLE EXTERNAL TABLE sales_out (LIKE sales)
   LOCATION ('gpfdist://etl1:8081/sales.out')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
   DISTRIBUTED BY (txn_id);
```

创建一个可写的外部 Web 表，其将 Segment 接收的输出数据管理到名为 `to_adreport_etl.sh` 的可执行脚本：

```
CREATE WRITABLE EXTERNAL WEB TABLE campaign_out
(LIKE campaign)
 EXECUTE '/var/unload_scripts/to_adreport_etl.sh'
 FORMAT 'TEXT' (DELIMITER '|');
```

使用上面定义的可写外部表来卸载所选数据：

```
INSERT INTO campaign_out SELECT * FROM campaign WHERE
customer_id=123;
```

注解

指定 LOG ERRORS 子句时，HashData 数据库会捕获读取外部表数据时发生的错误。用户可以查看和管理捕获的错误日志数据。

- 使用内置的SQL函数 gp_read_error_log('table_name')。它需要对table_name具有 SELECT特权。 此示例显示使用 COPY命令加载到表 ext_expenses 中的数据的错误日志信息：

  ```
  SELECT * from gp_read_error_log('ext_expenses');
  ```

  有关错误日志格式的信息，请参阅 HashData 数据库管理员指南中的在错误日志中查看不正确的行。

  如果 table_name 不存在，该函数返回 FALSE。

- 如果指定的表存在错误日志数据，新的错误日志数据将附加到现有的错误日志数据。错误日志信息不会复制到镜像 Segment。
- 使用内置的 SQL 函数 `gp_truncate_error_log('table_name')` 删除 table_name 的错误日志数据。它需要表所有者权限,此示例删除将数据移动到表中时捕获的错误日志信息 ext_expenses:

  ```
  SELECT gp_truncate_error_log('ext_expenses');
  ```

  如果 table_name 不存在，该函数返回 FALSE。

  指定 * 通配符以删除当前数据库中现有表的错误日志信息。指定字符串 `*.*` 以删除所有数据库错误日志信息，包括由于以前的数据库问题而未被删除的错误日志信息。如果指定 *，则需要数据库所有者权限。 如果指定了 `*.*` 则需要操作系统超级用户权限。

HDFS 文件格式支持 gphdfs 协议

如果用户指定了 gphdfs 协议将文件读取或写入 Hadoop 文件系统（HDFS），则可以通过使用 FORMAT 子句指定文件格式来读取或写入 Avro 或 Parquet 格式文件。

要从 Avro 或 Parquet 文件读取数据或将数据写入到 Avro 或 Parquet 文件中，用户可以使用 CREATE EXTERNAL TABLE

命令创建一个外部表，并指定 LOCATION 子句中的 Avro 文件的位置和 FORMAT 子句中 'AVRO' 。此示例用于从 Avro 文件读取的可读外部表。

```
CREATE EXTERNAL TABLE tablename (column_spec) LOCATION ( 'gphdfs://location') **FORMAT 'AVRO'**
```

该位置可以是包含一组文件的文件名或目录。对于文件名，用户可以指定通配符 `*` 来匹配任意数量的字符。 如果位置在读取文件时指定了多个文件，HashData 数据库将使用第一个文件中的模式作为其他文件的模式。

作为 LOCATION 参数的一部分，用户可以指定读取或写入文件的选项。在文件名后，用户可以使用 HTTP 查询字符串语法指定参数 \? 并在字段值对之间使用 \&。

对于此示例 LOCATION 参数，此 URL 设置 Avro 格式可写外部表的压缩参数。

```
'gphdfs://myhdfs:8081/avro/singleAvro/array2.avro?compress=true&compression_type=block&codec=snappy' FORMAT 'AVRO'
```

有关使用外部表读取和编写 Avro 和 Parquet 格式文件的信息，请参阅 HashData 数据库管理员指南中的 “装载和卸载数据”。

Avro文件

对于可读外部表，唯一有效的参数是 schema。阅读多个 Avro 文件时，可以指定包含 Avro 模式的文件。

对于可写外部表，可以指定 schema, namespace, 压缩参数。

表**1. Avro** 格式外部表 **location** 参数

参数	值	可读/可写	默认值
schema	`URL_to_schema_file`	Read and Write	None. For a readable external table,The specified schema overrides the schema in the Avro file. See "Avro Schema Overrides" If not specified, Hashdata Database uses the Avro file schema.For a writable external table,Uses the specified schema when creating the Avro file.If not specified, Hashdata Database creates a schema according to the external table definition.
namespace	`avro_namespace`	Write only	public.avro If specified, a valid Avro namespace.
compress	true or false	Write only	false
`compression_type`	block	Write only	Optional. For avro format, compression_type must be block if compress is true.
codec	deflate or snappy	Write only	deflate
codec_level (deflate codec only)	integer between 1 and 9	Write only	6 The level controls the trade-off between speed and compression. Valid values are 1 to 9, where 1 is the fastest and 9 is the most compressed.

这组参数指定 snappy 压缩:

```
'compress=true&codec=snappy'
```

这两组参数指定 deflate 压缩并且等效:

```
'compress=true&codec=deflate&codec_level=1'
'compress=true&codec_level=1'
```

gphdfs Avro文件的限制

对于 HashData 数据库可写外部表定义，列不能指定 NOT NULL 子句。

HashData 数据库仅支持 Avro 文件中的单个顶级模式，或者在 CREATE EXTERNAL TABLE 命令中使用 schema 参数指定。 如果 HashData 数据库检测到多个顶级模式，则会返回错误。

HashData 数据库不支持 Avro 地图数据类型，并在遇到错误时返回错误。

当 HashData 数据库从 Avro 文件读取数组时，数组将转换为文本文本值。例如，数组 [1,3] 转换为 '{1,3}'。

支持用户定义的类型（UDT），包括数组UDT。 对于可写外部表，类型将转换为字符串。

Parquet 文件

对于外部表，可以在该位置指定的文件之后添加参数。此表列出了有效的参数和值。

表 **2. Parquet** 格式外部表位置参数

选项	值	可读/可写	默认值
schema	`URL_to_schema`	Write only	None.If not specified, HashData Database creates a schema according to the external table definition.
pagesize	> 1024 Bytes	Write only	1 MB
rowgroupsize	> 1024 Bytes	Write only	8 MB
version	v1, v2	Write only	v1
codec	UNCOMPRESSED, GZIP, LZO, snappy	Write only	UNCOMPRESSED
dictionaryenable	true, false	Write only	false
dictionarypagesize	> 1024 Bytes	Write only	512 KB

> 注解：
>
> 1. 创建一个内部字典。 如果文本列包含相似或重复的数据，启用字典可能会改进 Parquet 文件压缩。

s3协议配置

s3 协议与 URI 一起使用以指定 Amazon Simple Storage Service（Amazon S3）存储区中文件的位置。该协议创建或下载 LOCATION 子句指定的所有文件。每个 HashData 数据库的 Segment 实例一次使用多个线程下载或上传一个文件。

配置和使用S3外部表

按照以下基本步骤配置 S3 协议并使用 S3 外部表，使用可用链接获取更多信息：

1. 配置每个数据库以支持 s3 协议：
    i. 在每个将使用 s3 协议, 访问 S3 存储区的数据库中，为 s3 协议库创建读写功能：

```
CREATE OR REPLACE FUNCTION write_to_s3() RETURNS integer AS
   '$libdir/gps3ext.so', 's3_export' LANGUAGE C STABLE;
```

```
CREATE OR REPLACE FUNCTION read_from_s3() RETURNS integer AS
   '$libdir/gps3ext.so', 's3_import' LANGUAGE C STABLE;
```

i. 在访问 S3 存储区的每个数据库中，声明 s3 协议并指定在上一步中创建的读写功能：

```
CREATE PROTOCOL s3 (writefunc = write_to_s3, readfunc = read_from_s3);
```

> 注解：协议名称 s3 必须与为创建用于访问 S3 资源的外部表指定的 URL 的协议相同。

每个 HashData 数据库的 Segment 实例调用相应的功能。所有 Segment 主机都必须具有访问 S3 存储桶的权限。

2. 在每个 HashData 数据库的 Segment，创建并安装 s3 协议配置文件：

i. 使用 gpcheckcloud 实用程序创建模板 s3 协议配置文件：

```
gpcheckcloud -t > ./mytest_s3.config
```

i. 编辑模板文件以指定连接到 S3 位置所需的 accessid 和 secret。

ii. 将文件复制到所有主机上所有 HashData 数据库 Segment 的相同位置和文件名。默认的文件位置是 `gpseg_data_dir/gpseg_prefixN/s3/s3.conf` 。 `gpseg_data_dir` 是 HashData 数据库 Segment 的数据目录的路径， `gpseg_prefix` 是 Segment 前缀， N 是 Segment 的 ID。在初始化 HashData 数据库系统时，会设置 Segment 数据目录，前缀和ID。

如果将文件复制到其他位置或文件名，则必须使用 s3 协议 URL 中的配置参数。

iii. 使用 gpcheckcloud 实用程序验证与 S3 存储桶的连接：

```
gpcheckcloud -c "s3://<s3-endpoint>/<s3-bucket> config=./mytest_s3.config"
```

指定系统配置文件的正确路径，以及要检查的 S3 端点名称和存储桶。 gpcheckcloud 尝试连接到 S3 端点并列出 s3 存储区中的任何文件（如果可用）。 成功的连接以消息结束：

```
Your configuration works well.
```

用户可以选择使用 gpcheckcloud 来验证是否从 S3 存储区中上载和下载。

3. 完成以前创建和配置 S3 协议的步骤之后，用户可以在 CREATE EXTERNAL TABLE 命令中指定 S3 协议 URL 来定义 S3 外部表。 对于只读 S3 表，URL 定义用于选择构成 S3 表的现有数据文件的位置和前缀。 例如：

```
CREATE READABLE EXTERNAL TABLE S3TBL (date text, time text, amt int)
   LOCATION('s3://s3-us-west-2.amazonaws.com/s3test.example.com/dataset1/normal/
      config=/home/gpadmin/aws_s3/s3.conf')
   FORMAT 'csv';
```

对于可写的 S3 表，协议 URL 定义了 HashData 数据库存储表的数据文件的 S3 位置，以及为表 INSERT 操作创建文件时使用的前缀。例如：

```
CREATE WRITABLE EXTERNAL TABLE S3WRIT (LIKE S3TBL)
   LOCATION('s3://s3-us-west-2.amazonaws.com/s3test.example.com/dataset1/normal/
      config=/home/gpadmin/aws_s3/s3.conf')
   FORMAT 'csv';
```

s3协议限制

这些是 s3 的协议限制：

- 只支持 S3 路径样式的 URL。

```
s3://S3_endpoint/bucketname/[S3_prefix]
```

- 只支持 S3 端点。该协议不支持 S3 桶的虚拟主机（将域名绑定到 S3 桶）。

- 支持 AWS 签名版本 2 和版本 4 的签名过程。
- CREATE EXTERNAL TABLE 命令的 LOCATION 子句中只支持一个 URL 和可选的配置文件。
- 如果在 CREATE EXTERNAL TABLE 命令中未指定 NEWLINE 参数，则在特定前缀的所有数据文件中，换行符必须相同。如果某些具有相同前缀的数据文件中的换行字符不同，则对文件的读取操作可能会失败。
- 对于可写入的 S3 外部表，只支持 INSERT 操作。不支持 UPDATE, DELETE, 和 TRUNCATE 操作。
- 由于 Amazon S3 允许最多 10,000 个部分用于多部分上传,所以最大支持每个数据块为 128MB HashData 数据库 Segment 的可写入 s3 表的最大插入大小 1.28TB。 用户必须确保 chunksize 设置可以支持表的预期表大小。 有关上传到 S3 的更多信息，请参阅 S3 文档中的多部分上传概述 。
- 利用 HashData 数据库执行的并行处理 Segment 实例中，只读 S3 表的 S3 位置中的文件的大小应类似，文件数量应允许多个 Segment 从 S3 位置下载数据。 例如，如果 HashData 数据库系统由16个 Segment 组成，网络带宽足够，在 S3 位置创建 16 个文件允许每个 Segment 从一个文件下载 S3 位置。 相比之下，如果位置只包含 1 或 2 个文件，则只有 1 个或 2 个 Segment 下载数据。

关于S3协议URL

对于 s3 协议，用户可以在 CREATE EXTERNAL TABLE 命令的 LOCATION 子句中指定文件的位置和可选的配置文件位置。语法如下：

```
's3://S3_endpoint[:port]/bucket_name/[S3_prefix] [region=S3_region] [config=config_file_location]'
```

s3 协议要求用户指定 S3 端点和 S3 桶名称。 每个 HashData 数据库 Segment 实例必须能够访问 S3 位置。 可选的 `S3_prefix` 值用于为只读 S3 表选择文件，也可用作为 S3 可写表上传文件时使用的文件名前缀。

> 注意： HashData 数据库 s3 协议 URL 必须包含 S3 端点主机名。

要在 LOCATION 子句中指定 ECS 端点（Amazon S3兼容服务），必须将 s3 配置文件参数设置为版本 2。 版本参数控制 LOCATION 中是否使用 region 参数。 用户还可以在版本参数为 2 时指定 Amazon S3 位置。

> 注意： 虽然 `S3_prefix` 是语法的可选部分， 但是用户应该始终为可写和只读 S3 表包含一个 S3 前缀，以分隔数据集作为 CREATE EXTERNAL TABLE 语法的一部分。

对于可写的 S3 表，s3 协议 URL 指定了 HashData 数据库上传表的数据文件的端点和存储桶名称。 对于上传文件的 S3 用户标识， S3 桶权限必须为 Upload/Delete。 由于将数据插入到表中，S3 文件前缀用于上传到 S3 位置的每个新文件。

对于只读 S3 表，S3 文件前缀是可选的。 如果指定了 `S3_prefix` ，则 s3 协议将以指定前缀开头的所有文件选择为外部表的数据文件。 s3 协议不使用斜杠字符 (/) 作为分隔符，因此前缀后面的斜杠字符将被视为前缀本身的一部分。

例如，考虑以下 5 个文件，每个文件都有名为 `S3_endpoint` 的 `s3-us-west-2.amazonaws.com` 和 `bucket_name` test1:

```
s3://s3-us-west-2.amazonaws.com/test1/abc
s3://s3-us-west-2.amazonaws.com/test1/abc/
s3://s3-us-west-2.amazonaws.com/test1/abc/xx
s3://s3-us-west-2.amazonaws.com/test1/abcdef
s3://s3-us-west-2.amazonaws.com/test1/abcdefff
```

- 如果 S3 URL 是作为 `s3://s3-us-west-2.amazonaws.com/test1/abc` 提供的， 那么 abc 前缀会选择所有 5 个文件。
- 如果 S3 URL 被提供为 `s3://s3-us-west-2.amazonaws.com/test1/abc/` ，则 `abc/ prefix` 选择文件 `s3://s3-us-west-2.amazonaws.com/test1/abc/` 和 `s3://s3-us-west-2.amazonaws.com/test1/abc/xx` .
- 如果 S3 URL 以 `s3://s3-us-west-2.amazonaws.com/test1/abcd` 提供，则 abcd 前缀会选择文件 `s3://s3-us-west-2.amazonaws.com/test1/abcdef` 和 `s3://s3-us-west-2.amazonaws.com/test1/abcdefff`

`S3_prefix` 中不支持通配符；然而，S3 前缀的功能就好像一个通配符紧随着前缀本身。

由 S3 URL (S3_endpoint/bucket_name/S3_prefix) 选择的所有文件都用作外部表的源，因此它们必须具有相同的格式。 每个文件还必须包含完整的数据行。数据行不能在文件之间分割。对于正在访问文件的 S3 用户标识，S3 文件权限必须是 Open/Download 和View

有关 Amazon S3 端点的信息，请参阅 http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region。 有关 S3 桶和

文件夹的信息，请参阅 Amazon S3 文档https://aws.amazon.com/documentation/s3/。 有关 S3 文件前缀的信息，请参阅Amazon S3文档列出密钥分层使用前缀和分隔符。

config 参数指定包含AWS连接凭据和通信参数的所需 s3 协议配置文件的位置。

s3协议配置文件

使用 s3 协议时，所有的 HashData 数据库 Segment都需要一个s3 协议配置文件。 默认位置是：

```
gpseg_data_dir/gpseg-prefixN/s3/s3.conf
```

`gpseg_data_dir` 是 HashData 数据库 Segment 数据目录的路径， gpseg-prefix 是 Segment 前缀， N 是 Segment 的 ID。在初始化 HashData 数据库系统时，会设置 Segment 数据目录，前缀和 ID。

如果 Segment 主机上有多个 Segment 实例，则可以通过在每个 Segment 主机上创建单个位置来简化配置。 然后可以使用 s3 协议 LOCATION 子句中的 config 参数指定位置的绝对路径。但是，请注意，只读和可写S3外部表对于它们的连接使用相同的参数值。 如果要为只读和可写 S3 表配置协议参数不同，则在创建每个表时，必须使用两种不同的 s3 协议配置文件，并在 CREATE EXTERNAL TABLE语句中指定正确的文件。

次示例指定 gpadmin 主目录的 the s3 目录中的单个文件位置：

```
config=/home/gpadmin/s3/s3.conf
```

主机上的所有 Segment 实例都使用文件 `/home/gpadmin/s3/s3.conf` 。

s3 协议配置文件是由 [default] 部分和参数组成的文本文件。这是一个配置文件示例：

```
[default]
secret = "secret"
accessid = "user access id"
threadnum = 3
chunksize = 67108864
```

用户可以使用 HashData 数据库 gpcheckcloud 实用程序来测试S3配置文件。

s3配置文件参数

accessid

必需。AWS S3 ID 访问s3的桶

secret

必需。AWS S3的S3代码访问S3存储区。

autocompress

对于可写入的S3外部表，此参数指定在上传到S3之前是否压缩文件（使用gzip）。如果不指定此参数，文件将默认压缩。

chunksize

每个Segment线程用于读取或写入到S3服务器的缓冲大小。默认值为64 MB。最小为8MB，最大为128MB。

当将数据插入到可写S3表中时，每个 HashData 数据库Segment将数据写入其缓冲区(using multiple threads up to the threadnum value) 直到缓冲区满为止，然后将缓冲区写入S3存储区中的一个文件。 然后根据需要在每个Segment上重复该过程，直到插入操作完成。

由于Amazon S3允许最多10,000个部分用于多部分上传，因此最小的 块 大小为 8MB， 支持每个 HashData 数据库Segment的最大插入大小为80GB。 最大的 块大小是128MB 支持每个 HashData 数据库Segment的最大插入大小为1.28TB. 对于可写的S3表，用户必须确保 chunksize 设置可以支持表的预期表格大小。

encryption

使用通过安全套接字层（SSL）保护的连接。 默认值为true。 值为true, t, on, yes, 和 y（不区分大小写）被视为true。 任何其他值被视为false。

如果在 CREATE EXTERNAL TABLE命令的 LOCATION 子句中的URL中未指定端口，则配置文件的 加密参数会影响 s3 协议使用的端口（HTTP端口80或HTTPS端口443）。 如果指定端口，则使用该端口，而不管加密设置如何。

low_speed_limit

上传/下载速度下限，以字节/秒为单位。 默认速度为10240（10K）。 如果上传或下载 速度低于由 low_speed_time, 指定的时间长于限制，则连接将中止并重试。 3次重试后，s3 协议返回错误。 值0指定没有下限。

low_speed_time

当连接速度小于 low_speed_limit时，此参数指定中止上载或从S3存储桶下载之前等待的时间（以秒为单位）。默认值为60秒。 值0指定没有时间限制。

server_side_encryption

已为桶配置的S3服务器端加密方法。 HashData 数据库仅支持使用Amazon S3管理的密钥进行服务器端加密，由配置参数值sse-s3标识。 默认情况下禁用服务器端加密(无)。

threadnum

在将数据上传到S3桶中的数据或从S3桶中下载数据时，Segment可以创建的最大并发线程数。 默认值为4.最小值为1，最大值为8。

verifycert

控制在客户端和s3 数据源之间通过HTTPS建立加密通信时，s3协议如何处理身份验证。 该值为 true 或 false。默认值为 true。

- verifycert=false - 忽略身份验证错误，并通过HTTPS进行加密通信。
- verifycert=true - 需要通过HTTPS进行加密通信的有效身份验证（特有的证书）。

将值设置为 false 可用于测试和开发环境， 以允许通信而不更改证书。

警告： 将值设置为 false通过在客户端和S3数据存储之间建立通信时忽略无效凭据来暴露安全风险。

version

指定 CREATE EXTERNAL TABLE 命令的LOCATION 子句中 指定的信息版本。值为1或 2。 默认值为1。

如果值为1，则LOCATION 子句支持Amazon S3 URL， 并且不包含 region 参数。如果值为 2，the LOCATION 子句支持S3兼容服务，并且必须包含 region 参数。region 参数指定S3数据源区域。 对于这个S3 URL s3://s3-us-west-2.amazonaws.com/s3test.example.com/dataset1/normal/, AWS S3区域是 us-west-2。

如果版本 是1或者没被指定，则这是一个示例 LOCATION 子句， CREATE EXTERNAL TABLE命令指定一个Amazon S3端点。

LOCATION ('s3://s3-us-west-2.amazonaws.com/s3test.example.com/dataset1/normal/
config=/home/gpadmin/aws_s3/s3.conf')

如果版本是2, 一个示例 LOCATION子句，其中包括AWS兼容服务的 region 参数。 .

LOCATION ('s3://test.company.com/s3test.company/test1/normal/ region=local-test
config=/home/gpadmin/aws_s3/s3.conf')

如果版本 是2，LOCATION 子句也可以指定一个Amazon S3端点。 此示例指定使用该Amazon S3端点 region 参数。

LOCATION ('s3://s3-us-west-2.amazonaws.com/s3test.example.com/dataset1/normal/ region=us-west-2
config=/home/gpadmin/aws_s3/s3.conf')

注解： 当上传或下载S3文件时， HashData 数据库可能需要在每个分Segment主机上最多可以使用 threadnum * chunksize 的内存。 当用户配置总体的 HashData 数据库内存时，请考虑此s3 协议内存要求，并根据需要增加 gp_vmem_protect_limit 的值。

兼容性

CREATE EXTERNAL TABLE 是 HashData 数据库扩展。SQL标准没有规定外部表。

另见

CREATE TABLE AS, CREATE TABLE, COPY, SELECT INTO, INSERT

上级主题： SQL命令参考

CREATE FUNCTION

建立函数

定义一个新函数。

概要

```
CREATE [OR REPLACE] FUNCTION name
    ( [ [argmode] [argname] argtype [ { DEFAULT | = } defexpr ] [, ...] ] )
      [ RETURNS { [ SETOF ] rettype
        | TABLE ([{ argname argtype | LIKE other table }
          [, ...]])
        } ]
    { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINE
    | COST execution_cost
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol' } ...
    [ WITH ({ DESCRIBE = describe_function
           } [, ...] ) ]
```

描述

CREATE FUNCTION 定义了一个新的函数。CREATE OR REPLACE FUNCTION 将会要么创建一个新的函数，要么替换一个存在的定义。

新函数的名称不能与同一模式中的具有相同参数的任何现有函数匹配。但是，不同参数类型的函数可能会共享一个名称（重载)。

为了更新一个已经存在函数的定义，可以使用 CREATE OR REPLACE FUNCTION 语句。不可能以这种方式更改函数的名称或参数类型（这实际上会创建一个新的，不同的函数）。此外，CREATE OR REPLACE FUNCTION 不会让用户更改现有函数的返回类型。为此，用户必须删除并重新创建该函数。如果用户删除然后重建一个函数，用户将不得不删除引用该旧函数的现有对象（如规则，视图，触发器等等）用户可以使用 CREATE OR REPLACE FUNCTION 去改变一个函数的定义，而不会破坏引用该函数的对象。

VOLATILE 和 **STABLE** 函数的限制使用

为了防止数据在 HashData 数据库的各个 segments 中变得不同步，任何分类为 STABLE 或 VOLATILE 的函数，如果它包含 SQL 或以任何方式修改数据库，都不能在 segment 级别执行。例如，random() 或 timeofday() 函数不允许在 HashData 数据库中的分布式数据上执行，因为他们可能会导致 segment 实例之间的数据不一致。

为了确保数据的一致性，VOLATILE 和 STABLE 函数可以安全地用在主机上进行评估和执行的语句中。例如，接下来的语句总是在主机（master）上执行（没有 FROM 子句的语句）：

```
SELECT setval('myseq', 201);
SELECT foo();
```

在一个语句包含分布式表的 FROM 子句而且 FROM 子句中使用的函数只简单的返回一组行的情况下，该操作则可以在 segment 上允许执行。

```
SELECT * FROM foo();
```

此规则的一个例外就是返回表引用（rangeFuncs）或使用 refCursor 数据类型的函数。请注意用户不能从 HashData 数据库中的任何类型函数返回 refcursor 。

参数

name

创建函数的名称（可选方案限定）

argmode

参数的模式：要么 IN，OUT，INOUT， 或者 VARIADIC。只有 OUT 参数可以遵循声明为 VARIADIC 的参数。如果省略，默认值为 IN。

argname

参数的名字。一些语言（目前只有 PL/pgSQL）能让用户在函数体中使用这个名字。 对于其他语言输入参数的名称只是额外的文档。但是，输出参数的名称很重要，因为它定义了结果行类型中的列名。（如果用户省略了输出参数的名称，系统将选择默认列名称。）

argtype

函数参数的数据类型（可选方案限定），如果有，该参数可以是 base, composite, 或 domain 类型，或可以引用表列的类型。

根据实现的语言，也可以允许指定诸如 cstring 之类的假型。伪类型表示实际的参数类型是未完全指定的，或者在普通的 SQL 数据类型之外。

列的类型通过写入 `tablename.columnname%TYPE` 来引用。使用此功能有时可以帮助函数独立于表的定义的更改。

defexpr

如果没有指定参数，则用作默认值的表达式。表达式必须对参数类型是强制的。只有 IN 和 INOUT 参数可以有一个默认值。参数列表中具有默认值的参数后面的每个输入参数也必须具有默认值。

rettype

返回数据类型（可选方案限定）。该返回值可以是 base，composite，或 domain 类型，或引用表列的类型。根据实现语言，也可以允许它们指定诸如 cstring 之类的伪类型。如果该函数不返回值，可以指定 void 作为其返回类型。

当有 OUT 和 INOUT 参数时， 该 RETURNS 子句可以被省略。如果存在，它必须与输出参数所暗示的结果类型一致：如果存在多个输出参数或与单个输出参数相同的类型，则为 RECORD。

该 SETOF 修饰符表示该函数将返回一组条目，而不是单个条目。

列的类型通过写入 tablename.columnname%TYPE 来引用。

langname

该函数实现的语言名称可以是 SQL，C，internal，或用户自定义过程语言的名称。更多 HashData 数据库支持过程语言请参见 CREATE LANGUAGE。 为了向后兼容，该名称可以用括号括起来。

IMMUTABLE

STABLE

VOLATILE

这些属性通知查询优化器有关该函数的行为。最多可以指定一个选择。如果这些都不出现，则 VOLATILE 是默认的假设。由于 HashData 数据库目前有使用 VOLATILE 函数的性质，因此如果一个函数是真正的 IMMUTABLE，那么用户必须声明它以能够限制的使用。

IMMUTABLE 表示该函数不能修改数据库，并且在给定相同的参数值时始终返回相同的结果。它不会进行数据库查找或以其他方式使用参数列表中不直接显示的信息。如果给出这个选项。则可以使用函数值替换具有全常量参数的函数的任何调用。

STABLE 表示该数据库不能修改数据库，并且在单个表扫描中，对于相同的参数值，它将返回相同的结果。但是其结果可能会夸SQL语句更改。 这是对结果取决于数据库查询，参数值（例如当前时区），等等的函数的适当选择。还是要注意 current_timestamp 系列的函数符合稳定状态，因为他们的值在事务中不会改变。

VOLATILE 表示即使在单个表扫描中函数值可能发生变化，因此不能进行优化。在这个意义上来说，相对较少的数据库函数是不稳定的。例如 random()，currval()，timeofday()。 但是请注意，任何具有副作用的函数必须归类为 volatile，即使它的结果相当可预测，以此来防止调用被优化，一个例子是 setval()。

CALLED ON NULL INPUT

RETURNS NULL ON NULL INPUT

STRICT

CALLED ON NULL INPUT（默认值）表示当其某些参数为 NULL 时，该函数仍能被正常调用。那么该函数的作者有责任在必要时检查空值，并作出适当的响应。 RETURNS NULL ON NULL INPUT or STRICT 表示当任何参数为空时，如果指定了该参数，则当有空参数时，该函数将不会执行。 而自动假设为 null 结果。

[EXTERNAL] SECURITY INVOKER

[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER（默认值）表示该函数将以调用它的用户的权限执行。SECURITY DEFINER 指定该函数将以创建它的用户的权限执行。允许关键词 EXTERNAL 以用于 SQL 的一致性，但是它是可选的，因为不像 SQL，此功能不仅适用于外部函数，它适用于所有函数。

COST *execution_cost*

确定估计成本的整数，以 cpu_operator_cost 为单位。 如果函数返回一个集合，execution_cost 会标识每个返回行的成本。如果没有指定成本，C 语言和内部函数默认为 1 个单位，然而其他语言的函数默认 100 个单位，当用户指定较大的 execution_cost 值时，计划程序会尝试较少的函数评估。

configuration_parameter

value

当输入该功能时，该 SET 子句将一个值应用于会话配置参数。当函数退出时，配置参数恢复到先前的值。当进入函数时，SET FROM CURRENT 应用会话的当前值。

definition

定义函数的字符串常量；意思取决于语言。它可能是内部函数名称，对象文件的路径， 过程化语言文本中的 SQL 命令。

obj_file, link_symbol

当 C 语言源代码中的函数名称和 SQL 函数的名称不同时，这种形式的 AS 子句用于动态加载 C 语言函数。字符串 obj_file 是包含动态可加载对象文件的名称， link_symbol 是 C 语言源代码中的函数名称。如果省略了连接符号，则假定与正在定义的 SQL 函数的名称相同。建议相对于 `$libdir`（位于 `$GPHOME/lib`）或通过动态库路径（由 `dynamic_library_path` 服务器配置参数所设置）来定位共享库。如果新安装的版本位于不同的位置，则可以简化升级。

describe_function

当调用此函数的查询被解析时要执行回调函数的名称。回调函数返回一个指示结果类型的元组描述符。

注意

必须将自定义函数的任何编译的代码（共享库文件）放置在 HashData 数据库的数组的（master 主机和 segment 主机）的每个主机上的相同位置。该位置必须位于 `LD_LIBRARY_PATH` 中，以便服务器找到文件。建议相对于 `$libdir`（位于 `$GPHOME/lib`）或者通过 HashData 数组的所有 master 和 segment 实例中的动态库路径（由 `dynamic_library_path` 服务器配置参数设置）来定位共享库。

对于输入参数和返回值，允许使用完整的 SQL 类型语法。但是，该类型规范的一些细节（如（数字）numeric 类型的精度字段) 是底层函数实现的责任，并且不能被 CREATE FUNCTION 命令识别或强制执行。

HashData 数据库允许函数重载。只要具有不同的参数类型，相同的名字可以用于几个不同的函数。但是，所有函数的 C 名称必须不同，所以用户必须给重载 C 函数不同 C 名称（例如， 使用参数类型作为 C 名称的一部分）。

两个函数如果他们具有相同的名字和输入的参数类型，那他们就被认为是相同的，而不用考虑任何 OUT 参数。因此例如以下声明会产生冲突：

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

具有不同参数类型列表的函数在创建的时候不会产生冲突，但是如果提供了参数的默认值，它们在使用中可能会产生冲突。例如，考虑：

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

调用 foo(10)，由于不知道该调用那个函数，所以会产生函数调用失败，（缺失可可用默认值代替）。

当重复 CREATE FUNCTION 函数调用时，引用同一个对象文件，该文件只被加载一次。当卸载并重新加载文件时，请使用 LOAD 命令。

用户必须对具有语言的 USAGE 权限才能使用该语言定义函数。

使用美元引用来编写定义字符串，而不是普通单引号语法，这是有帮助的。没有美元引用， 函数定义中的任何单引号或反斜杠必须通过双写来实现转义。美元引用的字符串由美元符号（ `$` ），零个或者多个字符的可选标记， 另一个美元符号，组成字符串内容的任意字符串序列组成，一个美元符号，一个开始此美元引用的相同的标签。 和一个美元符号。在美元引用的字符串中，可以使用单引号， 反斜杠和任何不需要转义的字符。字符串内容始终以字面形式写入。例如，这里有两种不同的方法来指定字符串 “Dianne's horse” 使用美元引用：

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

如果一个 SET 字句被附加到一个函数上，则在同一个变量的函数内部执行 SET LOCAL 命令的作用仅限于该函数；当函数退出时，配置参数的先前值仍然被恢复。但是， 普通的 SET 命令（不带 LOCAL）将覆盖 CREATE FUNCTION SET 字句，与之前的 SET LOCAL 命令一样。这样命令的效果将在函数退出后持续，除非当前的事务被回滚。

如果具有 VARIADIC 参数的函数声明为 STRICT，则严格性检查将测试作为整体的可变数组是非空值。如果数组有空元素，PL/pgSQL 仍然会调用该函数。

使用分布式数据查询功能

一些情况下， HashData 数据库不支持在查询中使用函数，如果该查询的 FROM 子句指定表中的数据分布在 HashData 数据库段上的话。例如，该 SQL 查询包含函数 func()：

```
SELECT func(a) FROM table1;
```

如果遇到以下所有情况的话，则该函数不支持在查询中使用：

- 该 table1 表的数据分布在 HashData 数据库 segment 主机上。
- 函数 func() 从分布式表上读或修改数据。
- 函数 func() 返回多个行，或者使用来自 table1 的参数（a）。

如果任何以下的条件没有满足，则支持该函数，具体来说，如果满足以下任一条件，则支持该函数：

- 函数 func() 不能访问来自分布表的数据，也不能访问仅存在 HashData 数据库主机上的数据。
- 表 table1 仅仅是主机上的表。
- 函数 func() 仅返回一行并且仅接受常量值的输入参数。如果可以将该函数改为不需要输入参数，则支持此功能。

示例

一个简单的加法函数：

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

在 PL/pgSQL 中增加使用参数名称的整数

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS
integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

为 PL/pgSQL 函数增加每个查询的默认段主机内存：

```
CREATE OR REPLACE FUNCTION function_with_query() RETURNS
SETOF text AS $$
        BEGIN
                RETURN QUERY
                EXPLAIN ANALYZE SELECT * FROM large_table;
        END;
$$ LANGUAGE plpgsql
SET statement_mem='256MB';
```

使用多态返回 ENUM 数组：

```
CREATE TYPE rainbow AS ENUM('red','orange','yellow','green','blue','indigo','violet');
CREATE FUNCTION return_enum_as_array( anyenum, anyelement, anyelement )
    RETURNS TABLE (ae anyenum, aa anyarray) AS $$
    SELECT $1, array[$2, $3]
$$ LANGUAGE SQL STABLE;
SELECT * FROM return_enum_as_array('red'::rainbow, 'green'::rainbow, 'blue'::rainbow);
```

返回多个输出参数的记录：

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;
SELECT * FROM dup(42);
```

用户可以使用明确命名的复合类型更详细地做同样的事情:

```
CREATE TYPE dup_result AS (f1 int, f2 text);
CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;
SELECT * FROM dup(42);
```

兼容性

CREATE FUNCTION 被定义在 SQL:1999 和之后。 该 HashData 数据库版本是相似的，但是不完全兼容。属性不可移植，不同可用语言也不可移植。

为了和其他数据库系统的兼容，argmode 可以写在 argname 之前或者之后。但是只有第一种是复合标准的。

SQL 标准没有指明参数的默认值。

另见

ALTER FUNCTION，DROP FUNCTION，LOAD

上级话题：SQL命令参考

CREATE GROUP

创建组

定义一个新的数据库角色。

概要

```
CREATE GROUP name [ [WITH] option [ ... ] ]
```

该 *option* 可以是：

```
    SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | CREATEUSER | NOCREATEUSER
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
  | VALID UNTIL 'timestamp'
  | IN ROLE rolename [, ...]
  | IN GROUP rolename [, ...]
  | ROLE rolename [, ...]
  | ADMIN rolename [, ...]
  | USER rolename [, ...]
  | SYSID uid
```

描述

从 HashData 数据库 2.2 版开始，CREATE GROUP 已经被 CREATE ROLE 所取代，不过为了向后兼容，它仍然被接受。

兼容性

在 SQL 标准中没有 CREATE GROUP 语句。

另见

CREATE ROLE

上级话题： SQL命令参考

CREATE INDEX

创建索引

定义一个新的索引

概要

```
CREATE [UNIQUE] INDEX name ON table
       [USING btree|bitmap|gist]
       ( {column | (expression)} [opclass] [, ...] )
       [ WITH ( FILLFACTOR = value ) ]
       [TABLESPACE tablespace]
       [WHERE predicate]
```

描述

CREATE INDEX 构造指定表上的索引。索引主要用于增强数据库性能（尽管使用不当会导致性能降低）。

索引的关键字段被指定为列名称，或者替换为括号中的表达式。如果索引方法支持多列索引，则可以指定多个字段。

索引字段可以是从表行的一个或者多个列的值计算的表达式。 该功能可以基于基本数据的一些变换来获取对数据的快速访问。 列如，在 upper(col) 上计算的索引将允许子句 WHERE upper(col) = 'JIM' 来使用索引。

HashData 数据库提供索引方法 B-tree，位图，和 GiST。 用户可以自定义自己的索引方法，但这是相当复杂的。

当存在 WHERE 子句时，将创建部分索引，部分索引是仅包含表的一部分条目的索引， 通常是与表的其余部分相比对索引更有用的地方。例如，如果用户的表包含已结算订单和未结算订单，其中未列出的订单占总表中的一小部分，但最常选择，则可以通过在该部分上创建索引来提高性能。

WHERE 子句中使用的表达式可能仅指基础表的列，但它可以使用所有列，而不仅仅是索引的列。在 WHERE 中也禁止子查询和聚合表达式。 相同的限制适用于表达式的索引字段。

索引定义中使用的所有函数和操作符必须是不可改变的。他们的结果必须仅仅取决于他们的参数，没有任何外界的影响（如另一个表的内容或参数值）。此限制可以确保索引的行为被很好的定义。要在索引表达式或 WHERE 子句中使用用户定义的函数， 请记住在创建函数时标记函数为 IMMUTABLE。

参数

UNIQUE

当创建索引并添加每个时间数据时，检查表中的重复值。重复条目将产生错误。唯一索引仅适用于 B 树索引。在 HashData 数据库中，只有索引关键字与 HashData 分发密钥（或者它的超集）相同时才允许使用唯一索引。在分区表上，唯一索引仅在单个分区中支持，而不是跨越所有分区。

name

要创建索引的名称。索引始终与父表在统一模式中创建。

table

要索引表的名称（可选方案限定）

btree | bitmap | gist

要使用的索引方法，该有的选择是 b 树，位图，gist。 默认方法是 b 树。

column

表中要创建索引列的名称，仅 B-tree, 位图, GiST 索引支持多列索引。

expression

基于表的一个或多个列的表达式。 表达式通常用括号括起来，如语法所示。但是如果表达式有函数调用的形式，则可以省略括号。

opclass

操作符类的名称。操作符类指明了该列的索引要使用的操作符。例如，四字节整数的 B 树索引将使用 int4_ops 类（该操作包含了四字节整数的比较函数）。 实际上，列的数据类型的默认的操作符类通常是足够的。 操作符类的要点是对于某些数据类型，可能会有多个有意义的排序。例如，复杂数据类型可以按绝对值或实数排序。我们可以通过为数据类型定义两个操作符类，然后在进行索引的时候选择适当的操作符类来实现。

FILLFACTOR

索引的 fillfactor 是一个百分比，用于确定索引方法将索引页填充什么程度。 对于 B 树来说，在初始化索引编译期间和扩展右侧索引（最大键值）时将页面填充该百分比程度，如果页面随后变得充满，它将会被分隔，导致索引效率逐渐下降。

B 树使用默认的填充因子 90，但是可以选择 10 到 100 之间的任何值。如果表示静态的，那么100是最好的最小化索引的物理大小的填充因子。但是对于大量更新的表，较小的填充因子是最好最小化页面分隔的需要。其他索引方法使用不同填充因子，但大致是相似的选择方式。默认的填充因子在方法之间有所不同。

tablespace

创建索引的表空间，如果没有指定，将使用默认的表空间。

predicate

部分索引的约束表达式。

注解

当在分区表上建立索引时，该索引将传播到由 HashData 数据库创建的所有子表。在 HashData 数据库创建的表上建立由分区表使用的索引是不支持的。

仅当索引列与 HashData 分布键列相同（或者是其超集）时，才允许 UNIQUE 索引。

在追加优化表上不允许使用 UNIQUE 索引。

在分区表上可以创建 UNIQUE 索引。但是，唯一性仅在分区内执行；分区之间之间不执行唯一性。例如，对于基于年份的分区和基于季度的自分区的分区表，仅在每个单独的季度分区上执行唯一性。在季度之间不执行唯一性。

默认情况下，索引不用于 IS NULL 字句。这种情况下使用索引的最好方法是使用 IS NULL 谓词创建部分索引。

bitmap 索引对于具有 100 到 100000 个不同值的列的效果最好。对于有超过 100000 个不同值的列， 位图索引的性能和空间效率下降。位图索引的大小与表中行数成比例，是索引列中不同值数量的数倍。

少于 100 个不同值的列通常不会从任何类型的索引中受益，例如，男性和女性只有两个不同值的性别列不是索引的良好候选。

HashData 的先前版本也有一个 R-tree 索引方法。 该方法已经被删除，因为它相对于 GIST 方法相比没有显著的优点。 如果指定了 USING rtree， 则 CREATE INDEX 将会将其解释为使用 USING gist。

更多关于 GiST 索引类型的信息，请参考 PostgreSQL文档。

HashData 中已经禁止使用 hash 和 GIN 索引。

例子

创建一个 B 树索引在 films 表的 title 列中：

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

创建一个位图索引在 employee 表的 gender 列中:

```
CREATE INDEX gender_bmp_idx ON employee USING bitmap
(gender);
```

创建一个索引在表达式 lower（title）上，允许有效的区分大小写的搜索：

```
CREATE INDEX lower_title_idx ON films ((lower(title)));
```

使用非默认的填充因子创建一个索引：

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH
(fillfactor = 70);
```

创建一个索引在 films 表的 code 列上，并且使索引驻留在表空间的索引空间中：

```
CREATE INDEX code_idx ON films(code) TABLESPACE indexspace;
```

兼容性

CREATE INDEX 是 HashData 数据库的语言扩展。SQL 标准中没有索引规定。

HashData 数据库不支持并发创建索引（CONCURRENTLY 关键字不支持）。

另见

ALTER INDEX, DROP INDEX, CREATE TABLE, CREATE OPERATOR CLASS

上级话题： SQL命令参考

CREATE LANGUAGE

创建语言

定义一个新的程序化语言

概要

```
CREATE [PROCEDURAL] LANGUAGE name

CREATE [TRUSTED] [PROCEDURAL] LANGUAGE name
       HANDLER call_handler [ INLINE inline_handler ] [VALIDATOR valfunction]
```

描述

CREATE LANGUAGE 在 HashData 数据库中注册一个新的程序化语言。此后，可以使用这种新语言来定义函数和触发程序。

超级用户可以在 HashData 数据库中注册新的语言。数据库所有者也可以在该数据库中注册 pg_pltemplate 目录中列出的任何语言，其中tmpldbacreate 字段为真。默认配置值允许数据库所有者注册仅受信任的语言。语言的创建者变成它的所有者，之后也可以删除它，重命名它，或将其所有权分配给其他用户。

CREATE LANGUAGE 有效地将语言名称与负责执行以该语言编写的函数的调用程序相关联。对于以程序语言（C 或 SQL 以外的语言）编写的函数，数据库服务器没有关于如何解释数据库源码的内置知识。该任务被传递到一个知道语言细节的特殊处理程序。处理程序可以执行解析，语法分析，执行等所有工作。也可以作为 HashData 数据库与编程语言现有实现之间的桥梁。处理程序本身就是编译成一个共享对象并按需加载，就像任何其他 C 函数一样。目前已经有四种程序语言包包含在标准的 HashData 数据库分布中：PL/pgSQL, PL/Perl, and PL/Python. 还为 PL/Java and PL/R 添加了语言处理程序，但是这些语言未预先安装在 HashData 中。

该 PL/Perl, PL/Java, and PL/R 库要求分别安装 Perl, Java, 和 R 的正确版本。

有两种形式的 CREATE LANGUAGE 命令。在第一种形式中，用户指定了所需语言的名称，并指定 HashData 数据库服务器使用 pg_pltemplate 系统目录来确定正确的参数。在第二种形式中，用户指明了语言参数和语言名称，用户可以使用第二种形式创建一个未在 pg_pltemplate 中定义的语言。

当服务器在 pg_pltemplate 目录中找到给定语言名称的条目时，即使该命令包含语言参数，它也将使用目录数据。此行为简化了旧的储存文件的加载，那些文件可能包含有关语言支持功能的过时信息。

参数

TRUSTED

如果服务器在 pg_pltemplate 中具有指定语言名称的条目，则将忽视该语句。该关键字指定该语言的调用处理程序是安全的，并且不会提供无权用户任何用以绕过访问限制的功能，如果注册时省略此关键字，只有超级权限的用户才能使用此语言创建函数。

PROCEDURAL

这是一个申明（noise）字。

name

新程序语言的名字。该语言名字大小写不敏感。该名字在数据库语言中必须唯一。 内置的语言支持有 plpgsql, plperl, plpython, plpythonu, 和 plr。 plpgsql (PL/pgSQL) 和 plpythonu (PL/Python) 默认在 HashData 数据库中安装。

HANDLER *call_handler*

如果服务器在 pg_pltemplate 中具有指定语言名称的条目，则该关键字将被忽略。是将被调用去执行程序语言函数的先前注册函数的名称。程序语言的调用处理程序必须以编译的语言来写，例如版本 1 的 C 的调用约定，并在 HashData 数据库中注册成函数，该函数不需要参数并返回 language_handler 类型，language_handler 是一个用来将函数标识成调用处理程序的占位符类型。

INLINE *inline_handler*

先前注册函数的名称，该函数是使用 DO 命令创建的以该语言执行的匿名代码块。如果未指定 inline_handler 函数，则该函数不支持匿名代码块。处理程序函数必须使用一个 internal类型的参数，即 DO 命令内部的表现形式。内部匿名函数通常返回 void。处理程序的返回值将被忽略。

VALIDATOR *valfunction*

如果服务器在 pg_pltemplate 中具有指定语言名称的条目，则该字段将被忽略。该字段是之前注册函数的名称，该函数将被调用来执行程序语言函数。程序语言的调用处理程序必须以编译语言来写，例如版本 1 的 C 调用约定，并在 HashData 数据库中注册成函数，该函数不需要输入参数，并返回language_handler 类型，该类型是用来标识函数成调用处理程序的占位符类型。

注意

PL/pgSQL 和 PL/Python 语言扩展已经默认安装在 HashData 数据库中。

该系统目录 pg_language 记录了当前注册语言的信息。

在程序化语言中创建函数，一个用户必须要有关于该语言的 USAGE 权限。默认情况下，对信任的语言， USAGE 权限被授予给 PUBLIC（每个人）如果需要，该操作可以被撤销。

程序语言是个人数据库的本地语言。用户可以在个人数据库中创建和删除语言。

如果服务器没有 pg_pltemplate 的语言条目，那么调用处理函数和验证函数（如果有的话）必须已经存在。但是当有条目时，这些函数不一定存在；如果数据库中不存在，他们将被自动定义。

实现语言的任何共享库必须位于 HashData 数据库数组中所有 segments 主机的同一个 LD_LIBRARY_PATH 位置。

例子

创建任何标准程序语言的首选方式：

```
CREATE LANGUAGE plpgsql;
CREATE LANGUAGE plr;
```

对于一个在 pg_pltemplate 目录中不知名的语言:

```
CREATE FUNCTION plsample_call_handler() RETURNS
language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

兼容性

CREATE LANGUAGE 是 HashData 数据库的扩展。

另见

ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, DO

上级话题： SQL命令参考

CREATE OPERATOR CLASS

创建操作符类

定义一个新的操作符类。

概要

```
CREATE OPERATOR CLASS name [DEFAULT] FOR TYPE data_type
  USING index_method AS
  {
  OPERATOR strategy_number op_name [(op_type, op_type)] [RECHECK]
  | FUNCTION support_number funcname (argument_type [, ...] )
  | STORAGE storage_type
  } [, ... ]
```

描述

CREATE OPERATOR CLASS 创建一个新的操作符类。操作符类定义了能和索引一起使用的特殊的数据类型。该操作符类指定了某些操作符将填充此数据类型和索引方法的特定角色和策略。当操作符类被选定给索引列时，操作符类也指定索引方法所使用的支持的程序。操作符类所使用的所有操作符和函数必须在操作符创建之前就进行定义。用于实现操作符类的所有函数必须被定义为 IMMUTABLE（不可改变的）。

CREATE OPERATOR CLASS 目前不检查操作符类定义是否包括索引方法所需要的所有的操作符和函数。也不检查是否操作符和函数形成自我一致的集合。定义有效的操作符类是用户的责任。

用户必须是超级用户才能创建操作符类。

参数

name

要定义的操作符类的（可选方案限定）的名称。同一模式中的两个操作符只有在不同索引方法时才具有相同的名称。

DEFAULT

为其数据类型制定默认的操作符类。对于一个特定类型的数据类型和索引方法，最多可以指定一个默认操作符类。

data_type

该操作符类对应的列数据类型。

index_method

该操作符类所对应的索引方法的名称。索引方法有 btree, bitmap, 和 gist。

strategy_number

与操作符类相关联的操作符由 *strategy numbers* 来标识，用于在操作符类的上下文中识别每个操作符的语义。例如，B 树对关键字施加了严格的排序，从小到大，因此 小于 和 大于或者等于 的操作符对于 B 树来说是有趣的。这些策略可以被认为是广义的操作符，每个操作符针对特定数据类型指定与每个策略相符的实际操作符和索引语义的解释。每个指标的相应策略编号如下：

表 1. B-tree 和 Bitmap 策略

Operation	Strategy Number
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

表 **2. GiST** 二维策略（**R-Tree**）

Operation	Strategy Number
strictly left of	1
does not extend to right of	2
overlaps	3
does not extend to left of	4
strictly right of	5
same	6
contains	7
contained by	8
does not extend above	9
strictly below	10
strictly above	11
does not extend below	12

operator_name

和操作符类相关的操作符的（可选方案限定）名称。

op_type

操作符的操作数数据类型， 或 NONE 表示左一元或右一元操作符。该操作数数据类型一般情况下，在与操作符数据类型相同的情况下，可以省略操作数数据类型。

RECHECK

如果存在的话，则该索引对该操作符是“有损的”，因此必须重新检查使用索引检索过的行，以验证他们实际上满足涉及该操作符的限定子句。

support_number

索引方法需要额外的支持例程才能工作。这些操作是索引方法在内部使用的管理例程。与策略一样，该操作符类指定了哪些特定的函数应该为给定的数据类型和语义解释去使用这些角色。索引方法定义了所需的函数集合，操作符通过将他们分配支持函数号 来识别需要的正确的函数。如下所示：

表 **3. B-tree** 和 **Bitmap** 支持函数

Function	Support Number
Compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second.	1

表 **4. GiST** 支持函数

Function	Support Number
consistent - determine whether key satisfies the query qualifier.	1
union - compute union of a set of keys.	2
compress - compute a compressed representation of a key or value to be indexed.	3
decompress - compute a decompressed representation of a compressed key.	4
penalty - compute penalty for inserting new key into subtree with given subtree's key.	5
picksplit - determine which entries of a page are to be moved to the new page and compute the union keys for resulting pages.	6
equal - compare two keys and return true if they are equal.	7

注意

因为索引机制在使用函数之前不检查它们的访问权限，包括在操作符类中的函数和操作符与授予它的公共执行权限相同。这对于在操作符类中有用的各种函数通常不是问题。

操作符不应该由 SQL 函数定义。调用查询中可能会嵌入 SQL 函数，这会阻止优化程序识别出查询与索引匹配。

用于实现操作符类的任何函数必须定义为 IMMUTABLE。

例子

接下来的例子命令定义了数据类型 int4（int4 的数组）的 GiST 索引操作符类：

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR 3 &&,
        OPERATOR 6 = RECHECK,
        OPERATOR 7 @>,
        OPERATOR 8 <@,
        OPERATOR 20 @@ (_int4, query_int),
        FUNCTION 1 g_int_consistent (internal, _int4, int4),
        FUNCTION 2 g_int_union (bytea, internal),
        FUNCTION 3 g_int_compress (internal),
        FUNCTION 4 g_int_decompress (internal),
        FUNCTION 5 g_int_penalty (internal, internal, internal),
        FUNCTION 6 g_int_picksplit (internal, internal),
        FUNCTION 7 g_int_same (_int4, _int4, internal);
```

兼容性

CREATE OPERATOR CLASS 是 HashData 数据库扩展。 在 SQL 标准中没有 CREATE OPERATOR CLASS 语句。

另见

ALTER OPERATOR CLASS, DROP OPERATOR CLASS, CREATE FUNCTION

上级话题： SQL命令参考

CREATE OPERATOR FAMILY

创建操作符族

定义一个新的操作符族

概要

```
CREATE OPERATOR FAMILY name  USING index_method
```

描述

CREATE OPERATOR FAMILY 创建一个新的操作符族。操作符族定义了相关操作符类的集合，以及一些额外的操作符和支持函数，这些函数和这些操作符类兼容，但是对于任何单个索引的运行来说不是必需的。（对索引必不可少的操作符和函数应归入相应的操作符类而不是在操作符族中“松动”，通常，单数据类型操作符绑定到操作符类，然而跨数据类型操作符可能在包含该两种数据类型的系列中“松动”。）

新的操作符族最初是空的，应该通过发出后续的 CREATE OPERATOR CLASS 命令来增加包含的操作符类，以及可选的 ALTER OPERATOR FAMILY命令来添加 "松动" 的操作符和相应的支持函数。

如果给定了模式名称，则在指定的模式中创建操作符族。否则会在当前模式中创建操作符族。 同一个模式中的两个操作符族，只有在当他们所指定的索引方法不同时，才能重名。

定义操作符族的人成为其所有者。目前，创建用户必须是超级用户（该限制的制定是因为错误的操作符族会混淆服务器甚至致使其崩溃。）

参数

name

要定义的操作符族的（可选操作符限定）名字，该名字是方案限定的。

index_method

该操作符族的索引方法的名称。

兼容性

CREATE OPERATOR FAMILY 是 HashData 数据库的扩展，SQL标准中没有 CREATE OPERATOR FAMILY 语句。

另见

DROP OPERATOR FAMILY, CREATE FUNCTION, ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR CLASS

上级话题： SQL命令参考

CREATE OPERATOR

定义一个操作符

概要

```
CREATE OPERATOR name (
       PROCEDURE = funcname
       [, LEFTARG = lefttype] [, RIGHTARG = righttype]
       [, COMMUTATOR = com_op] [, NEGATOR = neg_op]
       [, RESTRICT = res_proc] [, JOIN = join_proc]
       [, HASHES] [, MERGES]
       [, SORT1 = left_sort_op] [, SORT2 = right_sort_op]
       [, LTCMP = less_than_op] [, GTCMP = greater_than_op] )
```

描述

CREATE OPERATOR 定义了一个新的操作符。定义操作符的用户变成它的拥有者。

操作符的名称是从以下列表中组合，最长 NAMEDATALEN-1（默认63）个字符的序列： + - * \/ \< > \= ~ ! \@ # \% \^ \& | ` \?

用户选择的名称有一些限制

- -- 和 /* 不能出现在操作符的任何位置，因为他们被认为是注释的开始。
- 多字符操作符不能以 + 或 - 结束， 除非该名字包含一个以下的字符： ~ ! @ # % ^ \& | ` ？

例如， @- 是一个合法的操作符名字，但是 *- 却不是一个合法的操作符名字，该限制允许 HashData 数据库去解析复合 SQL 的命令，而不需要字符之间的空格。

操作符 != 在输出上映射为 <> ，所以这两个名字总是相等的。

LEFTARG 和 RIGHTARG 至少有一个必须被定义。对于二进制操作符来说，两个都需要定义。对于右元操作符来说，只要 LEFTARG 需要定义，然而对于左元操作符来说，只要 RIGHTARG 被定义就行了。

该 funcname 程序必须使用 CREATE FUNCTION 预定义，必须是 IMMUTABLE， 必须定义为能够接受正确的指定类型的参数个数（一个或者两个）

其他条款指出了可选的操作符优化子句，应当提供这些条款，以加快使用操作符的查询。 但是用户提供了这些条款，用户必须确保他们是正确的。不正确的使用优化子句可能导致服务进程的崩溃，错误输出或是其他意外结果。如果用户不确定，用户可以随时省略优化的句子。

参数

name

将要定义的操作符的（可选方案限定）名字。同一个模式中的两个操作符如果他们操作不同的数据类型，则他们可以有相同的名字。

funcname

实现该操作符的函数（必须是 IMMUTABLE 函数）。

lefttype

该操作符左操作数（如果有的话）的数据类型。对于左元操作符省略此选项。

righttype

该操作符右操作数（如果有的话）的数据类型。对于右元操作符省略此选项。

com_op

可选的 COMMUTATOR 子句命名了一个操作符，该操作符是正在定义的操作符的交换器。我们说，如果对于所有可能的输入值 x，y，（x A y）等于（y B x），则操作符 A 是操作符 B 的交换器。请注意，B 也是 A 的交换器。例如 操作符 < 和 > 对于特殊的数据类型来说通常都是相互的转向器，操作符 + 通常是与其自身可交换。但是操作符 - 通常不与任何操作符可交换，可交换操作符的左操作数与其交换器的右操作数类型相同，反之亦然。因此在 COMMUTATOR 子句中需要提供交换器的名称。

neg_op

可选的 NEGATOR 子句命名了一个操作符，该操作符是正在被定义的操作符的否定操作符。我们说，对于所有可能的输入 x，y，如果（x A y）等于 NOT（y B x）而且两者都返回布尔类型的结果，则 A 操作符是 B 操作符的否定符，请注意，B 操作符也是 A 操作符的否定符。例如， < 和 >= 对于大多数数据类型来说，是一个否定符号对。操作符否定符的左右操作数类型必须要与操作符的左右操作数的数据类型一样。所以只需在 NEGATOR 子句中指出操作符名称。

res_proc

可选的 RESTRICT 命名了操作符的限制选择性估计函数，请注意，这只是一个函数名称，不是操作符的名称。RESTRICT 子句仅对返回 boolean 的二进制操作符才有意义。限制选择性估计器的想法是猜测表中的哪几行满足以下格式的 WHERE 子句条件：

```
column OP constant
```

对于当前的操作符和特定的常数值。这有助于优化器，通过给出具有此表单的 WHERE 子句将消除多少行的想法。

用户通常可以为自己的操作符使用以下系统估计器函数之一:

eqsel 用 于=

neqsel 用于 <>

scalarltsel 用于 < 或者 <=

scalargtsel 用于 > 或者 >=

join_proc

可选的 JOIN 子句命令了操作符的连接选择性估计功能。请注意，这是一个函数名称，而不是操作符名称。JOIN 子句仅对返回 boolean 类型的二进制操作符才有意义。连接选择性估计器的想法是猜测一对表中的几个部分行将满足表单的 WHERE 子句条件。

table1.column1 OP table2.column2

对目前的操作符来说，这帮助优化器，通过让它明白可能的连接序列中哪一个可能占用最少的工作。

用户通常可以为许多自己的操作符使用以下系统标准连接选择性估计函数之一。

eqjoinsel 用于 =

neqjoinsel 用于 <>

scalarltjoinsel 用于 < 或者 <=

scalargtjoinsel 用于 > 或者 >=

areajoinsel 用于基于 2D 区域的比较

positionjoinsel 用于基于 2D 位置的比较

contjoinsel 用于基于 2D 包含的比较

HASHES

可选的 HASHES 子句告诉系统，允许基于此操作符使用连接的哈希连接方法。HASHES 仅对返回 boolean 的二进制操作符

才有意义。哈希连接操作符只能对左右值对都哈希到相同哈希码的情况才返回 true。如果这两个值被放在了不同的哈希桶中，则该连接将永远不会比较他们。隐含的假设连接操作符的结果必须是假。因此，对于不代表相等性的操作符，指定 HASHES 是没有意义的。

要标记为 HASHES，该连接操作符必须出现在哈希索引操作符类中。如果不存在这样的操作符类，试图在运行时的哈希连接中使用操作符将导致失败。系统需要操作符类来找到操作符输入的数据类型的特定哈希函数。用户还必须提供合适的哈希函数，然后才能创建操作符号类，在准备哈希函数时要小心，因为有依赖机器的方法，该方法将可能无法做正确的事情。

MERGES

MERGES 子句（如果存在）告诉系统可以使用基于此操作符的连接的合并方法。 MERGES 仅对于返回布尔值的二进制操作符才是有意义的，实际上，操作符必须表示某些数据类型或数据类型对的等式。

合并连接是基于按顺序排列的左右表格，然后并行扫描他们，因此，两种数据类型都能被完全排序，并且连接操作符必须只能对在排序顺序中位于相同位置的值对才能成功。实践中，这意味着连接符必须具有相等性。只要逻辑上兼容，就可以合并——连接两个不同的类型。例如，小整型对整数相等操作符是合并连接。我们只需排序操作符，这两个数据类型都将在成逻辑上相容的序列。

合并操作要求系统能够识别与合并连接相等操作符相关的四个操作符：小于左操作数数据类型的比较，小于右操作数数据类型的比较，在两种数据类型的小于比较和在两种数据类型的大于比较，也可以通过名称指定这些操作符，分别为 SORT1，SORT2，LTCMP，和 GTCMP。如果在指定了 MERGES 时省略了这些名称，系统将填写默认名称。

left_sort_op

如果此操作符可以支持合并连接，则为对操作符左数据类型进行排序的小于操作符。如果没有指定，则为默认值 <

right_sort_op

如果此操作符支持合并连接，则为对操作符右数据类型进行排序的小于操作符。如果没有指定，则为 <

less_than_op

如果此操作符支持合并连接，则为比较该操作符的输入数据类型的小于操作符。如果没有指定，则为 <

greater_than_op

如果此操作符能支持合并连接，则为比较该操作符的输入数据类型的大于操作符。如果没有指定，则为默认值 >。

要在可选参数中给出方案限定的操作符名称，请使用 OPERATOR() 语法，例如：

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

注意

任何实现该操作符的函数必须定义成 IMMUTABLE。

列子

下面创建一个操作符来加两个复数的例子，假设我们已经创建了复数类型。 首先定义执行工作的函数，然后定义操作符：

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;
CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

在查询中使用操作符：

```
SELECT (a + b) AS c FROM test_complex;
```

兼容性

CREATE OPERATOR 是 HashData 数据库的语言扩展。 SQL 标准没有提供用户自定义操作符。

另见

CREATE FUNCTION, CREATE TYPE, ALTER OPERATOR, DROP OPERATOR

上级话题： SQL命令参考

CREATE PROTOCOL

创建协议

注册一个自定义数据访问协议，当定义 HashData 数据库外部表时能指定该协议。

概要

```
CREATE [TRUSTED] PROTOCOL name (
   [readfunc='read_call_handler'] [, writefunc='write_call_handler']
   [, validatorfunc='validate_handler' ])
```

描述

CREATE PROTOCOL 将数据访问协议名字和负责从外部数据源读取和写入数据的处理程序相关联。

该 CREATE PROTOCOL 必须指定一个读调用程序或者一个写调用程序，在 CREATE PROTOCOL 命令中指定的调用程序必须在数据库中定义。

该协议名称可以在 CREATE EXTERNAL TABLE 命令中指定。

更多关于创建和启用数据访问协议的信息，请参阅 HashData 数据库管理员指南中的“自定义数据库访问协议的例子”。

参数

TRUSTED

噪声词。

name

数据访问协议的名称，协议名称大小写敏感，数据库中协议的名称必须唯一。

readfunc= 'read_call_handler'

之前注册函数的名称， HashData 数据库会调用该函数从外部数据源中读数据，该命令必须指定一个读调用程序或写调程序。

writefunc= 'write_call_handler'

之前注册函数的名称， HashData 数据库会调用该函数将数据写入到外部数据源，该命令必须指定一个读调用程序或写调用程序。

validatorfunc='validate_handler'

可选的验证函数，该函数验证在 CREATE EXTERNAL TABLE 命令中指定的 URL。

注意

HashData 数据库安装自定义协议。 file, gpfdist, gpfdists, 和默认值 gphdfs 。可选择的是，s3 协议也能被安装。

任何实现了数据访问协议的共享库必须位于所有 HashData 数据库段主机的相同位置。例如，该共享库可以在由操作系统环境变量 LD_LIBRARY_PATH 指定的在所有主机上的位置。用户也可以指定一个位置，当用户定义处理函数的时候。例如，当用户在 CREATE PROTOCOL 命令中定义 s3 协议时候， 用户指定了 `$libdir/gps3ext.so` 作为共享对象的位置。 其中

`$libdir` 位于 `$GPHOME/lib` 。

兼容性

CREATE PROTOCOL 是 HashData 数据库扩展。

另见

ALTER PROTOCOL，CREATE EXTERNAL TABLE，DROP PROTOCOL

上级话题： SQL命令参考

CREATE RESOURCE QUEUE

创建资源队列

定义一个新的资源队列

概要

```
CREATE RESOURCE QUEUE name WITH (queue_attribute=value [, ... ])
```

其中 queue_attribute 是：

```
    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' ]
```

描述

对 HashData 数据库负载管理，创建一个新的资源队列。 资源队列必须要有 ACTIVE_STATEMENTS 或者 MAX_COST 值（也可以两个都有）。只有超级用户可以创建一个资源队列。

带有 ACTIVE_STATEMENTS 阀值的资源队列在查询数量上做了最大限制，该查询能够被分配到该队列上的角色所执行。它（阀值）控制了活跃查询的数量，活跃查询是在同一时间允许运行的查询。ACTIVE_STATEMENTS 的值应当是一个大于 0 的值。

带有 MAX_COST 阀值的资源队列在查询总代价上设置了一个最大限制，该查询能够被分配到该队列上的角色所执行。代价是按照 HashData 数据库查询计划器（正如查询的 EXPLAIN 输出显示的）确定的查询的 估计总成本 来进行衡量的。因此，管理员必须熟悉系统典型执行的查询，以对队列设置一个合适的代价阀值。代价是以磁盘页的提取为单位进行衡量的。1.0 等于一个顺序的磁盘页面的读取。MAX_COST 的值被指定为浮点数（例如 100.0）或者也可以被指定为指数（例如 1e+2）。如果基于成本阀值限制资源队列，则管理员可以允许 COST_OVERCOMMIT=TRUE（默认值）。这意味着超过成本阀值的查询将被允许查询，但只有当系统空闲的时候才行。如果指定 COST_OVERCOMMIT=FALSE，超过成本限制的查询将始终被拒绝，从不允许执行。对 MIN_COST 指定一个值，这将允许管理员定义小查询的成本，低于该成本的肖查询将免除排队的烦恼。

如果没有给 ACTIVE_STATEMENTS 或者 MAX_COST 设定值，将被设置为默认值 -1（意味着没有限制）。在定义了资源队列之后，用户必须使用 ALTER ROLE 或者 CREATE ROLE 命令向队列分配角色。

用户可以选择性的分配 PRIORITY 给一个资源队列来控制与（和其他资源队列相关的）队列相关查询使用的可用 CPU 资源的相对份额。如果没有指定 PRIORITY 的值，则和队列相关的查询默认优先级为 MEDIUM。

带有可选择的 MEMORY_LIMIT 阀值的资源队列对内存数量上设置了最大限制。该内存是 Segment 主机上被所有通过资源队列提交的查询所使用的。这决定了在 Segment 主机上，在一次查询执行中，一个查询的所有工作进程所能消耗的总内存的数量。 HashData 推荐 MEMORY_LIMIT 和 ACTIVE_STATEMENTS 联合使用而不是和 MAX_COST。 基于语句的队列每个查询分配的默认内存量为：MEMORY_LIMIT / ACTIVE_STATEMENTS。基于成本队列每个查询分配的默认内存量为：MEMORY_LIMIT * (query_cost / MAX_COST)。

默认内存分配可以使用 statement_mem 服务器配置参数在每个查询的基础上被覆盖。前提是不超过 MEMORY_LIMIT 或

max_statement_mem。例如，要为特定查询分配更多的内存。

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

该 MEMORY_LIMIT 值对于用户所有的资源队列都不应该超过 Segment 主机上的物理内存。如果工作负载在多个队列中交错，内存分配可以重新拟定。但是，如果 Segment 主机在 gp_vmem_protect_limit 指定的内存限制被超的话，执行中查询可以被取消。

参数

name

资源队列的名字。

ACTIVE_STATEMENTS integer

带有 ACTIVE_STATEMENTS 阀值的资源队列限制了分配到队列角色所能够执行的查询的数量。它（阀值）控制着活跃查询的数量，活跃查询是在同一时间允许运行的查询数量。ACTIVE_STATEMENTS 的值应该是一个大于 0 的整数值。

MEMORY_LIMIT 'memory_units'

对于所有从该资源队列中提交的语句设置总内存配额。内存单元可以指定为 kB, MB 或者 GB。对于一个资源队列来说最小的内存配额是 10MB，没有最大限值，但是查询执行的上边界由 Segment 主机的物理内存所限定。默认值时没有限制为（-1）。

MAX_COST float

带有 MAX_COST 阀值的资源队列对查询代价设置了一个最大限制。该查询能够被分配到该队列的用户所执行。代价由 HashData 数据库查询优化器（正如查询 EXPLAIN 输出显示的）确定的查询的 估计共代价 进行衡量的。 因此，管理员必须要熟悉在系统中执行的典型查询，以对队列设置一个合理的阀值。成本以磁盘页提取为单位进行衡量；1.0 等于顺序读取一个磁盘页。MAX_COST 的值可以被指定为浮点数（例如 100.0） 或者可以被指定为（例如 1e+2）。

COST_OVERCOMMIT boolean

如果基于 MAX_COST 限制资源队列，则管理员可以允许 COST_OVERCOMMIT（默认）。这意味着超过允许的成本阈值的查询将被允许运行，但只有在系统空闲时才能运行。如果指定 COST_OVERCOMMIT=FALSE ，超过成本限制的查询将始终被拒绝，从不允许运行。

MIN_COST float

该是最小查询的最小查询成本限制。成本低于此限制的查询将不会排队等待立即运行。成本由 HashData 数据库查询优化器（正如查询 EXPLAIN 输出所示）确定的查询的 估计总成本 所衡量。 因此，管理员必须熟悉通常在系统上执行的查询，以便为被认为是小型查询设置适当的成本。 成本是以磁盘页提取为单位来衡量的; 1.0 等于一个顺序的磁盘页面读取。MIN_COST 的值可以被指定为浮点数（例如 100.0）或也可以被指定为一个指数（例如 1e+2）。

PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX}

设置和资源队列相关查询的优先级。队里中拥有高优先级的查询和语句会在竞争中拥有更大的可用CPU资源份额。队列中拥有低优先级的查询将会被推迟，同时，更高优先级的查询将会被执行。如果没有指定优先级，和队列相关的查询的优先级为 MEDIUM。

注意

使用 `gp_toolkit.gp_resqueue_status` 系统视图查看限制设置和当前资源队列的状态：

```
SELECT * from gp_toolkit.gp_resqueue_status WHERE
  rsqname='queue_name';
```

还有一个叫 `pg_stat_resqueues` 的系统视图，这显示了资源队列随时间的统计指标。但是，为了使用该视图，用户必须启用 `stats_queue_level` 服务器配置参数。

CREATE RESOURCE QUEUE 不能再事务中运行。

此外，在 EXPLAIN ANALYZE 命令执行期间执行的 SQL 命令被从资源队列排除。

例子

创建一个活跃查询限制为 20 的资源队列：

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20);
```

创建一个活跃查询限制为20的资源队列并且总内存限制为 2000MB（在执行时，每个查询会被分配 100MB 端主机内存)：

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20,
  MEMORY_LIMIT='2000MB');
```

创建一个查询代价限制为 3000.0 的资源队列：

```
CREATE RESOURCE QUEUE myqueue WITH (MAX_COST=3000.0);
```

创建一个查询代价限制为 310 的资源队列（或者 30000000000.0）并且不允许复写。允许 500 以下的小查询立即运行：

```
CREATE RESOURCE QUEUE myqueue WITH (MAX_COST=3e+10,
  COST_OVERCOMMIT=FALSE, MIN_COST=500.0);
```

创建一个带有活跃查询限制和查询代价限制的资源队列：

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=30,
  MAX_COST=5000.00);
```

创建一个带有活跃查询限制为 5 并且有最大优先级设置的资源队列：

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=5,
  PRIORITY=MAX);
```

兼容性

CREATE RESOURCE QUEUE 是 HashData 数据库扩展，在 SQL 标准中没有资源队列和工作负载管理的规定。

另见

ALTER ROLE，CREATE ROLE，ALTER RESOURCE QUEUE，DROP RESOURCE QUEUE

上级话题：SQL命令参考

CREATE ROLE

创建角色

定义一个新的数据库角色（用户或组）。

概要

```
CREATE ROLE name [[WITH] option [ ... ]]
```

其中 option 可以是：

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEEXTTABLE | NOCREATEEXTTABLE
      [ ( attribute='value'[, ...] ) ]
           where attributes and value are:
           type='readable'|'writable'
           protocol='gpfdist'|'http'
    | 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
    | [ DENY deny_point ]
    | [ DENY BETWEEN deny_point AND deny_point]
```

描述

CREATE ROLE 添加了一个新的角色到 HashData 数据库系统。角色是一个实体，可以拥有数据库对象并且有数据库权利。角色可以是用户，组，或者两者都是，这取决于它是怎么用的。用户必须有 CREATEROLE 权限或者是超级用户才能用此命令。

注意角色是在系统层级定义的，对 HashData 数据系统所有数据库都有效。

参数

name

新角色的名字。

SUPERUSER

NOSUPERUSER

如果指定了 SUPERUSER，该定义的角色将是个超级用户，该超级用户重写数据库中的所有限制。超级用户状态是危险的，应当在仅真正需要的时候才去用。用户自己必须是超级用户才能创建新的超级用户。默认是 NOSUPERUSER。

CREATEDB

NOCREATEDB

如果指定了 CREATEDB，被定义的角色将被允许定义一个新数据库。NOCREATEDB（默认是）将会禁止掉用户创建数据库的能力。

CREATEROLE

NOCREATEROLE

如果指定了 CREATEDB，该被定义的角色将被允许创建新的角色，修改其他角色和删除其他角色。 NOCREATEROLE（默认值）将会禁止掉用户创建或修改（除了他们自己的）角色的能力。

CREATEEXTTABLE

NOCREATEEXTTABLE

如果指定了 CREATEEXTTABLE，被定义的角色将被允许创建外部表。该默认的类型是 readable ，如果没有指定，则默认的协议是 gpfdist。 NOCREATEEXTTABLE（默认值）禁止掉了角色创建外部表的能力。注意使用 file 或 execute 协议的外部表只能由超级用户创建。

INHERIT

NOINHERIT

如果指定，INHERIT （默认值）允许角色使用该直接或间接所属所有角色的数据库权利。使用 NOINHERIT，其他角色的成员资格仅通过 SET ROLE 授予到其他角色。

LOGIN

NOLOGIN

如果指定，LOGIN 允许该角色登录数据库，使用 LOGIN 属性的角色可以被认为是一个用户。使用 NOLOGIN（默认值）的角色对管理数据库非常有用，可以被认为是组。

CONNECTION LIMIT connlimit

该角色所能创建的最大并发连接数。默认值 -1 表示没有限制。

PASSWORD password

为使用 LOGIN 属性的角色设置用户密码。如果用户没有打算使用密码授权，用户可以省略该属性。如果没有指定密码，该密码将被置为 null，并且对该用户来说密码授权始终无效。null 密码可选择性的写明为 PASSWORD NULL。

ENCRYPTED

UNENCRYPTED

这些关键字控制密码是否加密存储在系统目录中。（如果没一个指定，则默认行为由配置参数 password_encryption 决定。）如果提供的密码字符串已经是 MD5 加密的格式，则无论是否指定了 ENCRYPTED 或 UNENCRYPTED 都会按原样加密存储（因为系统不能解密指定加密的密码字符串）。这允许在转储/恢复期间重新加载加密的密码。

注意老客户端可能缺乏对 MD5 认证机制的支持，系统需要该机制来和存储的密码一起工作。

VALID UNTIL 'timestamp'

该 VALID UNTIL 子句设置了日期和时间，在此时间日期之后角色的密码不再有效。如果省略了该子句，则密码将永远不会过期。

IN ROLE rolename

将新角色添加为命令角色们的成员。请注意无法添加管理员新角色；使用单独的 GRANT 的命令可以做到。

ROLE rolename

将命名的角色添加为该角色的成员，将该角色变成一个组。

ADMIN rolename

该 ADMIN 子句像 ROLE，但是命令的角色将被添加到新角色中 WITH ADMIN OPTION，给他们能授予其他用户该组成员的权利。

RESOURCE QUEUE queue_name

该新的用户级角色被分配到的资源队列名称。仅带有 LOGIN 权利的角色能被分配到资源队列。该特殊的关键字 NONE 意味着该角色被分配到默认的资源队列，一个角色只能被分配到一个资源队列。

带有 SUPERUSER 属性的角色免于资源队列的限制。对于一个超级用户角色来说，查询总是立即运行而不顾资源队列施加的限制。

DENY deny_point

DENY BETWEEN deny_point AND deny_point

DENY 和 DENY BETWEEN 关键字设置了基于时间的限制，该限制在登录的时候被强制。DENY 设置了天或天和时间来拒绝访问。 DENY BETWEEN 设置了访问拒绝后的间隔。两个都用参数 deny_point ，该参数有以下格式：

```
DAY day [ TIME 'time' ]
```

该 deny_point 参数的两个部分使用以下格式：

对于 day：

```
{'Sunday' | 'Monday' | 'Tuesday' |'Wednesday' | 'Thursday' | 'Friday' |
'Saturday' | 0-6 }
```

对于 time：

```
{ 00-23 : 00-59 | 01-12 : 00-59 { AM | PM }}
```

该 DENY BETWEEN 子句使用两个 deny_point 参数：

```
DENY BETWEEN deny_point AND deny_point
```

注意

添加删除角色成员（管理组）最偏爱的方式就是使用 GRANT 和 REVOKE。

VALID UNTIL 子句仅对密码定义了一个期限时间，而不是对角色。当使用非基于密码授权方式登录时，该期限时间是不生效的。

该 INHERIT 属性控制可授权权限的继承（数据库对象和角色成员资格的访问权限）。它不适用与由 CREATE ROLE 和 ALTER ROLE 设置的特殊角色属性。例如，成为具有 CREATEDB 权限的角色的成员，即使设置了 INHERIT，也不会授予创建数据库的功能。这些权限/属性从不继承。SUPERUSER, CREATEDB, CREATEROLE, CREATEEXTTABLE, LOGIN, 和 RESOURCE QUEUE。必须在每个用户级角色上设置属性。

INHERIT 属性是默认的为了向后兼容。在先前 HashData 数据库的发行版中，用户通常访问他们所属组的所有权限。但是，NOINHERIT 提供了一个与 SQL 标准中指定语义更合适的匹配。

小心使用 CREATEROLE 权限。对于一个 CREATEROLE 角色，并没有一个继承的概念。这意味着即使一个角色没有特定的权限，也允许创建其他角色，它可以轻松地创建具有不同于其自己权限的其他角色（除了使用超级用户创建角色）。例如，如果角色具有 CREATEROLE 权限但是不具有 CREATEDB 权限，它可以创建一个带有 CREATEDB 权限的角色。因此，将具有 CREATEROLE 权限的角色视为几乎具有超级用户角色。

该 CONNECTION LIMIT 选项从来不对超级用户执行。

当使用此命令指定一个未加密的密码时候，必须小心。该密码将以明文形式发送到服务器，也可能会记录在客户端的命令历史记录或服务器日志中。然而，客户端程序 createuser 发送加密的密码。此外，psql 还包含一个命令 password，该可以用来安全的更改密码。

示例

创建一个可以登录的角色，但是不给密码：

```
CREATE ROLE jonathan LOGIN;
```

创建一个属于资源队列的角色：

```
CREATE ROLE jonathan LOGIN RESOURCE QUEUE poweruser;
```

创建一个带密码的角色，改密码有效期到 2016 年底（CREATE USER 和 CREATE ROLE 相同除了前者暗示了 LOGIN）：

```
CREATE USER joelle WITH PASSWORD 'jw8s0F4' VALID UNTIL '2017-01-01';
```

创建一个可以创建 db 和管理其他角色的角色：

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

创建一个不允许在星期天登录的角色：

```
CREATE ROLE user3 DENY DAY 'Sunday';
```

兼容性

SQL 标准定了用户和角色的概念，但是把他们视为不同的概念并且让所有定义用户的命令由数据库实现指定。在 HashData 数据库中，用户和角色被统一为单个类型的对象。因此，角色有比标准中更多的可选属性。

CREATE ROLE 在 SQL 标准中，但是该标准只需要语法：

```
CREATE ROLE name [WITH ADMIN rolename]
```

允许多个初始管理员和 CREATE ROLE 的所有其他选项，是 HashData 数据库扩展。

由 SQL 标准指定的行为和给与用户 NOINHERIT 属性最为接近，然而角色则赋予 INHERIT 属性。

另见

SET ROLE，ALTER ROLE，DROP ROLE，GRANT，REVOKE，CREATE RESOURCE QUEUE

上级话题：SQL命令参考

CREATE RULE

创建规则

定义一个新的重写规则。

概要

```
CREATE [OR REPLACE] RULE name AS ON event
  TO table [WHERE condition]
  DO [ALSO | INSTEAD] { NOTHING | command | (command; command
  ...) }
```

描述

CREATE RULE 定义一个新的规则应用于指定的表或视图 CREATE OR REPLACE RULE 将创建一个新的规则，或者在同一张表中替代一个已经存在的同名的规则。

HashData 数据库规则系统允许定义对数据库表的插入，更新或删除的可选操作。当指定表上的命令执行时，规则会导致附加命令或可选命令将被执行。规则也能应用于视图。意识到规则实际上是命令转换机制，或是命令宏是很重要的。转换在执行命令开始之前发生，它不像触发器那样为每个物理行独立运行。

ON SELECT 规则必须是无条件的 INSTEAD 规则，并且必须具有由单个 SELECT 命令组成的操作。因此，ON SELECT 能有效的将表转换成视图，其视图内容是规则的 SELECT 命令返回的行，而不是存储在表中的任何内容（如果有的话）。编写 CREATE VIEW 命令的风格比创建真实的表并且在其上定义 ON SELECT 规则要好很多。

用户可以通过定义 ON INSERT， ON UPDATE，和 ON DELETE 规则来创建可更新的视图的错觉，以便在其他表带有更新操作的视图上替换更新操作。

如果用户尝试使用条件规则进行视图更新，则会有一个 catch：对于希望在视图中允许的每一个操作，必须要有一个无条件的 INSTEAD 规则。如果规则是有条件的或不是 INSTEAD，则系统仍将拒绝执行更新操作的尝试，因为它认为在某些情况下可能会尝试对视图的虚拟表执行操作。如果要处理条件规则中的所有可用的情况，请添加无条件的 DO INSTEAD NOTHING 规则来确保系统理解它将永远不会被调用来更新虚拟表。然后使条件规则非 INSTEAD；在应用他们的场景中，他们添加到默认的 INSTEAD NOTHING 操作。（但是，该方法目前无法支持 RETURNING 查询）

参数

name

创建规则的名称。改名字在同一张表中必须和其他的规则的名字不相同。同一表和相同事件类型上的多个规则按字母顺序排序。

event

该事件是以下 SELECT，INSERT， UPDATE，或 DELETE 之一。

table

该规则所应用的表或视图的名称（选择性方案限定）。

condition

任何 SQL 的条件表达式（返回布尔值）。该条件表达式可能不会引用任何表除了 NEW 和 OLD，并且可能不会包含聚集函数。 NEW 和 OLD 引用参考表中的值。NEW 表在 ON INSERT 和 ON UPDATE 规则中有效，以引用要插入或更新的新

行。OLD 表在 ON UPDATE 和 ON DELETE 规则中有效，以引用将要被更新或者删除的行。

INSTEAD

INSTEAD 提示该命令应当替代原命令运行。

ALSO

ALSO 提示该命令应当附加到原命令上运行。如果不指定是 ALSO 也不指定 INSTEAD，则默认是 ALSO。

command

构成规则操作的命令或命令组。有效的命令是 SELECT，INSERT，UPDATE，或 DELETE。特殊表名 NEW 和 OLD 可能会被用来引用在参考表中的值。 NEW 在 ON INSERT 和 ONUPDATE 规则中有效，以引用被更新或插入的新行。OLD 在 ON UPDATE 和 ON DELETE 规则中有效，以引用将被更新和删除的行。

注意

用户必须是表的拥有者才能为其创建和改变规则。

注意避免递归规则是很重要的，递归规则在规则创建时不会被验证，但会在执行时报告错误。

例子

创建一个规则向字表 b2001 中插入行，当用户试图向其分区的父表 rank 中插入的时候:

```
CREATE RULE b2001 AS ON INSERT TO rank WHERE gender='M' and
year='2001' DO INSTEAD INSERT INTO b2001 VALUES (NEW.id,
NEW.rank, NEW.year, NEW.gender, NEW.count);
```

兼容性

就如同整个查询重写系统一样，CREATE RULE 是 HashData 数据库语言的扩展。

另见

DROP RULE， CREATE TABLE， CREATE VIEW

上级话题： SQL命令参考

CREATE SCHEMA

创建方案

定义一个新的方案

概要

```
CREATE SCHEMA schema_name [AUTHORIZATION username]
   [schema_element [ ... ]]

CREATE SCHEMA AUTHORIZATION rolename [schema_element [ ... ]]
```

描述

CREATE SCHEMA 将新的方案输入到当前的数据库中。方案名称必须与当前数据库中现有的名称不同。

方案本质上就是一个命名空间：它包含命名对象（表，数据类型，函数，和操作符）这些名字可能会与该方案中其他对象的名字重叠。命名对象通过将其名称以方案名称作为其前缀进行限定，或者通过设置包含所需方案的搜索路径进行访问。指定非限定对象名称的 CREATE 命令在当前方案中创建对象（搜索路径前面的对象，可以使用函数 current_schema 确定）。

可选择地是， CREATE SCHEMA 可以包含子命令来在新方案中创建对象。这些子命令本质上和创建方案之后发出的单独命令相同，除了使用 AUTHORIZATION 子句，所有创建的对象由该角色有用。

参数

schema_name

创建方案的名称，如果省略，则用户名被用作方案名。该名字不能以 pg_ 开头，因为这些名字是为系统目录方案保存的。

rolename

拥有该方案的角色的名称。如果省略，默认为执行该命令的角色。只有超级用户可以创建属于其他角色的方案，除了他们自己。

schema_element

定义在方案中创建对象的 SQL 语句。目前，只有 CREATE TABLE，CREATE VIEW，CREATE INDEX，CREATE SEQUENCE，CREATE TRIGGER 和 GRANT 被认作为 CREATE SCHEMA 中的子句。其他对象可能在方案创建之后以单独的命令来创建。

> 注意： HashData 数据库不支持触发器

注意

要创建方案，该调用用户必须要有当前数据库的 CREATE 权限或者是超级用户。

示例

创建一个方案:

```
CREATE SCHEMA myschema;
```

为角色 joe 创建一个方案（该方案也叫 joe）：

```
CREATE SCHEMA AUTHORIZATION joe;
```

兼容性

SQL 标准允许 DEFAULT CHARACTER SET 子句在 CREATE SCHEMA 中，还有比现在 HashData 支持的更多的子命令类型。

SQL 标准指定了在 CREATE SCHEMA 中的子命令可以出现在任何顺序。目前 HashData 数据库实现不能处理子命令中所有转发引用的问题。有时，有必要重新排序子命令，以避免转发引用。

根据 SQL 标准，方案的拥有者通常拥有里面的所有对象。 HashData 数据库允许对象包含不是方案对象本身拥有的对象。这当且仅当方案拥有者给其他人授权了 CREATE 权限。

另见

ALTER SCHEMA，DROP SCHEMA

上级话题：SQL命令参考

CREATE SEQUENCE

创建序列

定义一个新的序列生成器。

概要

```
CREATE [TEMPORARY | TEMP] SEQUENCE name
       [INCREMENT [BY] value]
       [MINVALUE minvalue | NO MINVALUE]
       [MAXVALUE maxvalue | NO MAXVALUE]
       [START [ WITH ] start]
       [CACHE cache]
       [[NO] CYCLE]
       [OWNED BY { table.column | NONE }]
```

描述

CREATE SEQUENCE 创建一个序列数字生成器。这包含了创建和初始化一个新的特殊的单行表。该生成器将被执行该命令的用户所拥有。

如果给定了模式名称，则序列在指定的模式上建立。否则在当前的模式中建立。临时序列存在于特殊的模式中，所以创建临时序列时一般不会给定模式名。序列名称不能和同模式中的其他序列，表，索引或视图同名。

创建序列之后，用户可以使用 nextval 函数来操作序列。例如，插入一行到表中要获取序列的下一个值：

```
INSERT INTO distributors VALUES (nextval('myserial'), 'acme');
```

用户也可以使用函数 setval 来操作序列，但是只能对不操作分布式数据的查询。例如，以下的查询是允许的，因为它对主机上的序列生成器进程重置了序列计数器的值：

```
SELECT setval('myserial', 201);
```

但是以下 HashData 数据库中的查询将会拒绝，因为它操作了分布式的数据：

```
INSERT INTO product VALUES (setval('myserial', 201), 'gizmo');
```

在常规数据库中（非分布式的），操作序列的函数去本地序列表获取他们需要的值。但是，在 HashData 数据库中，要注意每个段都是它自己特有的数据库进程。因此段需要单个真值点来获取序列值，以至于所有的段都能正确的递增，并且使得序列能以正确的顺序向前移动。序列服务器进程在主机上运行，是 HashData 分布式数据库中序列的真值点（point-of-truth），段主机在运行时从主机获取序列值。

因为该分布式序列的设计，在对 HashData 数据库操作序列的函数上有一些限制：

- lastval 和 currval 函数不支持。
- setval 只能用来设置主机上序列生成器的值，不能用来在子查询中更新分布式表数据的记录。
- nextval 有时候，根据查询，有时会从主数据库中获取一块值供段数据库使用。所以，如果所有块在段级别不需要，则在序列中跳过该值。注意，常规的 PostgresSQL 数据库也是这样做，所以这不是 HashData 数据库独有的。

尽管用户不能直接更新一个序列，但是用户可以使用以下查询：

```
SELECT * FROM sequence_name;
```

来检查参数和当前序列的状态。尤其是， 序列的 last_value 字段显示了任何会话所分配的最后值。

参数

TEMPORARY | TEMP

如果指定了该字段，则该序列对象仅为该会话所创建，并且会在会话退出时，自动删除。当临时序列存在时，同样名字的存在的永久序列是不可见的（在该会话中）。出给他们引用了该方案限定的名字。

name

所创建的序列的名字（可选方案限定）

increment

指明了在当前序列值上加上一个什么样的增量来得到一个新的值。一个正值会产生一个递增序列，一个负值会产生一个递减序列，默认值是1。

minvalue

NO MINVALUE

决定该序列所能生成的最小的值，如果没有提供该子句或者指定了 NO MINVALUE，则使用默认值。该升序，降序默认值分别为 1 和 -263-1。

maxvalue

NO MAXVALUE

决定该序列产生的最大值，如果没有提供该子句或者指定了 NO MAXVALUE，则将使用默认值。该升序，降序的默认值分别为 263-1 and -1。

start

允许序列能在任何地方开始，该升序的默认开始值为 minvalue，降序的默认开始值为 maxvalue。

cache

指明了有多少预先分配并存在内存中供快速访问的序列值。最小值（也是默认值）是1（不用cache）。

CYCLE

NO CYCLE

序列允许轮回当达到了 maxvalue（对升序来说）或 minvalue（对降序来说。如果达到了限制，下个生成的值将是 minvalue（对升序来说）或者 maxvalue（对降序来说）。如果指定了 NO CYCLE ，任何在序列达到了极限时对 nextval 的调用都将会返回错误。如果没有指定，则默认是 NO CYCLE。

OWNED BY table.column

OWNED BY NONE

会导致序列和一个特定表列相联系，因此如果删除该表列（或者删除整个表），该序列也将会自动删除，该指定的表必须和序列有相同的拥有者，并且在同一模式中。默认值为 OWNED BY NONE 指明没有这样的联系。

注意

序列是基于 bigint 的算术，所以该范围不能超过 8 字节整数的范围（-9223372036854775808 到 9223372036854775807）。

尽管保证多个会话来分配不同的序列值，但当考虑所有会话时必须保证该值按序列所出。例如，会话 A 保留值 1..10 并且返回 nextval=1，然后会话 B 可能在会话 A 生成nextval=2之前保留值 11..20 并且返回 nextval=11。因此，用户应该只假设 nextval 都是不同的，而不是纯粹按顺序生成。此外， last_value 会反应任何会话保留的最新值。无论其是否已被 nextval 返

回。

示例

创建名为 myseq 的序列：

```
CREATE SEQUENCE myseq START 101;
```

使用 next value 向表中插入一行数据：

```
INSERT INTO distributors VALUES (nextval('myseq'), 'acme');
```

重置主机上序列计数器的值：

```
SELECT setval('myseq', 201);
```

HashData 数据库中非法使用 setval（在分布式数据上设置序列的值）：

```
INSERT INTO product VALUES (setval('myseq', 201), 'gizmo');
```

兼容性

CREATE SEQUENCE 遵循 SQL 标准，但是以下除外：

- 不支持该在 SQL 标准中指定的 AS data_type 表达式
- 使用 nextval() 函数替代 SQL 标准中指定的 NEXT VALUE FOR 表达式获得下一个值。
- 该 OWNED BY 子句是 HashData 数据库扩展。

另见

ALTER SEQUENCE，DROP SEQUENCE

上级话题：SQL命令参考

CREATE TABLE AS

创建表如

从查询结果中定义一个新的表

概要

```
CREATE [ [GLOBAL | LOCAL] {TEMPORARY | TEMP} ] TABLE table_name
   [(column_name [, ...] )]
   [ WITH ( storage_parameter=value [, ... ] ) ]
   [ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP}]
   [TABLESPACE tablespace]
   AS query
   [DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY]
```

该 `storage_parameter` 如下：

```
   APPENDONLY={TRUE|FALSE}
   BLOCKSIZE={8192-2097152}
   ORIENTATION={COLUMN|ROW}
   COMPRESSTYPE={ZLIB|QUICKLZ}
   COMPRESSLEVEL={1-9 | 1}
   FILLFACTOR={10-100}
   OIDS[=TRUE|FALSE]
```

描述

CREATE TABLE AS 创建一个表并且使用由 SELECT 命令计算所得的数据进行填充。该表列和 SELECT 命令出来的列有相同的名字和数据类型，但是用户可以通过给出明确的新列名列表复写列名。

CREATE TABLE AS 创建一个新表，并且初始计算查询一次来填充新表。该新表不会同步追踪源表查询子序列的变化。

参数

GLOBAL | LOCAL

这些关键字用于 SQL 标准的兼容性，但是在 HashData 数据库中不起作用。

TEMPORARY | TEMP

如果指定了，该新表被创建成为临时表。临时表在会话结束之时自动删除，或者可选地在当前事务结束时（请参阅 ON COMMIT）。当临时表存在时，在当前会话中，同名的永久表是不可见的。除非使用方案限定的名字来引用，临时表中创建的索引也自动是临时的。

`table_name`

所创建新表的名字（可选方案限定）。

`column_name`

新表的列名，如果没有提供列名，则使用查询输出的列名。如果表从 EXECUTE 命令中所创建，不能指定一个列名列表。

`WITH ( storage_parameter=value )`

该 WITH 子句可以用来为表和索引设置存储选项。注意，用户可以在特定的分区或子分区中设置不同的存储参数，通过在分区中指定 WITH 子句。有以下存储选项：

APPENDONLY — 设置为 TRUE 来创建一个表作为追加优化表。如果设定为 FALSE 或者没有声明，则该表将会创建为常规的堆存储表。

BLOCKSIZE — 对表中每个块设置字节大小。该 BLOCKSIZE 必须在 8192 和 2097152 字节之间，为 8192 的倍数，默认值是 32768。

ORIENTATION — 对面向列存储设置列，或为面向行存储的设置行（默认值）。该选项仅在 APPENDONLY=TRUE 的时候有效。堆存储表只能是面向行的。

COMPRESSTYPE — 设置 ZLIB（默认值）或者 QUICKLZ1 来指定使用的压缩类型。QuickLZ 使用较少的 CPU 功率，并且比 zlib 具有更快的数据压缩和更低的压缩比。相反，zlib 在在较低速率下提供更紧凑的压缩比。此选项仅在 APPENDONLY=TRUE 时有效。

COMPRESSLEVEL — 对追加优化表的 zlib 压缩，设置压缩在 1（最快压缩）到 9（最高压缩比）之间。QuickLZ 压缩等级只能设置到 1。如果没有声明，则默认是 1。该选项仅在 APPENDONLY=TRUE 时才有效。

FILLFACTOR — 更多关于此索引存储参数的信息请参阅 CREATE INDEX。

OIDS — 设置为 OIDS=FALSE（默认值），则行没有分配的对象标识符。 HashData 强烈推荐用户在创建表的时候不要启用 OIDS 。在大型表中，如在典型的 HashData 数据库中表，对表行使用 OIDs 能引起 32OID 计数器环绕。一旦计数器环绕，OIDs 就被认为是不再唯一的，这不仅使它们对用户应用程序无用，而且还可能在 HashData 数据库系统目录表中引起问题。此外，从表中排出 OIDs 减少了磁盘上存储区表所需的空间，每行 4 字节，稍微地提升了性能。不允许在面向列的表上使用 OIDs。

ON COMMIT

事务块结束时的临时表的行为可以使用 ON COMMIT 来控制。该三个选项如下：

PRESERVE ROWS — 在临时表事务结束时不采取特别行动。这是默认行为。

DELETE ROWS — 临时表中的所有行将在每个事务块结束时删除。本质上，在每次提交时候执行了 TRUNCATE。

DROP — 在当前事务块结束时将删除临时表。

TABLESPACE tablespace

创建的新表所在的表空间的名字。如果没有指定，则使用数据库默认的表空间。

AS query

SELECT 或 VALUES 命令，或者运行准备好的SELECT 或 VALUES 查询的 EXECUTE 命令。

DISTRIBUTED BY (column, [...])

DISTRIBUTED RANDOMLY

用来声明 HashData 数据库表的分布策略。 DISTIBUTED BY 使用在分布键中声明一列或者多列进行哈希分布。对于大多数的均匀数据分布，该分布键应当是表的主键或是唯一列（或列的集合。如果不可能，则用户可以选择 DISTRIBUTED RANDOMLY，这会将数据随机循环发送到 Segment 实例。

该 HashData 数据库服务器配置参数 `gp_create_table_random_default_distribution` 控制默认的表分布策略，如果当用户创建表的时候没有指定 DISTRIBUTED BY 子句。如果没有指定分布策略，则 HashData 遵循以下规则来创建表

- 如果查询优化器创建表，并且该参数的值是 off，则该表的分布策略由命令所决定。
- 如果遗传查询优化器创建表，并且该参数的值为 on，则该表的分布策略是随机的。
- GPORCA 创建表，则表分布策略是随机的，该参数值没有影响。

在 HashData 数据库管理员指南中，更多关于该参数的信息，参阅“服务器配置参数”。更多关于遗传查询优化器和 GPORCA 的信息，参阅“查询数据”。

注意

该命令功能上和 SELECT INTO 相似，但是该命令更常用因为它相比使用 SELECT INTO 的语法不太可能产生混淆。此外，

CREATE TABLE AS 提供了包含 SELECT INTO 功能在内超集。

CREATE TABLE AS 可以用于从外部表数据源中快速加载数据。参阅 CREATE EXTERNAL TABLE。

实例

创建一个新表 `films_recent` 仅由表 films 最近的条目组成：

```
CREATE TABLE films_recent AS SELECT * FROM films WHERE
date_prod >= '2007-01-01';
```

创建一个临时表 `films_recent` ，仅由 films 表的最近的条目组成，使用预编译（prepared）语句。新表拥有 OIDs 并在提交时删除：

```
PREPARE recentfilms(date) AS SELECT * FROM films WHERE
date_prod > $1;
CREATE TEMP TABLE films_recent WITH (OIDS) ON COMMIT DROP AS
EXECUTE recentfilms('2007-01-01');
```

兼容性

CREATE TABLE AS 服从 SQL 标准，但以下例外：

- 该标准要求将子查询用括号括起来；在 HashData 数据库中，这些括号是可选的。
- 该标准定义了 WITH [NO] DATA 子句；这目前并没有由 HashData 数据库实现。 HashData 数据库提供的行为和标准的 WITH DATA 情况是一样的。WITH NO DATA 可以通过给查询追加 LIMIT 0 来模拟。
- HashData 数据库处理临时表不同于标准，更多细节参阅 CREATE TABLE。
- 该 WITH 子句是 HashData 数据库扩展； 存储参数和 OIDs 都不在标准中。
- 该 HashData 数据库表空间的概念不是标准的一部分。 该 TABLESPACE 子句是一个扩展。

另见

CREATE EXTERNAL TABLE, CREATE EXTERNAL TABLE, EXECUTE, SELECT, SELECT INTO, VALUES

上级话题： SQL命令参考

CREATE TABLE

创建表

定义一张新表

> 注意：引用完整性约束（外键约束）被接受但不被强制执行。

概要

```
CREATE [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name (
[ { column_name data_type [ DEFAULT default_expr ]
   [column_constraint [ ... ]
[ ENCODING ( storage_directive [,...] ) ]
]
   | table_constraint
   | LIKE other_table [{INCLUDING | EXCLUDING}
                      {DEFAULTS | CONSTRAINTS}] ...}
   [, ... ] ]
   )
   [ 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
           [(...)]
        ) ]
    )
```

其中 `column_constraint` 是：

```
   [CONSTRAINT constraint_name]
   NOT NULL | NULL
   | UNIQUE [USING INDEX TABLESPACE tablespace]
            [WITH ( FILLFACTOR = value )]
   | PRIMARY KEY [USING INDEX TABLESPACE tablespace]
                 [WITH ( FILLFACTOR = value )]
   | CHECK ( expression )
   | REFERENCES table_name [ ( column_name [, ... ] ) ]
            [ key_match_type ]
            [ key_action ]
```

其中一列的 `storage_directive` 是：

```
   COMPRESSTYPE={ZLIB | QUICKLZ | RLE_TYPE | NONE}
    [COMPRESSLEVEL={0-9} ]
    [BLOCKSIZE={8192-2097152} ]
```

其中一个表的 `storage_parameter` 是：

```
    APPENDONLY={TRUE|FALSE}
    BLOCKSIZE={8192-2097152}
    ORIENTATION={COLUMN|ROW}
    CHECKSUM={TRUE|FALSE}
    COMPRESSTYPE={ZLIB|QUICKLZ|RLE_TYPE|NONE}
    COMPRESSLEVEL={0-9}
    FILLFACTOR={10-100}
    OIDS[=TRUE|FALSE]
```

`table_constraint` 是：

```
    [CONSTRAINT constraint_name]
    UNIQUE ( column_name [, ... ] )
           [USING INDEX TABLESPACE tablespace]
           [WITH ( FILLFACTOR=value )]
    | PRIMARY KEY ( column_name [, ... ] )
                 [USING INDEX TABLESPACE tablespace]
                 [WITH ( FILLFACTOR=value )]
    | CHECK ( expression )
    | FOREIGN KEY ( column_name [, ... ] )
            REFERENCES table_name [ ( column_name [, ... ] ) ]
            [ key_match_type ]
            [ key_action ]
            [ key_checking_mode ]
```

其中 `key_match_type` 是：

```
     MATCH FULL
   | SIMPLE
```

其中 `key_action` 是：

```
     ON DELETE
   | ON UPDATE
   | NO ACTION
   | RESTRICT
   | CASCADE
   | SET NULL
   | SET DEFAULT
```

其中 `key_checking_mode` 是：

```
     DEFERRABLE
   | NOT DEFERRABLE
   | INITIALLY DEFERRED
   | INITIALLY IMMEDIATE
```

其中 `partition_type` 是：

```
     LIST
   | RANGE
```

其中 `partition_specification` 是：

```
  partition_element [, ...]
```

`partition_element` 是：

```
   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 ]
```

其中 `subpartition_spec` 或 `template_spec` 是：

```
subpartition_element [, ...]
```

`subpartition_element` 是：

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

其中一个分区的 `storage_parameter` 是：

```
   APPENDONLY={TRUE|FALSE}
   BLOCKSIZE={8192-2097152}
   ORIENTATION={COLUMN|ROW}
   CHECKSUM={TRUE|FALSE}
   COMPRESSTYPE={ZLIB|QUICKLZ|RLE_TYPE|NONE}
   COMPRESSLEVEL={1-9}
   FILLFACTOR={10-100}
   OIDS[=TRUE|FALSE]
```

描述

CREATE TABLE 在当前数据库中创建一个初始为空的表。发出该命令的人拥有该表。

如果用户指定了一个模式名字，HashData 在特定的模式中创建该表。否则 HashData 数据库在当前的模式中创建该表。临时表存在于特殊的模式中，所以用户在创建临时表时不能指定一个模式名。在同一个模式中，表必须和其他表，外部表，序列，索引或者视图不同名。

该可选约束子句指定了条件，该新行或者更新的行必须满足该条件才能成功进行插入或更新操作。约束是一个 SQL 对象，该对象以不同的方式在表中帮助定义有效值集合。约束应用于表，而不是分区，用户不能添加一个约束到分区或者子分区。

引用完整性约束（外键）被接受但不强制执行。该信息保持在系统目录表中，否则将被忽略。

有两种定义约束的方式：表约束和列约束t。列约束被定义为列定义的一部分。表约束不与特定的列相关联，并且它可以包含多个列，每个列约束也能写成表约束；当表约束仅仅影响一列时，列约束仅仅是一种符号方便。

当创建一张表的时候，还有额外的子句来声明 HashData 数据库分布策略。如果没有提供 DISTRIBUTED BY 或者 DISTRIBUTED RANDOMLY 子句，则 HashData 使用 PRIMARY KEY（如果表有的话）或者将该表的第一列作为分布键，给表分配一种哈希分布策略。几何或用户定义的数据类型的列不符合 HashData 分布键列的资格。如果表没有一个符合条件的数据类型的列，那么这些行则是基于循环或随机分布分配的。为了确保用户的 HashData 数据库系统中的数据均匀分配，用户需要选择对每个记录唯一的分布键，否则，请选择随机分布 DISTRIBUTED RANDOMLY。

该 PARTITION BY 子句允许用户将表划分成多个子表（或部分），他们组合在一起构成父表并共享其模式。尽管子表以独立表的形式存在，但是 HashData 数据库以重要的方式限制了他们的使用。在内部，分区被实现为一种特殊形式的继承。每个子表分区的创建带有不同 CHECK 约束，该约束根据一些定义标准限制表可以包含的数据。查询优化器也使用 CHECK 约束来确定要扫描的分区，以满足给定的谓词。这些分区约束由 HashData 数据库自动管理。

参数

GLOBAL | LOCAL

这些参数用于 SQL 标准兼容性，但在 HashData 数据库中不起作用。

TEMPORARY | TEMP

如果指定，则该表被创建成临时表。临时表在会话结束时自动删除，或选择性地在当前事务结束的时候删除（参阅 ON COMMIT）。当临时表存在时当前会话中具有同名的永久表是不可见的，除非他们使用方案限定的名字来引用。任何在临时表中创建的索引也都是临时的。

`table_name`

所创建表的名字（可选方案限定）。

`column_name`

新表中所要创建的列名。

`data_type`

列的数据类型，这可能包含数据说明符。

对于包含文本数据的表列，通过使用 VARCHAR 或 TEXT 指定数据类型。不建议指定数据类型 CHAR。在 HashData 数据库中，该数据类型 VARCHAR 或 TEXT 将添加到数据之后的补充（最后一个非空格字符之后添加的空格字符）作为有效字符，而数据类型 CHAR 却不会。

DEFAULT `default_expr`

该 DEFAULT 子句给其列定义出现的列分配一个默认值。该值任何无变量表达式（不允许对当前表中其他列的子查询或者交叉引用）。该默认表达式的数据类型必须匹配该列的数据类型。该默认表达式将会在任何对该列没有指定值的插入操作中使用。如果对该列没有默认值，则该默认值为 null。

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

对于列，可选的 ENCODING 子句指定列数据的压缩类型和块大小。

该子句仅对追加优化，面向列的表(AOCS)有效。

列压缩设置从表级继承到分区级再到子分区级别。最低级设置优先。

INHERITS

该可选 INHERITS 子句指定了新表自动继承所有列的表的列表（list of tables）。使用 INHERITS 在新的子表和其父表之间创建持久的关系。对父表的模式修改通常也传播给子表，默认情况下，子表的数据包含在父表的扫描中。

在 HashData 数据库中，在创建分区表的时候该 INHERITS 子句是不用的。尽管在分区层次体系中使用了继承的概念，但是分区表的继承结构是由 PARTITION BY 子句所创建的。

如果同样的列名存在于多个父表中，则会报告错误，除非列数据类型在每个父表中都匹配。如果没有冲突，则将重复的列合并在新表中形成单一的列。如果新表的列名列表包含继承的列名称，则数据类型必须与继承的列一致，列定义合并为一。但是，相同名字的继承列和新列的声明不需要指定相同约束：任何声明中提供的所有约束都合并在一起，并且都应用于新表。如果新表为一列明确指定了默认值，则该默认值覆盖该列继承声明中的任何默认值。否则，任何为该列指定默认值的父表都必须指定相同的默认值。否则将报告错误。

LIKE `other_table` [{INCLUDING | EXCLUDING} {DEFAULTS | CONSTRAINTS}]

该 LIKE 子句指定新表将自动复制所有列名，数据类型，非空约束和分发策略的表。诸如追加优化或分区结构之类的存储属性不被复制。与 INHERITS 不同，新表创建完成之后，新表和原始表完全解耦。

复制的列定义的默认表达式仅在指定了 INCLUDING DEFAULTS 时才会被复制。默认行为是排除默认表达式，该操作导致复制的列在新表中使用 null 的默认值。

`Not-null` 约束总是会被复制到新表中。仅当指定了 INCLUDING CONSTRAINTS 时，才会复制 CHECK 约束。其他类型的约束将 永远不会被复制。此外，当请求约束时，列约束和表约束之间不作区分，所有的检查约束都将会被复制。

还要注意，不像 INHERITS，复制的列和约束不会与命名相似的列和约束合并。如果明确指定相同的名字，或在另一个 LIKE 子句中，则会提示错误。

CONSTRAINT `constraint_name`

列或表约束的可选名称。如果违反该约束，则该约束名称会出现在错误消息中，因此诸如 该列必须为正数 的约束名称，可以用来向客户端应用传达有用的约束信息。（需要用双引号来指定包含空格的约束名称）。如果没有指定约束名称，则该系统将生成一个名称。

> 注意： 该指定的 `constraint_name` 用于约束，但是系统生成的唯一名字是用于索引名称。在先前的版本中，提供的名称用于约束名称和索引名称（both）。

NULL | NOT NULL

指明该列是否允许包含 null。默认值是 NULL。

UNIQUE (column constraint)

UNIQUE (`column_name` [, ...]) (table constraint)

该 UNIQUE 约束指定表中一列或多列的组合只包含唯一值。该唯一表约束的行为和列约束相同，具有跨多个列的附加功能。为了唯一约束的目的，空值不被认为是相等的。该唯一列必须包含 HashData 分布键的所有列。此外，如果表被分区，该 `<key>` 必须包含分区键中的所有列。请注意，分区表中的 `<key>` 约束和简单的 simple UNIQUE INDEX 不同。

PRIMARY KEY (column constraint)

PRIMARY KEY (`column_name` [, ...]) (table constraint)

该主键约束指定表的一列或多列只能包含唯一（非重复）非空值。技术上，PRIMARY KEY 仅仅是 UNIQUE 和 NOT NULL 的组合，但是将一组列标识为主键还提供了关于设计模式的元数据，因为主键意味着其他表可能依赖这组列作为行的唯一标识符。要使表具有主键，它必须是哈希分布（不是随机分布），唯一的主键列必须包含 HashData 分布键的所有列。另外，如果表被分区，则 `<key>` 必须包含分区键的所有列。请注意，分区表中的 `<key>` 约束和简单的 UNIQUE INDEX不同。

CHECK (expression)

该 CHECK 指定了一个表达式，该表达式产生布尔结果，新的或更新行必须满足该表达式才能成功插入或更新操作。 被评估为 TRUE 或者 UNKNOWN 的表达式算是成功。 如果插入或更新的任何行产生 FALSE 结果，则会引发错误异常，并且插入或更新不会更改数据库。指定为列约束的检查约束应仅引用该列的值，而表约束中出现的表达式可能会引用多个列。CHECK 表达式不能包含子查询也不能引用当前行以外的值。

```
REFERENCES table_name [ ( column_name [, ... ] ) ]

[ key_match_type ] [ key_action ]

FOREIGN KEY ( column_name [, ... ] )

REFERENCES table_name [ ( column_name [, ... ] )

[ key_match_type ] [ key_action [ key_checking_mode ]
```

该 REFERENCES 和 FOREIGN KEY 子句指定引用完整性约束（外键约束）。 HashData 接受PostgreSQL 语法中指定的引用完整性约束，但是不执行他们。

`WITH ( storage_option=value )`

该 WITH 子句可以用来为表或索引设置存储选项。注意用户也能在分区说明中，通过声明 WITH 来在特定的分区或子分区上设置存储参数。最低级设置优先。

某些表存储选项的默认值可以使用服务器配置参数 `gp_default_storage_options` 来指定。

有以下存储选项：

APPENDONLY — 设置为 TRUE 来创建一个表作为追加优化表。如果设置为 FALSE 或者没有指定，则该表将会被创建为常规的堆存储表。

BLOCKSIZE — 对表中的每个块设置大小。该 BLOCKSIZE 必须在 8192 和 2097152 字节之间并且必须是 8192 字节的倍数。该默认值是 32768。

ORIENTATION — 设置为 column 是面向列存储，或设置为 row（默认值）是面向行存储。该选项仅当 APPENDONLY=TRUE 时才有效。堆存储表只能是面向行的。

CHECKSUM — 该存储表仅对追加优化表（append-optimized tables）才是有效的，即（APPENDONLY=TRUE）。该值 TRUE 时默认的并且为追加优化表启用 CRC 校验和验证。该校验在块创建期间计算并存储在磁盘上。校验验证在块读取期间执行。如果读取期间计算的校验和存储的校验不匹配，则事务终止。如果用户设置该值为 FALSE 来禁用校验验证，则会启动检查表数据防止磁盘损坏。

COMPRESSTYPE — 设置为 ZLIB（默认值），`RLE-TYPE`，或 QUICKLZ1 来指明使用的压缩类型。该值 NONE 禁用压缩。QuickLZ 比 zlib 使用更少量的 CPU 功率，具有更低的压缩比和更快的数据压缩。zlib 提供了在低压缩速度下高压缩比。该选项仅当 APPENDONLY=TRUE 时才有效。

该值 `RLE_TYPE` 仅在指定 ORIENTATION =column 的时候才支持。 HashData 数据库使用运行长度编码（RLE）压缩算法。当同样的数据出现在连续的行中，RLE 比 zlib 和 QuickLZ 压缩算法能更好压缩数据。

对于类型为 BIGINT， INTEGER，DATE，TIME，或 TIMESTAMP 的列，如果 COMPRESSTYPE 选项设置为 RLE-TYPE 压缩，同样会应用增量压缩。该增量压缩算法是基于连续的行中列值之间的增量，并且设计为当加载的数据是有序或者要压缩的列数据是有序时，能提高压缩性能。

COMPRESSLEVEL — 对于追加优化表的 zlib 压缩，设置为 1（最快压缩）到 9（最高压缩比）之间的整数。QuickLZ 压缩等级只能设置为 1，如果没有指定，则默认为 1。对于 `RLE_TYPE`，该压缩比能设置在 1（最快压缩）到 4（最高压缩比）之间的整数。

该选项仅当 APPENDONLY=TRUE 时才有效。

FILLFACTOR — 参阅 CREATE INDEX 获取更多关于该索引存储参数的信息。

OIDS — 设置为 OIDS=FALSE（默认值），以便行没有分配对象表示符。 HashData 强烈推荐用户在创建表时不启用 OIDs。在大型表中（如典型的 HashData 数据库系统中的表），对表行使用 OIDs 会导致 32 位 OID 计数器环绕。一旦计数器环绕，OID 就不能被认为是唯一的，这不仅会导致他们对用户程序无用，而且还可能在 HashData 数据库系统目录中引起问题。此外，从表中排出 OID 会减少表在磁盘所需存储的空间，每行 4 字节，这会稍微提升性能。OIDs 不允许在分区表或者追加优化表面向列的表中使用。

ON COMMIT

在事务块结束时临时表的行为可以通过使用 ON COMMIT 来控制。该三个选项如下：

PRESERVE ROWS - 在事务结束时候没有对临时表特别的操作，这是默认的行为。

DELETE ROWS - 在每个事务块结束时，临时表的所有行都将被删除。本质上，在每次提交时，执行了自动删除 TRUNCATE。

DROP - 在当前事务结束时，会删除临时表。

TABLESPACE tablespace

要创建的新表所在的表空间的名字，如果没有指定，使用数据库的默认空间。

```
USING INDEX TABLESPACE tablespace
```

该子句允许表空间的选择，在该表空间中和 UNIQUE 或 PRIMARY KEY 相联系的索引将会被创建。如果没有指定，则使用数据库默认表空间。

```
DISTRIBUTED BY (column, [ ... ] )

DISTRIBUTED RANDOMLY
```

使用来声明 HashData 数据库中表的分布策略。 DISTIBUTED BY 使用在分布键中声明的一行或者多行做哈希分布。对于最均匀的数据分布，该分布键应当是表的主键或者一个唯一列（或者唯一多列）。如果这是不可能的，则用户可能会选择 DISTRIBUTED RANDOMLY，该策略会循环发送数据到Segment实例。

该 HashData 数据库服务器配置参数 `gp_create_table_random_default_distribution` 控制默认表分布策略，如果当用户创建表的时没有指定 DISTRIBUTED BY 子句。 HashData 遵循以下规则来创建表，当没有指定分布策略时。

如果参数的值时 off（也是默认值）， HashData 数据库基于命令选择表分布键。如果在表的创建命令中指定了该 LIKE 或 INHERITS 子句，则该创建的表使用同源表或者父级表的分布键相同的分布键。

如果该参数设置成了 on，则 HashData 数据库遵循以下规则：

- 如果没有指定 PRIMARY KEY 或 UNIQUE 列，则该表分布式随机的（DISTRIBUTED RANDOMLY）。即使表的创建命令包含 LIKE 或者 INHERITS 子句，该表的分布还是随机的。
- 如果指定了 PRIMARY KEY 或 UNIQUE 列，则 DISTRIBUTED BY 子句必须也要指定。如果没有指定 DISTRIBUTED BY 子句作为表创建命令的一部分，则命令失败。

PARTITION BY

声明一列或多列，根据该列来将表分区。

当创建一个分区表， HashData 数据库使用指定的表名创建根分区表（该根分区）。 HashData 数据库也会根据用户指定的分区选项创建子分区包括里面表的层次结构，子表。该 HashData 数据库 `pg_partition_*` 系统视图包含子分区表的信息。

对于每个分区级别（每个表的层次结构），分区表做多可以有 32,767 个分区。

> 注意：HashData 数据库将分区表数据存储在叶子表中，子表的层次结构中的最低级表用于分区表。

`partition_type`

声明分区类型：LIST（值列表）或者 RANGE（数字或日期范围）。

`partition_specification`

声明要创建各个分区，每个分区可以被单独定义或者为范围分区，用户可以使用 EVERY 子句（用一个 START 和可选的 END 子句）来定义用于创建各个分区的增量模式。

DEFAULT PARTITION name — 声明一个默认分区，当数据不匹配存在的分区时，则被插入到默认分区中。没有默认分区的设计将拒绝与现有分区不匹配的新的数据行的插入。

PARTITION name — 声明要用于分区的名字，使用以下命名约定创建分区： `parentname_level#_prt_givenname` .

VALUES — 对于列表分区，定义分区会包含的值。

START — 对于范围分区，定义分区的开始范围值。默认的是，开始值为 INCLUSIVE。 例如，如果用户声明了 `'2016-01-01'` 的开始日期，则该分区会包含所有大于等于 `'2016-01-01'` 的日期。通常， START 表达式的数据类型和分区键列的数据类型是一样的。如果不是这样，用户必须显式地转换为预期的数据类型。

END — 对于范围分区，定义分区的结束范围值。默认的是，结束值是 EXCLUSIVE。 例如，如果用户定义了 `'2016-02-01'` 的结束日期，则该分区将会包含所有小于该 `'2016-02-01'` 的日期。通常， END 表达式的数据类型和分区键列的数据类型是一样的。如果不是这样，用户必须显式地转换为预期的数据类型。

EVERY — 对于范围分区，定义如何将值从 START 增加到 END 以创建单个分区。通常，EVERY 表达式的数据类型和分区键列的数据类型相同。如果不是这样，则用户必须显式的准换为预期的数据类型。

WITH — 设置分区的存储选项，例如，用户可能希望老分区是追加优化表，而较新的分区则为常规的堆表。

TABLESPACE — 要创建的分区所在的表空间的名字。

SUBPARTITION BY

声明一列或多列来对表的第一级分区进行子分区。子分区规范的格式与上述分区规范的格式相似。

SUBPARTITION TEMPLATE

用户可以选择在声明一个用于创建自分区（较低级字表）的子分区模板，而不是为每个分区单独声明每个子分区定义。然后，此子分区规范将应用于所有父分区。

注意

- 在 HashData 数据库中（一个基于 Postgres 的系统）数据类型 VARCHAR 或 TEXT 处理添加到文本数据后的补充（添加到最后一个非空格字符后面的空格）作为有效的字符。而数据类型 CHAR 则不会。

  HashData 数据库中，CHAR(n) 类型的值被填充尾部空格来达到指定的宽度 n。该值存储并显示空格。然而T，该添加的空格被视为语义不重要的。当值分布时，尾部的空格被忽略。当比较 CHAR 类型的值时，尾部的空格也被视为语义不重要的，并且将字符值转化为其他字符串类型之一时，尾部空格被删除。

- 不建议在新的应用中使用 OIDs：在可能的情况下，使用 SERIAL 或其他序列生成器作为表的主键的首选。但是，如果用户的应用确实使用 OID 来标识表的特定行，则推荐在该 OID 列上创建唯一的约束，以确保表中的 OID 确实唯一标识行，即使在计数器轮回之后。避免假设 OID 在多表之间（across）是唯一的；如果用户需要数据库范围内的唯一标识符号，则可以使用表 OID 和行 OID 的组合。

- HashData 数据库对主键和涉及 HashData 表中分布键的列的唯一约束有一些特殊的条件。对 HashData 数据库中强制执行的唯一约束，表必须是哈希分布的（不是 DISTRIBUTED RANDOMLY），约束列必须表的分布键列（或超集）相同。此外，分布键必须是列约束的左子集并且列的顺序是正确的。例如，如果主键是（a，b，c），该分布键只能是以下集合之一：（a），（a，b），或（a，b，c）。

  主键约束仅仅是唯一约束和非空约束的组合。

  HashData 数据库自动创建 UNIQUE 索引为每个 UNIQUE 或 PRIMARY KEY 约束来强制执行唯一性。因此，无需为主键列显式创建索引。在追加优化表上不允许使用 UNIQUE 和 PRIMARY KEY 约束，因为由约束创建的 UNIQUE 索引在追加优化表上不允许。

  HashData 数据库不支持外键约束。

  对于继承的表，唯一约束，主键约束，索引和表权限在当前实现中不会继承。

- 对于追加优化表，UPDATE 和 DELETE 在序列化事务中不被允许，并且会导致事务中断。
  CLUSTER，DECLARE...FORUPDATE，和 触发器在追加优化表中不支持。

- 要将数据插入到分区表中，请指定根分区表，即用 CREATE TABLE 命令创建的表。用户也可以在 INSERT 命令中指定分区表的叶子表。如果数据不符合指定的叶子表，则会返回错误。不支持在 INSERT 命令中指定不是叶子表的子表。不支持在分区表任何子表上执行 DML 命令，如 UPDATE 和 DELETE。这些命令必须在根分布表上执行，即用 CREATE TABLE 命令创建的表。

- 这些表存储选项的默认值可以通过服务器配置参数 `gp_default_storage_option` 指定。

  - APPENDONLY
  - BLOCKSIZE
  - CHECKSUM
  - COMPRESSTYPE
  - COMPRESSLEVEL
  - ORIENTATION

  该默认值可以设置于数据库，模式和用户。关于设置存储选项的信息，请参阅服务器配置参数 `gp_default_storage_options`。

> 重要： 当前 HashData 数据库遗传优化器允许具有多列（复合）分区键的列表分区。GPORCA 不支持复合键，所以不建议使用复合分区键。

示例

创建一个名为 rank 的表在名为 baby 的模式中并且使用 rank，gender，和 year 来分布数据：

```
CREATE TABLE baby.rank (id int, rank int, year smallint,
gender char(1), count int ) DISTRIBUTED BY (rank, gender,
year);
```

创建表 films 和表 distributors（默认会使用主键作为 HashData 的分布键）：

```
CREATE TABLE films (
code        char(5) CONSTRAINT firstkey PRIMARY KEY,
title       varchar(40) NOT NULL,
did         integer NOT NULL,
date_prod   date,
kind        varchar(10),
len         interval hour to minute
);

CREATE TABLE distributors (
did    integer PRIMARY KEY DEFAULT nextval('serial'),
name   varchar(40) NOT NULL CHECK (name <> '')
);
```

创建 gzip 压缩的追加优化表：

```
CREATE TABLE sales (txn_id int, qty int, date date)
WITH (appendonly=true, compresslevel=5)
DISTRIBUTED BY (txn_id);
```

创建 3 级分区表使用，使用子分区模板和每级的默认分区：

```
CREATE TABLE sales (id int, year int, month int, day int,
region text)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)

  SUBPARTITION BY RANGE (month)
    SUBPARTITION TEMPLATE (
       START (1) END (13) EVERY (1),
       DEFAULT SUBPARTITION other_months )

  SUBPARTITION BY LIST (region)
    SUBPARTITION TEMPLATE (
       SUBPARTITION usa VALUES ('usa'),
       SUBPARTITION europe VALUES ('europe'),
       SUBPARTITION asia VALUES ('asia'),
       DEFAULT SUBPARTITION other_regions)

( START (2008) END (2016) EVERY (1),
  DEFAULT PARTITION outlying_years);
```

兼容性

CREATE TABLE 命令服从 SQL 标准，还有以下的例外：

- 临时表 — 在 SQL 标准中，在需要他们的每个会话中，临时表只定义一次并自动存在（从空的内容开始）。 HashData 数据库需要每个会话为每个要使用的临时表发出自己的 CREATE TEMPORARY TABLE 命令。这允许不同的会话为不同目的使用相同的临时表名称，而标准的方法限制给定临时表名的实例具有相同的表结构

  在全局和本地临时表之间的标准区别不存在于 HashData 数据库中。 HashData 数据库会在临时表的声明中接收 GLOBAL 和 LOCAL 关键字，但是他们没有影响。

  如果 ON COMMIT 子句被省略，该SQL标准指定默认行为为 ON COMMIT DELETE ROWS。而然， HashData 数据库中的默认行为为 ON COMMIT PRESERVE ROWS。该 ON COMMIT DROP 选项不存在与SQL标准中。

- 列检查约束 — 该 SQL 标准说 CHECK 列约束可能仅指所应用到的列；只有 CHECK 表约束可能指定到多列。 HashData 数据不强制执行该约束，它对待列和表的约束一样。

- **NULL** 约束 — 该 NULL 是 HashData 数据库对 SQL 标准的扩展，为了是和其他数据库系统的兼容（为了和 NOT NULL 约束的对称）。因为它是任何列的默认值，因此不需要它的存在。
- 继承 — 多重继承通过 INHERITS 子句是 HashData 数据库语言的扩展。SQL:1999 和更高的版本使用不同的语法和语义来定义单个继承。 HashData 数据库不支持 SQL:1999 版的继承。
- 分区 — 表分区通过使用 PARTITION BY 子句是 HashData 数据库语言扩展。
- 零列表 — HashData 数据库允许 0 列表的创建（例如，CREATE TABLE foo();）。这是从不支持 0 列表的 SQL 标准来的扩展。0 列表本身不是非常有用，但是不允许他们对 ALTER TABLE DROP COLUMN 语句会创建奇怪的特殊情况的表，所以 HashData 决定忽视此规范限制。
- **WITH**子句 — 该 WITH 子句是 HashData 数据库扩展；存储参数和 OID 都不是 SQL 标准中的。
- 表空间 — 该 HashData 表空间的概念 SQL 标准的一部分。 该子句 TABLESPACE 和 USING INDEX TABLESPACE 是扩展。
- 数据分布 — 该 HashData 数据库并行和分布数据库的概念不是 SQL 标准的一部分。该 DISTRIBUTED 子句是扩展。

另见

ALTER TABLE, DROP TABLE, CREATE EXTERNAL TABLE, CREATE TABLE AS

上级话题： SQL命令参考

CREATE TABLESPACE

创建表空间

定义一个新的表空间

概要

```
CREATE TABLESPACE tablespace_name [OWNER username]
       FILESPACE filespace_name
```

描述

CREATE TABLESPACE 为用户的 HashData 数据库系统注册一个新的表空间。该表空间的名字必须和系统中已存在其他任何表空间不同名。

一个表空间允许超级用户在文件系统上定义可选择的位置，该位置中的驻有包含数据库对象（表和索引）在内的数据文件。

有适当权利的用户可以传递一个表空间名到 CREATE DATABASE，CREATE TABLE，或 CREATE INDEX 中来使这些对象的数据文件存储在指定的表空间中。

在 HashData 数据库中，必须对 Master、每个 Segment，每个 Segment 镜像都有一个定义好的文件系统，为的就是让表空间在整个 HashData 系统中有位置来存储它的对象。该文件系统位置的集合定义在文件空间对象中。在用户创建一个表空间之前，必须定义一个文件空间。

参数

tablespacename

要创建表空间名字，该名字不能以 `pg_` 或 `gp_` 开头，因为这些名字是为系统表空间预留的。

OWNER username

拥有该表空间用户的名字。如果省略，默认为执行该命令的用户。只有超级用户可以创建表空间，但是他们能分配表空间的所属权给非超级用户。

FILESPACE

HashData 数据库表空间的名字，该表空间由 gpfilespace 管理实用程序所定义。

注意

用户必须首先创建一个由表空间使用的文件空间。

表空间仅在支持符号链接的系统上支持。

CREATE TABLESPACE 不能在事务块中执行。

示例

通过指定相应的要用文件空间，创建一个新的表空间。

```
CREATE TABLESPACE mytblspace FILESPACE myfilespace;
```

兼容性

CREATE TABLESPACE 是 HashData 数据库扩展。

另见

CREATE DATABASE, CREATE TABLE, CREATE INDEX, DROP TABLESPACE, ALTER TABLESPACE, gpfilespace

上级话题： SQL命令参考

CREATE TYPE

创建类型

描述一个新的数据类型。

概要

```
CREATE TYPE name AS ( attribute_name data_type [, ... ] )

CREATE TYPE name AS ENUM ( 'label' [, ... ] )

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [, RECEIVE = receive_function]
    [, SEND = send_function]
    [, TYPMOD_IN = type_modifier_input_function ]
    [, TYPMOD_OUT = type_modifier_output_function ]
    [, INTERNALLENGTH = {internallength | VARIABLE}]
    [, PASSEDBYVALUE]
    [, ALIGNMENT = alignment]
    [, STORAGE = storage]
    [, DEFAULT = default]
    [, ELEMENT = element]
    [, DELIMITER = delimiter] )

CREATE TYPE name
```

描述

CREATE TYPE 在当前数据库中注册了一个新的要使用的数据类型。定义该类型的用户是他的拥有者。

如果给定了模式，则该类型就会在指定的模式中创建。否则，会在当前的模式中创建。该类型的名字必须和该模式中任何存在的类型或域的名字不同。该类型的名字也必须和同模式中存在的表的名字不同。

复合类型

该 CREATE TYPE 的第一个形式创建了一个复合类型。该复合类型由属性名字和数据类型的列表指定。这本质上和表的行类型一样，但是，当仅需要定义一个类型时，使用 CREATE TYPE 避免了需要去创建实际的表。独立复合类型作为参数或者函数的返回类型来说是有用的。

枚举类型

该 CREATE TYPE 的第二种形式创建了一个（ENUM）的枚举类型，正如在 PostgreSQL 文档中描述的 枚举类型。枚举类型列出一个或多个引用的标签，每个标签必须小于 NAMEDATALEN 个字节长（标准中为 64）。

基础类型

该 CREATE TYPE 的第三种形式创建了一个新的基础类型（标量类型）。该参数可能会以任何顺序出现，不仅在语法中显示，大多数都是可选的。用户在定义类型之前，必须注册 2 个或更多函数（使用 CREATE FUNCTION）。支持函数 `input_function` 和 `output_function` 时必须的， 然而 `receive_function` ， `send_function` ， `type_modifier_input_function` ， `type_modifier_output_function` 和 `analyze_function` 函数是可选的。一般来说，这些函数必须是以 c 编写或者其他低级语言。在 HashData 数据库中，任何用于实现一个数据类型的函数必须定义成 IMMUTABLE。

该 `input_function` 将类型的外部文本表示转换为为之定义的操作符和函数使用的内部表示形式。 `output_function` 执行反向转换，该输入函数可以被声明为采用一个 cstring 类型参数，或者声明为采用 cstring, oid，integer 三个参数。第一个参数是

输入文本作为 C 字符串，第二个参数是类型自己的 OID（数组类型除外，它是接收它元素类型的 OID），第三个参数是目标列的 typmod，（如果已知的话）（如果未知的话，会传递-1）。该输入函数必须返回数据类型本身的值。通常，一个输入函数应该被声明为 STRICT；如果不是，当读取 NULL 输入值时，将使用 NULL 作为第一个参数来调用函数。这种情况下，该函数必须仍然返回 NULL 值，除非他引发错误。（这种情况主要是为了支持域输入功能，该可能需要拒绝 NULL 输入。）输出函数必须声明为采用新数据类型的一个参数。该输出函数必须返回 cstring。输出函数不会为 NULL 值调用。

该可选函数 `receive_function` 转换类型的外部二进制表示到内部表示。如果不支持这个函数，则该类型不能参与二进制输入。该二进制表示应该被选择为便于转换为内部形式，同时可以相当便携。（例如，标准整数数据类型使用网络字节顺序作为外部二进制表示，而内部表示是本地机器字节顺序）接收函数应执行足够的检查来确保值是有效的。接收函数可以被声明为使用一个参数类型为 internal 的函数，或者使用 internal，oid，integer 三个参数类型。第一个参数是一个指向 StringInfo 缓冲区的指针，它保存接收到的字节串；可选参数与文本输入函数相同。接收函数必须返回该数据类型的值。通常，接收函数应声明为 STRICT；如果不是，当读取 NULL 输入值时，将使用 NULL 为第一个参数调用此函数，这种情况下，函数必须返回 NULL，除非它引发错误。（这种情况主要是为了支持域接收功能，可能需要拒绝 NULL 输入。)同样，可选的 `send_function` 将从内部表示转换为外部二进制表示。如果未提供此功能，则该类型不能参与二进制输出。发送函数必须声明为采用新数据类型为一个参数，该发送函数必须返回类型为 bytea 的数据。发送函数必须为 NULL 值调用。

如果该类型支持修饰符，则需要可选的 `type_modifier_input_function` 和 `type_modifier_output_function` 。修饰符是附加到类型声明上的约束，例如 char(5) 或 numeric(30,2)。然而 HashData 数据库允许用户定义的类型将一个或多个简单常量或者标识符作为修饰符，但是该信息必须适合单个非负整数值，以便在系统目录中存储。 HashData 数据库将声明的修饰符以cstring 数组的形式传递给 `type_modifier_input_function` 。该修饰符输入函数必须检查值的有效性，如果不正确抛出异常。如果值正确，该修饰符输入函数返回单个非负整数值， HashData 数据库将把之作为列 typmod 存储。如果该类型未使用 `type_modifier_input_function` 定义，则类型修饰符将被拒绝。该 `type_modifier_output_function` 将内部整数 typmod 值转换为正确的表单，以供用户显示。该修饰符输出函数必须返回一个 cstring 值，该值是要附加到类型名称的确切的字符串。例如，numeric's 函数可能返回 (30,2)。 该 `type_modifier_output_function` 是可选的。当没有指定时，该默认显示形式是存储的括号中的 typmod 整数值。

当必须在创建新类型之前创建他们，用户应该在这一点上想知道输入和输出函数如何被声明为具有新类型的结果或参数。答案应该是首先将该类型定义为一个 shell 类型，他是一个占位符，除名字和所有者之外没有其他属性。这可以通过发出命令 CREATE TYPE name 来完成，没有其他参数。然后可以引用 shell 类型定义 I/O 函数。最后，具有完整定义的 CREATE TYPE 将使用完整的有效的类型定义替换 shell 条目，之后，可以正常使用新的类型定义了。

虽然新类型的内部表示的细节只有 I/O 函数和用户创建的该类型的其他函数才了解，但是内部表示的几个属性必须声明到 HashData 数据库。其中最重要的额是 internallength。基本数据类型可以使固定长度的，这种情况下，internallength 是一个正整数，或者可变长度，通过将 internallength 设置为 VARIABLE 来表示。（在内部，这通过将 typlen 设置成 -1 来表示。) 所有可变长度类型的内部表示必须以 4 字节整数开头，给出此类型值的总长度。

该可选标志 PASSEDBYVALUE 表示此数据类型的值通过值传递而不是引用传递。用户不能通过类型来传递，该类型的内部表示比 Datum 类型内部表示大小（size）大（大多数机器上为 4 个字节，少数机器上为 8 字节）。

该 aligment 参数指定数据类型所需的存储对齐。允许的值等于 1，2，4 或 8 字节边界上的对齐。注意，可变长度类型必须具有至少 4 的对齐，因为它们必须包含 int4 作为其第一个组成数。

该 storage 参数允许变长数据类型存储策略的选择。（对于固定长度类型，只允许 plain 策略）plain 指定类型的数据将始终线性存储，而不是压缩。extended 指定系统将首先尝试压缩长数据值，如果它仍然太长，将值移动到主表行之外。external 允许将值移出主表，但是该系统不会尝试压缩它。main 允许压缩，但是不鼓励将数据移出主表。（如果没有其他的方式使得数据适应行，指定该策略的数据项可能还是会被移出主表，但是它会优先于 extended 和 external 选项将之保留在主表中)

可以指定默认值，以防用户希望数据类型的列默认为非空值。使用 DEFAULT 关键字来指定默认值。（这样的默认值可能会被附加到该列显式的 DEFAULT 子句覆盖。）

要指定类型为数组，请使用关键字 ELEMENT 关键字指定数组元素的类型。例如，要定义一个 4 字节整数的（int4）的数组，请指定 ELEMENT = int4。更多关于数组类型的更多细节如下所示。

要指定在该类型数组内部表示值之间的分隔符，可以将 delimiter 设定为特殊的字符。默认的分隔符是逗号（，）。注意，该分隔符号与数组元素类型相关联，而不是数组类型本身。

数组类型

每当创建用户定义的基本数据类型时， HashData 数据库自动创建相关联的数组类型，其名称由前缀为下划线的基本类型名

称组成。解析器了解这个命名约定，并将类型为 foo[] 的列的请求转换为类型为 _foo 的请求。隐式创建的数组类型是可变长度，并使用内置的输入和输出函数 `array_in` 和 `array_out` 。

如果系统自动创建正确的数组类型，用户可能会合理地问为什么有一个 ELEMENT 选项。使用 ELEMENT 有用的唯一的情况是当用户创建固定长度的类型时，恰好在内部是相同数字的数组，用户希望允许直接通过下标访问这些东西，除了用户计划为该类型整体提供的任何操作。例如，类型 name 允许以这种方式访问其组成的 char 元素。2-D 点类型可以允许其两个组成数字可以像 point[0] 和 point[1] 被访问。注意，该便利，仅适用于固定长度类型，其内部格式正好是相同固定长度字段的序列。可下标表示的可变长度类型必须有由 `array_in` 和 `array_out` 使用的广义内部表示。由于历史原因，固定长度数组类型的下标从零开始，而可变长度类型却不行。

参数

name

要创建类型的名字（可选方案限定）。

`attribute_name`

复合类型属性的（列）名字。

`data_type`

成为复合类型列的存在的数据类型的名字。

label

表示和枚举类型一个值相关联的文字标签的字符串文字。

`input_function`

将数据从类型的外部文本形式转换为内部表单的函数的名称。

`output_function`

将数据从类型的内部表单转换为外部文本形式的函数的名称。

`receive_function`

将数据类型的外部二进制形式转换为其内部形式的函数的名称。

`send_function`

将数据类型的内部表单转换为其外部二进制形式的函数的名称。

`type_modifier_input_function`

将类型的修饰符数组转化为内部形式的函数的名称。

`type_modifier_output_function`

将类型修饰符的内部形式转化为外部文本形式的函数的名称。

internallength

一个数字常量，用于指定新类型内部表示的字节长度。默认假设是变长的。

alignment

数据类型的存储对齐要求。必须是 char，int2，int4， 或 double 之一。默认值是 int4。

storage

数据类型的存储策略，必须是 plain， external，extended，或 main 之一。 默认为 plain。

default

数据类型的默认值，如果省略，默认值为 null。

element

正在创建的类型是数组；这指定了数组元素的类型。

delimiter

此类型数组中值之间要使用的分隔符。

注意

用户定义的类型名称不能以下划线字符（_）开头，并且必须是 62 个字符长（或者一般为 NAMEDATALEN - 2，而不是允许的其他名称的 NAMEDATALEN - 1 字符）。以下划线开头的类型名称为内部创建的数组类型名称保留。

因为一旦数据类型被闯创建，使用数据类型就没有任何限制，所以创建基础类型就等同于为类型定义中提到的函数授予公共执行权限。（因此，该类型的创建者需要拥有这些函数。）对于类型定义中有用的各种函数，这通常不是问题。但是，在设计一种类型之前，用户可能需要三思而后行，因为该设计思路上，在将其作为转化的源或者目标类型时，可能需要一些“秘密”信息。

创建新基础类型的方法就是先创建一个输入函数。在这种方法中， HashData 数据库会首先看到新数据类型的名称作为输入函数的返回类型。这种情况下，shell 类型是隐式创建的，然后可以在其余的 I/O 函数的定义中引用它。这种方法仍然有效，但是已经弃用了，可能会在将来的某个版本中禁用掉。另外，为了避免由于函数定义中简单拼写错误而导致 shell 类型目录的异常混乱，当输入函数是用 c 写的时候，shell 类型只能以这种方式创建。

示例

该示例创建了一个复合类型，并且在函数定义中使用了它：

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

该例子创建了枚举类型 mood 并且在表定义中使用了它。

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

该例子创建了一个基础数据类型 box 然后在表定义中使用了它：

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS
... ;

CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS
... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

如果 box 的内部结构是 4 个 float4 元素的数组，我们可以替代使用：

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

这将允许通过下标访问一个 box 值的成员数字。否则类型的行为与以前相同。

该例子创建了一个大对象类型并在表的定义中使用了它

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);

CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

兼容性

该 CREATE TYPE 命令是 HashData 数据库扩展。 SQL 标准中有 CREATE TYPE 语句，但是细节上很是不同。

另见

CREATE FUNCTION, ALTER TYPE, DROP TYPE, CREATE DOMAIN

上级话题： SQL命令参考

CREATE USER

定义一个新的默认带有 LOGIN 权限的数据库角色

概要

```
CREATE USER name [ [WITH] option [ ... ] ]
```

该 option 可以是：

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | IN GROUP rolename [, ...]
    | ROLE rolename [, ...]
    | ADMIN rolename [, ...]
    | USER rolename [, ...]
    | SYSID uid    | RESOURCE QUEUE queue_name
```

描述

CREATE USER 已经由 CREATE ROLE 所替代，尽管为了向后兼容，它仍然被接受实用。

在 CREATE ROLE 和 CREATE USER 之间仅有的区别是 LOGIN 权限默认和 CREATE USER 一起创建，然而 NOLOGIN 权限默认和 CREATE ROLE 一起创建。

兼容性

在 SQL 标准中没有 CREATE USER 语句。

另见

CREATE ROLE

上级话题： SQL命令参考

CREATE VIEW

创建视图

定义一个新的视图。

概要

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

描述

CREATE VIEW 定义一个视图查询。该视图没有实际物。相反，每次在查询中引用该视图时，都会运行查询。

CREATE OR REPLACE VIEW 是相似的，但是如果同名的视图已经存在，则会被替换。用户只能使用生成相同列（相同列名称和数据类型）的新查询替换视图。

如果给定了模式名称，则在指定的模式名称中创建视图。否则，在当前的模式中创建该视图。临时视图存在于特殊的模式中，所以创建临时模式时不会给定模式名。该视图的名称与同模式中其他任何视图，表，序列或索引的名称必须不同。

参数

TEMPORARY | TEMP

如果指定，该视图创建为临时视图。临时视图在当前事务结束时自动删除。当临时视图存在时，具有相同的名字的永久视图在当前会话中不可见，除非他们使用方案限定名来引用。如果视图引用的任何表是临时的话，该视图会创建成一个临时视图（（不管）是否指定 TEMPORARY ）。

name

要创建的视图的名字（可选方案限定）。

`column_name`

可选名字列表用于视图的列。如果没有给定，该列名将从查询中推导而来。

query

SELECT 或 VALUES 命令，二者会提供视图的列和行。

注意

HashData 数据库中的视图是只读的。该系统不允许在视图上插入，更新，或者删除。用户可以通过在视图上创建重写规则到其他表上的适当操作来获得可更新视图的效果。更多信息，请参阅 CREATE RULE。

请注意，视图列的名字和数据类型可以按照用户想要的方式分配。例如：

```
CREATE VIEW vista AS SELECT 'Hello World';
```

是两种不好的形式：该列名默认为 ?column?, 而且列的数据类型默认为 unknown。如果用户想在视图的结果中使用字符串文字，请使用以下内容：

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

视图中引用的表的访问由视图所有者的权限决定的，而不是当前用户（即使当前用户是超级用户）。这在超级用户的情况下可能会令人困惑，因为超级用户通常可以访问所有对象。在视图的这种情况下，如果用户不是视图的所有者，那么即使是超级用户也必须要明确被授予对视图引用表的访问权限才能访问。

但是，视图中调用函数的处理方式与使用视图从查询中直接调用的方式相同。因此，视图的用户必须要有具有调用视图使用的任何函数的权限。

如果用户使用 ORDER BY 子句创建了一个视图，则当用户从视图中执行 SELECT 时将忽略 ORDER BY 子句。

例子

创建所有包含 comedy 电影的视图：

```
CREATE VIEW comedies AS SELECT * FROM films WHERE kind =
'comedy';
```

创建一个获取排名前 10 婴儿的名字：

```
CREATE VIEW topten AS SELECT name, rank, gender, year FROM
names, rank WHERE rank < '11' AND names.id=rank.id;
```

兼容性

该 SQL 标准指定了不在 HashData 数据库中的 CREATE VIEW 语句的一些附加功能。标准完整 SQL 命令的可选子句包括：

- **CHECK OPTION** — 该选项和可更新视图有关。 将检查视图上所有 INSERT 和 UPDATE 命令来确保数据满足视图定义条件（即，新的数据将通过视图可见）。如果不满足条件，则拒绝该更新。
- **LOCAL** — 检查视图上的完整性.
- **CASCADED** — 检查此视图的和任何从属视图的完整性，假定为 CASCADED 如果不指定 CASCADED 或 LOCAL。

CREATE OR REPLACE VIEW 是 HashData 数据库语言扩展，临时视图的概念也是如此。

另见

SELECT, DROP VIEW

上级话题： SQL命令参考

DEALLOCATE

释放

释放预预备语句

概要

```
DEALLOCATE [PREPARE] name
```

描述

DEALLOCATE 用来释放一个预先准备好对的 SQL 语句。如果用户不明确释放预备语句，它将会在会话结束时释放。

更多关于预备语句的信息，请参阅 PREPARE。

参数

PREPARE

被忽略的可选关键字

name

要释放的预备语句的名字

示例

释放预先准备好的叫做 `insert_names` 的语句：

```
DEALLOCATE insert_names;
```

兼容性

该 SQL 标准包含了 DEALLOCATE 语句，但是它只在嵌入式 SQL 中使用。

另见

EXECUTE, PREPARE

上级话题： SQL命令参考

DECLARE

声明

定义一个游标

概要

```
DECLARE name [BINARY] [INSENSITIVE] [NO SCROLL] CURSOR
    [{WITH | WITHOUT} HOLD]
    FOR query [FOR READ ONLY]
```

描述

DECLARE 允许用户创建游标，可以使用游标来从大查询中一次检索少量的行。游标可以使用 FETCH 返回文本数据或者二进制形式的数据。

普通游标以文本格式返回数据，和 SELECT 相同。由于数据本身以二进制格式存储，所以系统必须进行转换以产生文本格式。一旦信息以文本形式返回，客户端应用程序可能需要将其转换为二进制格式来操作它。另外，文本格式的数据通常比二进制格式大。二进制游标可以返回更容易地操作二进制形式数据。然而，如果用户打算以文本形式显示数据，则以文本形式检索数据将在客户端上节省一些精力。

例如，如果一个查询从一个整数列返回 1，那么用户将获得带有默认有游标的字符串 1，而使用二进制游标，用户将会获得一个包含该值内部表示的 4 字节字段（以大端字节顺序）。

应当仔细使用二进制游标，许多应用程序（包括 psql）并不准备处理二进制游标，并希望以文本格式返回数据。

> 注意：当客户端应用程序使用'扩展查询'协议发出 FETCH 命令时，绑定协议信息指定数据是以文本还是二进制格式检索。此选项将覆盖游标定义的方式。因此，当使用扩展查询协议时，二进制游标的概念就会过时-任何游标都可以被视为是文本或二进制的。

可以在 UPDATE 或 DELETE 语句中的 WHERE CURRENT OF 子句中指定游标，以更新或删除表数据。 该 UPDATE 或 DELETE 语句只能在服务器上执行， 例如在交互式 psql 会话或脚本中。语言扩展（如 PL/pgSQL）不支持可更新的游标。

参数

name

要创建有游标的名字

BINARY

使游标返回二进制数据而不是文本形式。

INSENSITIVE

表示当数据存在时，指出游标检索的数据不受游标所指表的更新的影响。在 HashData 数据库，所有游标都不敏感。这个关键词目前没有任何效果，仅为了和SQL的表准的兼容。

NO SCROLL

游标不能以非顺序方式检索行。这是 HashData 数据库中的默认行为，因为不支持可滚动的游标（SCROLL）。

WITH HOLD

WITHOUT HOLD

WITH HOLD 指出了该表可能会继续使用，在创建该游标的事务成功提交之后。WITHOUT HOLD 指出游标在脱离创建它的事务之后，就不能再使用了。默认值为 WITHOUT HOLD。

WITH HOLD 不能指定，当 query 包含 FOR UPDATE 或 FOR SHARE 子句。

query

SELECT 或 VALUES 命令会提供供游标返回的行。

如果游标在 UPDATE 或 DELETE 命令的 WHERE CURRENT OF 子句中使用，该 SELECT 命令必须要满足以下条件：

- 不能引用视图或者外部表
- 仅引用一张表:

  该表必须是可更新的，例如，以下是不可更新的：表函数，设置了返回值的函数，仅附加表，列表

- 不能包含任何以下的：
  - 分组语句
  - 例如 UNION ALL 或 UNION DISTINCT的集合操作
  - 排序子句
  - 窗口子句
  - 连接或者左连接

  指定 FOR UPDATE 子句在 SELECT 命令中可以阻止元组在获取和更新之间被其他会话更改行。没有该 FOR UPDATE 子句，那么随后（同会话中）带有 WHERE CURRENT OF 子句的 UPDATE 或 DELETE 命令就不起作用了，如果该行在创建游标之前已经更改（被其他会话更改）。（如数据删除之后用户去更新会找不到）

  > 注意： 指定 FOR UPDATE 子句在 SELECT 命令中锁定的是整个表，而不是仅仅是用户选择的行。

FOR READ ONLY

FOR READ ONLY 指明该游标仅仅用于只读模式

注意

除非指定了 WITH HOLD，由该命令创建的游标只能在当前的事务中使用。因此，没有 WITH HOLD 的DECLARE 语句，除非在事务块之外：否则该游标仅生存到该语句结束。因此如果该命令用于事务块之外， HashData 数据库会报告了一个错误。使用 BEGIN，COMMIT 和 ROLLBACK 来定义一个事务块。

如果指定 WITH HOLD 而且创建有游标的事务成功提交，该游标能够继续被同一个会话中的子事务访问。（但是如果创建事务被终止，则该游标会被删除）创建的带有 WITH HOLD 的游标，当被发出明确的 CLOSE 指令或者在会话结束时就会关闭。在当前的实现中，由保留游标表示的行 t 被复制到临时文件或者存储区域中，以便他们可以用于后续的事务。

如果用户在事务中使用 DECLARE 创建了游标，用户不能在事务中使用 SET 命令直到用户用 CLOSE 命令关闭了游标。

HashData 数据库不支持游标滚动。用户只能使用 FETCH 来向前移动游标，不能向后。

追加优化表不支持 DECLARE...FOR UPDATE。

用户可以通过查询 `pg_cursors` 系统视图来查看所有可用的游标。

例子

声明一个游标：

```
DECLARE mycursor CURSOR FOR SELECT * FROM mytable;
```

兼容性

SQL 标准只允许在嵌入式 SQL 和模块中使用游标。 HashData 数据库数据库允许交互式使用游标（interactively）。

HashData 数据库没有实现有游标的 OPEN 语句。游标在声明时就被认为是打开的。

SQL 标准允许游标向前向后移动。所有 HashData 数据库游标仅向前移动（非滚动）。

二进制游标是 HashData 数据库扩展。

另见

CLOSE, DELETE, FETCH, MOVE, SELECT, UPDATE

上级话题： SQL命令参考

DELETE

删除

从表中删除行

概要

```
DELETE FROM [ONLY] table [[AS] alias]
      [USING usinglist]
      [WHERE condition | WHERE CURRENT OF cursor_name ]
```

描述

DELETE 从指定的表中删除满足 WHERE 子句的行，如果该 WHERE 子句是空的，则会删除整个表行，该结果是有效的，只不过是个空表。

默认情况下，DELETE 会删除指定表中的行和他所有的子表。如果用户希望仅从特定指定的表中删除，请使用 ONLY 子句。

有两种使用包含在数据库中其他表的信息来删除行的方式：使用 sub-selects，或者在 USING 子句中指定额外的表。哪种技术更适合取决于具体情况。

如果指定了 WHERE CURRENT OF 子句，则删除的行是从指定的游标中最近读取的行。

用户必须有对表的 DELETE 权限才能从中删除。

Outputs

成功完成后， DELETE 命令返回表单的命令标签。

```
DELETE count
```

删除行的数量，如果数量为 0，没有满足删除条件的行（这并不认为是一个错误）。

参数

ONLY

如果指定，仅从提名的表中删除行。如果没有指定，也会处理任何从提名的表继承的表。

table

存在表的名字（可选方案限定）。

alias

目标表的别名。当提供了别名，它完全隐藏了表的实际名字。例如，给定 DELETE FROM foo AS f，该 DELETE 语句的其他部分必须将此表称为 f 而不是 foo。

usinglist

表达式列表，允许其他表的列出现在 WHERE 条件中。这和 SELECT 语句中的 FROM 子句中可以指定的表的列表相似：例如，可以指定表的别名。不要重复使用 usinglist 中的目标表，除非用户希望设置自连接。

condition

返回类型为 boolean 的表达式，该表达式决定那些要删除的行。

`cursor_name`

要在 WHERE CURRENT OF 条件下使用的游标的名字。要删除的行是从该有游标最近读取的行。游标必须是DELETE 目标表上的简单（非连接，非聚合）查询。

WHERE CURRENT OF 不能喝布尔条件一起指定。

该 DELETE...WHERE CURRENT OF 游标语句只能在服务器上执行，例如交互式 psql 会话或脚本。语言扩展（如 PL/pgSQL）不支持可更新游标。

更多有关创建有游标的信息，请参阅 DECLARE。

注意

HashData 数据库使用户在 WHERE 条件中通过指定在 USING 指定其他表来引用其他表的列。 例如，对 来自 rank 表的名为 Hannah 的列，可能会这样做：

```
DELETE FROM rank USING names WHERE names.id = rank.id AND
name = 'Hannah';
```

这里发生的本质是一个在 rank 和 names表之间的连接，所有成功连接的行都被标记为删除。该语法是不标准的。但是，相比子查询风格，该连接风格通常比较容易写，也能更快执行，例如：

```
DELETE FROM rank WHERE id IN (SELECT id FROM names WHERE name
= 'Hannah');
```

当使用 DELETE 来删除表中的所有行（例如： DELETE * FROM table;），HashData 数据库添加了隐式的 TRUNCATE 命令（当用户权限允许时）。 该添加的 TRUNCATE 命令释放了被删除行所占的空间，而没有执行表的 VACUUM 操作。这提升了后续查询的扫描性能，有助于临时表经常插入和删除的ELT工作负载。

不支持在特定分区（子表）上直接执行 UPDATE 和 DELETE 命令。相反，这些命令必须在根分区表（使用 CREATE TABLE 命令创建的表）上执行。

示例

删除除了 musicals 之外的所有 films：

```
DELETE FROM films WHERE kind <> 'Musical';
```

清除表 films：

```
DELETE FROM films;
```

使用连接删除：

```
DELETE FROM rank USING names WHERE names.id = rank.id AND
name = 'Hannah';
```

兼容性

该命令服从 SQL 标准，除了 HashData 数据库的扩展—— USING 子句。

另见

DECLARE, TRUNCATE

上级话题： SQL命令参考

DISCARD

丢弃

丢弃会话状态。

概要

```
DISCARD { ALL | PLANS | TEMPORARY | TEMP }
```

描述

DISCARD 释放与数据库会话相关联的内部资源。这些资源通常在会话结束时释放。 DISCARD TEMP 删除在当前会话中创建的所有临时表。DISCARD PLANS 释放所有内部缓存的查询计划 DISCARD ALL 重置会话到原始状态，丢弃临时资源并重置会话本地配置更改。

参数

TEMPORARY, TEMP

删除所有在当前会话中创建的临时表。

PLANS

释放所有缓存的查询计划

ALL

释放所有和当前会话相关联的临时资源并且重置会话到初始状态。目前，该和执行以下语句序列具有相同的效果：

```
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
CLOSE ALL;
UNLISTEN *;
SELECT pg_advisoryd_unlock_all();
DISCARD PLANS;
DISCARD TEMP;
```

注意

DISCARD ALL 不能在事务块中执行。

兼容性

DISCARD 是 HashData 数据库扩展

上级话题： SQL命令参考

DO

执行匿名的代码块作为暂时匿名函数。

概要

```
DO [ LANGUAGE lang_name ] code
```

描述

DO 执行了一个匿名代码块，或换而言之为一个程序化语言的暂时的匿名函数

该代码块被认为似乎是一个没有参数的函数体，返回空值。它被解析被执行了一次。

可选参数 LANGUAGE 子句可以出现在代码块的前或者后。

匿名块是程序化语言结构，该结构提供在运行时创建和执行程序代码的能力，而不用将代码作为数据库对象持久的存储在系统目录中。匿名代码块的概念和 UNIX shell 脚本相似，该能允许将多个手动输入命令分组并作为一步骤执行。顾名思义，匿名块没有名字，因此不能被其他对象引用。虽然动态创建，匿名块可以轻松的作为脚本存储在操作系统文件中以便重复执行。

匿名块是标准程序语言块。他们携带语法并且遵循适用于该程序语言的规则，包括变量的声明和作用于，执行，异常处理和语言使用。

匿名块的编译和执行结合在一个步骤中，而用户定义的函数需要在每次定义变更之前重新定义。

参数

code

需要执行的程序化语言代码。这必须指定为字符串文本。正如使用 CREATE FUNCTION 命令。建议使用美元引用文字。可选关键字无效。支持这些程序语言：PL/pgSQL（ plpgsql ），PL/Python （ plpythonu ），and PL/Perl （ plperl 和 plperlu ）。

lang_name

代码所用程序语言的名字。该语言默认是 plpgsql。该语言必须在 HashData 数据库中安装并且在用户数据库中注册。

注意

PL/pgSQL 语言安装在 HashData 数据库系统中并且注册在用户创建的数据库中。PL/Python 语言是默认安装的，但是没有注册。其他语言没有安装也没有注册。系统目录 pg_language 包含了在数据库中注册语言的信息。

如果语言不可信，则用户必须有对程序语言有 USAGE 权限，或者必须是超级用户。这和使用该语言创建函数的权限要求一样。

示例

该 PL/pgSQL 例子对 webuser 用户赋予在 public 模式中所有视图的所有权限：

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
             WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.table_name) || ' TO webuser';
    END LOOP;
END$$;
```

该 PL/pgSQL 例子决定是否 HashData 数据库用户是超级用户。在例子中，匿名块检索从临时表的输入值。

```
CREATE TEMP TABLE list AS VALUES ('gpadmin') DISTRIBUTED RANDOMLY;

DO $$
DECLARE
  name TEXT := 'gpadmin' ;
  superuser TEXT := '' ;
  t1_row   pg_authid%ROWTYPE;
BEGIN
  SELECT * INTO t1_row FROM pg_authid, list
     WHERE pg_authid.rolname = name ;
  IF t1_row.rolsuper = 'f' THEN
    superuser := 'not ';
  END IF ;
  RAISE NOTICE 'user % is %a superuser', t1_row.rolname, superuser ;
END $$ LANGUAGE plpgsql ;
```

> 注意： 例子 PL/pgSQL 使用 SELECT 和 INTO 子句。它和 SQL 命令的 SELECT INTO 语句不同。

兼容性

SQL 标准中没有 DO 语句。

另见

CREATE LANGUAGE

上级话题： SQL命令参考

DROP AGGREGATE

删除聚集函数

删除一个聚集函数

概要

```
DROP AGGREGATE [IF EXISTS] name ( type [, ...] ) [CASCADE | RESTRICT]
```

描述

DROP AGGREGATE 会删除一个存在的聚集函数。要执行该命令，当前用户必须是该聚集函数的拥有者。

参数

IF EXISTS

如果该聚集函数不存在不会抛出异常。在异常情况下会发出通知。

name

存在的聚集函数的名称（可选方案限定）。

type

聚集函数操作的输入数据类型。要引用一个 0 个参数的聚集函数，写 * 替代输入数据类型列表。

CASCADE

自动删除依赖该聚集函数的对象。

RESTRICT

如果有任何对象依赖于它，则拒绝删除聚集函数。这是默认的。

示例

删除参数为 integer 的聚集函数 myavg：

```
DROP AGGREGATE myavg(integer);
```

兼容性

SQL 标准中没有 DROP AGGREGATE 语句。

另见

ALTER AGGREGATE, CREATE AGGREGATE

上级话题： SQL命令参考

DROP CAST

删除一个造型

概要

```
DROP CAST [IF EXISTS] (sourcetype AS targettype) [CASCADE | RESTRICT]
```

描述

DROP CAST 会删除一个之前定义的造型，为了能够删除造型，用户必须要拥有资源或目标数据类型。用户需要获得以下权限来创建一个造型。

参数

IF EXISTS

如果造型不存在不会抛出错误，这种情况会发出一个通知。

sourcetype

造型的源数据类型的名称。

targettype

造型的目标数据类型的名称。

CASCADE

RESTRICT

这些关键词没有效果，因为没有对造型的依赖。

示例

删除从 text 到 int 类型的造型：

```
DROP CAST (text AS int);
```

兼容性

该 DROP CAST 命令服从 SQL 标准。

另见

CREATE CAST

上级话题： SQL命令参考

DROP CONVERSION

删除转换

删除一个转换

概要

```
DROP CONVERSION [IF EXISTS] name [CASCADE | RESTRICT]
```

描述

DROP CONVERSION 删除之前定义的一个转换。 为了能够删除一个转换，用户必须先拥有该转换。

参数

IF EXISTS

如果该转换不存在不会抛出异常，这种情况下会发出一个通知。

name

转换的名字，该转换名字可以是方案限定的

CASCADE

RESTRICT

这些关键词没有作用，因为他们是不依赖该转换。

例子

删除叫 myname 的转换：

```
DROP CONVERSION myname;
```

兼容性

SQL 标准中没有 DROP CONVERSION 的语句。

另见

ALTER CONVERSION, CREATE CONVERSION

上级话题： SQL命令参考

DROP DATABASE

删除数据库

删除一个数据库。

概要

```
DROP DATABASE [IF EXISTS] name
```

描述

DROP DATABASE 删除了一个数据库。它删除了数据的目录条目并且删除了包含该数据的目录。它只能被拥有该数据库的用户执行。另外，当用户或者是其他用户正连接到该数据库时候，是不能执行删除该数据库操作的。（连接到 postgres 或其他数据库来发出此命令）

> 警告： DROP DATABASE 不能被撤销，请小心使用。

参数

IF EXISTS

如果数据库不存在不会抛出异常。这种情况下发出通知。

name

要删除数据库的名字。

注意

DROP DATABASE 不能在事务块中执行。

当连接到目标数据库时，不能执行该命令。因此，使用程序 dropdb 来替代可能更方便一点。该命令是在此命令上的封装。

示例

删除叫 testdb 的数据库：

```
DROP DATABASE testdb;
```

兼容性

SQL 标准中没有 DROP DATABASE 语句。

另见

ALTER DATABASE, CREATE DATABASE

上级话题： SQL命令参考

DROP DOMAIN

删除域

删除一个域

概要

```
DROP DOMAIN  [IF EXISTS ] name  [, ... ]  [CASCADE | RESTRICT ]
```

描述

DROP DOMAIN 删除一个之前定义的域。用户必须是该域的所有者才能删除它。

参数

IF EXISTS

如果该域不存在不会抛出异常。这种情况下会发出通知。

name

存在的该域的名字（可选方案限定）。

CASCADE

自动删除依赖于域的对象（如表列）。

RESTRICT

如果有任何对象依赖于该域则拒绝删除该域，这是默认的。

示例

删除叫 zipcode 的域：

```
DROP DOMAIN zipcode;
```

兼容性

该命令是服从 SQL 标准的，除了 IF EXISTS 选项，该选项是 HashData 数据库扩展的。

另见

ALTER DOMAIN, CREATE DOMAIN

上级话题： SQL命令参考

DROP EXTENSION

删除扩展

从 HashData 数据库中删除一个扩展。

概要

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

描述

DROP EXTENSION 从数据库中移除扩展。删除扩张也会导致其组件对象也被删除。

> 注意：用来创建扩展所需的支持扩展文件不会被删除。这些文件必须从 HashData 数据库主机中手动删除。

用户必须拥有该扩展才能使用 DROP EXTENSION。

如果该扩展的任何对象正在数据库中使用，则该命令失败。例如，如果一个表是用扩展类型的列定义的，则要添加 CASCADE 选项来强制删除这些依赖对象。

重要提示：在发出 DROP EXTENSION 带有 CASCADE 关键词的命令时，用户必须明白所有依赖于该扩展的对象，以避免意外的后果。

参数

IF EXISTS

如果扩展不存在，不会抛出异常。会发出通知。

name

安装的扩展的名字。

CASCADE

自动删除依赖于该扩展的对象，并且依次删除依赖于这些对象的所有对象。

RESTRICT

如果有对象依赖于该扩展，则拒绝删除该扩展，除了扩展成员对象。这是默认的。

兼容性

DROP EXTENSION 是 HashData 数据库扩展。

另见

CREATE EXTENSION

上级话题： SQL命令参考

DROP EXTERNAL TABLE

删除外部表

删除一个外部表定义

概要

```
DROP EXTERNAL [WEB] TABLE [IF EXISTS] name [CASCADE | RESTRICT]
```

描述

DROP EXTERNAL TABLE 从数据库中删除一个存在的外部表定义。外部的数据资源和文件不会被删除。要执行这个命令用户必须是外部表的所有者。

参数

WEB

删除外部 web 表的可选参数

IF EXISTS

如果外部表存在，不会抛出异常。这种情况下会发出一个通知。

name

存在的外部表的名字（可选方案限定）。

CASCADE

自动删除依赖于外部表的对象（例如视图）。

RESTRICT

如果有对象依赖于外部表，则拒绝删除该外部表。这是默认的选项。

示例

删除叫 staging 的外部表，如果存在的话：

```
DROP EXTERNAL TABLE IF EXISTS staging;
```

兼容性

SQL 标准中没有 DROP EXTERNAL TABLE 语句。

另见

CREATE EXTERNAL TABLE

上级话题： SQL命令参考

DROP FILESPACE

删除文件空间

删除一个文件空间。

概要

```
DROP FILESPACE [IF EXISTS] filespacename
```

描述

DROP FILESPACE 从数据库中删除一个文件空间定义和它的系统生成的数据目录。

文件空间只能被它的拥有者或超级用户删除。该文件空间必须清空所有的表空间对象之后才能被删除。可能情况是其他数据库的表空间还在使用该文件空间，即使当前数据库中没有使用该文件空间的表空间。

参数

IF EXISTS

如果文件空间不存在，也不会抛出错误。这种情况下会发出通知。

tablespacename

要删除的文件空间的名字。

示例

删除表空间 myfs：

```
DROP FILESPACE myfs;
```

兼容性

SQL 标准中或者 PostgreSQL 中没有 DROP FILESPACE 语句。

另见

ALTER FILESPACE、DROP TABLESPACE

上级话题： SQL命令参考

DROP FUNCTION

删除函数

删除一个函数。

概要

```
DROP FUNCTION [IF EXISTS] name ( [ [argmode] [argname] argtype
    [, ...] ] ) [CASCADE | RESTRICT]
```

描述

DROP FUNCTION 删除一个存在函数的定义。要执行这个命令该用户必须是该函数的拥有者。必须要指定参数类型，因为几个不同的函数可能存在同名，但其参数列表不同。

参数

IF EXISTS

如果函数不存在，不会抛出错误。这种情况下会发出通知。

name

存在函数的名称（可选方案限定）。

argmode

参数的模式：要么 IN，OUT，INOUT，或 VARIADIC。如果省略，则默认是 IN。注意 DROP FUNCTION 实际上并不会注意 OUT 参数，因为仅需要输入参数就能决定函数的身份。索引列出 IN， INOUT，和 VARIADIC 参数就足够了。

argname

参数的名字。注意 DROP FUNCTION 实际上并不注意参数名字，因为仅需要参数数据类型就能决定函数的身份。

argtype

函数参数的数据类型（可选方案限定）如果有的话。

CASCADE

自动删除依赖于该函数的对象，例如操作符号。

RESTRICT

如果有任何对象依赖于该函数，则拒绝删除该函数，这是默认的。

例子

删除平方根函数：

```
DROP FUNCTION sqrt(integer);
```

兼容性

SQL 标准中定义了 DROP FUNCTION 语句，但是这和该命令不兼容。

另见

CREATE FUNCTION, ALTER FUNCTION

上级话题： SQL命令参考

DROP GROUP

删除一个数据库角色。

概要

```
DROP GROUP [IF EXISTS] name [, ...]
```

描述

DROP GROUP 是一个过时的命令，尽管为了向后兼容还被支持。组（和用户）已经被更一般的角色的概念所替代。 参阅 DROP ROLE 获取更多信息。

参数

IF EXISTS

如果角色不存在，也不会抛出错误。这种情况下会发出通知。

name

存在角色的名字。

兼容性

SQL标准中没有 DROP GROUP 语句。

另见

DROP ROLE

上级话题： SQL命令参考

DROP INDEX

删除一个索引。

概要

```
DROP INDEX [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

描述

DROP INDEX 从数据库系统上删除一个存在的索引。要执行这个命令用户必须是索引的拥有者。

参数

IF EXISTS

如果索引不存在，不会抛出错误。这种情况下会发出通知。

name

存在索引的名字（可选方案限定）。

CASCADE

自动删除依赖于该索引的对象。

RESTRICT

如果有任何对象依赖于该索引，则拒绝删除该对象。这是默认的。

示例

删除索引 title_idx：

```
DROP INDEX title_idx;
```

兼容性

DROP INDEX 是 HashData 数据库语言扩展。SQL 标准中没有该索引的规定。

另见

ALTER INDEX, CREATE INDEX, REINDEX

上级话题： SQL命令参考

DROP LANGUAGE

删除语言

删除一个程序语言。

概要

```
DROP [PROCEDURAL] LANGUAGE [IF EXISTS] name [CASCADE | RESTRICT]
```

描述

DROP LANGUAGE 会删除之前注册程序语言的定义。用户必须是超级用户或者是该语言的拥有者才能删除该语言。

参数

PROCEDURAL

可选关键字 - 没有效果。

IF EXISTS

如果该语言不存在，不会抛出错误。这种情况下发出通知。

name

存在的程序语言的名字。为了向后兼容，该名字必须要用但引号括起来。

CASCADE

自动删除依赖该语言的对象（例如用该语言写的函数）。

RESTRICT

如果有任何对象依赖于该语言，则拒绝删除该语言。这是默认的。

例子

删除程序语言 plsample：

```
DROP LANGUAGE plsample;
```

兼容性

SQL 标准中没有 DROP LANGUAGE 语句。

另见

ALTER LANGUAGE, CREATE LANGUAGE

上级话题： SQL命令参考

DROP OPERATOR CLASS

删除操作符类

删除操作符类

概要

```
DROP OPERATOR CLASS [IF EXISTS] name USING index_method [CASCADE | RESTRICT]
```

描述

DROP OPERATOR 删除一个操作符类。要执行这个命令用户必须是操作符类的拥有者。

参数

IF EXISTS

如果操作符类不存在，不会抛出异常。这种情况下会发出通知。

name

存在的操作符类的名字（可选方案限定）。

index_method

操作符类索引访问方法的名称。

CASCADE

自动删除依赖于该操作类的对象。

RESTRICT

如果有任何对象依赖于操作符类，则拒绝删除该操作符类。

示例

删除 B-tree 操作符类 widget_ops：

```
DROP OPERATOR CLASS widget_ops USING btree;
```

如果有任何存在的索引使用操作符类，则命令会失败。添加 CASCADE 来删除这些索引和操作符类。

兼容性

SQL 标准中没有 DROP OPERATOR CLASS 语句。

另见

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS

上级话题： SQL命令参考

DROP OPERATOR FAMILY

删除操作符族（family）

删除一个操作符族。

概要

```
DROP OPERATOR FAMILY [IF EXISTS] name USING index_method [CASCADE | RESTRICT]
```

描述

DROP OPERATOR FAMILY 删除了一个操作符族。要执行该命令，用户必须是该该操作符族的拥有者。

DROP OPERATOR FAMILY 包含删除在该操作符族中的任何操作符，但是它不会删除该系列引用的任何操作符或函数。如果有索引依赖于系列中 的操作符类，用户将需要指定 CASCADE 来完整的删除。

参数

IF EXISTS

如果一个操作符不存在，不会抛出错误。这种情况下会发出通知。

name

存在的操作符族的名字（可选方案限定）。

index_method

操作符族的索引访问方法的名字

CASCADE

自动删除依赖该操作符族的对象。

RESTRICT

如果有任何对象依赖于操作符族，则拒绝删除该操作符族。这是默认的。

例子

删除 B-tree 的操作符族 float_ops:

```
DROP OPERATOR FAMILY float_ops USING btree;
```

如果有任何存在索引使用该操作符族，则该命令失败。添加 CASCADE 来删除操作符族和索引。

兼容性

SQL 标准中没有 DROP OPERATOR FAMILY 。

另见

CREATE OPERATOR FAMILY, ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR CLASS

上级话题： SQL命令参考

DROP OPERATOR

删除一个操作符。

概要

```
DROP OPERATOR [IF EXISTS] name ( {lefttype | NONE} ,
    {righttype | NONE} ) [CASCADE | RESTRICT]
```

描述

DROP OPERATOR 从数据库系统中删除一个操作符。要执行这个命令用户必须是该操作符的拥有者。

参数

IF EXISTS

如果操作符不存在，不会抛出错误。这种情况下会发出通知。

name

存在操作符的名字（可选方案限定）。

lefttype

操作符左操作数的数据类型；如果没有左操作数写 NONE。

righttype

操作符右操作数的数据类型；如果没有右操作数写 NONE。

CASCADE

自动删除依赖于该操作符的对象。

RESTRICT

如果有任何对象依赖于该操作符，则拒绝删除该操作符。这是默认的。

示例

删除 integer 类型的 power 操作符 `a^b`：

```
DROP OPERATOR ^ (integer, integer);
```

删除左一元位补码 `~b`：

```
DROP OPERATOR ~ (none, bit);
```

删除右一元 bigint 的阶乘操作符 x!：

```
DROP OPERATOR ! (bigint, none);
```

兼容性

SQL 标准中没有 DROP OPERATOR 语句。

另见

ALTER OPERATOR, CREATE OPERATOR

上级话题： SQL命令参考

DROP OWNED

删除一个数据库角色拥有的数据库对象。

概要

```
DROP OWNED BY name [, ...] [CASCADE | RESTRICT]
```

描述

DROP OWNED 删除当前数据库中指定角色之一所拥有的所有对象。授予当前数据库中对象的给定角色的任何权限也将被撤销。

参数

name

要删除对象的拥有者的名字和要撤销权限的拥有者的名字。

CASCADE

自动删除依赖所影响对象的的对象。

RESTRICT

如果有任何其他数据库对象依赖所影响的对象之一，则拒绝删除该角色所拥有的对象。这是默认的。

注意

DROP OWNED 经常被使用来准备移除一个或多个角色。因为 DROP OWNED 仅影响当前数据的对象，通常需要在每个数据库中执行此命令，该数据库要包含要删除对象所拥有的对象。

使用 CASCADE 选项可能使命令递归到其他用户所拥有的对象。

该 REASSIGN OWNED 命令是重新分配由一个或多个角色所拥有的数据库对象所有权的替代方法。

示例

删除由 sally 角色所拥有的任何数据库对象：

```
DROP OWNED BY sally;
```

兼容性

该 DROP OWNED 语句是 HashData 数据库扩展。

另见

REASSIGN OWNED, DROP ROLE

上级话题： SQL命令参考

DROP PROTOCOL

从数据库中移除一个外部表数据访问协议。

概要

```
DROP PROTOCOL [IF EXISTS] name
```

描述

DROP PROTOCOL 从数据库中移除一个指定的协议。协议的名字可以通过命令 CREATE EXTERNAL TABLE 来指定，用来从外部数据源来读取数据或者将数据写到一个外部数据源中。

> 警告: 如果用户删除了一个数据访问协议，那么通过该访问协议定义的外部表将再也不能访问外部数据源。

参数

IF EXISTS

如果协议不存在，不会抛出一个错误。这种情况下会发出一个提示。

name

一个存在的数据访问协议的名字。

注解

如果用户删除一个数据访问协议，在数据库中定义的与该协议相关联的调用处理程序不会被删除。用户一定要手动地删除那些函数。

被协议用到的共享库也需要从 HashData 数据库的主机移除。

兼容性

DROP PROTOCOL 是一个 HashData 的扩展。

另见

CREATE EXTERNAL TABLE、CREATE PROTOCOL

上层主题 上级主题： SQL命令参考

DROP RESOURCE QUEUE

移除一个资源队列。

概要

```
DROP RESOURCE QUEUE queue_name
```

描述

该命令从 HashData 数据库中移除一个负载管理资源队列。为了删除一个资源队列，要删除的队列不能有任何角色分配给它，也不能有任何的语句在队列中等待。只有超级用户能够删除资源队列。

参数

queue_name

将要被删除的资源队列的名称。

注解

通过命令 ALTER ROLE 从一个资源队列中移除一个用户。

为了看到关于所有的资源队列的当前活动的查询，执行下面将 pg_locks 表和 pg_roles 以及 pg_resqueue 表相连接的查询：

```
SELECT rolname, rsqname, locktype, objid, transaction, pid,
mode, granted FROM pg_roles, pg_resqueue, pg_locks WHERE
pg_roles.rolresqueue=pg_locks.objid AND
pg_locks.objid=pg_resqueue.oid;
```

为了看到分配到一个资源队列的角色，可以在 pg_roles 和 pg_resqueue 两个系统目录表执行下面查询：

```
SELECT rolname, rsqname FROM pg_roles, pg_resqueue WHERE
pg_roles.rolresqueue=pg_resqueue.oid;
```

示例

从一个资源队列中移除一个角色（同时移动该角色到默认的资源队列，pg_default）：

```
ALTER ROLE bob RESOURCE QUEUE NONE;
```

移除一个名为 adhoc 的资源队列：

```
DROP RESOURCE QUEUE adhoc;
```

兼容性

DROP RESOURCE QUEUE 语句是一个 HashData 数据库的扩展。

另见

ALTER RESOURCE QUEUE、CREATE RESOURCE QUEUE、ALTER ROLE

上级主题： SQL命令参考

DROP ROLE

移除一个数据库角色。

概要

```
DROP ROLE [IF EXISTS] name [, ...]
```

描述

DROP ROLE 移除指定的角色。要删除一个超级用户角色，用户必须自己是一个超级用户。要删除一个非超级用户角色，用户必须要有 CREATEROLE 权限。

如果一个角色仍然被集簇中的任何一个数据库引用，它就不能被移除。如果尝试移除将会抛出一个错误。在删除角色之前，用户必须删除（或者重新授予所有权）它所拥有的所有对象并且收回该已经授予给该角色的在其他对象上的特权。REASSIGN OWNED 和 DROP OWNED 命令可以用于这个目的。

不过，没有必要移除涉及该角色的角色成员关系；DROP ROLE 会自动收回目标角色在其他角色中的成员关系，以及其他角色在目标角色中的成员关系。其他角色不会被删除也不会被影响。

参数

IF EXISTS

如果该角色不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的角色的名称。

示例

移除名为 sally 和 bob 的角色：

```
DROP ROLE sally, bob;
```

兼容性

SQL 标准定义了 DROP ROLE，但是它只允许一次删除一个角色并且它指定了和 HashData 数据库不同的特权需求。

另见

REASSIGN OWNED, DROP OWNED, CREATE ROLE, ALTER ROLE, SET ROLE

上级主题： SQL命令参考

DROP RULE

移除一个重写规则。

概要

```
DROP RULE [IF EXISTS] name ON relation [CASCADE | RESTRICT]
```

描述

DROP RULE 删除一个表或视图相关的重写规则。

参数

IF EXISTS

如果该规则不存在则不要抛出一个错误，而是发出一个提示。

name

要删除的规则的名称。

relation

该规则适用的表或视图的名称（可以是方案限定的）。

CASCADE

自动删除依赖于该规则的对象，然后删除所有依赖于那些对象的对象。

RESTRICT

如果有任何对象依赖于该规则，则拒绝删除它。这是默认值。

示例

移除一个在表 sales 上名为 sales_2006 的重写规则：

```
DROP RULE sales_2006 ON sales;
```

兼容性

DROP RULE 不是 SQL 标准中的语句。

另见

CREATE RULE

上级主题： SQL命令参考

DROP SCHEMA

移除一个方案。

概要

```
DROP SCHEMA [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

描述

DROP SCHEMA 从数据库中移除方案。一个方案只能由其拥有者或一个超级用户删除。注意即使拥有者不拥有该方案中的某些对象，也能删除该方案（以及所有含有的对象）。

参数

IF EXISTS

如果该方案不存在则不要抛出一个错误，而是发出一个提示。

name

要移除方案的名称。

CASCADE

自动删除包含在该方案中的对象（表、函数等），然后删除所有依赖于那些对象的对象。

RESTRICT

如果该方案含有任何对象，则拒绝删除它。这是默认值。

示例

从数据库中移除一个名为 mystuff 的方案及其中所包含的对象：

```
DROP SCHEMA mystuff CASCADE;
```

兼容性

DROP SCHEMA 完全符合 SQL 标准，不过该标准只允许在每个命令中删除一个方案并且没有 IF EXISTS 选项。该选项是 HashData 的一个扩展。

另见

CREATE SCHEMA, ALTER SCHEMA

上级主题： SQL命令参考

DROP SEQUENCE

移除一个序列。

概要

```
DROP SEQUENCE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

描述

DROP SEQUENCE 移除序数生成器表。一个序列只能被其拥有者（或超级用户删除）。

参数

IF EXISTS

如果该序列不存在则不要抛出一个错误，而是发出一个提示。

name

要移除序列的名称（可以是方案限定的）。

CASCADE

自动删除依赖于该序列的对象，然后删除所有依赖于那些对象的对象。

RESTRICT

如果有任何对象依赖于该序列，则拒绝删除它。这是默认值。

示例

移除一个名为 myserial 的序列：

```
DROP SEQUENCE myserial;
```

兼容性

DROP SEQUENCE 符合 SQL 标准，不过该标准只允许每个命令中删除一个序列并且没有 IF EXISTS 选项。该选项是一个 HashData 扩展。

另见

ALTER SEQUENCE, CREATE SEQUENCE

上级主题： SQL命令参考

DROP TABLE

移除一个表。

概要

```
DROP TABLE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

描述

DROP TABLE 从数据库移除表。只有表拥有者、模式拥有者和超级用户能删除一个表。要清空一个表中的行但是不销毁该表，可以使用 DELETE 或者 TRUNCATE。

DROP TABLE 总是移除目标表的任何索引、规则、触发器和约束。不过，要删除一个被视图或者另一个表的外键约束所引用的表，必须指定 CASCADE。 CASCADE 将会把依赖的视图也完全移除。

参数

IF EXISTS

如果该表不存在则不要抛出一个错误，而是发出一个提示。

name

要删除的表的名称（可以是方案限定的）。

CASCADE

自动删除依赖于该表的对象（例如视图），然后删除所有 依赖于那些对象的对象。

RESTRICT

如果有任何对象依赖于该表，则拒绝删除它。这是默认值。

示例

移除一个名为 mytable 的表：

```
DROP TABLE mytable;
```

兼容性

DROP TABLE 符合 SQL 标准，不过该标准只允许每个命令删除一个表并且没有 IF EXISTS 选项。该选项是 HashData 的一个扩展。

另见

CREATE TABLE, ALTER TABLE, TRUNCATE

上级主题： SQL命令参考

DROP TABLESPACE

移除一个表空间。

概要

```
DROP TABLESPACE [IF EXISTS] tablespacename
```

描述

DROP TABLESPACE 从系统中移除一个表空间。

一个表空间只能被其拥有者或超级用户删除。在一个表空间能被删除前，其中必须没有任何数据库对象。即使当前数据库中没有对象正在使用该表空间，也可能有其他数据库的对象存在于其中。

参数

IF EXISTS

如果该表空间不存在则不要抛出一个错误，而是发出一个提示。

tablespacename

要移除的表空间的名称。

示例

移除名为 mystuff 的表空间：

```
DROP TABLESPACE mystuff;
```

兼容性

DROP TABLESPACE 是 HashData 的一个扩展。

另见

CREATE TABLESPACE, ALTER TABLESPACE

上级主题： SQL命令参考

DROP TYPE

移除一个数据类型。

概要

```
DROP TYPE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

描述

DROP TYPE 移除一种用户定义的数据类型。只有一个类型的拥有者才能移除它。

参数

IF EXISTS

如果该类型不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的数据类型的名称（可以是方案限定的）。

CASCADE

自动删除依赖于该类型的对象（例如表列、函数、操作符），然后删除所有 依赖于那些对象的对象。

RESTRICT

如果有任何对象依赖于该类型，则拒绝删除它。这是默认值。

示例

移除一个名为 box 的数据库类型：

```
DROP TYPE box;
```

兼容性

这个命令类似于 SQL 标准中的对应命令，但 IF EXISTS 选项是 HashData 数据库的一个扩展。 但是要注意 HashData 数据库中的 CREATE TYPE 命令和数据类型扩展机制都与 SQL 标准不同。

另见

ALTER TYPE, CREATE TYPE

上级主题： SQL命令参考

DROP USER

移除一个数据库角色。

概要

```
DROP USER [IF EXISTS] name [, ...]
```

描述

DROP USER 是一个被弃用的命令，不过为了向后兼容仍然可以使用。组（用户）已经被更加通用的概念角色替代。 更多详细信息见 DROP ROLE 。

参数

IF EXISTS

如果该角色不存在则不要抛出一个错误，而是发出一个提示。

name

一个存在的角色的名称。

兼容性

在 SQL 标准中没有 DROP USER 命令。 SQL 标准把用户的定义留给具体实现自行解释。

另见

DROP ROLE

上级主题： SQL命令参考

DROP VIEW

移除一个视图。

概要

```
DROP VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

描述

DROP VIEW 移除一个现有的视图。只有一个视图的所有者才能移除它。

参数

IF EXISTS

如果该视图不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的视图的名称（可以是方案限定的）。

CASCADE

自动删除依赖于该视图的对象（例如其他视图），然后删除所有依赖于那些对象的对象。

RESTRICT

如果有任何对象依赖于该视图，则拒绝删除它。这是默认值。

示例

移除一个名为 topten 的视图：

```
DROP VIEW topten;
```

兼容性

DROP VIEW 这个命令符合 SQL 标准，不过该标准只允许在每个命令中删除一个视图并且没有 IF EXISTS 选项。该选项是 HashData 数据库的一个扩展。

另见

CREATE VIEW

上级主题： SQL命令参考

END

提交当前事务。

概要

```
END [WORK | TRANSACTION]
```

描述

END 提交当前事务。所有该事务做的更改变得对他人可见并且被保证发生崩溃时仍然是持久的。这个命令是一种 HashData 数据库的扩展，它等效于 COMMIT。

参数

WORK

TRANSACTION

可选关键词，它们没有效果。

示例

提交当前事务：

```
END;
```

兼容性

END 是一种 HashData 数据库的扩展，它提供和 COMMIT 等效的功能，后者是 SQL 标准中指定的。

另见

BEGIN、ROLLBACK、COMMIT

上级主题： SQL命令参考

EXECUTE

执行一个预备的 SQL 语句。

概要

```
EXECUTE name [ (parameter [, ...] ) ]
```

描述

EXECUTE 被用来执行一个之前准备好的语句。由于预备语句只在会话期间存在，该预备语句必须在当前会话中由一个更早执行的 PREPARE 语句所创建。

如果创建预备语句的 PREPARE 语句指定了一些参数，必须向 EXECUTE 语句传递一组兼容的参数，否则会发生错误。注意（与函数不同）预备语句无法基于其参数的类型或者数量重载。在一个数据库会话中，预备语句的名称必须唯一。

更多创建和使用预备语句的信息请见 PREPARE。

参数

name

要执行的预备语句的名称。

parameter

给预备语句的参数的实际值。这必须是一个能得到与该参数数据类型（ 在预备语句创建时决定）兼容的值的表达式。

示例

为一个 INSERT 语句创建一个预备语句，然后执行它：

```
PREPARE fooplan (int, text, bool, numeric) AS INSERT INTO
foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

兼容性

SQL 标准包括了一个 EXECUTE 语句，但是只被用于嵌入式 SQL。这个版本的 EXECUTE 语句也用了一种有点不同的语法。

另见

DEALLOCATE, PREPARE

上级主题： SQL命令参考

EXPLAIN

显示一个语句的查询计划

概要

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

描述

EXPLAIN 显示 HashData 计划器为提供的语句所生成的查询计划。查询计划是一颗节点计划树。在计划中的每个节点代表了一个操作，例如表扫描、连接、聚集或者是一个排序操作。

因为每个节点直接向它上面的节点提供行结果，所以计划应该从下往上进行阅读。最底层的节点通常是一些表扫描操作（顺序扫描、索引扫描或者是位图扫描）。如果查询要求连接、聚集或者排序（或者其他在原始行上的操作），那么需要再扫描节点增加这些操作的节点。计划最顶层的节点通常是 HashData 的 motion 节点（重分布、显式重分布、广播或者聚集 motion）。这些操作符代表了在查询处理期间在分片示例之间移动行数据。

EXPLAIN 的输出是树中每个节点有一行，显示基本的节点类型，紧接着是由计划器为执行该计划节点时的代价评估：

- **cost** — 通过读取磁盘页面的次数来度量。即，1.0 代表一个连续磁盘页面的读取。首先是启动代价（获取第一行的代价），第二个代价是总的代价（获取所有行的代价）。注意总的代价假设所有的行都将要被取回，但并不是总是这种情况（比如使用 LIMIT 语句）。
- **rows** — 该计划节点总的输出行数。这通常是小于实际由计划节点处理或者扫描的行数，主要是由于任何一个 WHERE 条件语句的评估选择性。理想情况下，最高层的节点估计近似等于实际由该查询返回、更新或者删除的行数。
- **width** — 该计划节点所有行输出的总的字节数。

值得注意的是更上一层的节点的代价包括了所有它孩子节点的代价。计划中最顶层节点是估计执行整个计划的代价。该数字是计划器寻求最小化的地方。同时也需要意识到代价仅仅反应了查询优化器关心的部分。特别是，代价没有考虑将结果行传输到客户端所花费的时间。

EXPLAIN ANALYZE 导致该语句被实际执行，而不仅仅是被计划。EXPLAIN ANALYZE 计划显示实际的结果以及计划器的评估。这个对于看是否计划器评估接近实际的情况非常有用。除了显示在 EXPLAIN 计划中的信息，EXPLAIN ANALYZE 还要外加显示下面的信息：

- 总的花费在执行该查询的时间间隔（以毫秒为单位）。
- 在一个计划节点操作中涉及到的 workers（Segment）的数量。只有返回行的 Segment 被计入。
- 一个操作中输出最多行的 Segment 返回的最大行数。如果多个 Segment 输出了相同数量的行数，取 time to end 最长的那个 Segment。
- 在一个操作中输出最多行的 Segment 的 ID。
- 对于相关的操作，该操作使用的 work_mem。如果 work_mem 不足以在内存中执行操作，计划将显示有多少数据溢出到磁盘上以及对于使用工作内存最少的执行 Segment 要求了多少趟对数据的处理。例如：

  ```
  Work_mem used: 64K bytes avg, 64K bytes max (seg0).
      Work_mem wanted: 90K bytes avg, 90K bytes max (seg0) to abate workfile
      I/O affecting 2 workers.
      [seg0] pass 0: 488 groups made from 488 rows; 263 rows written to
      workfile
      [seg0] pass 1: 263 groups made from 263 rows
  ```

- 产生最多行的 Segment 检索到第一行所花的时间（以毫秒计），以及在该 Segment 上获取所有行所花费的时间。如果 `<time> to first row` 同 `<time> to end` 相等，则前者有可能被省略。

> 重要： 记住当使用 EXPLAIN ANALYZE 语句会被实际执行。尽管 EXPLAIN ANALYZE 将丢弃 SELECT 所返回的任何输出，照例该语句的其他副作用还是会发生。如果用户希望在一个 DML 语句上执行EXPLAIN ANALYZE 而不希望它们影响用户的数据，可以使用下面的方法：

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

参数

name

将要执行的预备语句的名称。

parameter

预备语句的实际参数值。这一定是一个产生与该参数数据类型兼容的值的表达式，参数的数据类型则是在预备语句被创建时定义的。

注解

为了允许查询计划器在优化查询时能做出合理的知情决策，ANALYZE 语句需要被执行用来记录关于在表内数据的分布统计信息。如果用户还没有完成这个操作（或者如果表内数据从上一次执行 ANALYZE 语句后的统计分布发生了很大的变化），那么评估代价不会很符合实际的查询属性，同时因此会导致选择一个较差的计划被选中。

一个 SQL 语句在一个 EXPLAIN ANALYZE 命令被执行时执行会从 HashData 资源队列中排出。

更多关于资源队列的信息见 HashData 数据库管理员指南中的“用资源队列进行工作负载管理”部分。

示例

为了展示如何阅读一个 EXPLAIN 查询计划，考虑下面的一个简单查询的例子：

```
EXPLAIN SELECT * FROM names WHERE name = 'Joelle';
                     QUERY PLAN
-------------------------------------------------------------
Gather Motion 2:1 (slice1) (cost=0.00..20.88 rows=1 width=13)

   -> Seq Scan on 'names' (cost=0.00..20.88 rows=1 width=13)
         Filter: name::text ~~ 'Joelle'::text
```

如果我们从下往上阅读该计划，查询优化器开始于顺序扫描表names。注意 WHERE 子句作为一个过滤条件被应用。这意味着一个扫描操作要检验扫描中的每一行是否满足该条件，同时返回那些满足条件的行。

扫描操作的结果将向上传递到一个 gather motion 操作。在 HashData 数据库中， gather motion 是 Segment 向上传递行到 Master 的时机。在该例子中，我们有 2 个 Segment 实例发送到 1 个 Master 实例（2：1）。该操作工作在并行查询执行计划的 slice1 上。在 HashData 数据库中，一个查询计划被分成切片，这样每个查询计划的多个部分能够被 Segment 并行地执行。

该计划的估计启动代价是 00.00（没有代价）以及总代价为 20.88 次取磁盘页。计划器认为该查询将会返回一行。

兼容性

在 SQL 标准中没有定义 EXPLAIN 语句。

另见

ANALYZE

上级主题： SQL命令参考

FETCH

使用游标从查询中检索行。

概要

```
FETCH [ forward_direction { FROM | IN } ] cursorname
```

其中 forward_direction 可以为空或者为下列之一：

```
    NEXT
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
```

描述

FETCH 从之前创建的一个游标中检索行。

游标具有一个相关联的位置，FETCH 会用到该位置。游标位置可能会位于查询结果的第一行之前、结果中任意行之上或者结果的最后一行之后。在被创建时，游标被定位在第一行之前。在取出一些行后，该游标被定位在最近被取出的行上。如果 FETCH 运行超过了可用行的末尾，则该游标会被定位在最后一行之后。FETCH ALL 将总是让游标被定位于最后一行之后。

NEXT, FIRST, LAST, ABSOLUTE, RELATIVE 形式会在适当移动游标后取出一行。 如果没有这样一行，将返回一个空结果，并且视情况将游标定位在第一行之前或者最后一行之后。

使用形式 FORWARD 的形式会在向前移动的方向上检索指定数量的行，然后将游标定位在最后返回的行上（如果 count 超过可用的行数，则定位在所有行之后）。注意在 HashData 数据库中是不能够向后移动游标，因为不支持滚动游标。只能够用 FETCH 向前移动游标。

RELATIVE 0 、FORWARD 0 都会请求检索当前行但不移动游标，也就是重新取最近被取出的行。只要游标没有被定位在第一行之前或者最后一行之后，这种操作都会成功，否则不会返回任何行。

输出

如果成功完成，FETCH 命令返回一个下面形式的命令标签：

```
FETCH count
```

count 是取得的行数（可能为零）。注意在 psql 中，命令标签将不会实际显示，因为 psql 会显示被取得的行。

参数

forward_direction

定义获取方向以及要取得的行数。只有向前获取在 HashData 数据库中是支持的。它可以是下列之一：

NEXT

取出下一行。如果省略 direction，这将是默认值。

FIRST

取出该查询的第一行（和 ABSOLUTE 1 相同）。只有在该游标第一次使用 FETCH 时是允许的。

LAST

取出该查询的最后一行（和 ABSOLUTE -1 相同）。

ABSOLUTE count

取出查询的指定行。如果 count 的值超出范围就定位到最后一行之后。只有在被指定了 count 的行向前移动游标位置才能使用。

RELATIVE count

取出查询指定行 count 个后继行。RELATIVE 0 重新取出当前行（如果有）。只有在 count 向前移动游标位置时才允许。

count

获取后面 count 行（和 FORWARD count 相同）。

ALL

获取剩余的所有行（和 FORWARD ALL 相同）。

FORWARD

取出下一行（和 NEXT 相同）。

FORWARD count

取出接下来的 count 行。 FORWARD 0 重新取出当前行。

FORWARD ALL

取出所有剩下的行。

cursorname

一个已打开游标的名称。

注解

HashData 数据库不支持滚动游标，所以只能用 FETCH 向前移动游标位置。

ABSOLUTE 取行并不比用相对移动快多少：不管则样，底层实现都必须遍历所有的中间行。

DECLARE 用来定义一个游标。使用 MOVE 可以改变游标的位置而不检索数据。

示例

-- 开始一个事务：

```
BEGIN;
```

-- 建立一个游标：

```
DECLARE mycursor CURSOR FOR SELECT * FROM films;
```

-- 在游标 mycursor 中获取前五行：

```
FETCH FORWARD 5 FROM mycursor;
 code  |          title          | did | date_prod  |   kind   |  len
-------+-------------------------+-----+------------+----------+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28
```

-- 关闭游标并结束事务：

```
CLOSE mycursor;
COMMIT;
```

修改表 films 的游标 c_films 指向的当前位置的行的 kind 列的值：

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

兼容性

SQL 标准只在嵌入式 SQL 和模块中使用游标。 HashData 数据库允许游标在交互式的环境下使用。

这里描述的 FETCH 变体返回数据时就好像数据是一个 SELECT 结果，而不是被放在主变量中。除这一点之外，FETCH 完全向上兼容于 SQL 标准。

涉及 FORWARD 形式的 FETCH ，以及形式 FETCH count 和 FETCH ALL（其中 FORWARD 是隐式的）都是 HashData 数据库扩展。不支持 BACKWARD 。

SQL 标准只允许 FROM 在游标名之前。使用 IN 的选项是一种扩展。

另见

DECLARE, CLOSE, MOVE

上级主题： SQL命令参考

GRANT

定义访问特权。

概要

```
GRANT { {SELECT | INSERT | UPDATE | DELETE | REFERENCES |
TRIGGER | TRUNCATE } [,...] | ALL [PRIVILEGES] }
    ON [TABLE] tablename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {USAGE | SELECT | UPDATE} [,...] | ALL [PRIVILEGES] }
    ON SEQUENCE sequencename [, ...]
    TO { rolename | PUBLIC } [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | CONNECT | TEMPORARY | TEMP} [,...] | ALL
[PRIVILEGES] }
    ON DATABASE dbname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { EXECUTE | ALL [PRIVILEGES] }
    ON FUNCTION funcname ( [ [argmode] [argname] argtype [, ...]
] ) [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { USAGE | ALL [PRIVILEGES] }
    ON LANGUAGE langname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | USAGE} [,...] | ALL [PRIVILEGES] }
    ON SCHEMA schemaname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { CREATE | ALL [PRIVILEGES] }
    ON TABLESPACE tablespacename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT parent_role [, ...]
    TO member_role [, ...] [WITH ADMIN OPTION]

GRANT { SELECT | INSERT | ALL [PRIVILEGES] }
    ON PROTOCOL protocolname
    TO username
```

描述

GRANT 命令由两种基本的变体：一种授予在一个数据库对象（表、列、视图、外部表、序列、数据库、外部数据包装器、外部服务器、函数、过程语言、方案或表空间）上的特权，另一个授予一个角色中的成员关系。

在数据库对象上 **GRANT**

这种 GRANT 命令的变体将一个数据库对象上的指定特权交给一个或多个角色。如果有一些已经被授予，这些特权会被加入到它们之中。

关键词 PUBLIC 指示特权要被授予给所有角色，包括那些可能稍后会被创建的角色。 PUBLIC 可以被认为是一个被隐式定义的总是包含所有角色的组。任何特定角色都将具有直接授予给它的特权、授予给它作为成员所在的任何角色的特权以及被授予给 PUBLIC 的特权。

如果指定了 WITH GRANT OPTION ，特权的接收者可以接着把它授予给其他人。如果没有授权选项，接收者就不能这样做。授权选项不能被授予给 PUBLIC。

没有必要把权限授予给一个对象的拥有者（通常就是创建该对象的用户），因为拥有者默认具有所有的特权。 删除一个对象或者以任何方式修改其定义的权力是不被当作一个可授予特权的，它被固化在拥有者中，并且不能被授予和撤回。拥有者也

隐式地拥有该对象的所有授权选项。

会把某些类型的对象上的默认特权授予给 PUBLIC。默认不包括对表、方案以及表空间的公共访问；CONNECT 特权以及数据库中对 TEMP 表的创建特权；函数的 EXECUTE 特权；语言的 USAGE 特权。对象的拥有者当然可能 revoke 这些特权。

角色上的 **GRANT**

GRANT 命令的这种变体把一个角色中的成员关系授予一个或者多个其他角色。一个角色中的成员关系是有意义的，因为它会把授予给一个角色的特权带给该角色的每一个成员。

如果指定了 WITH ADMIN OPTION ，成员接着可以把该角色中的成员关系授予给其他用户，也可以撤回该角色中的成员关系。数据库超级用户能够授予或撤回任何角色中任何人的成员关系。具有 CREATEROLE 特权的角色能够授予或者撤回任何非超级用户角色中的成员关系。

和特权的情况不同，一个角色中的成员关系不能被授予 PUBLIC。

协议上的**GRANT**

当创建一个用户协议后，指定 specify CREATE TRUSTED PROTOCOL 能够允许任何一个除了其拥有者来访问它。如果协议是不可信的，用户不能给任何其他用户通过该协议来读或者写数据。当一个 TRUSTED 协议被创建后，可以用 GRANT 命令指定哪些用户可以访问它。

- 为了让一个用户能够创建带有可信协议的外部可读表，使用

```
GRANT SELECT ON PROTOCOL protocolname TO username
```

- 为了让一个用户能够创建带有可信协议的外部可写表，使用

```
GRANT INSERT ON PROTOCOL protocolname TO username
```

- 为了让一个用户能够创建带有可信协议的外部可读且可写表，使用

```
GRANT ALL ON PROTOCOL protocolname TO username
```

参数

SELECT

允许从指定表、视图或序列的任何列或者列出的特定列进行 SELECT。还允许使用 COPY TO。 对于序列，这个特权也允许使用 ccurrval 函数。

INSERT

允许 INSERT 一个新行到指定表中。 还允许 COPY FROM。

UPDATE

允许对指定表的特定列进行 UPDATE。 SELECT ... FOR UPDATE and SELECT ... FOR SHARE 也需要该特权（同时也要 SELECT 特权）。 对于序列，这个特权允许使用 nextval 和 setval 函数。

DELETE

允许从指定的表中 DELETE 一行。

REFERENCES

这个关键词是可以接受的，尽管现在外键约束在 HashData 数据库中还不支持。要创建一个外键约束，必须在引用列和被引用列上都有这个特权。

TRIGGER

允许在指定的表上创建触发器。

> 注解：HashData 数据库不支持触发器。

TRUNCATE

允许在指定表上使用 TRUNCATE 。

CREATE

对于数据库，允许在其中创建新方案。

对于方案，允许在其中创建新的对象。要重命名一个已有对象，用户必须拥有该对象并且具有所在方案的这个特权。

对于表空间，允许在其中创建表、索引，并且允许创建使用该表空间作为默认表空间的数据库（注意撤回这个特权将不会更改现有对象的放置位置）。

CONNECT

允许用户连接到指定数据库。在连接开始时会检查这个特权（除了检查由 pg_hba.conf 施加的任何限制之外）。

TEMPORARY

TEMP

允许在使用指定数据库时创建临时表。

EXECUTE

允许使用指定的函数以及使用在该函数之上实现的任何操作符。这是适用于函数的唯一一种特权类型（这种语法也可用于聚集函数）。

USAGE

对于过程语言，允许使用指定的语言创建函数。这是适用于过程语言的唯一一种特权类型。

对于方案，允许访问包含在指定方案中的对象（假定这些对象的拥有特权要求也满足）。本质上这允许被授权者在方案中"查阅"对象。

对于序列，这种特权允许使用 currval 和 nextval 函数。

ALL PRIVILEGES

一次授予所有的可用特权。在 HashData 数据库中，PRIVILEGES 关键词是可选的，但是在严格的 SQL 中是要求它的。

PUBLIC

一个特别的组级别的角色，它指示了那些授予给所有角色的特权，包括后来可能会被创建的。

WITH GRANT OPTION

特权接受者可以把该特权授予给其他的用户。

WITH ADMIN OPTION

成员接着可以把该角色中的成员关系授予给其他用户。

注解

数据库超级用户可以访问所有对象而不管对象特权的设置。对于该规则的一个例外是视图。访问一个视图中引用的表取决于视图的拥有者而不是当前用户（即使当前用户是超级用户）。

如果一个超级用户选择发出一个 GRANT 或者 REVOKE 命令，该命令将被执行，好像它是由被影响对象的拥有者发出的一样。特别地，通过这样一个命令授予的特权将好像是由对象拥有者授予的一样。对于角色成员关系，该成员关系好像是由该角色本身授予的一样。

GRANT 以及 REVOKE 也可以由一个不是受影响对象拥有者的角色完成，不过该角色是拥有该对象的角色的一个成员，或者是在该对象上持有特权的 WITH GRANT OPTION 的角色的一个成员。在这种情况下，特权将被记录为由实际拥有该对象的角色授予或者是由持有特权的 WITH GRANT OPTION 的角色授予。

授予一个表上的权限不会自动地扩展权限给该表使用的任何序列，包括绑定在 SERIAL 列上的序列。序列上的权限必须被独立设置。

HashData 数据库不支持授予或者回收一个表上单独列上的特权。一个可选的方案是在需要赋予的特权的列上创建视图，然后授予视图特权。

使用 psql 的 z 命令能够获取一个对象上现在存在的特权信息。

示例

授予所有角色在表 mytable 上的插入特权：

```
GRANT INSERT ON mytable TO PUBLIC;
```

授予所有可以的特权给角色 sally 在视图 topten 上。注意只有在超级用户或者视图 topten 拥有者执行时才会真正的将所有特权都授予，如果执行的是其他用户，那么只会授予那么授予角色现在拥有的可进行授予的特权。

```
GRANT ALL PRIVILEGES ON topten TO sally;
```

将角色 admins 授予用户 joe：

```
GRANT admins TO joe;
```

兼容性

在 SQL 标准中， PRIVILEGES 关键词是必需的，但是在 HashData 数据库中是可选的。SQL 标准不支持在每个命令中设置超过一个对象上的特权。

HashData 数据库允许一个用户的拥有者撤回它们拥有的的普通特权：一个表拥有者可以通过撤回其自身拥有的 INSERT、UPDATE、DELETE 和 TRUNCATE 特权让该表对它们自己只读。根据 SQL 标准这是不可能发生的。 原因在于 HashData 数据库认为拥有者的特权是由拥有者授予给它们自己的，因此它们也能够撤回它们。 在 SQL 标准中，拥有者的特权是有一个假设的实体 system 所授予。

SQL 标准允许在一个表的单独列上设置特权。

SQL 标准提供了其他对象类型上的 USAGE 特权：字符集、排序规则、翻译、域。

数据库、表空间、方案和语言上的特权都是 HashData 数据库扩展。

另见

REVOKE

上级主题： SQL命令参考

INSERT

在表中创建新行。

概要

```
INSERT INTO table [( column [, ...] )]
   {DEFAULT VALUES | VALUES ( {expression | DEFAULT} [, ...] )
   [, ...] | query}
```

描述

INSERT 将新行插入到一个表中。我们可以插入一个或者更多由值表达式指定的行，或者插入来自一个查询的零行或者更多行。

目标列的名称可以以任意顺序列出。如果没有给出列名列表，默认的方式是按照表在定义时列的顺序。 VALUES 子句或者 query 提供的值会被从左至右关联到这些显式或者隐式给出的目标列。

每一个没有出现在显式或者隐式列列表中的列都将被默认填充，如果为该列声明过默认值则用默认值填充，否则用空值填充。

如果任意列的表达式不是正确的数据类型，将会尝试自动类型转换。

为了向表中插入数据，用户必须拥有在其上的 INSERT 特权。

输出

成功完成时，INSERT 命令会返回以下形式的命令标签：

```
INSERT oid count
```

count 是被插入的行数。 如果 count 正好为 1 并且目标表具有 OID，那么 oid 就是分配给被插入行的 OID。否则 oid 为零。

参数

table

一个已有表的名称（可以被方案限定）。

column

在表中的列的名称。如有必要，列名可以用一个子域名或者数组下标限定（指向 一个组合列的某些列中插入会让其他域为空）。

DEFAULT VALUES

所有列都将被其默认值填充。

expression

要赋予给相应列的表达式或者值。

DEFAULT

相应的列将被其默认值填充。

query

提供要被插入行的查询（SELECT 语句）。 其语法描述参考 SELECT 语句。

注解

要插入数据到一个分区表中，需要指定根分区表，该表通过 CREATE TABLE 命令创建。也需要在 INSERT 命令中指定分区表的一个叶子表。如果对于指定的叶子表数据是无效将抛出错误。在 INSERT 命令指定一个非叶子表为子表还不支持。其他 DML 命令，例如 UPDATE 和 DELETE，在任何一个分区表的子表上的执行还不支持。这些命令一定是执行在根分区表上面，即那些通过 CREATE TABLE 创建的表。

对于追加优化表， HashData 数据库支持最大并行度为 127 的并行 INSERT 事务插入到一个追加优化表中。

对于可写 S3 外部表，INSERT 操作更新在配置 S3 桶中的一个或者更多的文件。通过 Ctrl-c能够取消 INSERT 同时停止更新到 S3。

示例

插入当条行到表 films 中：

```
INSERT INTO films VALUES ('UA502', 'Bananas', 105,
'1971-07-13', 'Comedy', '82 minutes');
```

在这个示例中，列 length 被忽略，因此它将被默认值填充：

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

这个示例中对列 date_prod 使用了 DEFAULT 子句而不是指定一个具体的值：

```
INSERT INTO films VALUES ('UA502', 'Bananas', 105, DEFAULT,
'Comedy', '82 minutes');
```

插入一条全部都是由默认值组成的行：

```
INSERT INTO films DEFAULT VALUES;
```

通过语法多行的 VALUES 语法同时插入多条行的数据：

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

在这个示例中，从具有相同表结构的表 tmp_films 插入若干的行到表 films：

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod <
'2004-05-07';
```

兼容性

INSERT 符合 SQL 标准。标准不允许省略列名列表但不通过 VALUES 或者 query 填充所有列的情况。

query 子句可能的限制在 SELECT 有介绍。

另见

COPY, SELECT, CREATE EXTERNAL TABLE

上级主题： SQL命令参考

LOAD

载入一个共享库文件。

概要

```
LOAD 'filename'
```

描述

该命令载入一个共享库文件到 HashData 数据库服务器的地址空间。如果文件之前已经被载入，它首先会被卸载。该命令主要用于卸载并重装载一个在服务器第一次装载它之后发生变化的共享库文件。要使用共享库文件，其中的函数需要由 CREATE FUNCTION 命令声明。

文件名采用和 CREATE FUNCTION 中的共享库名称相同的方式指定；特别地，文件名可能依赖于搜索路径以及对系统标准共享库文件名扩展的自动增加。

注意在 HashData 数据库中，共享库文件（.so文件）在 HashData 数据库阵列（Master、Segment 以及镜像）中的每一台主机上都必须位于相同的路径位置。

参数

filename

共享库文件的路径以及文件名。该文件在 HashData 阵列中所有的主机上都必须位于相同的位置。

示例

载入一个共享库文件：

```
LOAD '/usr/local/ HashData -db/lib/myfuncs.so';
```

兼容性

LOAD 是 HashData 扩展。

另见

CREATE FUNCTION

上级主题： SQL命令参考

LOCK

锁定一个表。

概要

```
LOCK [TABLE] name [, ...] [IN lockmode MODE] [NOWAIT]
```

lockmode 可以是下列之一：

```
    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
  | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

描述

LOCK TABLE 获得一个表级锁，必要时会等待任何冲突锁被释放。如果指定了 NOWAIT，LOCK TABLE 不会等待以获得想要的锁：如果它不能立刻得到，该命令会被中止并且发出一个错误。一旦获取到，该锁会被在当前事务中一直持有。没有 UNLOCK TABLE 命令，锁总是在事务结束时被释放。

在为引用表的命令自动获取锁时， HashData 数据库总是尽可能使用最不严格的锁模式。提供 LOCK TABLE 是用于想要更严格的锁定的情况。例如，假设一个应用运行一个 Read Committed 隔离级别的事务，并且需要确保一个表中的数据在该事务的期间保持稳定。要实现这个目的，必须在查询之前在表上获得 SHARE 锁模式。这将阻止并发的数据更改并且确保该表的后续读操作会看到已提交数据的一个稳定视图，因为 SHARE 锁模式与写入者所要求的 ROW EXCLUSIVE 锁有冲突，并且用户的 LOCK TABLE name IN SHARE MODE 语句将等待，直到任何并发持有 ROW EXCLUSIVE 模式锁的持有者提交或者回滚。因此，一旦得到锁，就不会有未提交的写入还没有解决。更进一步，在释放该锁之前，任何人都不能开始。

要在运行在 Serializable 隔离级别的事务中得到类似的效果，用户必须在执行任何 SELECT 或者数据修改语句之前执行 LOCK TABLE 语句。 一个事务的数据视图将在它的第一个 SELECT 或者数据修改语句开始时被冻结。在该事务中稍后的一个 LOCK TABLE 仍将阻止并发写 — 但它不会确保该事务读到的东西对应于最新的已提交值。

如果一个此类事务正要修改表中的数据，那么它应该使用 SHARE ROW EXCLUSIVE 锁模式来取代 SHARE 模式。这会保证一次只有一个此类事务运行。如果不用这种模式，死锁就可能出现：两个事务可能都要求 SHARE 模式，并且都不能获得 ROW EXCLUSIVE 模式来真正地执行它们的更新注意一个事务所拥有的锁不会冲突，因此一个事务可以在它持有 SHARE 模式时获得 ROW EXCLUSIVE 模式 —— 但是如果有其他人持有 SHARE 模式时则不能。为了避免死锁，确保所有的事务在同样的对象上以相同的顺序获得锁，并且如果在一个对象上涉及多种锁模式，事务应该总是首先获得最严格的那种模式。

参数

name

要锁定的一个现有表的名称（可以是方案限定的）。

如果给出了多个表，这些表会按照 LOCK TABLE 中的顺序一个一个被锁定。

lockmode

锁模式指定这个锁和哪些锁冲突。如果没有指定锁模式，那将使用最严格的模式 ACCESS EXCLUSIVE。 所模式包括下面的情况：

- ACCESS SHARE — 只与 ACCESS EXCLUSIVE 锁模式冲突。 SELECT 和 ANALYZE 命令在被引用的表上获得一个这种模式的锁。通常，任何只读取表而不修改它的查询都将获得这种锁模式。
- ROW SHARE — 与 EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。 SELECT FOR SHARE 命令在目标表上获得一个这种模式的锁（加上在被引用但没有 SELECT FOR SHARE 的任何其他表上的 ACCESS SHARE 锁）。

- ROW EXCLUSIVE — 与 SHARE、 SHARE ROW EXCLUSIVE、 EXCLUSIVE、和 ACCESS EXCLUSIVE 锁模式冲突。 命令 INSERT 和 COPY 在目标表上取得这种锁模式（加上在任何其他被引用表上的 ACCESS SHARE 锁）。
- SHARE UPDATE EXCLUSIVE — 与 SHARE UPDATE EXCLUSIVE、 SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。 这种模式保护一个表不受并发模式改变和 VACUUM 运行的影响。 由 VACUUM （不带 FULL) 取得。
- SHARE — 与 ROW EXCLUSIVE、 SHARE UPDATE EXCLUSIVE、 SHARE ROW EXCLUSIVE、 EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。这种模式保护一个表不受并发数据改变的影响。由 CREATE INDEX 取得。
- SHARE ROW EXCLUSIVE — 与 ROW EXCLUSIVE、 SHARE UPDATE EXCLUSIVE、 SHARE、 SHARE ROW EXCLUSIVE、 EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。该锁模式不能由 HashData 数据库中的命令直接取得。
- EXCLUSIVE — 与 ROW SHARE、ROW EXCLUSIVE、 SHARE UPDATE EXCLUSIVE、SHARE、 SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 锁模式冲突。 这种模式只允许并发的 ACCESS SHARE 锁，即只有来自于表的读操作可以与一个持有该锁模式的事务并行处理。在 HashData 数据库中，该锁模式由 UPDATE、SELECT FOR UPDATE 和 DELETE 命令取得（这里比 PostgreSQL 更加的严格）。
- ACCESS EXCLUSIVE — 与所有模式的锁冲突（ACCESS SHARE、 ROW SHARE、 ROW EXCLUSIVE、 SHARE UPDATE EXCLUSIVE、 SHARE、 SHARE ROW EXCLUSIVE、 EXCLUSIVE 和 ACCESS EXCLUSIVE）。 这种模式保证持有者是访问该表的唯一事务。由 ALTER TABLE、 DROP TABLE、REINDEX、 CLUSTER 和 VACUUM FULL 命令取得。这也是未显式指定模式的 LOCK TABLE 命令的默认锁模式。在处理过程中，该锁模式也可以由使用 VACUUM （不带FULL）命令追加优化表上取得。

NOWAIT

指定 LOCK TABLE 不等待任何冲突锁被释放：如果所指定的锁不能立即获得，那么事务就会中止。

注解

LOCK TABLE ... IN ACCESS SHARE MODE 要求目标表上的 SELECT 特权。所有其他形式的 LOCK 要求 UPDATE、DELETE 特权。

LOCK TABLE 只在一个事务块内部有用（BEGIN/COMMIT 对），因为一旦事务结束，锁就会被删除。出现在事务块之外的任何 LOCK TABLE 命令会形成一个自包含的事务，这样锁将在被获得时马上被删除。

LOCK TABLE 只处理表级锁，因此涉及到 ROW 的模式名称在这里都是不正当的。这些模式名称应该通常被解读为用户在被锁定表中获取行级锁的意向。还有，ROW EXCLUSIVE 模式是一个可共享的表锁。记住就 LOCK TABLE 而言，所有的锁模式都具有相同的语义，只有模式的冲突规则有所不同。关于如何获取一个真正的行级锁的信息， 请见 SELECT 参考文档中的 FOR UPDATE/FOR SHARE 。

示例

获取一个 SHARE 锁在 films 表上当在表 `films_user_comments` 上执行插入时：

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- 如果记录没有被返回就ROLLBACK
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

在将要执行一次删除操作前在表上取一个 SHARE ROW EXCLUSIVE 锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

兼容性

在 SQL 标准中没有 LOCK TABLE，SQL 标准中使用 SET TRANSACTION 指定事务上的并发层次。 HashData 数据库也支持这样做。

除 ACCESS SHARE、ACCESS EXCLUSIVE 和 SHARE UPDATE EXCLUSIVE 锁模式之外， HashData 数据库的锁模式和 LOCK TABLE 语法与 Oracle 中的兼容。

另见

BEGIN, SET TRANSACTION, SELECT

上级主题： SQL命令参考

MOVE

定位一个游标

概要

```
MOVE [ forward_direction {FROM | IN} ] cursorname
```

其中 forward_direction 可以为空或者下列之一：

```
    NEXT
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
```

描述

MOVE 重新定位一个游标而不检索任何数据。MOVE 的工作完全像 FETCH 命令，但是它只定位游标并且不返回行。

注意在 HashData 数据库中向后移动一个游标是不可能的，因为在 HashData 数据库中不支持滚动游标。只能够用 MOVE 向前移动游标位置。

输出

成功完成时，MOVE 命令返回的命令标签形式是

```
MOVE count
```

count 是一个具有同样参数的 FETCH 命令会返回的行数（可能为零）。

参数

forward_direction

见 FETCH 获取更多信息。

cursorname

一个打开的游标名称。

示例

-- 开始一个事务：

```
BEGIN;
```

-- 建立一个游标：

```
DECLARE mycursor CURSOR FOR SELECT * FROM films;
```

-- 使用游标 mycursor 向前移动 5 行：

```
MOVE FORWARD 5 IN mycursor;
MOVE 5
```

--获取之后的一行（第六行）

```
FETCH 1 FROM mycursor;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)
```

--关闭游标，结束事务：

```
CLOSE mycursor;
COMMIT;
```

兼容性

在 SQL 标准中没有 MOVE 语句。

另见

DECLARE, FETCH, CLOSE

上级主题： SQL命令参考

PREPARE

为执行准备一个语句

概要

```
PREPARE name [ (datatype [, ...] ) ] AS statement
```

描述

PREPARE 创建一个预备语句，有可能有未绑定参数。预备语句是一种服务器端对象，它可以被用来优化性能。一个预备语句可能后续给参数绑定值后被执行。HashData 数据库可能选择对同一预备语句的不同执行进行重新规划。

预备语句可以接受参数：在执行时会被替换到语句中的值。在创建预备语句时，可以用位置引用参数，如 `$1`、`$2` 等。也可以选择性地指定参数数据类型的一个列表。当一个参数的数据类型没有被指定或者被声明为 unknown 时，其类型会从该参数被使用的环境中推知（如果可能）。在执行该语句时，在 EXECUTE 语句中为这些参数指定实际值。

预备语句只在当前数据库会话期间存在。当会话结束时，预备语句会消失，因此在重新使用之前必须重新建立它。这也意味着一个预备语句不能被多个数据库客户端同时使用。不过，每一个客户端可以创建它们自己的预备语句来使用。预备语句可以用 DEALLOCATE 命令手工清除。

当一个会话要执行大量类似语句时，预备语句可能会有最大性能优势。如果该语句很复杂（难于规划或重写），例如，如果查询涉及很多表的连接或者要求应用多个规则，性能差异将会特别明显。如果语句相对比较容易规划和重写，但是执行起来开销相对较大，那么预备语句的性能优势就不那么显著了。

参数

name

给这个特定预备语句的任意名称。它在一个会话中必须唯一并且后续将被用来执行或者清除一个之前准备好的语句。

datatype

预备语句一个参数的数据类型。如果一个特定参数的数据类型没有被指定或者被指定为 unknown，将从该参数被使用的环境中推得。要在预备语句本身中引用参数，可以使用 $1、 $2 等。

statement

任何 SELECT, INSERT, UPDATE, DELETE 或者 VALUES 语句。

注解

在某些场景中，为一个预备语句产生的查询计划将比不上该语句被正常提交执行时的查询计划。这是因为当语句被规划并且规划器尝试确定最优查询计划时，该语句中指定的任何参数的实际值都还不可用。 HashData 数据库收集表中数据分布的统计信息，同时可以用语句中的常量值来猜测执行该语句的可能结果。由于在使用参数规划预备语句时，这种数据还不可用，被选中的计划可能是次优的。为了检测 HashData 数据库中为预备语句选择的查询计划，可以使用 EXPLAIN。

更多关于查询规划以及 HashData 数据库为此所收集的统计信息的内容，请见 ANALYZE 文档。

可以通过查询 pg_prepared_statements 系统视图来看到会话中所有可用的预备语句。

示例

为一个 INSERT 语句创建一个预备语句，然后执行它：

```
PREPARE fooplan (int, text, bool, numeric) AS INSERT INTO
foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

为一个 SELECT 语句创建一个预备语句，然后执行它。注意第二个参数的数据类型没有被指定，因此会从使用 `$2` 的环境中推知：

```
PREPARE usrrptplan (int) AS SELECT * FROM users u, logs l
WHERE u.usrid=$1 AND u.usrid=l.usrid AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

兼容性

SQL 标准包括一个 PREPARE 语句，但是它只用于嵌入式 SQL。这个版本的 PREPARE 语句也使用了一种有些不同的语法。

另见

EXECUTE, DEALLOCATE

上级主题： SQL命令参考

REASSIGN OWNED

更改一个数据库角色拥有的数据库对象的拥有关系。

概要

```
REASSIGN OWNED BY old_role [, ...] TO new_role
```

描述

REASSIGN OWNED 指示系统把 old_role 们拥有的任何数据库对象的拥有关系更改为 new_role。注意它并不改变数据库本身的拥有关系。

参数

old_role

一个角色的名称。这个角色在当前数据库中所拥有的所有对象以及所有共享对象（数据库、表空间）的所有权都将被重新赋予给 new_role。

new_role

将作为受影响对象的新拥有者的角色名称。

注解

REASSIGN OWNED 经常被用来为移除一个或者多个角色做准备。因为 REASSIGN OWNED 只影响当前数据库中的对象，通常需要在包含有被删除的角色所拥有的对象的每一个数据库中都执行这个命令。

DROP OWNED 命令可以简单地删掉一个或者多个角色所拥有的所有数据库对象。

REASSIGN OWNED 命令不会影响授予给 old_role 们的在它们不拥有的对象上的任何特权。DROP OWNED 可以回收那些特权。

示例

重新将 sally 和 bob 拥有的对象改为 admin 拥有的：

```
REASSIGN OWNED BY sally, bob TO admin;
```

兼容性

REASSIGN OWNED 语句是 HashData 数据库的扩展。

另见

DROP OWNED, DROP ROLE

上级主题： SQL命令参考

REINDEX

重建索引。

概要

```
REINDEX {INDEX | TABLE | DATABASE | SYSTEM} name
```

描述

REINDEX 使用索引的表里存储的数据重建一个索引，并且替换该索引的旧拷贝。有一些场景需要使用 REINDEX：

- 一个索引变得 “臃肿”，其中包含很多空的或者近乎为空的页面。 HashData 数据库中的 B-树索引在特定的非常规访问模式下可能会发生这种情况。REINDEX 提供了一种方法来减少索引的空间消耗，即制造一个新版本的索引，其中没有死亡页面。
- 修改了一个索引的存储参数（例如填充因子），并且希望确保这种修改完全生效。

参数

INDEX

重新创建指定的索引。

TABLE

重新创建指定表的所有索引。如果该表有一个二级 “TOAST” 表，它也会被重索引。

DATABASE

重新创建当前数据库内的所有索引。共享的系统目录上的索引也会被处理。这种形式的 REINDEX 不能在一个事务块内执行。

SYSTEM

重新创建当前数据库中在系统目录上的所有索引。共享系统目录上的索引也被包括在内。用户表上的索引则不会被处理。这种形式的 REINDEX 不能在一个事务块内执行。

name

要被重索引的特定索引、表或者数据库的名字。索引和表名可以被方案限定。当前，REINDEX DATABASE 和 REINDEX SYSTEM 只能重索引当前数据库，因此它们的参数必须匹配当前数据库的名称。

注解

REINDEX 类似于删除索引并且重建索引，在其中索引内容会被从头开始建立。不过，锁定方面的考虑却相当不同。REINDEX 会用锁排斥写，但不会排斥在索引的父表上的读。它也会在被处理的索引上取得一个排他锁，该锁将会阻塞对该索引的使用尝试。相反，DROP INDEX 会暂时在父表上取得一个排他锁，阻塞写和读。 后续的 CREATE INDEX 会排斥写但不排斥读，由于该索引不存在，所以不会有读取它的尝试，这意味着不会有阻塞但是读操作可能被强制成昂贵的顺序扫描。

重索引单独一个索引或者表要求用户是该索引或表的拥有者。重索引一个数据库要求用户是该数据库的拥有者（注意拥有者因此可以重建由其他 用户拥有的索引或者表）。当然，超级用户总是能够重索引任何东西。

如果怀疑共享全局系统表目录索引已经损坏，它们只能在 HashData 数据库的 utility 模式下进行重建索引。损坏一个共享索

引的典型症状会出现 “index is not a btree” 的错误，或者服务器在启动的时候由于依赖在已经损坏的索引上而立即崩溃。在这种情况下，联系 HashData 客服支持获取帮助。

示例

重建一个单独的索引：

```
REINDEX INDEX my_index;
```

重建表 my_table 上的所有索引：

```
REINDEX TABLE my_table;
```

兼容性

在 SQL 标准中没有 REINDEX 命令。

另见

CREATE INDEX, DROP INDEX, VACUUM

上级主题： SQL命令参考

RELEASE SAVEPOINT

销毁一个之前定义的保存点。

概要

```
RELEASE [SAVEPOINT] savepoint_name
```

描述

RELEASE SAVEPOINT 销毁在当前事务中之前定义的一个保存点。

销毁一个保存点会使得它不能再作为一个回滚点，但是它没有其他用户可见的行为。它不会撤销在该保存点被建立之后执行的命令的效果（要这样做，可见 ROLLBACK TO SAVEPOINT ）。当不再需要一个保存点时销毁它允许系统在事务结束之前回收一些资源。

RELEASE SAVEPOINT 也会销毁所有在该保存点建立之后建立的保存点。

参数

savepoint_name

要销毁的保存点的名称。

示例

建立并销毁一个保存点：

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上述事务将插入 3 和 4。

兼容性

该命令符合 SQL 标准。 标准指定了关键词 SAVEPOINT 是强制需要的，但是在 HashData 数据库中允许将其忽略。

另见

BEGIN, SAVEPOINT, ROLLBACK TO SAVEPOINT, COMMIT

上级话题： SQL命令参考

RESET

把一个运行时参数的值恢复到默认值。

概要

```
RESET configuration_parameter
RESET ALL
```

描述

RESET 把运行时参数恢复到它们的默认值。RESET 是 SET configuration_parameter TO DEFAULT 的另外一种写法。

默认值被定义为如果在当前会话中没有发出过 SET，参数必须具有的值。这个值的实际来源可能是一个编译在内部的默认值、配置文件（postgresql.conf）、命令行选项、或者针对每个数据库或者每个用户的默认设置。

参数

configuration_parameter

一个可设置的运行时参数名称。

ALL

把所有可设置的运行时参数重置为默认值。

示例

将 statement_mem 设置为其默认值：

```
RESET statement_mem;
```

兼容性

RESET 是 HashData 数据库的扩展。

另见

SET

上级话题： SQL命令参考

REVOKE

移除访问特权。

概要

```
REVOKE [GRANT OPTION FOR] { {SELECT | INSERT | UPDATE | DELETE
       | REFERENCES | TRIGGER | TRUNCATE } [,...] | ALL [PRIVILEGES] }
       ON [TABLE] tablename [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {USAGE | SELECT | UPDATE} [,...]
       | ALL [PRIVILEGES] }
       ON SEQUENCE sequencename [, ...]
       FROM { rolename | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {CREATE | CONNECT
       | TEMPORARY | TEMP} [,...] | ALL [PRIVILEGES] }
       ON DATABASE dbname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {EXECUTE | ALL [PRIVILEGES]}
       ON FUNCTION funcname ( [[argmode] [argname] argtype
                              [, ...]] ) [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {USAGE | ALL [PRIVILEGES]}
       ON LANGUAGE langname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [GRANT OPTION FOR] { {CREATE | USAGE} [,...]
       | ALL [PRIVILEGES] }
       ON SCHEMA schemaname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { CREATE | ALL [PRIVILEGES] }
       ON TABLESPACE tablespacename [, ...]
       FROM { rolename | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [ADMIN OPTION FOR] parent_role [, ...]
       FROM member_role [, ...]
       [CASCADE | RESTRICT]
```

描述

REVOKE 命令收回之前从一个或者更多角色授予的特权。关键词 PUBLIC 隐式定义的全部角色的组。

特权类型的含义见 GRANT

注意任何特定角色拥有的特权包括直接授予给它的特权、从它作为其成员的角色中得到的特权以及授予给 PUBLIC 的特权。因此，例如从 PUBLIC 收回 SELECT 特权并一定意味着所有角色都失去在该对象上的 SELECT 特权：那些直接被授予的或者通过另一个角色被授予的角色仍然会拥有它。

如果指定了 GRANT OPTION FOR ，只会回收该特权的授予选项，特权本身不被回收。否则，特权及其授予选项都会被回收。

如果一个用户持有一个带有授予选项的特权并且把它授予给了其他用户，那么被那些其他用户持有的该特权被称为依赖特

权。如果第一个用户持有的该特权或者授予选项正在被收回且存在依赖特权，指定 CASCADE 可以连带回收那些依赖特权，不指定则会导致回收动作失败。这种递归回收只影响通过可追溯到该 REVOKE 的主体的用户链授予的特权。因此，如果该特权经由其他用户授予给受影响用户，受影响用户可能实际上还保留有该特权。

在回收一个角色中的成员关系时，GRANT OPTION 被改称为 ADMIN OPTION 但行为是类似的。

参数

见 GRANT 部分。

示例

从 public 收回表 films 的插入特权：

```
REVOKE INSERT ON films FROM PUBLIC;
```

从角色 sally 收回视图 topten 上的所有特权，注意实际意味着回收所有当前角色（如果不是超级用户）授予的特权：

```
REVOKE ALL PRIVILEGES ON topten FROM sally;
```

从 joe 收回角色 admins 中的成员关系：

```
REVOKE admins FROM joe;
```

兼容性

GRANT 命令的兼容性注解同样适用于 REVOKE。

关键词 RESTRICT 或者 CASCADE 根据标准是必需的，但是 HashData 数据库假定 RESTRICT 为默认值。

另见

GRANT

上级话题： SQL命令参考

ROLLBACK TO SAVEPOINT

回滚当前事务到一个保存点。

概要

```
ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] savepoint_name
```

描述

回滚在该保存点被建立之后执行的所有命令。该保存点保持有效并且可以在以后再次回滚到它（如果需要）。

ROLLBACK TO SAVEPOINT 隐式地销毁在所提及的保存点之后建立的所有保存点。

参数

WORK

TRANSACTION

可选关键词。他们没有任何影响。

savepoint_name

要回滚到的保存点名称。

注解

使用 RELEASE SAVEPOINT 销毁一个保存点而丢弃在它建立之后被执行的命令的效果。

指定一个没有被建立的保存点是一种错误。

相对于保存点，游标有一点非事务的行为。在保存点被回滚时，任何在该保存点内被打开的游标将会被关闭。如果一个先前打开的游标在一个保存点内被 FETCH 命令影响，而该该保存点后来又被回滚，那么该游标将保持 FETCH 使它指向的位置（即，就是说由 FETCH 导致的游标动作不会被回滚）。回滚也不能撤销关闭一个游标。如果一个游标的执行导致事务中止，它会被置于一种不能被执行的状态，这样当事务被用 ROLLBACK TO SAVEPOINT 恢复后，该游标也不再能被使用。

示例

要撤销在 my_savepoint 建立后执行的命令的效果：

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

游标位置不会受保存点回滚的影响：

```
BEGIN;
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;
SAVEPOINT foo;
FETCH 1 FROM foo;
column
----------
       1
ROLLBACK TO SAVEPOINT foo;
FETCH 1 FROM foo;
column
----------
       2
COMMIT;
```

兼容性

SQL 标准指定了关键词 SAVEPOINT 是必需的，但是在 HashData 数据库（和 Oracle）中允许该关键词被省略。 SQL 只允许 WORK 而不是 TRANSACTION 作为一个 ROLLBACK 之后的造生词。另外，SQL 有一个可选的子句 AND [NO] CHAIN，当前在 HashData 数据库中是不支持的。其他方面，该命令符合 SQL 标准。

另见

BEGIN, COMMIT, SAVEPOINT, RELEASE SAVEPOINT, ROLLBACK

上级话题： SQL命令参考

ROLLBACK

中止当前事务。

概要

```
ROLLBACK [WORK | TRANSACTION]
```

描述

ROLLBACK 回滚当前事务并且导致该事务所作的所有更新都被丢弃。

参数

WORK

TRANSACTION

可选关键词。它们没有任何影响。

注解

使用 COMMIT 可成功的结束一个事务。

在一个事务块之外发出 ROLLBACK 会发出一个警告并且不会有任何效果。

示例

丢弃当前事务所做的所有修改：

```
ROLLBACK;
```

兼容性

SQL 标准中只指定了两种形式 ROLLBACK 和 ROLLBACK WORK。其他方面，该命令完全符合 SQL 标准。

另见

BEGIN, COMMIT, SAVEPOINT, ROLLBACK TO SAVEPOINT

上级话题： SQL命令参考

SAVEPOINT

在当前事务中定义一个新的保存点。

概要

```
SAVEPOINT savepoint_name
```

描述

SAVEPOINT 在当前事务中建立一个新保存点。

保存点是事务内的一种特殊标记，它允许所有在它被建立之后执行的命令被回滚，把该事务的状态恢复到它处于保存点时的样子。

参数

savepoint_name

新保存点的名字。

注解

使用 ROLLBACK TO SAVEPOINT 回滚到一个保存点。 使用 RELEASE SAVEPOINT 销毁一个保存点，但保持在它被建立之后执行的命令的效果。

保存点只能在一个事务块内建立。可以在一个事务内定义多个保存点。

示例

要建立一个保存点并且后来撤销在它建立之后执行的所有命令的效果：

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

上面的事务将插入值 1 和 3，但不会插入 2。

要建立并且稍后销毁一个保存点：

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上面的事务将插入 3 和 4。

兼容性

当建立另一个同名保存点时，SQL 要求之前的那个保存点自动被销毁。在 HashData 数据库中，旧的保存点会被保留，不过在进行回滚或释放时只能使用最近的那一个（释放较新的保存点将会导致较旧的保存点再次变得可以被 ROLLBACK TO SAVEPOINT 和 RELEASE SAVEPOINT 访问）。在其他方面，SAVEPOINT 完全符合 SQL。

另见

BEGIN, COMMIT, ROLLBACK, RELEASE SAVEPOINT, ROLLBACK TO SAVEPOINT

上级话题： SQL命令参考

SELECT INTO

从一个查询的结果定义一个新表。

概要

```
[ WITH with_query [, ...] ]
SELECT [ALL | DISTINCT [ON ( expression [, ...] )]]
    * | expression [AS output_name] [, ...]
    INTO [TEMPORARY | TEMP] [TABLE] new_table
    [FROM from_item [, ...]]
    [WHERE condition]
    [GROUP BY expression [, ...]]
    [HAVING condition [, ...]]
    [{UNION | INTERSECT | EXCEPT} [ALL] select]
    [ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}] [, ...]]
    [LIMIT {count | ALL}]
    [OFFSET start]
    [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT]
    [...]]
```

描述

SELECT INTO 创建一个新表并且用一个查询计算得到的数据填充它。这些数据不会像普通的 SELECT 那样被返回给客户端。新表的列具有和 SELECT 的输出列相关的名称和数据类型。

参数

SELECT INTO 主要的参数同 SELECT 是一样的。

TEMPORARY

TEMP

如果被指定，该表被创建为一个临时表。

new_table

要创建的表的名字（可以是方案限定的）。

示例

创建一个新表 films_recent 由表 films 的最近项构成：

```
SELECT * INTO films_recent FROM films WHERE date_prod >=
'2016-01-01';
```

兼容性

SQL 标准使用 SELECT INTO 表示把值选择到一个宿主程序的标量变量中，而不是创建一个新表。 HashData 数据库使用 SELECT INTO 来表示创建是有历史原因的。最好在新的应用中使用 CREATE TABLE AS 来实现该目的。

另见

SELECT, CREATE TABLE AS

上级话题： SQL命令参考

SELECT

从一个表或视图检索行。

概要

```
[ WITH with_query [, ...] ]
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] [NULLS {FIRST | LAST}] [, ...]]
  [LIMIT {count | ALL}]
  [OFFSET start]
  [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT] [...]]
```

其中 with_query 是：

```
  with_query_name [( column_name [, ...] )] AS ( select )
```

其中 grouping_element可以是下列之一：

```
  ()
  expression
  ROLLUP (expression [,...])
  CUBE (expression [,...])
  GROUPING SETS ((grouping_element [, ...]))
```

其中 window_specification 可以是：

```
  [window_name]
  [PARTITION BY expression [, ...]]
  [ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}] [, ...]
     [{RANGE | ROWS}
          { UNBOUNDED PRECEDING
          | expression PRECEDING
          | CURRENT ROW
          | BETWEEN window_frame_bound AND window_frame_bound }]]
                    其中window_frame_bound可以是下列之一：
                        UNBOUNDED PRECEDING
                        expression PRECEDING
                        CURRENT ROW
                        expression FOLLOWING
                        UNBOUNDED FOLLOWING
```

其中 from_item 可以是下列之一：

```
[ONLY] table_name [[AS] alias [( column_alias [, ...] )]]
(select) [AS] alias [( column_alias [, ...] )] with_query_name [ [AS] alias [( column_alias [, ...] )]]
function_name ( [argument [, ...]] ) [AS] alias
             [( column_alias [, ...]
                | column_definition [, ...] )]
function_name ( [argument [, ...]] ) AS
              ( column_definition [, ...] )
from_item [NATURAL] join_type from_item
          [ON join_condition | USING ( join_column [, ...] )]
```

描述

SELECT 从零或更多表中检索行。 SELECT 的通常处理如下：

1. WITH 子句中的所有查询都会被计算。这些查询实际充当了在 FROM 列表中可以引用的临时表。
2. FROM 列表中的所有元素都会被计算（FROM 中的每一个元素都是一个真实表或者虚拟表）。如果在 FROM 列表中指定了多于一个元素，它们会被交叉连接在一起。
3. 如果指定了 WHERE 子句，所有不满足该条件的行都会被从输出中消除。
4. 如果指定了 GROUP BY子句，输出会被组合成由在一个或者多个值上匹配的行构成的分组，并且在其上计算聚集函数的结果。如果出现了 HAVING 子句，它会消除不满足给定条件的分组。
5. 如果指定了窗口表达式（可选的 WINDOW 子句），输出会根据位置（行）或者基于值（范围）的窗口帧来组织。
6. DISTINCT 从结果中消除重复的行。DISTINCT ON 消除在所有指定表达式上匹配的行。ALL（默认）将返回所有候选行， 包括重复的行。
7. 对于每一个被选中的行，会使用 SELECT 输出表达式计算实际的输出行。
8. 通过使用操作符 UNION、 INTERSECT 和 EXCEPT，多于一个 SELECT 语句的输出可以被整合形成 一个结果集。UNION 操作符返回位于一个或者两个结果集中的全部行。INTERSECT 操作符返回同时位于两个结果集中的所有行。EXCEPT 操作符返回位于第一个结果集但不在第二个结果集中的行。在所有三种情况下，重复行都会被消除（除非指定 ALL）。
9. 如果指定了 ORDER BY 子句，被返回的行会以指定的顺序排序。如果没有给定 ORDER BY，系统会以能最快产生行的顺序返回它们。
10. 如果指定了 LIMIT 或者 OFFSET 子句，SELECT 语句只返回结果行的一个子集。
11. 如果指定了 FOR UPDATE 或者 FOR SHARE，SELECT 语句会把被选中的行锁定而不让并发更新访问它们。

用户必须拥有在要读取值的表上的 SELECT 特权。FOR UPDATE、 FOR SHARE 还要求 UPDATE 特权。

参数

WITH 子句

WITH 子句允许用户指定一个或者多个在主查询中可以其名称引用的子查询。在主查询期间子查询实际扮演了临时表或者视图的角色。每一个子查询都可以是一个 SELECT 或者 VALUES 语句。

对于每一个 WITH 子句，都必须指定一个名称（无需方案限定）。可选地，可以指定一个列名列表。如果省略该列表，会从该子查询中推导列名。主查询和 WITH 查询全部（理论上）都在同一时间被执行。

The SELECT List

SELECT 列表（位于关键词 SELECT 和 FROM 之间）指定构成 SELECT 语句输出行的表达式。这些表达式可以（并且通常确实会）引用 FROM 子句中计算得到的列。

另一个名字可以被指定用于一个输出列的名称，使用 [AS] output_name 。该名称最基本是出于显示目的标记列。一个输出列的名称可以被用来在 ORDER BY 以及 ORDER BY 子句中引用该列的值，但是不能用于 WHERE 和 HAVING 子句（在其中必须写出表达式）。在大多数场景下，AS 关键词是可选的（例如当为一个列名、常量、函数调用、简单一元操作表达式声明一个别名）。为了避免声明的别名与关键词冲突，输出名一定要使用双引号包含起来。推荐总是写上 AS 或者用双引号引用输出名称。

一个 SELECT 列表中的表达式可以为常量值、一个列引用、一个操作符调用、一个函数调用、一个窗口表达式、一个标量子查询（scalar subquery）等等。一些构造可以分类为一个表达式但是不符合通用的语法规则。这些通常有一个操作符或者函数的语义。关于 SQL 值表达式以及函数调用的信息，见在 HashData 数据库管理员指南中的“查询数据”。

可以在输出列表中写 来取代表达式，它是被选中行的所有列的一种简写方式。还可以写*table_name.*，它是只来自那个表的所有列的简写形式。

FROM 子句

FROM 子句为 SELECT 指定一个或者更多源表。如果指定了多个源表，结果将是所有源表的笛卡尔积（交叉连接）。但是通常会增加限定条件，来把返回的行限制为该笛卡尔积的一个小子集。FROM 子句可以包含下列元素：

table_name

一个现有表或视图的名称（可以是方案限定的）。如果在表名前指定了 ONLY，则只会扫描该表。如果没有指定 ONLY，该表及其所有后代表（如果有）都会被扫描。

alias

一个包含别名的 FROM 项的替代名称。别名被用于让书写简洁或者消除自连接中的混淆（其中同一个表会被扫描多次）。当提供一个别名时，表或者函数的实际名称会被隐藏。例如，给定 FROM foo AS f，SELECT 的剩余部分就必须以 f 而不是 foo 来引用这个 FROM 项。如果写了一个别名，还可以写一个列别名列表来为该表的一个或者多个列提供替代名称。

select

一个子查询可以出现在 FROM 子句中。这就好像把它的输出创建为一个存在于该 SELECT 命令期间的临时表。注意子查询必须用圆括号包围，并且必须为它提供一个别名。也可以在这里使用一个 VALUES 命令。见 兼容性 部分的 “非标准子句” 关于在 HashData 数据库中使用相关子查询的局限性。

with_query_name

在 FROM 子句中，可以通过写一个 WITH 查询的名称来引用 WITH 查询，就好像该查询的名称是一个表名。WITH 查询的名称不能包含一个方案限定词。可以像表一样，以同样的方式提供一个别名。

function_name

函数调用可以出现在 FROM 子句中（对于返回结果集合的函数特别有用，但是可以使用任何函数）。这就好像把该函数的输出创建为一个存在于该 SELECT 命令期间的临时表。可以用和表一样的方式提供一个别名。如果写了一个别名，还可以写一个列别名列表来为该函数的组合返回类型的一个或者多个属性提供替代名。如果函数被定义为返回 record 数据类型，那么必须出现一个别名或者关键词 AS，后面跟上形为 (column_name data_type [, ...]) 的列定义列表。列定义列表必须匹配该函数返回的列的实际数量和类型。

join_type

下列之一：

- **[INNER] JOIN**
- **LEFT [OUTER] JOIN**
- **RIGHT [OUTER] JOIN**
- **FULL [OUTER] JOIN**
- **CROSS JOIN**

对于 INNER 和 OUTER 连接类型，必须指定一个连接条件，即 NATURALON join_condition 或者 USING (join_column [, ...]) 之一（只能有一种）。其含义见下文。对于 CROSS JOIN，上述子句不能出现。

一个 JOIN 子句联合两个 FROM 项（ 为了方便我们称之为"表"，尽管实际上它们可以是任何类型的 FROM 项）。如有必要可以使用圆括号确定嵌套的顺序。在没有圆括号时，JOIN 会从左至右嵌套。在任何情况下， JOIN 的联合比分隔 JOIN -列表项的逗号更强。

CROSS JOIN 和 INNER JOIN 会产生简单的笛卡尔积，也就是与在 FROM 的顶层列出两个表得到的结果相同，但是要用连接条件（如果有）约束该结果。CROSS JOIN 与 INNER JOIN ON(TRUE) 等效，也就是说条件不会移除任何行。这些连接类型只是一种记号上的方便，因为没有什么是用户用纯粹的 FROM 和 WHERE 能做而它们不能做的。

LEFT OUTER JOIN 返回被限制过的笛卡尔积中的所有行（即所有通过了其连接条件的组合行），外加左手表中没有相应的通过了连接条件的右手行的每一行的拷贝。通过在右手列中插入空值，这种左手行会被扩展为连接表的完整行。注意在决定哪些行匹配时，只考虑 JOIN 子句自身的条件。之后才应用外条件。

相反，RIGHT OUTER JOIN 返回所有连接行，外加每一个没有匹配上的右手行（在左端用空值扩展）。这只是为了记号上的方便，因为用户可以通过交换左右表把它转换成一个 LEFT OUTER JOIN。

FULL OUTER JOIN 返回所有连接行，外加每一个没有匹配上的左手行（在右端用空值扩展），再外加每一个没有匹配上的右手行（在左端用空值扩展）。

ON join_condition

join_condition 是一个会得到 boolean 类型值的表达式（类似于一个 WHERE 子句），它说明一次连接中哪些行被认为相匹配。

USING (join_column [, ...])

形式 USING (a, b, ...) 的子句是 ON left_table.a = right_table.a AND left_table.b = right_table.b ... 的简写。还有，USING 表示每一对相等列中只有一个会被包括在连接输出中。

NATURAL

NATURAL 是列出在两个表中所有具有 相同名称的列的 USING 的简写。

WHERE子句

可选的 WHERE 子句的形式：

```
WHERE condition
```

其中 condition 是任一计算得到 boolean 类型结果的表达式。任何不满足这个条件的行都会从输出中被消除。如果用一行的实际值替换其中的变量引用后，该表达式返回真，则该行符合条件。

GROUP BY 子句

可选的 GROUP BY 子句的形式：

```
GROUP BY grouping_element [, ...]
```

grouping_element 可以为下列之一：

```
()
expression
ROLLUP (expression [,...])
CUBE (expression [,...])
GROUPING SETS ((grouping_element [, ...]))
```

GROUP BY 将会把所有被选择的行中共享相同分组表达式值的那些行压缩成一个行。一个被用在 expression 可以是输入列名、输出列（SELECT 列表项）的名称或序号或者由输入列值构成的任意表达式。在出现歧义时，GROUP BY 名称将被解释为输入列名而不是输出列名。

聚集函数（如果使用）会在组成每一个分组的所有行上进行计算，从而为每一个分组产生一个单独的值（如果有聚集函数但是没有 GROUP BY 子句，则查询会被当成是由所有选中行构成的一个单一分组）。当存在 GROUP BY 子句或者任何聚集函数时，SELECT 列表表达式不能引用非分组列（除非它出现在聚集函数中或者它函数依赖于分组列），因为这样做会导致返回非分组列的值时会有多种可能的值。

HashData 数据库有下面增加的 OLAG 分组扩展（通常被称为 supergroups）：

ROLLUP

一个 ROLLUP 分组是 GROUP BY 分组的扩展。该分组创建一个从最细的级别到一个粗粒度级别上卷聚集操作，后面紧跟着一系列的分组列（或者表达式）。 ROLLUP 接受一个有序的分组列，计算在 GROUP BY 中指定的标准聚集值，然后从右到左进一步创建高层次的部分和。最后创建了累积和。一个 ROLLUP 分组能够看做一系列的分组集。例如：

```
GROUP BY ROLLUP (a,b,c)
```

等价于：

```
GROUP BY GROUPING SETS( (a,b,c), (a,b), (a), () )
```

注意，一个有 n 个元素的 ROLLUP 翻译为 n+1 分组集。同时，在 ROLLUP 中指定分组表达式的顺序很重要。

CUBE

CUBE 分组是 GROUP BY 子句的一个扩展。它能够为给定的分组列（或者表达式）所有可能的组合创建部分和。在多维分析上，CUBE 为指定维度的、可计算的数据立方体计算出所有的部分和。例如：

```
GROUP BY CUBE (a,b,c)
```

等价于：

```
GROUP BY GROUPING SETS( (a,b,c), (a,b), (a,c), (b,c), (a),
(b), (c), () )
```

注意，一个有 n 个元素的 CUBE 翻译为 2n 个分组集。在任何需要交叉表报表的场景下，考虑使用 CUBE。CUBE 典型的适用于查询中从多个维度中使用列而不是一个列代表不同层次上使用列。例如，一个常用的交叉列表可能需要分类汇总为月，所有组合的状态，和产品。

GROUPING SETS

GROUP BY 子句中，可以在想要使用 GROUPING SETS 表达式的地方选择性指定分组集合。这允许精确的规范在多个维度而不用计算整个 ROLLUP 或 CUBE。例如：

```
GROUP BY GROUPING SETS( (a,c), (a,b) )
```

如果使用分组扩展子句 ROLLUP、CUBE 或者 GROUPING SETS，有两个挑战将会出现。首先，如何决定哪些结果行需要是部分和，以及给定的部分和的准确聚集层次。或者用户如何区别包含 NULL 或者由 ROLLUP 、CUBE 产生 "NULL" 值的结果行。第二，当在 GROUP BY 子句中指定了重复分组，如何决定哪些结果行是冗余的呢？有两个额外的分组函数可以使用在SELECT列表中帮助：

- **grouping(column [, ...])** — grouping 函数能够被应用到一个或者更多的分组属性上来从正规的分组行区分开超级聚集行（这对将一个超级聚集行中表示所有值集合的 “NULL” 与普通行中的 NULL 区分开很有用）。该函数中的每个参数产生一个位 - 要么为 1 或者 `0，其中 1 意味着结果行是超级聚集的，0 意味着结果行来自于普通聚集。grouping 函数通过将这些位当做一个二进制数然后将它们转换为一个十进制的书，返回一个整数。
- **group_id()** — 对于包含有冗余分组集，group_id 函数被用来鉴别在输出中的冗余行。所有 unique 分组集输出行将有一个为 0 的 group_id 值。对于每个检测到的冗余的分组集，group_id 函数 分配一个大于 0 的 group_id。在一个特定的冗余分组集中的所有行被有相同的 group_id 值。

WINDOW 子句

WINDOW 子句是用来定义一个能够被用在一个窗口函数（例如，rank 或者 avg）的 OVER() 表达式中的窗口。 例如：

```
SELECT vendor, rank() OVER (mywindow) FROM sale
GROUP BY vendor
WINDOW mywindow AS (ORDER BY sum(prc*qty));
```

一个 WINDOW 子句有一般的形式：

```
WINDOW window_name AS (window_specification)
```

其中 window_specification 可以为：

```
[window_name]
[PARTITION BY expression [, ...]]
[ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}] [, ...]
    [{RANGE | ROWS}
      { UNBOUNDED PRECEDING
      | expression PRECEDING
      | CURRENT ROW
      | BETWEEN window_frame_bound AND window_frame_bound }]]
            其中 window_frame_bound可以为下列之一：
              UNBOUNDED PRECEDING
              expression PRECEDING
              CURRENT ROW
              expression FOLLOWING
              UNBOUNDED FOLLOWING
```

window_name

给窗口说明一个名字。

```
PARTITION BY
```

The PARTITION BY 子句基于指定表表达式的唯一值将结果集组织为逻辑组。当同窗口函数使用，函数将被单独地应用到每个分片。例如，如果用户在一个列名后紧跟一个 PARTITION BY，结果集将会通过列的不同值进行分割。如果忽略，整个结果集被看做一个分片。

ORDER BY

The ORDER BY 子句定义如何对结果集的每个分片进行排序。如果忽略，返回的行按照效率高的方式返回，可能每次有所不同。 注意： 缺乏连贯顺序的数据类型的列，例如，time，在一个窗口说明的 ORDER BY 子句不适合作为排序字段。时间，有或者没有时区，缺少一个明确的顺序，由于加法和减法没有预期的效果。例如，下面的一般不为真： `x::time < x::time + '2 hour'::interval`

ROWS | RANGE

通过使用 ROWS 或者 RANGE 子句来表示窗口的界（bounds）。窗口的界可能为一个分区的一个，多个行或者所有行。可以根据一系列的值距离当前行的值偏移量来表达(RANGE)或者依据距离当前行的偏移行数来表达(ROWS)。当使用 RANGE 子句，一定要使用一个 ORDER BY 子句。这是因为执行产生窗口的计算需要值是排好序的。另外，ORDER BY 子句不能包含多于一个的表达式，同时表达式的必须为一个日期或者一个数值值。当使用 ROWS 或者 RANGE 子句，如果用户只指定了一个开始行，那么当前行会作为窗口的最后一行。

PRECEDING — PRECEDING 子句定义以当前行为参考点窗口的第一行的位置。开始行依据距离当前行的前驱行数来表达。例如，在 ROWS 框架中，5 PRECEDING 设置窗口开始于当前的第五个前驱行。在RANGE框架中，设置窗口开始于按照给定顺序的当前行的第五个前驱行。如果按照时间升序指定顺序，那么第一行为当前行五天前的行。UNBOUNDED PRECEDING 设置窗口中的第一行为分区中的第一行。

BETWEEN — BETWEEN 子句使用当前行作为参考点，定义了窗口的第一行和最后一行。第一行和最后一行依据当前行的前驱和后继的行的数目表达。例如，BETWEEN 3 PRECEDING AND 5 FOLLOWING 设置窗口开始于当前行前驱的第三个行，结束于当前行后面的第五行。使用 BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING 设置窗口的第一行和最后一行为该分区中的一行和最后一行。这等效于在没有ROW 或者RANGE子句指定是的默认行为。

FOLLOWING — FOLLOWING 子句定义了使用当前行作为参考点的窗口的最后一行。最后一行的表示依据跟在当前行后面的行的行号。例如，在 ROWS 框架中, 5 FOLLOWING 设置窗口的结束为止在当前行后的第五个行。在 RANGE 框架中，它设置窗口的结束为按在给定顺序跟在当前行后面的 5 行。如果指定属性为按照日期的升序，那么最一行为当前行之后 5 天的行。使用 UNBOUNDED FOLLOWING 设置窗口中的最后一行为分区中的最后一行。

如果没有指定一个 ROW 或者 RANGE 子句，窗口的界会从分区的第一行开始（UNBOUNDED PRECEDING） 同时以当前行为结束（CURRENT ROW），如果使用了 ORDER BY 。如果 ORDER BY 没有指定，那么窗口开始于分区（UNBOUNDED PRECEDING）的第一行同时结束语分区（UNBOUNDED FOLLOWING）的最后一行。

HAVING子句

可选 HAVING 子句的形式：

```
HAVING condition
```

其中 condition 与 WHERE 子句中指定的条件相同。HAVING 消除不满足该条件的分组行。 HAVING 与WHERE不同：WHERE 会在应用 GROUP BY 之前过滤个体行，而 HAVING 过滤由 GROUP BY 创建的分组行。condition 中引用的每一个列必须无歧义地引用一个分组列（除非该引用出现在一个聚集函数）。

即使没有 GROUP BY 子句，HAVING 的存在也会把一个查询转变成一个分组查询。这和查询中包含聚集函数但没有 GROUP BY 子句时的情况相同。所有被选择的行都被认为是一个单一分组，并且 SELECT 列表和 HAVING 子句只能引用聚集函数中的表列。如果该 HAVING 条件为真，这样一个查询将会发出一个单一行； 否则不返回行。

UNION子句

UNION 子句具有下面的形式：

```
select_statement UNION [ALL] select_statement
```

select_statement 是任何没有 ORDER BY、LIMIT、FOR UPDATE、FOR SHARE 和 FOR KEY SHARE 子句的 SELECT 语句（如果子表达式被包围在圆括号内，ORDER BY 和 LIMIT 可以被附着到其上。如果没有圆括号，这些子句将被应用到 UNION 的结果而不是右手边的表达式上）。

UNION 操作符计算所涉及的 SELECT 语句所返回的行的并集。如果一行至少出现在两个结果集中的一个内，它就会在并集中。作为 UNION 两个操作数的 SELECT 语句必须产生相同数量的列并且对应位置上的列必须具有兼容的数据类型。

UNION 的结果不会包含重复行，除非指定了 ALL 选项。ALL 会阻止消除重复（因此，UNION ALL 通常显著地快于 UNION，尽量使用 ALL）。

除非用圆括号指定计算顺序，同一个 SELECT 语句中的多个 UNION 操作符会从左至右计算。

当前，FOR UPDATE 和 FOR SHARE 不能用于 UNION 结果或者 UNION 的任何输入。

INTERSECT 子句

INTERSECT 子句具有下面的形式：

```
select_statement INTERSECT [ALL] select_statement
```

select_statement 是任何没有 ORDER BY、LIMIT、FOR UPDATE 以及 FOR SHARE 子句的 SELECT 语句。

INTERSECT 操作符计算所涉及的 SELECT 语句返回的行的交集。如果一行同时出现在两个结果集中，它就在交集中。

INTERSECT 的结果不会包含重复行，除非指定了 ALL 选项。如果有 ALL，一个在左表中有 m 次重复并且在右表中有 n 次重复的行将会在结果中出现 min(m, n) 次。

除非用圆括号指定计算顺序， 同一个 SELECT 语句中的多个 INTERSECT 操作符会从左至右计算。 INTERSECT 的优先级比 UNION 更高。也就是说，A UNION B INTERSECT C 被被读成 A UNION (B INTERSECT C)。

当前， FOR UPDATE 和 FOR SHARE 不能用于 INTERSECT 结果或者 INTERSECT 的任何输入。

EXCEPT 子句

EXCEPT 子句具有如下形式：

```
select_statement EXCEPT [ALL] select_statement
```

select_statement 是任何没有 ORDER BY、LIMIT、FOR UPDATE 以及 FOR SHARE 子句的 SELECT 语句。

EXCEPT 操作符计算位于左 SELECT 语句的结果中但不在右 SELECT 语句结果中的行集合。

EXCEPT 的结果不会包含重复行，除非指定了 ALL 选项。如果有 ALL，一个在左表中有 m 次重复并且在右表中有 n 次重复的行将会在结果集中出现 max(m-n,0) 次。

除非用圆括号指定计算顺序， 同一个 SELECT 语句中的多个 EXCEPT 操作符会从左至右计算。EXCEPT 的优先级与

UNION 相同。

当前，FOR UPDATE 和 FOR SHARE 不能用于 EXCEPT 结果或者 EXCEPT 的任何输入。

ORDER BY子句

ORDER BY 子句可选的形式如下：

```
ORDER BY expression [ASC | DESC | USING operator] [NULLS { FIRST | LAST}] [, ...]
```

每一个 expression 可以是输出列（SELECT 列表项）的名称或者序号，它也可以是由输入列值构成的任意表达式。

ORDER BY 子句导致结果行被按照指定的表达式排序。如果两行按照最左边的表达式是相等的，则会根据下一个表达式比较它们，依次类推。如果按照所有指定的表达式它们都是相等的，则它们被返回的顺序取决于实现。

序号指的是输出列的顺序（从左至右）位置。这种特性可以为不具有唯一名称的列定义一个顺序。这不是绝对必要的，因为总是可以使用 AS 子句为输出列赋予一个名称。

也可以在 ORDER BY 子句中使用任意表达式，包括没有出现在 SELECT 输出列表中的列。因此，下面的语句是合法的：

```
SELECT name FROM distributors ORDER BY code;
```

这种特性的一个限制是一个应用在 UNION、INTERSECT 或者 EXCEPT 子句结果上的 ORDER BY 只能指定输出列名称或序号，但不能指定表达式。

如果一个 ORDER BY 表达式是一个既匹配输出列名称又匹配输入列名称的简单名称，ORDER BY 将把它解读成输出列名称。这与在同样情况下 GROUP BY 会做出的选择相反。这种不一致是为了与 SQL 标准兼容。

可以为 ORDER BY 子句中的任何表达式之后增加关键词 ASC（上升）DESC（下降）。如果没有指定，ASC被假定为默认值。或者，可以在 USING 子句中指定一个特定的排序操作符名称。ASC 通常等价于 USING < 而 DESC 通常等价于 USING >（但是一种用户定义数据类型的创建者可以准确地定义默认排序顺序是什么，并且它可能会对应于其他名称的操作符）。

如果指定 NULLS LAST，空值会排在非空值之后；如果指定 NULLS FIRST，空值会排在非空值之前。如果都没有指定，在指定或者隐含 ASC 时的默认行为是 NULLS LAST， 而指定或者隐含 DESC 时的默认行为是 NULLS FIRST（因此，默认行为是空值大于非空值）。当指定 USING 时，默认的空值顺序取决于该操作符是否为小于或者大于操作符。

注意顺序选项只应用到它们所跟随的表达式上。例如 ORDER BY x, y DESC 和 ORDER BY x DESC, y DESC 是不同的。

字符串数据被根据区域相关的排序规则顺序排序，该顺序在 HashData 数据库系统被初始化时建立。

DISTINCT 子句

如果指定了 DISTINCT 子句，所有重复的行会被从结果集中移除（为每一组重复的行保留一行）。ALL 则 指定相反的行为：所有行都会被保留，这也是默认情况。

DISTINCT ON (expression [, ...]) 只保留在给定表达式上计算相等的行集合中的第一行。DISTINCT ON 表达式使用和 ORDER BY 相同的规则（见上文）解释。注意，除非用 ORDER BY 来确保所期望的行出现在第一位，每一个集合的 "第一行" 是不可预测的。例如

```
SELECT DISTINCT ON (location) location, time, report FROM
weather_reports ORDER BY location, time DESC;
```

为每个地点检索最近的天气报告。但是如果我们不使用 ORDER BY 来强制对每个地点的时间值进行降序排序，我们为每个地点得到的报告的时间可能是无法预测的。

DISTINCT ON 表达式必须匹配最左边的 ORDER BY 表达式。ORDER BY 子句通常将包含额外的表达式，这些额外的表达式用于决定在每一个 DISTINCT ON 分组内行的优先级。

当 HashData 数据库处理包含有 DISTINCT 子句的查询时，查询将会转换为 GROUP BY 查询。在很多场景中，变换能够提供显著的性能提升。然而，当 distinct 值的数量与总的行数相近时，该转换可能会导致多个层次的分组计划的产生。在这种情况下，由于引入的低聚集度开销，预期性能会降低。

LIMIT 子句

LIMIT 子句两个独立的子句构成：

```
LIMIT {count | ALL}
OFFSET start
```

count 指定要返回的最大行数，而 start 指定在返回行之前要跳过的行数。在两者都被指定时，在开始计算要返回的 count 行之前会跳过 start 行。

在使用 LIMIT 时，用一个 ORDER BY 子句把结果行约束到一个唯一顺序是个好办法。否则用户讲得到该查询结果行的一个不可预测的子集 — 用户可能要求从第 10 到第 20 行，但是在什么顺序下的第 10 到第 20 呢？除非指定 ORDER BY，用户是不知道顺序的。

查询规划器在生成一个查询计划时会考虑 LIMIT，因此根据用户使用的 LIMIT 和 OFFSET，用户很可能 得到不同的计划（得到不同的行序）。所以，使用不同的 LIMIT/OFFSET 值来选择一个查询结果的不同子集将会给出不一致的结果，除非用户用 ORDER BY 强制一种可预测的结果顺序。这不是一个缺陷，它是 SQL 不承诺以任何特定顺序（除非使用 ORDER BY 来约束顺序）给出一个查询结果这一事实造成的必然后果。

FOR UPDATE/FOR SHARE子句

FOR UPDATE 子句的形式如下：

```
FOR UPDATE [OF table_name [, ...]] [NOWAIT]
```

非常接近的 FOR SHARE 子句的形式为：

```
FOR SHARE [OF table_name [, ...]] [NOWAIT]
```

FOR UPDATE 导致被 SELECT 语句访问的表被锁定，就好像在做更新一样。这可以防止表在当前事务结束之前被其他事务修改或者删除。也就是说在这个表上尝试 UPDATE、DELETE 或者 SELECT FOR UPDATE 的其他事务都将被阻塞，直至当前事务结束。还有，如果来自于另一个事务的 UPDATE、DELETE 或者 SELECT FOR UPDATE 已经锁住了选择的表，SELECT FOR UPDATE 将会等待该其他事务完成，并且接着锁住并且返回更新过的表。

为了防止该操作等待其他事务提交，可使用 NOWAIT。使用 NOWAIT 时， 如果选中的行不能被立即锁定，该语句会报告错误而不是等待。注意 NOWAIT 只适合行级锁 — 所要求的 ROW SHARE 表级锁仍然会以常规的方式取得。如果想要不等待的表级锁，用户可以先使用带 NOWAIT 的 LOCK（见 LOCK ）。

FOR SHARE 的行为相似，除了需要一个在表上的共享锁而不是排它锁。一个共享锁阻塞其它在表上执行 UPDATE、DELETE 或者 SELECT FOR UPDATE 的事务，但是不禁止它们执行 SELECT FOR SHARE。

如果特定的表在 FOR UPDATE 或者 FOR SHARE 中，那么只有这些表被锁定；任何其它在 SELECT 中使用的表按照通常的方式进行读。一个不带表列表的 FOR UPDATE 或者 FOR SHARE 子句将会影响所有在命令中使用的表。如果 FOR UPDATE 或者 FOR SHARE 应用到了一个视图或者子查询上，那么它将影响所有在视图或者子查询中使用到的表。

FOR UPDATE 或者 FOR SHARE 不能应用到由一个基础查询引用的 with_query 上。如果需要行锁定在 with_query 上，可以在 with_query 中指定 FOR UPDATE 或者 FOR SHARE。

如果需要在不同的表上指定不同的锁行为，多个 FOR UPDATE 和 FOR SHARE 子句是可以写的。如果相同表同时由 FOR UPDATE 和 FOR SHARE 子句中提及（或者是隐式的影响），那么该表将会按照 FOR UPDATE 进行处理。相似的，如果一个表任何一个子句都会对它有影响，那么将会按照 NOWAIT 进行处理。

示例

把表 films 与表 distributors 连接：

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind FROM
distributors d, films f WHERE f.did = d.did
```

要对所有电影的 length 列求和并且用 kind 对结果分组：

```
SELECT kind, sum(length) AS total FROM films GROUP BY kind;
```

要对所有电影的 length 列求和、对结果按照 kind 分组并且显示总长小于 5 小时的分组：

```
SELECT kind, sum(length) AS total FROM films GROUP BY kind
HAVING sum(length) < interval '5 hours';
```

根据 kind 和 distributor 计算所有电影销售的部分和以及总和。

```
SELECT kind, distributor, sum(prc*qty) FROM sales
GROUP BY ROLLUP(kind, distributor)
ORDER BY 1,2,3;
```

基于总的销售对电影的发行商进行排名：

```
SELECT distributor, sum(prc*qty),
       rank() OVER (ORDER BY sum(prc*qty) DESC)
FROM sale
GROUP BY distributor ORDER BY 2 DESC;
```

下面的两个例子都是根据第二列（name）的内容排序结果：

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

接下来的例子展示了如何得到表 distributors 和 actors 的并集，把结果限制为那些在每个表中以字母 W 开始的行。只想要可区分的行，因此省略了关键词 ALL：

```
SELECT distributors.name FROM distributors WHERE
distributors.name LIKE 'W%' UNION SELECT actors.name FROM
actors WHERE actors.name LIKE 'W%';
```

这个例子展示了如何在 FROM 子句中使用函数，分别使用和不使用列定义列表：

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors
AS $$ SELECT * FROM distributors WHERE did = $1; $$ LANGUAGE
SQL;
SELECT * FROM distributors(111);

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS
$$ SELECT * FROM distributors WHERE did = $1; $$ LANGUAGE
SQL;
SELECT * FROM distributors_2(111) AS (dist_id int, dist_name
text);
```

这个例子展示了如何使用简单的 WITH 子句：

```
WITH test AS (
  SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM test
UNION ALL
SELECT * FROM test;
```

这个例子使用 WITH 子句来展示每个产品只在最好销售区域的销售总额。

```
WITH regional_sales AS
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
  ), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales) FROM
       regional_sales)
  )
SELECT region, product, SUM(quantity) AS product_units,
   SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

该示例能够不用 WITH 子句进行重写，但是将需要两层的 子-SELECT 语句。

兼容性

SELECT 语句是兼容 SQL 标准的。 但是也有一些扩展和缺失的特性。

省略的 **FROM** 子句

HashData 数据库允许我们省略 FROM 子句。一种简单的使用是计算简单表达式的结果，例如：

```
SELECT 2+2;
```

某些其他 SQL 数据库需要引入一个假的单行表放在该 SELECT 要选择的表上。

注意，如果没有指定一个 FROM 子句，该查询就不能引用任何数据库表。对于依赖这种行为的应用的兼容性，add_missing_from 配置参数能够可以启动。

AS 关键词

在 SQL 标准中，可选关键词 AS 只是一个噪声，能够直接被省略没有任何的语义影响。 HashData 数据库解析器在重命名输出列时需要该关键词，因为没有它在类型扩展特性上会出现解析歧义。只要新列名是一个合法的列名（就是说与任何保留关键词不同），就可以省略输出列名之前的可选关键词 AS。 PostgreSQL要稍微严格些：只要新列名匹配任何关键词（保留或者非保留）就需要 AS。推荐的习惯是使用 AS 或者带双引号的输出列名来防止与未来增加的关键词可能的冲突。然而，在 FROM 中 AS 是可选的。

GROUP BY 和 **ORDER BY** 可用的名字空间

在 SQL-92 标准中，一个 ORDER BY 子句只能使用输出列名或者序号，而一个 GROUP BY 子句只能使用基于输入列名的表达式。 HashData 数据库扩展了这两种子句以允许它们使用其他的选择（但如果有歧义时还是使用标准的解释）。HashData 数据库也允许两种子句指定任意表达式。注意出现在一个表达式中的名称将总是被当做输入列名而不是输出列名。

SQL:1999 及其后的标准使用了一种略微不同的定义，它并不完全向后兼容 SQL-92。不过，在大部分的情况下，HashData 数据库会以与 SQL:1999 相同的方式解释 ORDER BY 或 GROUP BY 表达式。

非标准子句

DISTINCT ON、LIMIT 以及 OFFSET 子句并没有在 SQL 标准中有定义。

STABLE 以及 **VOLATILE** 函数的限制使用

为了防止数据在在 HashData 数据库的多个 segment 间不同步，任何一个定义为 STABLE 或者 VOLATILE 的函数不能在 Segment 级别执行（如果它包含了 SQL 或者修改了数据库）。见 CREATE FUNCTION 获取更多信息。

另见

EXPLAIN

上级话题： SQL命令参考

SET ROLE

设置当前会话的当前用户标识符。

概要

```
SET [SESSION | LOCAL] ROLE rolename

SET [SESSION | LOCAL] ROLE NONE

RESET ROLE
```

描述

这个命令把当前 SQL 会话的当前用户标识符设置为 rolename。角色名可以写成一个标识符或者一个字符串。在 SET ROLE 后，对 SQL 命令的权限检查时就好像该角色就是原先登录的角色一样。

当前会话用户必须是指定的角色 rolename 的一个成员。如果会话用户是一个超级用户，则可以选择任何角色。

NONE 和 RESET 形式把当前用户标识符重置为当前会话用户标识符。这些形式可以由任何用户执行。

参数

SESSION

指明命令作用于当前会话。这是默认值。

LOCAL

指明命令只作用于当前事务中。在 COMMIT 或者 ROLLBACK 之后，会话级设置继续恢复影响。注意 SET LOCAL 好像没有任何的影响，如果该命令执行在事务外时。

rolename

在会话中用来进行权限检查的有角色名。

NONE

RESET

重置当前角色（用户）标识符成为当前会话角色（用户）标识符（即用来登录的角色）。

注解

使用这个命令可以增加特权或者限制特权。如果会话用户角色具有 INHERITS 属性，则它会自动具有它能 SET ROLE 到的所有角色的全部特权。在这种情况下。SET ROLE 实际会删除所有直接分配给会话用户的特 权以及分配给会话用户作为其成员的其他角色的特权，只留下所提及角色可用的特权。换句话说，如果会话用户没有 NOINHERITS 属性， SET ROLE 会删除直接分配给会话用户的特权而得到所提及角色可用的特权。

特别地，当一个超级用户选择 SET ROLE 到一个非超级用户角色时，它们会丢失其超级用户特权。

SET ROLE 的效果堪比 SET SESSION AUTHORIZATION，但是涉及的特权检查完全不同。 还有，SET SESSION AUTHORIZATION 决定后来的 SET ROLE 命令可以使用哪些角色，不过用 SET ROLE 更改角色并不会改变后续 SET ROLE 能够使用的角色集。

示例

```
SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 peter        | paul
```

兼容性

HashData 数据库允许标识符语法（rolename），而 SQL 标准要求角色名被写成字符串。 SQL 不允许在事务中使用这个命令；但 HashData 数据库并不做此限制。和 RESET 语法一样，SESSION 和 LOCAL 修饰符是 HashData 数据库的扩展。

另见

SET SESSION AUTHORIZATION

上级话题： SQL命令参考

SET SESSION AUTHORIZATION

置当前会话的会话用户标识符和当前用户标识符。

概要

```
SET [SESSION | LOCAL] SESSION AUTHORIZATION rolename

SET [SESSION | LOCAL] SESSION AUTHORIZATION DEFAULT

RESET SESSION AUTHORIZATION
```

描述

这个命令把当前 SQL 会话的会话用户标识符和当前用户标识符设置为 rolename。用户名可以被写成一个标识符或者一个字符串。例如，可以使用这个命令临时成为一个无特权用户并且稍后切换回来成为一个超级用户。

会话用户标识符初始时被设置为客户端提供的（可能已认证的）用户名。当前用户标识符通常等于会话用户标识符，但是可能在 setuid 函数和类似的机制的环境中发生临时更改。也可以使用 SET ROLE 更改它。当前用户标识符与权限检查相关。

会话用户标识符只能在初始会话用户已认证用户具有超级用户特权时被更改。否则，只有该命令指定已认证用户名时才会被接受。

DEFAULT 和 RESET 形式把会话用户标识符和当前用户标识符重置为初始的已认证用户名。这些形式可以由任何用户执行。

参数

SESSION

指明命令作用于当前会话。这是默认值。

LOCAL

指明命令只作用于当前事务中。在 COMMIT 或者 ROLLBACK 之后，会话级设置继续恢复影响。注意 SET LOCAL 好像没有任何的影响，如果该命令执行在事务外时。

rolename

假定的角色名。

NONE

RESET

重置当前角色（用户）标识符成为当前会话角色（用户）标识符（即用来登录的角色）。

示例

```
SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 paul         | paul
```

兼容性

SQL 标准允许一些其他表达式出现在文本 rolename 的位置上，但是实际上这些选项并不重要。 HashData 数据库允许标识符语法（rolename），而 SQL 标准不允许。 SQL 不允许在事务中使用这个命令，而 HashData 数据库并不做此限制。 和 RESET 语法一样，SESSION 和 LOCAL 修饰符是 HashData 数据库的扩展。

另见

SET ROLE

上级话题： SQL命令参考

SET TRANSACTION

设置当前事务的特性。

概要

```
SET TRANSACTION [transaction_mode] [READ ONLY | READ WRITE]

SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode
     [READ ONLY | READ WRITE]
```

其中 transaction_mode 为下列之一：

```
   ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED | READ UNCOMMITTED}
```

描述

SET TRANSACTION 命令设置当前会话的特性。它对任何子序列事务没有影响。

可用的事务特性是事务隔离级别和事务访问模式（读/写或只读）。

一个事务的隔离级别决定当其他事务并行运行时该事务能看见什么数据。

- **READ COMMITTED** — 一个语句只能看到在它开始前提交的行。这是默认值。
- **SERIALIZABLE** — 当前事务的所有语句只能看到这个事务中执行的第一个查询或者数据修改语句之前提交的行。

SQL 标准定义了两种额外的级别，READ UNCOMMITTED 和 REPEATABLE READ。在 HashData 数据库中 READ UNCOMMITTED 被当做为 READ COMMITTED。 REPEATABLE READ 还不支持；如果需要REPEATABLE READ 行为，使用 SERIALIZABLE。

一个事务执行了第一个查询或者数据修改语句（SELECT、INSERT、DELETE、UPDATE、FETCH 或 COPY）之后就无法更改事务隔离级别。

事务的访问模式决定该事务是否为读/写或者只读。读/写是默认值。当一个事务为只读时，如果 SQL 命令 INSERT、UPDATE、DELETE 和 COPY FROM 要写的表不是一个临时表，则它们不被允许。不允许 CREATE、ALTER 和 DROP 命令。不允许 GRANT、REVOKE、TRUNCATE。如果 EXPLAIN ANALYZE 和EXECUTE 要执行的命令是上述命令之一，则它们也不被允许。这是一种高层的只读概念，它不能阻止所有对 磁盘的写入。

参数

SESSION CHARACTERISTICS

为一个会话的子事务序列设置默认的事务特征。

SERIALIZABLE

READ COMMITTED

READ UNCOMMITTED

SQL 标准定义的四个事务隔离级别：READ COMMITTED、READ UNCOMMITTED、 SERIALIZABLE 和REPEATABLE READ。将一个语句只能看到在它开始前已经提交的行作为（samp class="ph codeph">READ COMMITTED）默认行为。在 HashData 数据库中， READ UNCOMMITTED 被看作为 READ COMMITTED。 REPEATABLE READ 是不支持的，使用 SERIALIZABLE 作为替代。 SERIALIZABLE 是最严格的事务隔离级别。该级别模拟了串行化的事务执行，好像事务一个接着一个执行，串行而不是并行的。实际应用中使用该级别要准备由于线性执行的失败而尝试重启事务。

READ WRITE

READ ONLY

决定事务是读/写还是只读。读/写是默认模式。当一个事务是只读的，下面的 SQL 命令： INSERT、UPDATE、DELETE 和 COPY FROM 执行将被禁止，如果他们将要写的表不是临时表。也不允许 CREATE、ALTER 和 DROP 命令执行。不允许执行 GRANT、REVOKE、TRUNCATE。同时 EXPLAIN ANALYZE 和 EXECUTE 要执行的命令是上诉命令之一也会不允许执行。

注解

如果执行 SET TRANSACTION 之前没有 START TRANSACTION 或者 BEGIN，它会发出一个警告并且不会有任何效果。

可以通过在 BEGIN 或者 START TRANSACTION 中指定想要的 transaction_modes 来省掉 SET TRANSACTION。

会话默认的事务模式也可以通过设置配置参数 default_transaction_isolation 和 default_transaction_read_only 来设置。

示例

为当前事务设置一个事务隔离级别：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

兼容性

SQL 标准中定义了这些命令。 SERIALIZABLE 在标准中是默认的隔离级别。在 HashData 数据库中默认的隔离级别是 READ COMMITTED。由于缺少断言锁， SERIALIZABLE 并不是真正的串行化。从本质上讲，一个断言锁系统通过严格限制写的内容来防止幻读，而 HashData 数据库中使用的多版本并发控制模型通过严格显示读的内容来方式幻读。

在 SQL 标准中，可以用这些命令设置一个其他的事务特性：诊断区域的尺寸。这个概念与嵌入式 SQL 有关，并且因此没有在 HashData 数据库服务器中实现。

SQL 标准要求连续的 transaction_modes 之间有逗号， 但是出于历史原因 HashData 数据库允许省略逗号。

另见

BEGIN, LOCK

上级话题： SQL命令参考

SET

更改一个 HashData 数据库配置参数的值。

概要

```
SET [SESSION | LOCAL] configuration_parameter {TO | =} value |
    'value' | DEFAULT}

SET [SESSION | LOCAL] TIME ZONE {timezone | LOCAL | DEFAULT}
```

描述

SET 命令更改运行时配置参数。任何被分类到 session 的参数可以用 SET 即时更改。SET 只影响当前会话所使用的值。

如果在一个事务内发出 SET（或者等效的 SET SESSION）而该事务后来中止，在该事务被回滚时 SET 命令的效果会消失。一旦所在的事务被提交，这些效果将会持续到会话结束（除非被另一个 SET 所覆盖）。

SET LOCAL 的效果只持续到当前事务结束，不管事务是否被提交。一种特殊情况是在一个事务内 SET 后面跟着 SET LOCAL：SET LOCAL 值将会在该事务结束前一直可见，但是之后（如果该事务被提交）SET 值将会生效。

如果在一个函数内使用 SET LOCAL 并且该函数还有对同一变量的 SET 选项（见 CREATE FUNCTION ），在函数退出时 SET LOCAL 命令的效果会消失。也就是说，该函数被调用时的值会被恢复。这允许用 SET LOCAL 在函数内动态地或者重复地更改一个参数，同时仍然能便利地使用 SET 选项来保存以及恢复调用者的值。不过，一个常规的 SET 命令会覆盖它所在的任何函数的 SET 选项，除非回滚，它的效果将一直保持。

如果在一个事务中，使用 DECLARE 创建了一个游标，那么就不能在该事务中使用 SET 命令，直到使用 CLOSE 命令关闭游标。

参数

SESSION

指定该命令对当前会话有效（这是默认值）。

LOCAL

指定该命令只对当前事务有效。在 COMMIT 或者 ROLLBACK 之后，会话级别的设置会再次生效。注意 SET LOCAL 出现在事务块外部执行，不会有效果。

configuration_parameter

一个可设置运行时参数的名称。只有被分类为 session 类别的参数能够通过 SET 命令就行修改。

value

参数的新值。根据特定的参数，值可以被指定为字符串常量、标识符、数字或者以上构成的逗号分隔列表。写 DEFAULT 可以指定把该参数重置成它的默认值。如果指定内存大小伙子时间单位，则值包含在单引号中。

TIME ZONE

SET TIME ZONE value 是 SET timezone TO value 的一个别名。语法 SET TIME ZONE 允许用于时区指定的特殊语法。这里是合法值的例子：

```
'PST8PDT'

'Europe/Rome'

-7 (UTC 以西 7 小时的时区)

INTERVAL '-08:00' HOUR TO MINUTE (UTC 以西 8 小时的时区)。

LOCAL

DEFAULT
```

把时区设置为用户的本地时区（也就是说服务器的 timezone 默认值）。 见 PostgreSQL文档中时区部分 获取更多关于 HashData 数据库中关于时区的信息。

示例

设置模式搜索路径：

```
SET search_path TO my_schema, public;
```

增加每个查询的段主机内存到 200MB：

```
SET statement_mem TO '200MB';
```

把日期风格设置为传统 POSTGRES 的 "日在月之前" 的输入习惯：

```
SET datestyle TO postgres, dmy;
```

设置时区为加州伯克利：

```
SET TIME ZONE 'PST8PDT';
```

设置时区为意大利：

```
SET TIME ZONE 'Europe/Rome';
```

兼容性

SET TIME ZONE 扩展了 SQL 标准定义的语法。标准只允许数字的时区偏移量而 HashData 数据库允许更灵活的时区说明。所有其他 SET 特性都是 HashData 数据库的扩展。

另见

RESET, SHOW

上级话题： SQL命令参考

SHOW

显示一个运行时参数的值。

概要

```
SHOW configuration_parameter

SHOW ALL
```

描述

SHOW 将显示当前 HashData 数据库系统配置参数的当前设置。 这些变量可以使用 SET 语句、编辑 HashData 数据库 Master 上的 postgresql.conf 配置参数。注意 ：一些能够使用 SHOW 查看的参数是只读的，它们的值只能被查看，不能被修改。见 HashData 数据库参考指南了解细节信息。

参数

configuration_parameter

系统配置参数的名称。

ALL

所有配置参数的当前值。

示例

显示参数 search_path 的当前设置：

```
SHOW search_path;
```

显示所有配置参数的当前的值：

```
SHOW ALL;
```

兼容性

SHOW 是 HashData 数据库的扩展。

另见

SET, RESET

上级话题： SQL命令参考

START TRANSACTION

开始一个事务块。

概要

```
START TRANSACTION [SERIALIZABLE | READ COMMITTED | READ UNCOMMITTED]
                  [READ WRITE | READ ONLY]
```

描述

START TRANSACTION 开始一个新的事务块。如果指定了隔离级别、读写模式，新的事务将会具有这些特性，就像执行了 SET TRANSACTION 一样。这和 BEGIN 命令一样。

参数

SERIALIZABLE

READ COMMITTED

READ UNCOMMITTED

SQL 标准定义四个事务隔离级别： READ COMMITTED、READ UNCOMMITTED、SERIALIZABLE 和REPEATABLE READ。默认行为：一个语句只能看到在它开始前已经提交的行（READ COMMITTED）。在 HashData 数据库中将 READ UNCOMMITTED 和 READ COMMITTED 一样。REPEATABLE READ 目前还不支持；如果需要该隔离级别则使用 SERIALIZABLE。在 SERIALIZABLE 隔离模式下，当前事务内的所有语句只能看到在事务中第一条语句执行前已经提交的行，是一种严格的事务隔离。该事务级别模拟串行事务执行，好像事务一个接一个的执行，而不是并行地。使用该隔离级别的应用一定要准备由于串行失败而导致重新执行事务。

READ WRITE

READ ONLY

决定事务是 read/write 还是 read-only。默认为 read/write。当一个事务是 read-only，下面的 SQL 命令是不被允许的： INSERT、 UPDATE、 DELETE 以及COPY FROM（这种情况是如果将要写的表不是临时表） ； 所有的 CREATE, ALTER 和 DROP 命令； GRANT、REVOKE、 TRUNCATE；以及 EXPLAIN ANALYZE 和 EXECUTE 要执行的命令是上述列举的中间。

示例

开始一个事务块：

```
START TRANSACTION;
```

兼容性

在标准中，没有必要发出 START TRANSACTION 来开始一个事务块：任何 SQL 命令会隐式地开始一个块。 HashData 数据库的行为可以被视作在每个命令之后隐式地发出一个没有跟随在 START TRANSACTION（或者BEGIN）之后的 COMMIT 并且因此通常被称作 "自动提交"。为了方便，其他关系型数据库系统也可能会提供自动提交特性。

SQL 标准要求在连续的 transaction_modes 之间有逗号，但是出于历史的原因， HashData 数据库中允许逗号省略。

另见 SET TRANSACTION 的兼容性部分。

另见

BEGIN, SET TRANSACTION

上级话题： SQL命令参考

TRUNCATE

清空一个表。

概要

TRUNCATE [TABLE] name [, ...] [CASCADE | RESTRICT]

描述

TRUNCATE 可以从一组表中快速地移除所有行。 它具有和在每个表上执行无条件 DELETE 相同的效果，不过它会更快，因为它没有实际扫描表。在大表上它最有用。

要截断一个表，用户必须具有其上的 TRUNCATE 特权。

参数

name

要截断的表的名字（可以是方案限定的）。

CASCADE

因为关键字应用到外键引用上 HashData 数据库还没有支持，所有没有效果。

RESTRICT

因为关键字应用到外键引用上 HashData 数据库还没有支持，所有没有效果。

注解

TRUNCATE 将不会引发表上可能存在的任何用户定义的 ON DELETE 触发器。

TRUNCATE 不会截断任何从命名的表继承而来的表。只有命名的表被截断，它的子表不会截断。

TRUNCATE 不会截断任何一个分区表的子表。如果用户指定了一个分区表的子表，TRUNCATE 不会从子表以及它的孩子表上移除行。

示例

清空表 films:

```
TRUNCATE films;
```

兼容性

在 SQL 标准中没有 TRUNCATE 命令。

另见

DELETE, DROP TABLE

上级话题： SQL命令参考

UPDATE

更新一个表的行。

概要

```
UPDATE [ONLY] table [[AS] alias]
   SET {column = {expression | DEFAULT} |
   (column [, ...]) = ({expression | DEFAULT} [, ...])} [, ...]
   [FROM fromlist]
   [WHERE condition | WHERE CURRENT OF cursor_name ]
```

描述

UPDATE 更改满足条件的所有行中指定列的值。只有要被修改的列需要在 SET 子句中提及，没有被显式修改的列保持它们之前的值。

默认地，UPDATE 命令将会更新指定表的所有列以及它的所有子表。如果只想要更新提到的指定表，必须使用 ONLY 子句。

有两种方法使用包含在数据库其他表中的信息来修改一个表：使用子查询或者在 FROM 子句中指定额外的表。这种技术只适合特定的环境。

如果 WHERE CURRENT OF 子句被指定，被更新的行则为最近从指定游标获取的行。

用户必须拥有在要更新表上的 UPDATE 特权。如果任何一列的值需要被 expressions 或者 condition 读取， 用户还必须拥有该列上的 SELECT 特权。

输出

成功完成时，一个 UPDATE 命令返回命令标签如下形式：

```
UPDATE count
```

count 是被更新的行数。 如果 count 为 0，没有行被该查询更新（这不是一个错误）。

参数

ONLY

如果指定，则只会更新所提及表中的匹配行。如果没有指定，任何从所提及表继承得到的表中的匹配的行也会被更新。

table

要更新的表的名称（可以是方案限定的）。

alias

目标表的一个替代名称。在提供了一个别名时，它会完全隐藏表的真实名称。例如，给定 UPDATE foo AS f，UPDATE 语句的剩余部分必须用 f 而不是 foo 引用该表。

column

所指定的表的一列的名称。如果需要，该列名可以用一个子域名称或者数组下标限定。不要在目标列的说明中包括表的名称。

expression

A 要被赋值给该列的一个表达式。该表达式可以使用该表中这一列或者其他列的旧值。

DEFAULT

将该列设置为它的默认值（如果没有为它指定默认值表达式，默认值将会为 NULL）。

fromlist

表表达式的列表，允许来自其他表的列出现在 WHERE 条件和更新表达式中。 这类似于可以在 SELECT 语句的 FROM 子句中指定的表列表。注意目标表不能出现在 fromlist 中，除非用户想做自连接（这种情况下它必须以 alias（别名）出现在 fromlist中）。

condition

一个返回 boolean类型值的表达式。让这个 表达式返回true的行将会被更新。

cursor_name

要在 WHERE CURRENT OF 条件中使用的游标名。要被更新的是从这个游标中最近取出的行。该游标必须是一个在 UPDATE 目标表上的简单（非连接，非聚集）的查询。

WHERE CURRENT OF 不能和一个布尔条件一起指定。

UPDATE...WHERE CURRENT OF 语句只能在服务器端执行，例如在一个交互式的 psql 会话中或者一个脚本中。 语言扩展，例如 PL/pgSQL 不支持可更新游标。

output_expression

在每一行被更新后，要被 UPDATE 命令计算并且返回的表达式。该表达式可以使用 FROM 列出的表中的任何列名。写*可以返回所有列。

output_name

用于一个被返回列的名称。

注解

SET 在 HashData 数据库一个表的分布主键列上是不允许的。

当存在 FROM 子句时，实际发生的是：目标表被连接到 from_list 中的表，并且该连接的每一个输出行表示对目标表的一个更新操作。在使用 FROM 时，用户应该确保该连接对每一个要修改的行产生至多一个输出行。换句话说，一个目标行不应该连接到来自其他表的多于一行上。如果发生这种情况，则只有一个连接行将被用于更新目标行，但是将使用哪一行是很难预测的。

由于这种不确定性，只在一个子选择中引用其他表更安全，不过这种语句通常很难写并且也比使用连接慢。

直接在一个分区表的特定分区（子表）执行 UPDATE 和 DELETE 还不支持。相反，可以在根分区表（由 CREATE TABLE 命令创建的表）上执行这些命令。

示例

把表 films 的列 kind 中的单词 Drama 改成 Dramatic：

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

在表 weather 的一行中调整温度项并且把沉淀物重置为它的默认值：

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi =
temp_lo+15, prcp = DEFAULT
WHERE city = 'San Francisco' AND date = '2016-07-03';
```

使用另一种列列表语法来做同样的更新：

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1,
temp_lo+15, DEFAULT)
WHERE city = 'San Francisco' AND date = '2016-07-03';
```

为管理 Acme Corporation 账户的销售人员增加销售量，使用 FROM 子句语法（假定所有连接的表在 HashData 数据库中是通过 id 列进行分布的）：

```
UPDATE employees SET sales_count = sales_count + 1 FROM
accounts
WHERE accounts.name = 'Acme Corporation'
AND employees.id = accounts.id;
```

执行相同的操作，在 WHERE 子句中使用子选择：

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT id FROM accounts WHERE name = 'Acme Corporation');
```

尝试插入一个新库存项及其库存量。如果该项已经存在，则转而更新已有项的库存量。要这样做并且不让整个事务失败，可以使用保存点：

```
BEGIN;
-- 其他操作
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- 假定上述语句由于未被唯一键失败，
-- 那么现在我们发出这些命令：
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau
Lafite 2003';
-- 继续其他操作，并且最终
COMMIT;
```

兼容性

这个命令符合 SQL 标准，不过 FROM 子句是 HashData 数据库的扩展。

根据标准，列语法应该允许列为从一个单一的行值表达式赋值，如子选择：

```
UPDATE accounts SET (contact_last_name, contact_first_name) =
    (SELECT last_name, first_name FROM salesmen
     WHERE salesmen.id = accounts.sales_id);
```

当前还没有实现 — 源必须是独立的表达式列表。

有些其他数据库系统提供了一个 FROM 选项，在其中在其中目标表 可以在 FROM 中被再次列出。但 HashData 数据库不是这样解释 FROM 的。在移植使用这种扩展的应用时要小心。

另见

DECLARE, DELETE, SELECT, INSERT

上级话题： SQL命令参考

VACUUM

垃圾收集并根据需要分析一个数据库。

概要

```
VACUUM [FULL] [FREEZE] [VERBOSE] [table]
VACUUM [FULL] [FREEZE] [VERBOSE] ANALYZE
             [table [(column [, ...] )]]
```

描述

VACUUM 收回由死亡元组占用的存储空间。在 HashData 数据库操作中，被删除或者被更新废弃的元组并没有在物理上从它们的表中移除，它们将一直存在直到一次 VACUUM 被执行。因此有必要周期性地做 VACUUM，特别是在频繁被更新的表上。

在不带任何参数的情况下，VACUUM 会处理当前用户具有清理权限的当前数据库中的每一个表。通过使用一个参数，VACUUM 可以只处理指定表。

VACUUM ANALYZE 对每一个选定的表执行一个 VACUUM 然后执行 ANALYZE。这是两种命令的一种方便的组合形式，可以用于例行的维护脚本。

VACUUM（不带 FULL）标记在表和索引中已经删除和废除的数据用于之后重用以及收回空间用于重用，只要空间位于一个表末端以及表上的排它锁能够容易地获取。在表开头和中间位置的没有使用的空间仍旧保留。在堆表中，这种形式命令能够并行读写表，因为没有获取排它锁。

使用追加优化表，VACUUM 首先通过清理索引来紧缩表，然后依次紧缩每个 Segment 文件，最后清理辅助关系同时更新统计信息。在每个 Segment 上，可见行从当前 Segment 文件复制到一个新的 Segment 文件，然后将当前的 Segment 文件计划为将删除同时将新的 Segment 文件设置为可用。一个追加优化表的简单 VACUUM 允许在 Segment 文件被紧缩时做 SELECT、INSERT、DELETE 以及 UPDATE 操作。不过，将会短暂地取得一个 Access Exclusive 锁以删除当前的 Segment 文件并且激活新的 Segment 文件。

VACUUM FULL 会做更多深入的处理，包括为了尝试把表紧缩成占用最少的磁盘块而在块间移动元组。这种形式会更慢一些并且在每个表被处理时都需要取得其上的 Access Exclusive 锁。Access Exclusive 锁确保持有者是唯一访问该表的事务。

输出

如果指定了 VERBOSE，VACUUM 会发出进度消息来表明当前正在处理哪个表。各种有关这些表的统计信息也会被打印出来。

参数

FULL

选择 "完全" 清理，它可以收回更多空间，并且需要更长时间和表上的排他锁。

FREEZE

指定 FREEZE 等价于参数 vacuum_freeze_min_age 设置为 0 的 VACUUM。

VERBOSE

为每个表打印一份详细的清理活动报告。

ANALYZE

更新优化器用以决定最有效执行一个查询的方法的统计信息。

table

要清理的表的名称（可以有模式修饰）。缺省时是当前数据库中的所有表。

column

要分析的指定列的名称。缺省是所有列。

注解

VACUUM 不能再一个事务块中执行。

我们建议经常清理活动的生产数据库（至少每晚一次），以保证移除失效的行。在增加或删除了大量行之后， 对受影响的表执行 VACUUM ANALYZE 命令是一个很好的做法。 这样做将把最近的更改更新到系统目录，并且允许 HashData 数据库查询规划器在规划用户查询时做出更好的选择。

> 重要：在 PostgreSQL 中有一个单独叫做 autovacuum 守护进程 的可选服务进程，用来自动执行 VACUUM 和 ANALYZE 命令。该特征在 HashData 数据库中被禁止了。

VACUUM 会导致 I/O 流量的大幅度增加，这可能导致其他活动会话性能变差。因此，有时建议使用基于代价的清理延迟特性。

对于堆表，无效行被记录在一个叫做 空闲空间映射 地方。空闲空间映射必须足够大能够覆盖数据库中所有堆表的无效行。如果空间不足，那些从空闲空间映射溢出的无效行占用的空间将不能被一个 VACUUM 回收。

VACUUM 命令略过外部包。

VACUUM FULL 回收所有无效行空间，然后它需要获取一个排它锁在它当前处理的每个表上面，这是一个非常昂贵的操作，有可能会耗费很长的时间在那些大的，分布式的 HashData 数据库的表上。执行 VACUUM FULL 在维护数据库期间。

作为一个 VACUUM FULL 的替代方案，用户可以通过使用 CREATE TABLE AS 重新构建表，然后删除旧的表。

将空闲空间映射设置为适当的大小。通过下面的配置参数来配置空闲空间映射：

- max_fsm_pages
- max_fsm_relations

对于追加优化表， VACUUM 要求有足够的磁盘空间来容纳新的 Segment 文件在 VACUUM 执行过程中。如果在一个 Segment 文件中隐藏行的行数占总行数的比率小于一个阈值（默认值，10），那么 Segment 文件不会进行压缩。阈值可以通过 gp_appendonly_compaction_threshold 来进行配置。VACUUM FULL 忽略阈值同时重写段文件而不考虑比率。可以通过 gp_appendonly_compaction 参数来关闭在追加优化表上 VACUUM 的使用。

在追加优化表被清理时如果检测到一个并发的可序列化事务，那么当前和后续的 Segment 文件都不会被紧缩。如果一个 Segment 文件已经被紧缩但是在删除原始 Segment 文件的事务中检测到在一个并发可序列化事务，则会跳过删除。这样在清理完成后，可能会留下一个或两个状态为 “awaiting drop” 的 Segment 文件。

示例

清理当前数据库下的所有表：

```
VACUUM;
```

只清理一张特定的表：

```
VACUUM mytable;
```

清理当前数据库下的所有表同时为查询优化器收集统计信息：

```
VACUUM ANALYZE;
```

兼容性

在 SQL 标准中没有 VACUUM。

另见

ANALYZE

上级话题： SQL命令参考

VALUES

计算一个行集合。

概要

```
VALUES ( expression [, ...] ) [, ...]
    [ORDER BY sort_expression [ASC | DESC | USING operator] [, ...]]
    [LIMIT {count | ALL}] [OFFSET start]
```

描述

VALUES 计算由值表达式指定的一个行值或者一组行值。 更常见的是把它用来生成一个大型命令内的 "常量表"，但是它也可以被独自使用。

当多于一行被指定时，所有行都必须具有相同数量的元素。 结果表的列数据类型由出现在该列的表达式的显式或者推导类型组合决定，决定的规则与 UNION 相同。

在大型的命令中，在语法上允许 VALUES 出现在 SELECT 出现的任何地方。因为语法把它当做一个 SELECT，可以为一个 VALUES 命令使用 ORDER BY、LIMIT 和 OFFSET 子句。

参数

expression

要在结果表（行集合）中指定位置计算并且插入的一个常量或者表达式。在一个出现于 INSERT 顶层的 VALUES 列表中，expression 可以被 DEFAULT 替代以表示应该插入目标列的默认值。当 VALUES 出现在其他环境中时，不能使用 DEFAULT。

sort_expression

一个指示如何排序结果行的表达式或者整型常量。这个表达式可以用 column1、column2 等来引用该 VALUES 结果的列。

operator

一个排序操作符。

LIMIT count

OFFSET start

要返回的最大行数。

注解

应该避免具有大量行的 VALUES 列表，否则可能会碰到内存不足失败或者很差的性能。出现在 INSERT 中的 VALUES 是一种特殊情况（因为想要的列类型可以从 INSERT 的目标表得知，并且不需要通过扫描该 VALUES 列表来推导），因此它可以处理比其他环境中更大的列表。

示例

一个纯粹的 VALUES 命令：

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

这将返回一个具有两列、三行的表。它实际等效于：

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

更常用地，VALUES 可以被用在一个大型 SQL 命令中。 在 INSERT 中最常用：

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

在 INSERT 环境中，一个 VALUES 列表的项可以是 DEFAULT 指示应该使用该列的默认值而不是指定一个值：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82
minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

VALUES 也可以被用在可以写子 SELECT 的地方，例如在一个 FROM 子句中：

```
SELECT f.* FROM films f, (VALUES('MGM', 'Horror'), ('UA',
'Sci-Fi')) AS t (studio, kind) WHERE f.studio = t.studio AND
f.kind = t.kind;
UPDATE employees SET salary = salary * v.increase FROM
(VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno,
target, increase) WHERE employees.depno = v.depno AND
employees.sales >= v.target;
```

注意当 VALUES 被用在一个 FROM 子句中时， 需要提供一个 AS 句，与 SELECT 相同。 不需要为所有的列用 AS 子句指定名称，但是那样做是一种好习惯。在 HashData 数据库中，VALUES 的默认列名是 column1、column2 等，但在其他数据库系统中可能会不同。

当在 INSERT 中使用 VALUES 时，值都会被自动地强制为相应目标列的数据类型。当在其他环境中使用时，有必要指定正确的数据类型。如果项都是带引号的字符串常量，强制第一个就足以为所有项假设数据类型：

```
SELECT * FROM machines WHERE ip_address IN
(VALUES('192.168.0.1'::inet), ('192.168.0.10'),
('192.0.2.43'));
```

tips：对于简单的 IN 测试，最好使用 IN 的 list-of-scalars 形式而不是写一个上面那样的 VALUES 查询。标量列表方法的书写更少并且常常更加高效。

兼容性

VALUES 符合 SQL 标准，但 LIMIT 和 OFFSET 是 HashData 数据库的扩展。

另见

INSERT, SELECT

上级话题： SQL命令参考

数据类型

HashData 数据库有丰富的本地数据类型集供用户使用。 用户还可以使用 CREATE TYPE 命令定义新的数据类型。该引用显示所有的内置数据类型。除了这里列出的类型之外， 还有一些内部使用的数据类型，例如 *oid*（对象标识符），但是在本指南中没有记录。

以下数据类型由SQL指定： *bit*, *bit varying*, *boolean*, *character varying*, *varchar*, *character*, *char*, *date*, *double precision*, *integer*, *interval*, *numeric*, *decimal*, *real*, *smallint*, *time*（有或没有时区）和 *timestamp* (有或没有时区)。

每种数据类型都具有由其输入和输出功能决定的外部表现。 许多内置类型具有明显的外部格式。 但是有些数据类型只为 PostgreSQL (和 HashData 数据库)所用，例如几何路径，或者一些格式可能的情况， 例如日期和时间类型。一些输入和输出功能不可逆。也就是说，与原始输入相比，输出功能的结果可能会失去准确性。

表 1. HashData 数据库内建数据类型

Name	Alias	Size	Range	Description
bigint	int8	8 bytes	-922337203 6854775808 to 922337203 6854775807	large range integer
bigserial	serial8	8 bytes	1 to 922337203 6854775807	large autoincrementing integer
bit [(n)]		n bits	bit string constant	fixed-length bit string
bit varying [(n)]1	varbit	actual number of bits	bit string constant	variable-length bit string
boolean	bool	1 byte	true/false, t/f, yes/no, y/n, 1/0	logical boolean (true/false)
box		32 bytes	((x1,y1),(x2,y2))	rectangular box in the plane - not allowed in distribution key columns.
bytea1		1 byte +*binary string*	sequence ofoctets	variable-length binary string
character [(n)]1	char [(n)]	1 byte +*n*	strings up to_n_characters in length	fixed-length, blank padded
character varying [(n)]1	varchar [(n)]	1 byte +*string size*	strings up to_n_characters in length	variable-length with limit
cidr		12 or 24 bytes		IPv4 and IPv6 networks
circle		24 bytes	<(x,y),r> (center and radius)	circle in the plane - not allowed in distribution key columns.
date		4 bytes	4713 BC - 294,277 AD	calendar date (year, month, day)
decimal [(p, s)]1	numeric [(p, s)]	variable	no limit	user-specified precision, exact
double precision	float8float	8 bytes	15 decimal digits precision	variable-precision, inexact
inet		12 or 24		IPv4 and IPv6 hosts and networks

		bytes		
integer	int, int4	4 bytes	-2147483648 to +2147483647	usual choice for integer
interval [(p)]		12 bytes	-178000000 years - 178000000 years	time span
lseg		32 bytes	((x1,y1),(x2,y2))	line segment in the plane - not allowed in distribution key columns.
macaddr		6 bytes		MAC addresses
money		4 bytes	-21474836.48 to +21474836.47	currency amount
path1		16+16n bytes	[(x1,y1),...]	geometric path in the plane - not allowed in distribution key columns.
point		16 bytes	(x,y)	geometric point in the plane - not allowed in distribution key columns.
polygon		40+16n bytes	((x1,y1),...)	closed geometric path in the plane - not allowed in distribution key columns.
real	float4	4 bytes	6 decimal digits precision	variable-precision, inexact
serial	serial4	4 bytes	1 to 2147483647	autoincrementing integer
smallint	int2	2 bytes	-32768 to +32767	small range integer
text1		1 byte *+string size*	strings of any length	variable unlimited length
time [(p)] [without time zone]		8 bytes	00:00:00[.000000] - 24:00:00[.000000]	time of day only
time [(p)] with time zone	timetz	12 bytes	00:00:00+1359 - 24:00:00-1359	time of day only, with time zone
timestamp [(p)] [without time zone]		8 bytes	4713 BC - 294,277 AD	both date and time
timestamp [(p)] with time zone	timestamptz	8 bytes	4713 BC - 294,277 AD	both date and time, with time zone
xml1		1 byte *+xml size*	xml of any length	variable unlimited length

伪类型

HashData 数据库支持统称为 伪类型的专用数据类型项。 伪类型不能用作列数据类型，但可以用于声明函数的参数或结果类型。每个可用的伪类型在函数的行为与简单地取得或返回特定SQL数据类型的值不对应的情况下很有用。

以程序语言编码的函数只能使用其实现语言所允许的伪类型。 程序语言都禁止使用伪类型作为参数类型，并且只允许使用 *void* 和 *record*作为结果类型。

伪类型记录 作为返回数据类型的函数返回未指定的行类型。 该 记录 表示可能是匿名复合类型的数组。由于 复合数据带有自己的类型识别，因此在数组级别不需要额外的知识。

伪类型*void* 表示函数无返回值。

> 注解： HashData 数据库不支持触发器和伪类型 触发器。

types *anyelement*, *anyarray*, *anynonarray*, 和 *anyenum* 多态类型的伪类型。 一些程序语言还支持使用类型 *anyarray*, *anyelement*, *anyenum*, 和 *anynonarray*的多态函数。

有关伪类型的更多信息， 请参阅有关 伪类型的Postgres文档。

多态类型

特别有趣的四种伪类型是 *anyelement*, *anyarray*, *anynonarray*, 和 *anyenum*, 它们统称为多态 类型。使用这些类型声明的任何函数都被称为多态函数。一个多态函数可以对许多不同的数据类型进行操作，特定数据类型由在运行时实际传递给它的数据类型确定。

当一个调用多态函数的查询被解析时，多态参数和结果会被绑在一起并且解析为一种特定的数据类型。声明为 *anyelement* 的每个位置（参数或返回值）声明为anyelement的每个位置（参数或返回值）都被允许具有任何特定的实际数据类型，但在任何给定的调用中，它们必须都是相同的实际类型 声明为*anyarray* c的每个位置都可以有任何数组数据类型，但类似地，它们必须都是相同的类型。 如果有声明为*anyarray*也有声明为 *anyelement*， 则再*anyarray* 位置中的实际数组类型 必须是一个数组的，其元素在*anyelement* 位置上显示相同的类型。 *anynonarray* 被视为与 *anyelement*完全相同，但添加了实际类型不能是数组类型的附加约束。*anyenum* 被视为与*anyelement*完全相同，但添加的实际类型必须是 enum类型的约束。

当多个参数位置被声明为多态类型时，net效果是仅允许实际参数类型的某些组合。 例如，一个声明为equal(*anyelement*, *anyelement*)的函数只要具有相同的数据类型，就可以使用任何两个输入值。

当函数的返回值被声明为多态类型时，还必须至少有一个参数位置也是多态的，并且作为参数提供的实际数据类型确定该调用的实际结果类型。 例如，如果没有数组下标机制，可以定义一个实现下标的函数 subscript(*anyarray*, integer) 返回 *anyelement*。此声明将实际的第一个参数限制为数组类型，并允许解析器从实际的第一个参数的类型推断出正确的结果类型。另一个例子是一个声明的函数 myfunc(*anyarray*) returns *anyenum* 将只接受数组enum 类型。

请注意，*anynonarray* 和*anyenum* 不代表单独的类型变量; 它们与*anyelement*类型相同，只是附加约束。 例如，声明函数为 myfunc(*anyelement*, *anyenum*) 相当于将其声明为 myfunc(*anyenum*, *anyenum*): 两个实际参数必须相同 enum 类型。

当其最后一个参数被声明为一个可变函数（一个取可变数量的参数）是多态的VARIADIC *anyarray*。为了参数匹配和确定实际的结果类型， 这样的函数的行为与用户已经声明了适当数量的 *anynonarray* 参数的行为相同。

有关多态类型的更多信息，请参阅有关 多态参数和返回类型的Postgres文档。

上级标题： HashData 数据库参考指南

1 对于可变长度数据类型，如果数据大于等于127字节，存储开销是每个字节需要4个字节存储。

字符集支持

HashData 数据库里面的字符集支持用户能够以各种字符集存储文本，包括单字节字符集， 比如 ISO 8859 系列，以及多字节字符集 ，比如EUC（扩展 Unix 编码 Extended Unix Code）、UTF-8 和 Mule 内部编码。 所有被支持的字符集都可以被客户端透明地使用，但少数只能在服务器上使用（即作为一种服务器端编码）。 初始化用户的HashData 数据库数组时，会选择默认字符集 gpinitsystem。 在用户创建一个数据库时可以重载它，因此用户可能会有多个数据库并且每一个使用不同的字符集。

表 **1. HashData** 数据库字符集 1

Name	Description	Language	Server?	Bytes/Char	Aliases
BIG5	Big Five	Traditional Chinese	No	1-2	WIN950, Windows950
EUC_CN	Extended UNIX Code-CN	Simplified Chinese	Yes	1-3	
EUC_JP	Extended UNIX Code-JP	Japanese	Yes	1-3	
EUC_KR	Extended UNIX Code-KR	Korean	Yes	1-3	
EUC_TW	Extended UNIX Code-TW	Traditional Chinese, Taiwanese	Yes	1-3	
GB18030	National Standard	Chinese	No	1-2	
GBK	Extended National Standard	Simplified Chinese	No	1-2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	Latin/Cyrillic	Yes	1	
ISO_8859_6	ISO 8859-6, ECMA 114	Latin/Arabic	Yes	1	
ISO_8859_7	ISO 8859-7, ECMA 118	Latin/Greek	Yes	1	
ISO_8859_8	ISO 8859-8, ECMA 121	Latin/Hebrew	Yes	1	
JOHAB	JOHA	Korean (Hangul)	Yes	1-3	
KOI8	KOI8-R(U)	Cyrillic	Yes	1	KOI8R
LATIN1	ISO 8859-1, ECMA 94	Western European	Yes	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	Central European	Yes	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	South European	Yes	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	North European	Yes	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	Turkish	Yes	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Nordic	Yes	1	ISO885910
LATIN7	ISO 8859-13	Baltic	Yes	1	ISO885913

LATIN8	ISO 8859-14	Celtic	Yes	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 with Euro and accents	Yes	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Romanian	Yes	1	ISO885916
MULE_INTERNAL	Mule internal code	Multilingual Emacs	Yes	1-4	
SJIS	Shift JIS	Japanese	No	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SQL_ASCII	unspecified2	any	No	1	
UHC	Unified Hangul Code	Korean	No	1-2	WIN949, Windows949
UTF8	Unicode, 8-bit	all	Yes	1-4	Unicode
WIN866	Windows CP866	Cyrillic	Yes	1	ALT
WIN874	Windows CP874	Thai	Yes	1	
WIN1250	Windows CP1250	Central European	Yes	1	
WIN1251	Windows CP1251	Cyrillic	Yes	1	WIN
WIN1252	Windows CP1252	Western European	Yes	1	
WIN1253	Windows CP1253	Greek	Yes	1	
WIN1254	Windows CP1254	Turkish	Yes	1	
WIN1255	Windows CP1255	Hebrew	Yes	1	
WIN1256	Windows CP1256	Arabic	Yes	1	
WIN1257	Windows CP1257	Baltic	Yes	1	
WIN1258	Windows CP1258	Vietnamese	Yes	1	ABC, TCVN, TCVN5712, VSCII

上级主题： HashData 数据库参考指南

设置字符集

gpinitsystem通过在初始化时间读取gp_init_config 文件中的 ENCODING参数的设置来定义HashData 数据库系统的默认字符集。 默认的字符集是UNICODE 或UTF8.

除了用作系统级默认值之外，还可以创建一个不同字符集的数据库。例如:

```
=> CREATE DATABASE korean WITH ENCODING 'EUC_KR';
```

重点： 虽然用户可以指定数据库所需的任何编码， 但选择不是用户所选择的语言环境所期望的编码是不明智的。该LC_COLLATE和 LC_CTYPE设置意味着特定的编码，并且依赖于区域设置的操作（例如排序）可能会误解处于不兼容编码的数据。

由于这些区域设置被 gpinitsystem 系统冻结，因此在不同数据库中使用不同编码的灵活性明显比实际更理论

一种安全使用多个编码的方法是在初始化时间内将语言环境设置为C 或POSIX ，从而禁止任何真正的区域意识。

服务器和客户端之间的字符集转换

HashData 数据库支持服务器和客户端之间的某些字符集组合的自动字符集转换。 转换信息存储在主 *pg_conversion*系统目录表中。 HashData 数据库带有一些预定义的转换，或者用户可以使用SQL命令创建一个新的转换CREATE CONVERSION.

表 2. 客户端/服务器字符集转换

Server Character Set	Available Client Character Sets
BIG5	not supported as a server encoding
EUC_CN	EUC_CN, MULE_INTERNAL, UTF8
EUC_JP	EUC_JP, MULE_INTERNAL, SJIS, UTF8
EUC_KR	EUC_KR, MULE_INTERNAL, UTF8
EUC_TW	EUC_TW, BIG5, MULE_INTERNAL, UTF8
GB18030	not supported as a server encoding
GBK	not supported as a server encoding
ISO_8859_5	ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	ISO_8859_6, UTF8
ISO_8859_7	ISO_8859_7, UTF8
ISO_8859_8	ISO_8859_8, UTF8
JOHAB	JOHAB, UTF8
KOI8	KOI8, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
LATIN1	LATIN1, MULE_INTERNAL, UTF8
LATIN2	LATIN2, MULE_INTERNAL, UTF8, WIN1250
LATIN3	LATIN3, MULE_INTERNAL, UTF8
LATIN4	LATIN4, MULE_INTERNAL, UTF8
LATIN5	LATIN5, UTF8
LATIN6	LATIN6, UTF8
LATIN7	LATIN7, UTF8
LATIN8	LATIN8, UTF8
LATIN9	LATIN9, UTF8
LATIN10	LATIN10, UTF8
MULE_INTERNAL	MULE_INTERNAL, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	not supported as a server encoding
SQL_ASCII	not supported as a server encoding
UHC	not supported as a server encoding
UTF8	all supported encodings
WIN866	WIN866
ISO_8859_5	KOI8, MULE_INTERNAL, UTF8, WIN1251
WIN874	WIN874, UTF8

WIN1250	WIN1250, LATIN2, MULE_INTERNAL, UTF8
WIN1251	WIN1251, ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866
WIN1252	WIN1252, UTF8
WIN1253	WIN1253, UTF8
WIN1254	WIN1254, UTF8
WIN1255	WIN1255, UTF8
WIN1256	WIN1256, UTF8
WIN1257	WIN1257, UTF8
WIN1258	WIN1258, UTF8

要启用自动字符集转换，用户必须告诉HashData 数据库用户要在客户端中使用的字符集（编码）。有几种方法可以实现这一点：

- 使用\encoding 命令在psql中, 它允许用户即时更改客户端编码。
- 使用SETclient_encoding TO。

  要设置客户端编码，请使用以下SQL命令：

```
=> SET CLIENT_ENCODING TO 'value';
```

要查询当前的客户端编码：

```
=> SHOW client_encoding;
```

要返回到默认编码：

```
=> RESET client_encoding;
```

- 使用PGCLIENTENCODING 环境变量。当在客户端的环境变量定义为 PGCLIENTENCODING 时，与服务器建立连接时会自动选择该客户端编码。(这可以随后使用上述其他任何方法重写)
- 设置配置参数client_encoding。 如果client_encoding被设置为主 postgresql.conf 文件，当建立与HashData 数据库的连接时，会自动选择该客户端编码。 (这可以随后使用上述其他任何方法重写。)

如果特定字符的转换是不可能的 " 假设 用户为服务器选择了 EUC_JP 而为客户端选择了 LATIN1 则一些日文字符在LATIN1 没有表现形式" 则会报告错误。

如果客户端字符集被定义为SQL_ASCII， 无论服务器的字符集如何，禁止编码转换。 除非用户使用所有ASCII数据，否则 SQL_ASCII 是不明智的。 不支持SQL_ASCII 作为服务器编码。

1 并非所有API都支持所有列出的字符集。例如，JDBC驱动程序不支持MULE_INTERNAL，LATIN6，LATIN8和LATIN10。

2 SQL_ASCII设置与其他设置的行为大不相同。字节值0-127根据ASCII标准进行解释，字节值128-255作为未解释的字符。如果使用任何非ASCII数据，则将SQL_ASCII设置用作客户端编码是不明智的。不支持SQL_ASCII作为服务器编码。

系统配置参数

HashData 数据库服务器配置参数的字典序描述。

- add_missing_from
- application_name
- array_nulls
- authentication_timeout
- backslash_quote
- block_size
- bonjour_name
- check_function_bodies
- client_encoding
- client_min_messages
- cpu_index_tuple_cost
- cpu_operator_cost
- cpu_tuple_cost
- cursor_tuple_fraction
- custom_variable_classes
- DateStyle
- db_user_namespace
- deadlock_timeout
- debug_assertions
- debug_pretty_print
- debug_print_parse
- debug_print_plan
- debug_print_prelim_plan
- debug_print_rewritten
- debug_print_slice_table
- default_statistics_target
- default_tablespace
- default_transaction_isolation
- default_transaction_read_only
- dynamic_library_path
- effective_cache_size
- enable_bitmapscan
- enable_groupagg
- enable_hashagg
- enable_hashjoin
- enable_indexscan
- enable_mergejoin
- enable_nestloop
- enable_seqscan
- enable_sort
- enable_tidscan
- escape_string_warning
- explain_pretty_print
- extra_float_digits
- filerep_mirrorvalidation_during_resync
- from_collapse_limit
- gp_adjust_selectivity_for_outerjoins
- gp_analyze_relative_error

- gp_appendonly_compaction
- gp_appendonly_compaction_threshold
- gp_autostats_mode
- gp_autostats_mode_in_functions
- gp_autostats_on_change_threshold
- gp_backup_directIO
- gp_backup_directIO_read_chunk_mb
- gp_cached_segworkers_threshold
- gp_command_count
- gp_connection_send_timeout
- gp_connections_per_thread
- gp_content
- gp_create_table_random_default_distribution
- gp_dbid
- gp_debug_linger
- gp_default_storage_options
- gp_dynamic_partition_pruning
- gp_email_from
- gp_email_smtp_password
- gp_email_smtp_server
- gp_email_smtp_userid
- gp_email_to
- gp_enable_adaptive_nestloop
- gp_enable_agg_distinct
- gp_enable_agg_distinct_pruning
- gp_enable_direct_dispatch
- gp_enable_exchange_default_partition
- gp_enable_fallback_plan
- gp_enable_fast_sri
- gp_enable_gpperfmon
- gp_enable_groupext_distinct_gather
- gp_enable_groupext_distinct_pruning
- gp_enable_multiphase_agg
- gp_enable_predicate_propagation
- gp_enable_preunique
- gp_enable_relsize_collection
- gp_enable_sequential_window_plans
- gp_enable_sort_distinct
- gp_enable_sort_limit
- gp_external_enable_exec
- gp_external_max_segs
- gp_filerep_tcp_keepalives_count
- gp_filerep_tcp_keepalives_idle
- gp_filerep_tcp_keepalives_interval
- gp_fts_probe_interval
- gp_fts_probe_retries
- gp_fts_probe_threadcount
- gp_fts_probe_timeout
- gp_gpperfmon_send_interval
- gpperfmon_log_alert_level
- gp_hadoop_home
- gp_hadoop_target_version
- gp_hashjoin_tuples_per_bucket

- gp_idf_deduplicate
- gp_initial_bad_row_limit
- gp_interconnect_fc_method
- gp_interconnect_hash_multiplier
- gp_interconnect_queue_depth
- gp_interconnect_setup_timeout
- gp_interconnect_snd_queue_depth
- gp_interconnect_type
- gp_log_format
- gp_log_fts
- gp_log_gang
- gp_max_csv_line_length
- gp_max_databases
- gp_max_filespaces
- gp_max_local_distributed_cache
- gp_max_packet_size
- gp_max_plan_size
- gp_max_tablespaces
- gp_motion_cost_per_row
- gp_num_contents_in_cluster
- gp_reject_percent_threshold
- gp_reraise_signal
- gp_resqueue_memory_policy
- gp_resqueue_priority
- gp_resqueue_priority_cpucores_per_segment
- gp_resqueue_priority_sweeper_interval
- gp_role
- gp_safefswritesize
- gp_segment_connect_timeout
- gp_segments_for_planner
- gp_server_version
- gp_server_version_num
- gp_session_id
- gp_set_proc_affinity
- gp_set_read_only
- gp_snmp_community
- gp_snmp_monitor_address
- gp_snmp_use_inform_or_trap
- gp_statistics_pullup_from_child_partition
- gp_statistics_use_fkeys
- gp_vmem_idle_resource_timeout
- gp_vmem_protect_limit
- gp_vmem_protect_segworker_cache_limit
- gp_workfile_checksumming
- gp_workfile_compress_algorithm
- gp_workfile_limit_files_per_query
- gp_workfile_limit_per_query
- gp_workfile_limit_per_segment
- gpperfmon_port
- 整数_datetimes
- IntervalStyle
- join_collapse_limit
- keep_wal_segments

- krb_caseins_users
- krb_server_keyfile
- krb_srvname
- lc_collate
- lc_ctype
- lc_messages
- lc_monetary
- lc_numeric
- lc_time
- listen_addresses
- local_preload_libraries
- log_autostats
- log_connections
- log_disconnections
- log_dispatch_stats
- log_duration
- log_error_verbosity
- log_executor_stats
- log_hostname
- log_min_duration_statement
- log_min_error_statement
- log_min_messages
- log_parser_stats
- log_planner_stats
- log_rotation_age
- log_rotation_size
- log_statement
- log_statement_stats
- log_timezone
- log_truncate_on_rotation
- max_appendonly_tables
- max_connections
- max_files_per_process
- max_fsm_pages
- max_fsm_relations
- max_function_args
- max_identifier_length
- max_index_keys
- max_locks_per_transaction
- max_prepared_transactions
- max_resource_portals_per_transaction
- max_resource_queues
- max_stack_depth
- max_statement_mem
- optimizer
- optimizer_array_expansion_threshold
- optimizer_analyze_root_partition
- optimizer_control
- optimizer_cte_inlining_bound
- optimizer_enable_master_only_queries
- optimizer_force_multistage_agg
- optimizer_force_three_stage_scalar_dqa
- optimizer_join_order_threshold

- optimizer_mdcache_size
- optimizer_metadata_caching
- optimizer_minidump
- optimizer_nestloop_factor
- optimizer_parallel_union
- optimizer_print_missing_stats
- optimizer_print_optimization_stats
- optimizer_sort_factor
- password_encryption
- password_hash_algorithm
- pgstat_track_activity_query_size
- pljava_classpath
- pljava_classpath_insecure
- pljava_statement_cache_size
- pljava_release_lingering_savepoints
- pljava_vmoptions
- port
- random_page_cost
- readable_external_table_timeout
- repl_catchup_within_range
- replication_timeout
- regex_flavor
- resource_cleanup_gangs_on_wait
- resource_select_only
- runaway_detector_activation_percent
- search_path
- seq_page_cost
- server_encoding
- server_version
- server_version_num
- shared_buffers
- shared_preload_libraries
- ssl
- ssl_ciphers
- standard_conforming_strings
- statement_mem
- statement_timeout
- stats_queue_level
- superuser_reserved_connections
- tcp_keepalives_count
- tcp_keepalives_idle
- tcp_keepalives_interval
- temp_buffers
- TimeZone
- timezone_abbreviations
- track_activities
- track_counts
- transaction_isolation
- transaction_read_only
- transform_null_equals
- unix_socket_directory
- unix_socket_group
- unix_socket_permissions

- update_process_title
- vacuum_cost_delay
- vacuum_cost_limit
- vacuum_cost_page_dirty
- vacuum_cost_page_hit
- vacuum_cost_page_miss
- vacuum_freeze_min_age
- validate_previous_free_tid
- vmem_process_interrupt
- wal_receiver_status_interval
- writable_external_table_bufsize
- xid_stop_limit
- xid_warn_limit
- xmlbinary
- xmloption

add_missing_from

自动向 FROM 字句添加缺少的表引用。当前与 8.1 之前的版本的 PostgreSQL 的版本兼容，默认情况下允许此行为。

值范围	默认	设置分类
Boolean	off	master session reload

application_name

设置客户端会话的应用程序名称。例如，如果通过 psql 连接，这将会被设置为 psql。设置应用程序名称可以在日志消息和统计视图中进行报告。

值范围	默认	设置分类
string		master session reload

array_nulls

该参数控制输入解析器是否将非引用的 NULL 识别为指定空数组元素。默认情况下，这是打开的，允许输入数组值包含空值。因此可以将 NULL 视为指定字符串值为 “NULL” 的普通数组元素。

值范围	默认	设置分类
Boolean	on	master session reload

authentication_timeout

完成客户端认证的最大时间，这样可以防止挂起的客户端无限期占用连接。

值范围	默认	设置分类
任何有效的时间表达式（数字和单位）	1分钟	local system restart

backslash_quote

这可以控制是否在字符串中可以用 \'表示引号。代表引号的首选 SQL 标准是用（''）表示，但是 PostgreSQL 历来也使用 \'。但是，使用 \' 会导致安全风险，因为在一些客户端字符集编码中，有很多多字节字符，其中最后一个字节等同于 ASCII 字符 \。

值范围	默认	设置分类
on（总是允许 \'） off（总是拒绝） safe_encoding（只有客户端编码不允许多字节中的ASCII字符\才允许）	safe_encoding	master session reload

block_size

报告磁盘块大小。

值范围	默认	设置分类
字节数	32768	只读

bonjour_name

指定 Bonjour 广播名称。默认情况下，使用计算机名称，指定为空字符串。如果服务器未支持 Bonjour 服务，则忽略此选项。

值范围	默认	设置分类
string	未设置	master system restart

check_function_bodies

当设置为关闭时，禁用在 CREATE FUNCTION 期间函数体字符串的验证。当从 dump 恢复函数定义时禁用验证对避免诸如转发引用之类的问题是有时有用的。

值范围	默认	设置分类
Boolean	on	master session reload

client_encoding

设置客户端编码（字符集）。默认是使用与数据库相同的编码。请参阅 PostgreSQL 文档中的 支持的字符集。

值范围	默认	设置分类
字符集	UTF8	master session reload

client_min_messages

控制哪些消息级别发送到客户端。每个级别包括跟它随后的所有级别，越往后的级别，发送的消息就越少。

值范围	默认	设置分类
DEBUG5 DEBUGz4 DEBUG3 DEBUG2 DEBUG1 LOG NOTICE WARNING ERROR FATAL PANIC	NOTICE	master session reload

cpu_index_tuple_cost

对于传统的查询优化器（planner），在索引扫描期间设置对处理每个索引行代价的估计。这是作为顺序页面提取代价的一部分来衡量的。

值范围	默认	设置分类
浮点	0.005	master session reload

cpu_operator_cost

对于传统的查询优化器（planner），设置对处理 WHERE 语句中每个操作符代价的估计。这是作为顺序页面提取代价的一部分来衡量的。

值范围	默认	设置分类
浮点	0.0025	master session reload

cpu_tuple_cost

对于传统的查询优化器（planner），设置对处理一个查询中每行（元组）代价的估计。这是作为顺序页面提取代价的一部分来衡量的。

值范围	默认	设置分类
浮点	0.01	master session reload

cursor_tuple_fraction

告知传统查询优化器（planner）预期在游标查询中提取多少行，从而允许传统优化器使用此信息来优化查询计划。默认值为 1 表示获取所有行。

值范围	默认	设置分类
整数	1	master session reload

custom_variable_classes

指定要用于自定义变量的一个或多个类名。自定义变量通常是服务器不知道的变量，但是由一些附加模块使用。这些变量的名字必须由类名，点和变量名组成。

值范围	默认	设置分类
逗号分隔的类名列表	未设置	local system restart

DateStyle

设置日期和时间值的显示格式，以及解释模糊日期输入值的规则。该变量值包含两个独立的而部分：输出格式规范和输入输出规范中年月日的顺序。

值范围	默认	设置分类
\, \ 其中： \ 是 ISO、Postgres、SQL 或者 German \ 是 DMY、MDY 或者 YMD	ISO, MDY	master session reload

db_user_namespace

这启用了每个数据库的用户名。如果打开，用户应该以 *username@dbname* 创建用户。要创建普通的全局用户，只需要在客户端指定用户名时附加 @。

值范围	默认	设置分类
Boolean	off	local system restart

deadlock_timeout

在检查以查看是否存在死锁情况之前等待锁的时间。在一个比较重的服务器上，用户可能希望提高此值。理想的情况下，设置的值应该超过用户的典型处理时间，以此提高在等待线程在决定检查死锁之前自动解锁的几率。

值范围	默认	设置分类
任何有效时间的表达式（数字或者单位）。	1 s	local system restart

debug_assertions

打开各种断言检查。

值范围	默认	设置分类
Boolean	off	local system restart

debug_pretty_print

缩进调试输出产生更可读但是更长的输出格式。 *client_min_messages* 或者 *log_min_messages* 必须是 DEBUG1 或者更低。

值范围	默认	设置分类
Boolean	off	master session reload

debug_print_parse

对于每一个执行的查询，打印出结果分析树。 *client_min_messages* 或 *log_min_messages* 必须是 DEBUG1 或者更低。

值范围	默认	设置分类
Boolean	off	master session reload

debug_print_plan

对于每个执行的查询，打印出 HashData 并行查询执行计划。 *client_min_messages* 或 *log_min_messages* 必须是 DEBUG1 或者更低。

值范围	默认	设置分类
Boolean	off	master session reload

debug_print_prelim_plan

对每个执行的查询，打印出初步查询计划。 *client_min_messages* 或 *log_min_messages* 必须是 DEBUG1 或者更低。

值范围	默认	设置分类
Boolean	off	master session reload

debug_print_rewritten

对于每个执行的查询，打印出查询重写输出。 *client_min_messages* 或 *log_min_messages* 必须是 DEBUG1 or lower.

值范围	默认	设置分类
Boolean	off	master session reload

debug_print_slice_table

对于每个执行的查询，打印 HashData 查询分片计划。 *client_min_messages* 或 *log_min_messages* 必须是 DEBUG1 或者更低。

值范围	默认	设置分类
Boolean	off	master session reload

default_statistics_target

通过 ALTER TABLE SET STATISTICS 为没有指定列目标集的表的列设置默认统计目标。较大的值会增加 ANALYZE 所需的时间，但是可能会提高传统查询优化器（planner）估计的质量。

值范围	默认	设置分类
整数 > 0	25	master session reload

default_tablespace

当 CREATE 命令没有明确指定一个表空间，会在默认的表空间创建对象（表和索引）。

值范围	默认	设置分类
表空间的名字	未设置	master session reload

default_transaction_isolation

控制每个新事务的默认隔离级别

值范围	默认	设置分类
读已提交（read committed） 读未提交（read uncommitted） 序列化（serializable）	读已提交（read committed）	master session reload

default_transaction_read_only

控制每个新事务的默认只读状态。只读的SQL事务 不能修改非临时表。

值范围	默认	设置分类
Boolean	off	master session reload

dynamic_library_path

如果需要打开动态加载的模块，并且在 CREATE FUNCTION 或 LOAD 命令中指定的文件名没有目录部分（即：目录不包括斜杠），系统会搜索该路径以获取所需的文件。此时， PostgreSQL 内置编译的包库目录会替换 $libdir。这是由标准 PostgreSQL 发行版提供的模块安装位置。

值范围	默认	设置分类
由冒号分隔的绝对目录路径列表	$libdir	local system restart

effective_cache_size

设置对于遗传查询优化器（计划器）的单个查询可用的磁盘缓存的有效大小的假设。这是对使用索引成本估计的因素；较高的值使之更可能使用索引扫描，较低的值使之更可能使用顺序扫描。该参数对 HashData 服务器实例分配的共享内存大小没有影响。也不会保留内核磁盘缓存；它仅用于估计目的。

值范围	默认	设置分类
浮点	512MB	master session reload

enable_bitmapscan

启用或禁用遗传查询优化器（计划程序）使用位图扫描计划类型。请注意，这不同于位图索引扫描。位图扫描意味着索引将在适当时候动态转换为内存中的位图，从而使得针对大型表的复杂查询的索引性能更快。当不同索引上有多个谓词时使用它。可以比较每列的每个位图，以创建所选元组的最终列表。

值范围	默认	设置分类
Boolean	on	master session reload

enable_groupagg

启用或禁用遗传查询优化器（计划器）使用组聚集计划类型

值范围	默认	设置分类
Boolean	on	master session reload

enable_hashagg

启用或禁用遗传查询优化器（计划器）使用哈希聚集计划类型。

值范围	默认	设置分类
Boolean	on	master session reload

enable_hashjoin

启用或者禁用遗传查询优化器（计划器）使用散列聚集计划类型。

值范围	默认	设置分类
Boolean	on	master session reload

enable_indexscan

启用或禁用遗传查询优化器（计划器）使用索引扫描计划类型。

值范围	默认	设置分类
Boolean	on	master session reload

enable_mergejoin

启用或禁用遗传查询优化器（计划器）使用合并连接计划类型。合并连接是基于左右表的顺序排序，然后并行扫描它们的想法。因此，两种数据类型必须能够被完全排序，并且连接操作符必须是只能在排序顺序中位于“相同位置”的值对的连接操作符号。在实践中，这意味着连接操作符具有相等的性质。

值范围	默认	设置分类
Boolean	off	master session reload

enable_nestloop

启用或禁用遗传查询优化器（计划器）使用嵌套循环连接计划。完全禁止嵌套循环连接是不可能的，但是如果有其他方法可用，则关闭此变量可能会使得遗传优化器放弃使用该方法。

值范围	默认	设置分类
Boolean	off	master session reload

enable_seqscan

启用或者禁用遗传查询优化器（计划器）使用顺序扫描类型。完全禁止顺序扫描是不可能 的，但是如果有其他方法可用，则关闭此变量将阻止遗传查询优化器使用该方法。

值范围	默认	设置分类
Boolean	on	master session reload

enable_sort

启用或禁用遗传查询优化器（计划器）使用显式的排序步骤。完全禁止显式排序是不可能的，但是，如果有其他可用的方法，关闭此变量将阻止遗传查询优化器使用该方法。

值范围	默认	设置分类
Boolean	on	master session reload

enable_tidscan

启用或者禁止遗传查询优化器（计划器）使用元组标识符。

值范围	默认	设置分类
Boolean	on	master session reload

escape_string_warning

打开的时候，如果在普通字符串文字（‘...’语法）中出现反斜杠（\）,则会发出警告。转义字符语法（E'...'）应用于转义，因为在将来的版本中，普通字符串将具有字面上处理反斜杠的符合SQL标准的行为。

值范围	默认	设置分类
Boolean	on	master session reload

explain_pretty_print

确定 EXPLAIN VERBOSE 是否使用缩进或非缩进格式显示详细的查询树存储。

值范围	默认	设置分类
Boolean	on	master session reload

extra_float_digits

调整浮点值显示的位数，包括float4，float8， 和几何数据类型。将参数将加到数位上。该值可以设置为高达2，包括部分有效位。这对于转储需要精确恢复的浮点数据尤其有用。或者用以用来设置为负摒弃不需要的位。

值范围	默认	设置分类
整数	0	master session reload

filerep_mirrorvalidation_during_resync

该默认设置值 false 在段镜像增量重新同步期间可以提供数据库性能。设置为 true 可以在增量重新同步期间检查段镜像上所有关系的文件是否存在。检查文件会降低增量重新同步过程的性能，但是提供了对数据库对象的最小检查。

值范围	默认	设置分类
Boolean	false	master session reload

from_collapse_limit

如果生成的 FROM 列表不超过这么多项，则遗传查询优化器（计划器）将把子查询合并到上层查询中。较小的值会较少计划时间，但是可能会产生较差的查询计划。

值范围	默认	设置分类
1- *n*	20	master session reload

gp_adjust_selectivity_for_outerjoins

启用对外连接的NULL测试的选择性。

值范围	默认	设置分类
Boolean	on	master session reload

gp_analyze_relative_error

设置表的基数的估计可接受的误差，0.5应该等于可50%的接受的误差（这是PostgreSQL中使用的默认值）。如果在 ANALYZE 期间收集的统计数据没有对特定表属性产生良好的基数估计，减少相对误差值（接受较少的错误）告诉系统对更多行进行采样。

值范围	默认	设置分类
浮点 < 1.0	0.25	master session reload

gp_appendonly_compaction

在 VACUUM 命令期间启用压缩段文件。当禁用时， VACUUM 只会将段文件清为EOF值，与当前的行为一样。管理员可能希望在高I/O负载情况或低空闲空间的情况下禁用压缩。

值范围	默认	设置分类
Boolean	on	master session reload

gp_appendonly_compaction_threshold

当在没有指定FULL选项情况下运行VACUUM时，指明隐藏行和总行的阀值比率（以百分比表示），该比率会触发段文件的压缩。如果段中的段文件中的隐藏行的比例小于该阀值，则段文件不能压缩，并且发出日志消息。

值范围	默认	设置分类
整数 (%)	10	master session reload

gp_autostats_mode

指定使用 ANALYZE触发自动统计信息收集的模式。on_no_stats 选项可以触发对任何没有统计信息的表上的 CREATE TABLE AS SELECT，INSERT，或 COPY 操作的统计信息收集。

当受影响的行数超过由 gp_autostats_on_change_threshold 定义的阀值时，on_change 选项才会触发统计信息收集。可以使用 on_change 触发自动统计信息收集的操作有：

```
CREATE TABLE AS SELECT

UPDATE

DELETE

INSERT

COPY
```

默认值是 on_no_stats。

> 注意： 对于分区表来说，如果从分区表的顶级父表插入数据，则不会触发自动统计信息收集。

如果数据直接插入到分区表的叶表（数据的存储位置）中，则触发自动统计信息收集。统计数据仅在叶表上收集。

值范围	默认	设置分类
none on_change on_no_stats	on_no_ stats	master session reload

gp_autostats_mode_in_functions

指定使用过程语言函数中的 ANALYZE 语句触发自动统计信息收集的模式。 none 选项禁用统计信息收集。 on_no_stats 选项在任何没有现有统计信息表上的函数中执行的 CREATE TABLE AS SELECT，INSERT，或 COPY 操作触发统计信息收集。

只有当受影响的行数超过由gp_autostats_on_change_threshold定义的阀值时，on_change 选项才会触发统计信息收集。可以使用 on_change 触发自动信息统计收集功能的操作有：

```
CREATE TABLE AS SELECT

UPDATE

DELETE

INSERT

COPY
```

值范围	默认	设置分类
none on_change on_no_stats	none	master session reload

gp_autostats_on_change_threshold

当 gp_autostats_mode 设定为 on_change时，指明自动统计信息收集的阀值。当触发表操作影响超过此阀值的行数时，将添加ANALYZE 并收集表的统计信息。

值范围	默认	设置分类
整数	2147483647	master session reload

gp_backup_directIO

直接 I/O 允许 HashData 数据库绕过文件系统缓存中的内存缓冲区进行备份。当直接 I/O 用于文件时，数据将直接从磁盘传输到应用程序缓冲区，而不使用文件缓冲区缓存。

仅在 Red Hat Enterprise Linux，CentOS，和SUSE上支持直接I/O。

值范围	默认	设置分类
on, off	off	local session reload

gp_backup_directIO_read_chunk_mb

当使用 gp_backup_directIO启用直接I/O时，以MB为单位设置块的大小。默认的块大小是20MB。

默认值是最佳设置。减少该值会增加备份时间，增加该值会导致备份时间几乎不变。

值范围	默认	设置分类
1-200	20 MB	local session reload

gp_cached_segworkers_threshold

当用户启动与 HashData 数据库的会话并发出查询时，系统将在每个段上创建工作进程的组或”帮派“，以进行工作。完成工作以后，除了由此参数设置的缓存数字外，段工作进程将被销毁。较低的设置节省了段主机上的系统资源，但更高的设置可能会提高高级用户（希望在一行中发出许多复杂查询）的性能。

值范围	默认	设置分类
整数 > 0	5	master session reload

gp_command_count

显示主机从客户端收到的命令数量。请注意，单个SQLcommand可能在内部实际涉及多个命令，因此对于单个查询，计数器可能会增加多个命令。该计数器也由在命令上执行的所有段进程共享。

值范围	默认	设置分类
整数 > 0	1	read only

gp_connection_send_timeout

在查询处理期间发送数据到无响应的 HashData 数据库用户客户端的超时时间。值为0将禁用超时， HashData 数据库将无限期地等待客户端。到达超时时，将使用此消息取消查询：

无法向客户端发送数据：连接超时。

值范围	默认	设置分类
秒数	3600（1小时）	master system reload

gp_connections_per_thread

当处理SQL查询时，调度工作以查询段实例上的执行段实例上的执行程序进程时，控制 HashData 数据库查询调度程序（QD）生成的异步线程（工作线程）的数量。当处理查询时，该值设置工作线程连接到的主段实例的数量。例如，当值是2的时并且有64个段实例时，QD将生成32个工作线程来调度查询计划工作。每个线程分派给2个段。

对于默认值0，查询分派器生成两种类型的线程：管理查询计划工作分派的主线程和互连线程。主线程也作为工作线程。

对于大于0的值，QD 生成3中类型的线程：主线程，一个或者多个工作线程和互连线程。当该值等于或者大于段实例的数量时，QD将生成3个线程：主线程，单个工作线程和互连线程。

不需要改动该默认值，除非有已知的吞吐性能问题。

该参数仅适用于主机，并更改它需要需要重新启动服务器。

值范围	默认	设置分类
整数 >= 0	0	master restart

gp_content

如果是段的话则为本地内容。

值范围	默认	设置分类
整数		read only

gp_create_table_random_default_distribution

使用不包含DISTRIBUTED BY子句的CREATE TABLE 或 CREATE TABLE AS 创建 HashData 数据库表时，控制表的创建。

对于 CREATE TABLE，如果参数的值为 off（默认值），并且表创建命令不包含 DISTRIBUTED BY 子句，则 HashData 数据库将根据该命令选择分布键。如果表创建命令中指定了 LIKE 或者 INHERITS 子句，则创建的表使用与源表或父表相同的分布键。

如果该参数值设置为真 on，当未指定DISTRIBUTED BY子句时， HashData 数据库遵循以下规则创建表：

- 如果 PRIMARY KEY 或者 UNIQUE 列未指定，则该表的分布式随机的（DISTRIBUTED RANDOMLY）。即使创建表的命令中包含 LIKE 或 INHERITS 子句，表分布也是随机的。
- 如果指定了 PRIMARY KEY 或者 UNIQUE 列，还必须指定 DISTRIBUTED BY 子句。如果没有指定 DISTRIBUTED BY 子句作为表创建命令的一部分，则该命令失败。

对于不包含分布子句的 CREATE TABLE AS 命令：

- 如果遗传查询优化器创建表，并且参数的值为 off，则表分布策略将根据该命令确定。
- 如果遗传查询优化器创建表，并且参数的值为 on，则表分布策略是随机的。
- 如果 GPORCA 创建表，则表分布策略是随机的，该参数无效。

更多关于遗传查询优化器和GPORCA的信息，请参阅 HashData 数据库管理员指南的“查询数据”。

值范围	默认	设置分类
boolean	off	master system reload

gp_dbid

如果是段，为本地内容的dbid。

值范围	默认	设置分类
整数		read only

gp_debug_linger

在致命内部错误之后， HashData 进程停留的秒数。

值范围	默认	设置分类
任何有效的时间表达式（数字和单位）	0	master session reload

gp_default_storage_options

当使用 CREATE TABLE 命令创建表之后，为以下的表存储选项设置默认值。

- APPENDONLY
- BLOCKSIZE
- CHECKSUM
- COMPRESSTYPE
- COMPRESSLEVEL

- ORIENTATION

以逗号分隔列表指定多个存储选项值。

用户可以使用此参数设置存储选项，而不是在CREATE TABLE命令的WITH中指定表存储选项。使用 CREATE TABLE 命令指定的表存储选项将覆盖此参数指定的值。

并非存储选项值的所有组合都有效。如果指定的存储选项无效，则返回错误。有关表存储选项的信息，请参阅 CREATE TABLE 命令。

可以为数据库和用户设置默认值。如果服务器配置参数设置在不同的级别，则当用户登录到数据库并创建表时候，表存储值的优先顺序从最高到最低：

1. 在 CREATE TABLE 命令中用 WITH 子句或者 ENCODING 子句指定的值。
2. 使用 ALTER ROLE...SET 命令为用户设置的 gp_default_storage_options 的值。
3. ALTER DATABASE...SET 命令为用户创建的 gp_default_storage_options 的值。
4. 使用 gpconfig 实用功能为 HashData 数据库系统创建的 gp_default_storage_options 的值。

参数值不是累积的。例如，如果，参数指定数据库和用户登录的 APPENDONLY 和 COMPRESSTYPE 选项，并且设置参数以指定 ORIENTATION 选项的值，则将忽略在数据库级别设置的 APPENDONLY，和 COMPRESSTYPE 值。

此示例 ALTER DATABASE 命令为数据库mystest设置了默认的 ORIENTATION 和 COMPRESSTYPE 表存储选项。

```
ALTER DATABASE mytest SET gp\_default\_storage\_options = 'orientation=column, compresstype=rle\_type'
```

要在 mytest 数据库中使用面向列的表和RLE压缩方式创建一个追加优化表。用户需要仅在WITH子句中指定 APPENDONLY=TRUE。

该例子 gpconfig 实用程序命令为 HashData 数据库系统设置默认存储选项。如果用户为多表存储选项设置了默认值，则该值必须要用单引号括起来，然后再用双引号括起来。

```
gpconfig -c 'gp\_default\_storage_options' -v "'appendonly=true, orientation=column'"
```

此示例 gpconfig 实用程序命令显示参数的值。该参数的值必须在 HashData 数据库主机和所有段之间一致。

```
gpconfig -s 'gp\_default\_storage_options'
```

值范围	默认	**Set Classifications 1**			
APPENDONLY= TRUE \	FALSE BLOCKSIZE= integer between 8192 and 2097152 CHECKSUM= TRUE \	FALSE COMPRESSTYPE= ZLIB \	QUICKLZ2 \	RLE_TYPE \	NONE COMPRESSLEVEL= integer between 0 and 9 ORIENTATION= ROW \

> 注意： 1当参数使用 gpconfig 实用程序实用程序设置在系统级时，为set classification（集合分类）

gp_dynamic_partition_pruning

启用可以动态消除分区扫描的计划。

值范围	默认	设置分类
on/off	on	master session reload

gp_email_from

邮件地址用发送邮件警告，以该形式：

'username@example.com'

或

'Name \username@example\.com\'

值范围	默认	设置分类
string		master system reload superuser

gp_email_smtp_password

用于与SMTP服务器进行身份验证的密码/密令。

值范围	默认	设置分类
string		master system reload superuser

gp_email_smtp_server

用于发送电子邮件警报的SMTP服务器的完全限定域名或IP地址和端口。必须的格式为：

smtp_servername.domain.com:port

值范围	默认	设置分类
string		master system reload superuser

gp_email_smtp_userid

用于和SMTP服务器进行认证的用户id。

值范围	默认	设置分类
string		master system reload superuser

gp_email_to

分号（;）分隔的电子邮件地址列表接受电子邮件警报消息，格式为：‘username\@example.com’

或

'Name \username@example\.com\'

如果没有设置此参数，则禁用邮件警告。

值范围	默认	设置分类
string		master system reload superuser

gp_enable_adaptive_nestloop

在遗传查询优化器的执行时间内，启用使用名为“自适应嵌套式”的新类型的连接节点。这将导致遗传优化器相比嵌套连接更倾向于哈希连接，如果连接外侧的行数超过预先计算的阀值。这参数提高索引操作的性能，该遗传查询优化器以前喜欢慢的嵌套循环连接。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_agg_distinct

启用或者禁用两阶段聚合以计算单个不同合格的聚合。这仅适用于包含单个不同合格的聚合函数的子查询。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_agg_distinct_pruning

启用或者禁用三阶段聚合和连接来计算单个不同合格的聚合。这仅应用在包括一个或多个单个不同合格的聚合函数的子查询上。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_direct_dispatch

启用或者禁用针对访问单个段上的数据查询的目标查询计划的分派。打开时，目标行在单个段上的查询仅将他们的查询计划分配到该段（而不是所有段）。这显著地减少了限定查询的响应时间，因为没有涉及互连设置。直接分派的话确实需要主机上更多的CPU利用率。

值范围	默认	设置分类
Boolean	on	master system restart

gp_enable_exchange_default_partition

控制 ALTER TABLE 的EXCHANGE DEFAULT PARTITION 子句的可用性。 该参数默认值为 off。如果该子句在 ALTER TABLE 命令中指定了，则该子句不可用并且 HashData 数据库返回一个错误。

如果该值为 on， HashData 数据库返回一个警告声明由于默认分区中无效的数据，交换默认分区可能会导致错误的结果。

警告： 在更换默认分区之前，必须确保要交换表中的数据，新的默认分区对默认分区有效。例如，新的默认分区中的数据不能包含在分区表的其他叶子分区中有效的数据。否则，使用有GPORCA执行交换的默认分区的分区表的查询可能返回不正确的结果。

值范围	默认	设置分类
Boolean	off	master session reload

gp_enable_fallback_plan

允许使用禁用计划类型，当查询没有该类型不可行的时候。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_fast_sri

当设置为 on，该遗传查询优化器（计划器）计划单行插入，因此他们被直接送到了正确的段实例上（无需引导操作）。这很大的提高了当行插入语句的性能。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_gpperfmon

启用或者禁用数据收集代理，该代理为 HashData 命令中心填充 gpperfmon 数据库。

值范围	默认	设置分类
Boolean	off	local system restart

gp_enable_groupext_distinct_gather

启用或禁用向单个节点收集数据，以便在组扩展查询上计算分别不同合格的聚集。当这个参数和 gp_enable_groupext_distinct_pruning 都启用了，该遗传查询优化器（计划器）使用代价较小的计划。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_groupext_distinct_pruning

启用或者禁用三阶段聚集和连接以在组扩展查询上计算不同合格的聚合。通常，启用此参数将生成较便宜（代价小）的查询计划，该计划相较于存在的计划将会被遗传查询优化器所使用。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_multiphase_agg

启用或者禁用两或三阶段并行聚合方案的遗传查询优化器（计划程序）。此方法适用于具有聚合的任何子查询。如果 gp_enable_multiphase_agg 是off，则 gp_enable_agg_distinct 和 gp_enable_agg_distinct_pruning 将被禁用。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_predicate_propagation

当被启用时，该遗传查询优化器（计划器）会在表上分布键连接的地方将谓词应用于两个表的表达式。在进行连接前过滤这两个表（如果可以的话）会更有效率。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_preunique

启用 SELECT DISTINCT 查询的两阶段重复删除（不是 SELECT COUNT(DISTINCT)）。启用后，它会在移动之前增加一个额外的 SORT DISTINCT 计划点集。在distinct操作大大较少行数的情况下，这种额外的 SORT DISTINCT 比跨Interconnect发送行代价小的多。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_sequential_window_plans

如果启用，启用包含窗口函数调用的查询的非并行查询计划。如果关闭，将并行评估兼容窗口函数并重新加入结果。这是一个实验参数。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_relsize_collection

如果没有表的统计信息，则可以使遗传查询优化器（计划程序）使用表的估计大小（pg_relation_size 函数）。默认情况下，如果统计信息不可用，计划器将使用默认值来估计行数。默认行为可以提高查询计划时间，减少繁重工作负载中的资源的使用情况，但是可能导致次优计划。

值范围	默认	设置分类
Boolean	off	master session reload

gp_enable_sort_distinct

排序的时候启用删除的重复项。

值范围	默认	设置分类
Boolean	on	master session reload

gp_enable_sort_limit

在排序时启用 LIMIT 操作。当计划最多需要前 *limit_number* 行时候排序会更有效。

值范围	默认	设置分类
Boolean	on	master session reload

gp_external_enable_exec

启用或禁用在段主机上执行os命令或脚本的外部表的使用（CREATE EXTERNAL TABLE EXECUTE 语法）。如果使用 Command Center或 MapReduce功能，必须启用。

值范围	默认	设置分类
Boolean	on	master system restart

gp_external_max_segs

设置在外部表操作期间将扫描外部表数据段的数量，目的是不使系统因扫描数据过载，并从其他并发操作中夺取资源。这仅适用于使用 the gpfdist:// 协议来访问外部表数据的外部表。

值范围	默认	设置分类
整数	64	master session reload

gp_filerep_tcp_keepalives_count

在连接被认为死亡之前，可能会丢失多少个keepalives。值为0是默认使用系统默认值。如果不支持TCP_KEEPCNT，该参数必须是0。

对于主段和镜像段之间的所有连接，使用此参数。对于不在主段和镜像段之间的设置，请使用 tcp_keepalives_count。

值范围	默认	设置分类
丢失的keepalives数量	2	local system restart

gp_filerep_tcp_keepalives_idle

在空间连接上发送keepalive之间的秒数。值为0使用系统默认值。如果不支持TCP_KEEPIDLE ，该参数必须是0。

对于主段和镜像段之间的所有连接，使用此参数。对于不再主段和镜像段之间的设置，请使用 tcp_keepalives_idle。

值范围	默认	设置分类
秒数	1 分钟	local system restart

gp_filerep_tcp_keepalives_interval

在重新传输之前等待响应 keepalive 的秒数。值为 0 使用默认值。如果不支持 TCP_KEEPINTVL ，则此参数必须为0。

对于主段和镜像段之间的所有连接，使用此参数，对于不在主镜像和镜像段之间的设置，请使用 tcp_keepalives_interval。

值范围	默认	设置分类
秒数	30 秒	local system restart

gp_fts_probe_interval

指定故障检测过程的轮询间隔（ftsprobe）。 该 ftsprobe 进程大概需要此量的时间来检测分段故障。

值范围	默认	设置分类
10 - 3600 秒	1min	master system restart

gp_fts_probe_retries

在报告段失败之前，指定故障检测进程（ftsprobe）尝试连接到段的次数。

值范围	默认	设置分类
整数	5	master system restart

gp_fts_probe_threadcount

指定要创建的 ftsprobe 线程数。该参数应设置为等于或大于每个主机的段数。

值范围	默认	设置分类
1 - 128	16	master system restart

gp_fts_probe_timeout

指定故障检测过程（ftsprobe）允许的超时，以在声明它之前建立与段的连接。

值范围	默认	设置分类
10 - 3600 秒	20 秒	master system restart

gp_log_fts

控制故障检测进程（ftsprobe）写入日志文件的细节数量。

值范围	默认	设置分类
OFF TERSE VERBOSE DEBUG	TERSE	master system restart

gp_log_gang

控制写入到日志文件中的关于查询工作进程创建和查询管理的信息量。默认值为 OFF，不记录信息。

值范围	默认	设置分类
OFF TERSE VERBOSE DEBUG	OFF	master session restart

gp_gpperfmon_send_interval

设置 HashData 数据库服务进程将查询执行更新发送到用数据收集代理进程的频率，该代理进程为Cmmand Center填充 gpperfmon 数据库。 此期间隔期间执行的查询操作通过UDP发送到段监视代理。如果发现在长时间运行的复杂查询中丢了过多的UDP的数据包用户可以考虑增加此值。

值范围	默认	设置分类
任何有效的时间表达式（数字和单位）	1sec	master system restart

gpperfmon_log_alert_level

控制哪些消息级别写入 gpperfmon 日志。每个级别包括跟随它的所有级别。级别越后，发送到日志的消息越少。

> 注意： 如果 gpperfmon 数据库已经安装并正在监视数据库，则默认值为警告。

值范围	默认	设置分类
none warning error fatal panic	none	local system restart

gp_hadoop_home

使用 Pivotal HD 时，指定 Hadoop 的安装目录。例如，该默认安装目录是 /usr/lib/gphd。

使用 HashData HD 1.2 或更早的版本时，请指定与 HADOOP_HOME 环境变量相同的值。

值范围	默认	设置分类
有效的目录名	HADOOP_HOME 的值	local session reload

gp_hadoop_target_version

HashData Hadoop 目标的安装版本。

值范围	默认	设置分类
gphd-1.0 gphd-1.1 gphd-1.2 gphd-2.0 gphd-3.0 gpmr-1.0 gpmr-1.2 hadoop2 hdp2 cdh5 cdh3u2 cdh4.1	gphd-1.1	local session reload

gp_hashjoin_tuples_per_bucket

设置 HashJoin 操作使用的哈希表的目标密度。较小的值将倾向于产生较大的哈希表，这可能增加连接性能。

值范围	默认	设置分类
整数	5	master session reload

gp_idf_deduplicate

改变计算和处理 MEDIAN 和 PERCENTILE_DISC 的策略。

值范围	默认	设置分类
auto none force	auto	master session reload

gp_initial_bad_row_limit

对于参数值 *n* ，当用户使用 COPY 命令或从外部表导入数据时，如果处理的前 *n* 行包括格式错误，则 HashData 数据库停止处理输入行。如果前 *n* 行中有处理的有效的行， HashData 数据库将继续处理输入行。

设置该值为 0 来禁止该限制。

也可以为 COPY 命令或者外部表定义指定 SEGMENT REJECT LIMIT 子句来限定被拒绝的行。

INT_MAX 是最大能够存储在用户系统上的 integer 值。

值范围	默认	设置分类
整数 0 - INT_MAX	1000	master session reload

gp_interconnect_fc_method

指定用于默认的 HashData 数据库 UDPIFC 互连的流控制方法。

对于基于容量的流量控制，当接收机不具有容量时，发送方不发送数据包。

基于丢失的流量控制是基于基于容量的流量控制的，并根据包的丢失情况调整发送速度。

值范围	默认	设置分类
CAPACITY LOSS	LOSS	master session reload

gp_interconnect_hash_multiplier

设置由 HashData 数据库用于具有默认 UDPIFC 互连的互连连接的哈希表的大小。该数字乘以段数，以确定哈希表中的桶数。增加该值会增加复杂多片层查询的互连性能（同时在片段主机上占用更多内存）。

值范围	默认	设置分类
2-25	2	master session reload

gp_interconnect_queue_depth

设置由接收器上的 HashData 数据库互连排队的每个对等体的数据量（当数据被接收但没有空间可用于接收数据时，数据将被丢弃，并且发送器将需要重新发送），用于默认的 UDPIFC 互连。增加深度的默认值将导致系统使用多的内存，但可能提高性能。将此值设置在 1 到 10 之间是合理的。具有数据偏移的查询可能会随着队列深度的增加而更好地执行。增加这个值可能会大大增加系统使用的内存量。

值范围	默认	设置分类
1-2048	4	master session reload

gp_interconnect_setup_timeout

指定等待 HashData 数据库互连在超时之前完成设置的时间量。

值范围	默认	设置分类
任何有效的时间表达式（数字和单位）	2 hours	master session reload

gp_interconnect_snd_queue_depth

设置发送方默认的 UDPIFC 互连排队的每个对等端口上的数据量。增加深度默认值将导致系统使用更多内存，但是可能会提高性能。该参数的合理值在1到4之间。增加值可能会大大增加系统使用的内存量。

值范围	默认	设置分类
1 - 4096	2	master session reload

gp_interconnect_type

设置用于 HashData 数据库互连流量的网络协议。UDPIFC 指定使用UDP与互连流量的流量控制，并且这是唯一支持的值。

UDPIFC（默认值）指定使用UDP和互连流量的流量控制。使用 gp_interconnect_fc_method 指定互连流量控制方法。

使用TCP作为互连协议， HashData 数据库具有 1000 非分段实例的上限-如果查询包含复杂的多分片查询，上限就会小于该值。

值范围	默认	设置分类
UDPIFC TCP	UDPIFC	local system restart

gp_log_format

指定服务器日志文件的格式。如果使用 *gp_toolkit* 管理模式，日志文件必须是 CSV 格式。

值范围	默认	设置分类
csv text	csv	local system restart

gp_max_csv_line_length

将导入到系统的 CSV 格式文件中的行的最大长度。默认值为 1MB（1048576 字节）。最大允许为 4MB（4194184 字节）。如果使用 *gp_toolkit* 管理模式读取 HashData 数据库日志文件，则可能需要增加默认值。

值范围	默认	设置分类
字节数	1048576	local system restart

gp_max_databases

在 HashData 数据库系统中允许的最大数据库数。

值范围	默认	设置分类
整数	16	master system restart

gp_max_filespaces

HashData 数据库系统中允许的最大文件空间数。

值范围	默认	设置分类
整数	8	master system restart

gp_max_local_distributed_cache

将分布式实务日志条目的最大数量设置到段实例的后台进程内存的 cache 中。

日志条目包含正在被 SQL 语句访问的行状态的信息。在 MVCC 环境中执行多个同时进行的 SQL 语句时，该信息用于决定那些行对于 SQL 实务是可见的。通过提高行可见性确定进程的性能，本地缓存分布式实务日志条目可以提高事务处理速度。

默认值对于不同的 SQL 处理环境是最佳的。

值范围	默认	设置分类
整数	1024	local system restart

gp_max_packet_size

设置 HashData 数据库互连的元组序列块大小。

值范围	默认	设置分类
512-65536	8192	master system restart

gp_max_plan_size

指定查询执行计划的总的最大未压缩的大小乘以计划中移动操作符（切片）的数量。如果查询计划的大小超过该值，则查询将被取消，并返回错误。值为0表示不监视计划的大小。

用户可以指定一个以 kB，MB，或者 GB。默认单位是 kB。例如，值 200 是 200kB。值 1GB 等同于 1024MB 或者 1048576kB。

值范围	默认	设置分类
整数	0	master superuser session

gp_max_tablespaces

HashData 数据库系统允许的最大表空间数量。

值范围	默认	设置分类
整数	16	master system restart

gp_motion_cost_per_row

设置移动操作符的遗传查询优化器（计划器）的成本估计，该移动操作符将一个行从一个段传输到另一个段，以顺序页面提取的成本为单位。如果为 0，则使用的值是 *cpu_tuple_cost* 的两倍。

值范围	默认	设置分类
浮点	0	master session reload

gp_num_contents_in_cluster

HashData 数据库系统主段数目。

值范围	默认	设置分类
-	-	read only

gp_reject_percent_threshold

对于 COPY 和外部表 SELECT 上的单行错误处理，设置在 SEGMENT REJECT LIMIT *n* PERCENT 开始计算之前处理的行数。

值范围	默认	设置分类
1- *n*	300	master session reload

gp_reraise_signal

如果启用，如果发生致命的服务器错误，将尝试转存内核。

值范围	默认	设置分类
Boolean	on	master session reload

gp_resqueue_memory_policy

启用 HashData 内存管理的功能。分布算法 eager_free 利用了一个事实，该事实是不是所有的操作符都在统一个时间执行。查询计划被分成几个阶段， HashData 数据库将在该阶段结束时急速释放分配给前一个阶段的内存，然后将新的内存分配给新阶段。

当设置为 none，内存管理和 HashData 数据库 4.1 之前的版本一样。

当设置为 auto，查询内存使用情况由 statement_mem 和资源队列内存限制来控制。

值范围	默认	设置分类
none, auto, eager_free	eager_free	local system restart/reload

gp_resqueue_priority

启用或者禁用查询优先级。禁用此参数时，不会再查询运行时评估现有的优先级设置。

值范围	默认	设置分类
Boolean	on	local system restart

gp_resqueue_priority_cpucores_per_segment

指定每个段实例分配的 CPU 单元数。例如，如果 HashData 数据库集群具有 4 个段的 10 个核心段主机，则将段实例的值设置为 2.5。对于主实例，该值将为 10。主机上通常只有主实例运行在用户其上，因此主机的值应反应所有可用 CPU 内核的使用情况。

不正确的设置可能导致 CPU 利用率不足或查询优先级不能按照设计工作。

HashData Data Computing Appliance V2 上的段默认值是4，而主机上是25。

值范围	默认	设置分类
0.1 - 512.0	4	local system restart

gp_resqueue_priority_sweeper_interval

指定清理进程评估当前 CPU 使用情况的间隔。当新语句变成活动状态时，将在下一个时间间隔到来之前完成优先级评估和 CPU 共享的决定。

值范围	默认	设置分类
500 - 15000 ms	1000	local system restart

gp_role

该服务进程的角色 '' 对主机设置为 *dispatch* ，对段设置为 *execute* 。

值范围	默认	设置分类
dispatch execute utility		read only

gp_safefswritesize

指定用于非成熟文件系统中追加优化表的安全写入操作的最小大小。当指定大于 0 的字节数时候，追加优化写入程序将附加数据添加到该数上，以防止由于文件系统错误导致的数据损坏。当使用具有该类型的文件系统的 HashData 数据库时，每个非成熟文件系统都具有已知的安全写入大小，该大小必须在这里指定。这通常是设置为文件系统的大小的倍数；例如，Linux ext3 是 4096 字节，所以值 32768 时经常使用的。

值范围	默认	设置分类
整数	0	local system restart

gp_segment_connect_timeout

在超时之前， HashData 互连将尝试通过网络连接到段实例的时间。控制主段（master）和主要段（primary）之间的网络连接超时，以及主要段（primary）到镜像段之间的复制进程。

值范围	默认	设置分类
任何有效的时间表达式 （数字和单位）	10min	local system reload

gp_segments_for_planner

假设在其成本和大小估计中，设置遗传查询优化器（计划器）主要段实例数。如果为0，则使用的值是实际的主要段数。该变量影响传统优化器对移动操作符中每个发送和接受进程处理的行数的估计。

值范围	默认	设置分类
0- *n*	0	master session reload

gp_server_version

以字符串报告服务器的版本号。版本修饰符参数可能会追加到版本字符串的数字部分，例如： *5.0.0beta* 。

值范围	默认	设置分类
String. Examples: *5.0.0*	n/a	read only

gp_server_version_num

以数字的形式报告服务器的版本号。该数保证永远对每个版本都是增加的，可以用来数字比较。主要版本是表示为两位数，次要和补丁版本是填充0。

值范围	默认	设置分类
Mmmpp where *M* is the major version, *mm* is the minor version zero-padded and *pp* is the patch version zero-padded. Example: 50000	n/a	read only

gp_session_id

系统为客户端会话分配ID号。当主实例首次启动时，从1开始计数。

值范围	默认	设置分类
1- *n*	14	read only

gp_set_proc_affinity

如果启动，当 HashData 服务器（postmaster）开始时，它会绑定到CPU。

值范围	默认	设置分类
Boolean	off	master system restart

gp_set_read_only

设置为 on 以禁用对数据库的写入。任何正在进行的事务必须必须在只读模式生效之前完成。

值范围	默认	设置分类
Boolean	off	master system restart

gp_snmp_community

设置为用户的环境指定的社区名称。

值范围	默认	设置分类
SNMP community name	public	master system reload

gp_snmp_monitor_address

用户网络监视程序的主机名：端口。通常，端口是 162。如果有多个监视器地址，以逗号来分隔。

值范围	默认	设置分类
hostname:port		master system reload

gp_snmp_use_inform_or_trap

陷阱通知是一个应用程序发送到另一个应用程序之间的 SMTP 消息（例如，在 HashData 数据库和网络监视应用程序之间）。这些消息不被监视应用程序所确认，但会产生更少的网络开销。

提醒通知和陷阱消息一样，除了应用程序会生成警告的应用程序发送确认。

值范围	默认	设置分类
inform trap	trap	master system reload

gp_statistics_pullup_from_child_partition

当使用遗传查询优化器（计划器）对父表进行计划查询的时，启用子表统计信息的使用。

值范围	默认	设置分类
Boolean	on	master session reload

gp_statistics_use_fkeys

当启用时，允许遗传查询优化器（计划器）使用存储在系统目录中的外键信息来优化在外键和主键之间的连接。

值范围	默认	设置分类
Boolean	off	master session reload

gp_vmem_idle_resource_timeout

如果一个数据库会话空间时间超过指定时间，会话将释放系统资源（如共享内存），但仍然保持连接到数据库。这允许一次并发连接更多的连接到数据库。

值范围	默认	设置分类
任何有效的时间表达式（数字和单位）	18s	master system reload

gp_vmem_protect_limit

设置活跃段实例的所有 postgres 进程可以使用的内存量（以 MB 为单位）。如果查询超出该限制，则不会分配内存，并且查询将失败。请注意，这是本地参数，必须为系统中的每个段（主要段和镜像段）设置。设置参数值时，仅指定数值。例如，要指定 4096MB，请使用 4096。不要将单位 MB 添加到该值。

为了防止内存的过度分配，这些计算可以估计一个安全的 gp_vmem_protect_limit 值。

首先计算 gp_vmem的值。这是主机上可用的 HashData 数据库内存

```
gp_vmem = ((SWAP \+ RAM) - (7.5GB + 0.05 * RAM)) / 1.7
```

其中 SWAP 是主机交换空间，RAM 是主机上的 RAM，以 GB 为单位。

接下来，计算 max_acting_primary_segments。这是当镜像段由于故障而被激活时，可以在主机上运行的主要段（primary）的最大数量。对于每个主机，配置有4个主机块，8个段实例。例如，单个段主机故障将，在故障的主机模块每个剩余主机上，激活2或3个镜像段。此配置的 max_acting_primary_segments 值为11（8个主要段加上3个故障后激活的镜像段）。

这是 gp_vmem_protect_limit的计算。该值应转化为MB。

```
gp\_vmem\_protect_limit = gp_vmem / acting\_primary\_segments
```

对于生成大量工作文件的情况，这是负责工作文件 gp_vmem 的计算。

```
gp_vmem = ((SWAP \+ RAM) - (7.5GB + 0.05 * RAM \- (300KB * total_#_workfiles))) / 1.7
```

有关监视和管理工作文件使用情况的信息，请参阅 HashData 数据库管理员指南。

基于 gp_vmem 值用户可以计算 vm.overcommit_ratio 操作系统的内核参数。当配置每个 HashData 数据库主机时，此参数被设置。

```
vm.overcommit_ratio = (RAM \- (0.026 * gp_vmem)) / RAM
```

> 注意： 该内核参数 vm.overcommit_ratio 在 Red Hat Enterprise Linux 中默认值是50。

有关内核参数的信息，请参阅 HashData 数据库安装指南。

值范围	默认	设置分类
整数	8192	local system restart

gp_vmem_protect_segworker_cache_limit

如果查询执行器进程的消耗此配置的数量，则该进程将不会被缓存以备在进程完成后的后续查询中使用。具有大量连接或空闲进程的系统可能希望减少此数量以释放段上的更多内存。请注意，这是一个本地参数，必须为每个段设置。

值范围	默认	设置分类
兆字节数	500	local system restart

gp_workfile_checksumming

向 HashAgg 和 HashJoin 查询操作符使用的工作文件（或益出文件）的每个块添加校验和值。这增加了从错误的OS磁盘驱动程序将错误的块写入磁盘的额外保护措施。当校验和操作失败时，查询将取消并回滚，而不是潜在地将数据写入磁盘。

值范围	默认	设置分类
Boolean	on	master session reload

gp_workfile_compress_algorithm

当查询处理期间散列聚合或散列连接操作溢出到磁盘时，指定要在溢出文件上要使用的压缩算法。如果使用zlib，它必须在所有段的 `$PATH` 中。

如果用户的 HashData 数据库安装使用串行 ATA (SATA) 磁盘驱动器，则将此参数的值设置为 zlib 可能有助于避免IO操作超载磁盘子系统。

值范围	默认	设置分类
none zlib	none	master session reload

gp_workfile_limit_files_per_query

设置每个段每个查询允许的临时溢出文件（也称工作文件）的最大数量。当执行需要比分配内存更多的内存的查询时，会创建溢出文件。当超过限制时，当前查询终止。

将值设置为 0（零）以允许无限数量的溢出文件。主会话重新加载。

值范围	默认	设置分类
整数	100000	master session reload

gp_workfile_limit_per_query

设置单个查询允许用于在每个段创建临时溢出文件的最大磁盘大小。默认值为0，这意味着不强制执行限制。

值范围	默认	设置分类
千字节	0	master session reload

gp_workfile_limit_per_segment

设置允许所有运行查询用于在每个段创建临时溢出文件的最大磁盘大小。默认值为0，这意味着不强制执行限制。

值范围	默认	设置分类
千字节	0	local system restart

gpperfmon_port

设置所有数据收集代理（对于CommandCenter）与主机通信的端口。

值范围	默认	设置分类
整数	8888	master system restart

整数_datetimes

报告 PostgreSQL 是否支持64位整数日期和时间。

值范围	默认	设置分类
Boolean	on	read only

IntervalStyle

设置间隔值的显示形式。 *sql_standard* 值产生输出匹配 SQL 标准间隔字面值。当 DateStyle 参数设置为ISO时， *postgres* 值产生输出匹配PostgreSQL8.4之前的发行版本。

当 DateStyle 参数设置为 non-ISO 输出时， *postgres_verbose* 值产生输出匹配 HashData 3.3之前的发行版本。

iso_8601 值产生与ISO 8601第4.4.3.2节中定义的 *format with designators* 时间间隔相匹配的输出。有关详细信息，请参阅 PostgreSQL 文档 。

值范围	默认	设置分类
postgres postgres_verbose sql_standard iso_8601	postgres	master session reload

join_collapse_limit

每当列表总数不超过这么多项目时，该遗传查询优化器（计划器）将会重写明确的内部 JOIN 结构到 FROM 项目的列表中。默认的是，这个变量与 *from_collapse_limit* 相同，这适用于大多数用途，将此变量设置为1可防止内部链接重新排序。将此变量设置为1和 *from_collapse_limit* 之间的值可能有助于根据所选计划的质量来衡量计划时间。

值范围	默认	设置分类
1- *n*	20	master session reload

keep_wal_segments

对于 HashData 数据库主镜像，如果检查点操作发生，则设置由活跃的 HashData 数据库主机保存的最大的WAL段文件数。

段文件用来在备用主机上同步活跃的主机。

值范围	默认	设置分类
整数	5	master system reload superuser

krb_caseins_users

设置Kerberos用户名是否应该被区分大小写。默认值是区分大小写（off状态）。

值范围	默认	设置分类
Boolean	off	master system restart

krb_server_keyfile

设置 Kerberos 服务器秘钥（key）文件的位置。

值范围	默认	设置分类
路径和文件名	unset	master system restart

krb_srvname

设置 Kerberos 服务名称。

值范围	默认	设置分类
服务名	postgres	master system restart

lc_collate

报告文本数据排序完成的场所。该值是在 HashData 数据库数组初始化时确定的。

值范围	默认	设置分类
\<依赖于系统>		read only

lc_ctype

报告确定字符分类的范围。该值是在 HashData 数据库数组初始化时确定的。

值范围	默认	设置分类
\<依赖于系统>		read only

lc_messages

设置显示消息的语言。可用的范围取决于用户的操作系统安装的内容-使用 *locale -a* 来列出所有可用的范围。默认值是从服务器执行环境中继承的。在某些系统上，此设置类型不存在。设置此变量仍然有效，但是不会有任何影响。此外，有一个机会没有翻译信息所需语言的存在，这种情况下，用户讲继续看到英文讯息。

值范围	默认	设置分类
\<依赖于系统>		local system restart

lc_monetary

设置用于格式化金额的区域设置（部分，locale），例如使用 *to_char* 函数系列。可用的区域设置取决于用户安装操作系统的内容 -使用 *locale -a* 可以列出可用的区域设置。默认值从服务器的执行环境继承。

值范围	默认	设置分类
\<依赖于系统>		local system restart

lc_numeric

设置用于格式化数字的区域设置，例如使用 *to_char* 函数系列。可用的区域设置取决用户的操作系统安装的内容-使用 *locale -a* 来列出所有可用的区域设置。默认值是从服务器的执行环境继承。

值范围	默认	设置分类
\<依赖于系统>		local system restart

lc_time

该参数目前没有什么用，但是将来可能会有用。

值范围	默认	设置分类
\<依赖于系统>		local system restart

listen_addresses

指定服务器要监听客户端程序的TCP/IP地址- 逗号分隔的主机名和，或数字IP地址。特殊的项 * 对应于所有可用的IP接口。如果列表为空，则只能连接UNIX域的套接字。

值范围	默认	设置分类
localhost, host names, IP addresses, * (all available IP interfaces)	*	master system restart

local_preload_libraries

逗号分隔的共享库文件的列表，该列表将在客户端会话开始时预加载。

值范围	默认	设置分类
		local system restart

log_autostats

记录有关 gp_autostats_mode 和 gp_autostats_on_change_threshold的自动 ANALYZE 操作的信息的日志。

值范围	默认	设置分类
Boolean	off	master session reload superuser

log_connections

这将向服务器日志输出一条详细说明每个成功连接的详细信息。某些客户端程序（向psql）在确定是否需要密码时尝试连接两次，因此重复的"连接接收"消息并不总是指示着问题。

值范围	默认	设置分类
Boolean	off	local system restart

log_disconnections

这将在客户端会话终止时在服务器日志中输出一行，并包括会话的持续时间。

值范围	默认	设置分类
Boolean	off	local system restart

log_dispatch_stats

当设置为"on"时候，该参数添加一条有关语句发送的详细信息的日志消息。

值范围	默认	设置分类
Boolean	off	local system restart

log_duration

会导致每个满足 *log_statement* 的已完成语句的持续时间都会被记录。

值范围	默认	设置分类
Boolean	off	master session reload superuser

log_error_verbosity

控制记录在每个消息的服务器日志中写入的详细信息量。

值范围	默认	设置分类
TERSE DEFAULT VERBOSE	DEFAULT	master session reload superuser

log_executor_stats

对于每个查询，将查询执行程序的性能统计信息写入服务器日志。这是一个粗糙的剖析仪器，无法和 *log_statement_stats* 一起使用。

值范围	默认	设置分类
Boolean	off	local system restart

log_hostname

默认情况下，连接日志消息仅显示连接主机和IP地址。打开此选项可以记录 HashData 数据库主机的IP地址和主机名。请注意，根据用户主机名决定设置，这可能会施加一个不可忽视的性能惩罚，无法和 *log_statement_stats* 一起使用。

值范围	默认	设置分类
Boolean	off	master system restart

log_min_duration_statement

如果其持续时间大于或等于指定的毫秒数，则将该语句及其持续时间记录子单个日志行上。将其设置为0会打印所有语句和它们的持续时间。例如，如果用户设置其为250，然后所有执行时间大于250ms的SQL语句将会记录到日志上。启用该选项可用于在用户的应用中追踪未优化查询。

值范围	默认	设置分类
毫秒数, 0, -1	-1	master session reload superuser

log_min_error_statement

控制是否会在服务器日志中记录引发错误情况的SQL的语句。所有导致指定级别或者更高级别错误的SQL语句都被记录。默认值为PANIC（有效的关闭此功能以正常使用）。启用此选项有助于追踪出现在服务器日志中的任何错误的来源。

值范围	默认	设置分类
DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 INFO NOTICE WARNING ERROR FATAL PANIC	ERROR	master session reload superuser

log_min_messages

控制那些消息级别写入服务器日志。每个级别包括跟随它（它之后）的所有级别，级别越靠后，发送到日志的消息越少。

值范围	默认	设置分类
DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 INFO NOTICE WARNING LOG ERROR FATAL PANIC	WARNING	master session reload superuser

log_parser_stats

对于每个查询，将查询解析器的性能统计信息写入服务器日志。这是一个粗糙的剖析仪器。无法与 *log_statement_stats* 一起使用。

值范围	默认	设置分类
Boolean	off	master session reload superuser

log_planner_stats

对于每个查询，将遗传查询优化器（计划程序）的性能统计信息写入服务器日志。这是一个粗糙的剖析仪器。无法与 *log_statement_stats* 一起使用。

值范围	默认	设置分类
Boolean	off	master session reload superuser

log_rotation_age

确定单个日志文件的最长生命周期，在这段时间之后，将创建一个新的日志文件。设置为0以禁用基于时间的日志文件的创建。

值范围	默认	设置分类
任何有效的时间表达式（数字和单位）	1d	local system restart

log_rotation_size

确定单个日志文件的最大大小。这么多千字节被发射到日志文件之后，将创建一个新的日志文件。设置为0以禁用基于大小的新的日志文件的创建。

最大值为 INT_MAX/1024。如果指定了无效值，则使用默认值。INT_MAX 是可以作为整数存储在系统上的最大值。

值范围	默认	设置分类
千字节数	0	local system restart

log_statement

控制记录那些SQL语句。DDL记录所有数据定义命令，如 CREATE，ALTER，和 DROP 命令。MOD记录所有DDL语句，加上 INSERT，UPDATE，DELETE，TRUNCATE和COPY FROM。如果它们包含的命令是适当的类型，则还会记录 PREPARE 和 EXPLAIN ANALYZE 语句。

值范围	默认	设置分类
NONE DDL MOD ALL	ALL	master session reload superuser

log_statement_stats

对于每个查询，将查询解析器，计划程序和执行程序的总体性能统计信息写入服务器日志。这是一个粗糙的剖析仪器。

值范围	默认	设置分类
Boolean	off	master session reload superuser

log_timezone

设置用于记录在日志中的时间戳使用的时区。与 TimeZone不同，该值是系统范围内的，因此所有会话将一致的报告时间戳。默认值是 unknown，这意味着使用系统环境指定的时区作为时区。

值范围	默认	设置分类
string	unknown	local system restart

log_truncate_on_rotation

清空（覆盖），而不是附加到任何现有的同名的日志文件。清空将仅在由于基于时间的轮换打开新文件时，才会发生。例如，将此设置与log_filename（例如 gpseg#-%H.log）组合将导致生成二十四小时日志文件，然后循环覆盖他们。关闭时，在所有情况下，将附加在预先存在的文件。

值范围	默认	设置分类
Boolean	off	local system restart

max_appendonly_tables

设置可以写入或更新追加优化表的并发事务的最大数量。超过最大数量的事务将返回错误。

计数的操作是 INSERT，UPDATE， COPY 和 VACUUM 操作。该限制仅限于正在进行的事务。一旦事务结束（终止或者提交），它不再计入此限制。

对于针对分区表的操作，作为追加优化表并被更改的每个子分区（字表）被计为最大值的单个表。例如，分区表 p_tbl 被定义为具有追加优化表 p_tbl_ao1，p_tbl_ao2，和 p_tbl_ao3的三个自分区。针对改变追加优化表 p_tbl_ao1 和 p_tbl_ao2的分区表p_tbl 的 INSERT 或 UPDATE 命令被计为两个事务。

增加该限制值，将在服务器开始时，在主机上分配更多的共享内存。

值范围	默认	设置分类
整数 > 0	10000	master system restart

max_connections

与数据库服务器并发连接的最大数量。在 HashData 数据库系统中，用户客户端仅连通 HashData 主实例。段实例应允许的数量应该是主实例数量的5-10倍。增加此参数时，max_prepared_transactions 也必须要增加。更多关于限制并发连接的信息，参阅 *HashData* 数据库管理员指南的“配置客户端身份验证”。

增加此参数可能会导致 HashData 数据库要求更多的共享内存。有关 HashData 服务器实例共享内存缓存区的信息，请参阅 shared_buffers。

值范围	默认	设置分类
10- *n*	Master上是250 Segment上是750	local system restart

max_files_per_process

设置允许每个服务器子进程同时打开文件的最大数量。如果内核正在执行每个进程的安全限制，则不需要担心此设置。一些平台，例如BSD，内核将允许单个进程打开比系统真正支持的更多文件数目。

值范围	默认	设置分类
整数	1000	local system restart

max_fsm_pages

设置在共享空闲内存映射中追踪的可用空间的最大磁盘页数。每个页面槽都占用了6个字节的共享内存。

值范围	默认	设置分类
整数 > 16 * *max_fsm_relations*	200000	local system restart

max_fsm_relations

设置在共享内存空间映射中可追踪空闲空间的最大关系数。应设置为大于总数的值：

表 + 索引 + 系统表。

对于每个段实例的每个关系，它花费了大约60个字节的内存。最好放大一点空间，设置高一点而不是太低。

值范围	默认	设置分类
整数	1000	local system restart

max_function_args

报告函数参数的最大数目。

值范围	默认	设置分类
整数	100	read only

max_identifier_length

报告最大标识符长度。

值范围	默认	设置分类
整数	63	read only

max_index_keys

报告索引键的最大数目。

值范围	默认	设置分类
整数	32	read only

max_locks_per_transaction

共享锁表的创建带有描述在 *max_locks_per_transaction* * (*max_connections* + *max_prepared_transactions*) 对象上的锁，所以在一次不能将多于这么多的不同的对象锁住。这不是任何一个事务所占锁数量的硬限制，而是最大平均值。如果客户端在单个事务中涉及多张不同的表，用户可能需要提升该值。

值范围	默认	设置分类
整数	128	local system restart

max_prepared_transactions

设置能同时处于准备状态的最大事务数。 HashData 在内部使用准备好的事务来确保数据跨段的完整性。该值至少和主机上的 max_connections 值一样大。段实例应设置与主机相同的值。

值范围	默认	设置分类
整数	250 on master 250 on segments	local system restart

max_resource_portals_per_transaction

设置每个事务允许同时打开用户声明游标的最大数量。请注意，打开的游标将在资源维持一个查询节点。用于工作负载管理。

值范围	默认	设置分类
整数	64	master system restart

max_resource_queues

设置可在 HashData 数据库系统中创建的资源队列的最大数量。请注意，资源队列是系统范围内的（和角色一样），因此他们适用于系统中的所有数据库。

值范围	默认	设置分类
整数	9	master system restart

max_stack_depth

指定服务器栈的最大安全执行深度。此参数的理想设置是内核执行的实际的栈大小限制（由 *ulimit -s* 或本地等效设置），少于一个兆字节的安全边距。将参数设置为高于实际内核限制将意味着失控的递归函数可能导致单个后端进程崩溃。

值范围	默认	设置分类
千字节数	2MB	local system restart

max_statement_mem

设置查询的最大内存限制。将 statement_mem 设置得太高，有助于在查询处理期间避免段主机上的内存不足的错误。当 gp_resqueue_memory_policy=auto时， statement_mem 和 资源队列内存限制了控制查询内存的使用。考虑到单个段主机的配置，计算该设置如下：

(seghost_physical_memory) / (average_number_concurrent_queries)

值范围	默认	设置分类
千字节数	2000MB	master session reload superuser

optimizer

运行SQL查询时启用或禁用GPORCA。 默认值为 on。如果禁用GPORCA， HashData 数据库仅使用遗传查询优化器。

GPORCA 和遗传查询优化器共存。启用GPORCA后， HashData 数据库使用GPORCA在可能时为查询生成执行计划。如果GPORCA不可用，则使用遗传查询优化器。

可以为数据库系统，单个数据库，会话或查询 optimizer（优化器）参数。

有关遗传查询优化器和GPORCA的信息，请参阅 HashData 数据库管理员指南中的“查询数据”。

值范围	默认	设置分类
Boolean	on	master session reload

optimizer_analyze_root_partition

对于分区表，当在表上运行 ANALYZE 命令时收集根分区的统计信息。 GPORCA 使用根分区统计信息来生成一个查询计划。而遗传查询优化器并不使用这些数据。如果用户设置服务器配置参数 optimizer 的值为on，请将此参数设置为on，并在分区表上运行ANALYZE 或 ANALYZE ROOTPARTITION命令来确保收集到正确的统计信息。

有关遗产查询优化器和GPORCA的信息，请参阅 HashData 数据库管理员指南的“查询数据”。

值范围	默认	设置分类
Boolean	off	master session reload

optimizer_array_expansion_threshold

当启用（默认值）GPORCA时候并且正在处理包含具有常量数组的谓词的查询时，该 optimizer_array_expansion_threshold 参数将根据数组中的常量数限制优化过程。如果当前查询谓词中的数组包含多于参数指定的数字元素，则GPORCA在查询优化期间禁用将谓词转换为其分离的正常格式。

默认值为25。

例如，当GPORCA正在执行超过25个元素的 IN 子句查询时，GPORCA 在查询优化期间不会将谓词转化为其分离的正常格式，以减少优化时间，消耗更小的内存。查询处理的差异可以在查询EXPLAIN 计划的IN子句的过滤条件中看到。

更改此参数的值会换来更小的优化时间和更少的内存消耗，以及在查询优化期间的约束导出的潜在优势，例如冲突监测和分区消除。

可以为数据库系统，单个数据库，会话或查询设置该参数。

值范围	默认	设置分类
Integer > 0	25	master session reload

optimizer_control

控制是否可以使用SET，RESET 命令或 HashData 数据库实用程序 gpconfig更改服务器配置参数优化程序。如果 optimizer_control 参数值为 on，则用户可以设置 optimizer 参数。如果 optimizer_control 参数值为 off，则 optimizer 参数不能被改变。

值范围	默认	设置分类
Boolean	on	master system restart superuser

optimizer_cte_inlining_bound

启用GPORCA（默认值）时, 此参数控制对公共表达式（CTE）查询（包含 WHERE 子句的查询）执行的内联量。默认值为“0”禁用内联。

可以为数据库系统，单个数据库或会话或查询设置该参数。

值范围	默认	设置分类
Decimal >= 0	0	master session reload

optimizer_enable_master_only_queries

启用GPORCA（默认值）时，此参数允许GPORCA执行仅在 HashData 数据库主机上运行的目录查询。对于默认值 off，只有遗传查询优化器才能执行仅在 HashData 数据库主机上运行的目录查询。

可以为数据库系统，单个数据库会会话或查询设置该参数。

注意： 启用此参数会降低运行中的短目录查询的性能。为了避免此问题，请仅为会话或查询设置此参数。

有关GPORCA的信息，请参阅 HashData 数据库管理员指南。

值范围	默认	设置分类
Boolean	off	master session reload

optimizer_force_multistage_agg

对于默认设置，启用GPORCA并且此参数为 true，当生成此类计划备选项时，GPORCA 会选择标量不同的合格聚合的3个阶段聚合计划。当值为 false时，GPORCA 会基于代价作出选择，而不是启发式选择。

可以为数据库系统，单个数据库，会话或查询设置该参数。

值范围	默认	设置分类
Boolean	true	master session reload

optimizer_force_three_stage_scalar_dqa

对于默认设置，启用GPORCA并且此参数为 true，当生成此类计划备选方案时，GPORCA 会选择具有多级聚合的计划。当值为false时，GPORCA会基于成本作出选择，而不是启发式选择。

可以为数据库系统，单个数据库，会话或查询设置该参数。

值范围	默认	设置分类
Boolean	true	master session reload

optimizer_join_order_threshold

当启用GPORCA（默认值）时，该参数设置连接子节点的最大数量，GPORCA 将使用基于动态编程连接排序算法为该子节点排序。用户可以为单个查询或整个会话设置此值。

值范围	默认	设置分类
0 - INT_MAX	10	master session reload

optimizer_mdcache_size

设置GPORCA在查询优化期间用于缓存查询元数据（优化数据）的 HashData 数据库主机上的最大内存量。基于内存限制会话。GPORCA 使用默认设置在查询优化期间缓存元数据：启用GPORCA并且 optimizer_metadata_caching 为 on。

默认值为 16384 (16MB)。这是通过性能分析确定的最佳值。

用户可以以KB，MB，或GB为单位指定值。默认单位是KB。例如，值16384 为 16384KB。值1GB 等同为 1024MB 或 1048576KB。如果值为0，则缓存大小不收限制。

可以为数据库系统，单个数据库，会话或查询设置此参数。

值范围	默认	设置分类
Integer >= 0	16384	master session reload

optimizer_metadata_caching

当启用GPORCA（默认值）时，该参数指定在查询优化期间，GPORCA是否在 HashData 数据库主机的内存中缓存查询元数据（优化数据）。 该参数的默认值为 on，启用缓存。该缓存是基于会话的。当会话结束时，如果查询元数据量超过缓存大小，则旧的未使用的元数据将从缓存中释放。

如果该值为 off，GPORCA 在查询优化期间不缓存元数据。

可以为数据库系统，单个数据库，会话或查询设置此参数。

服务器配置参数 optimizer_mdcache_size 控制查询元数据缓存的大小。

值范围	默认	设置分类
Boolean	on	master session reload

optimizer_minidump

GPORCA 生成 minidump 文件来描述给定查询的优化上下文。文件中的信息不是可以用来方便地调试和排除故障的格式。minidump 文件位于主数据目录下，并使用以下命名格式：

Minidump_date_time.mdp

minidump 文件包含查询相关的信息：

- 目录对象包括GPORCA要求的数据类型，表格，操作符和统计信息。
- 查询的内部表示（DXL）。
- GPORCA制定的计划的内部表示（DXL）。
- 传递给GPORCA的系统配置信息，如服务器配置参数，成本和统计信息配置，以及段的数量。
- 在优化查询时，生成的错误堆栈跟踪。

将此参数设置为 ALWAYS 会为所有查询生成一个minidump。将此参数设置为ONERROR 以最小化总优化时间。

关于GPORCA的信息，参阅 HashData 数据库管理员指南的“查询数据”。

值范围	默认	设置分类
ONERROR ALWAYS	ONERROR	master session reload

optimizer_nestloop_factor

启用GPORCA（默认值）时，该参数控制在查询优化期间应用的嵌套循环连接代价因子，该默认值为 1，指定了默认排序代价因子。该值表示的是从默认因子中减去或者增加的一个比率。例如，2.0 将代价因子设置为默认值的2倍。此外，值 0.5 将代价因子设置为默认值的一半。

可以为数据库系统，单个数据库，会话或查询设置该参数。

值范围	默认	设置分类
Decimal > 0	1	master session reload

optimizer_parallel_union

启用GPORCA（默认值）时，optimizer_parallel_union 控制对包含 UNION 或 UNION ALL 子句的查询的并行化的数量。

当该值为 off时，默认值GPORCA会生成一个查询计划，其中APPEND(UNION) 操作符的每个子节点与在同一个片段，该片段和APPEND操作符一样。 在查询执行期间，子节点以顺序的方式执行。

当该值是 on，GPORCA 生成一个查询计划，其中再分配移动节点在 APPEND（UNION）操作符之下。在查询优化期间，子节点和父APPEND操作符在不同的片段上，允许APPEND（UNION）操作符的子进程在段实例上并行执行。

可以为数据库系统，单个数据库，会话或查询设置该参数。

值范围	默认	设置分类
Boolean	off	master session reload

optimizer_print_missing_stats

当启用 GPORCA（默认值），该参数控制关于查询缺少统计信息的列的表列信息的显示。默认值为 true，将列信息显示给客户端。当为 false，该信息不回发送到客户端。

信息在查询执行期间显示，或使用 EXPLAIN 或 EXPLAIN ANALYZE 命令显示。

可以为数据库系统，单个数据库会会话设置该参数。

值范围	默认	设置分类
Boolean	true	master session reload

optimizer_print_optimization_stats

当启用GPORCA（默认值）时，该参数可以启用对查询各种优化阶段的GPORCA查询优化统计信息的日志记录。默认值为 off，不记录优化统计信息。要记录优化统计信息，必须将此参数设置为 on ，并将参数 client_min_messages 必须设置为 log。

- set optimizer_print_optimization_stats = on;
- set client_min_messages = 'log';

在查询执行期间或使用 EXPLAIN 或 EXPLAIN ANALYZE 命令记录信息。

可以为数据库系统，单个数据库，会话或查询设置此参数。

值范围	默认	设置分类
Boolean	off	master session reload

optimizer_sort_factor

当启用GPORCA（默认值）时，optimizer_sort_factor 控制在查询优化期间应用于排序操作的代价因子。该默认值为 1 指定了默认排序的代价因子。该值是在默认因子增加或者减少的比率。例如，值 2.0 将成本因子设置为默认值的2倍，值 0.5 将成本因子设置为默认值的一半。

可以为数据库系统，单个数据库，会话或查询设置该参数。

值范围	默认	设置分类
Decimal > 0	1	master session reload

password_encryption

当在 CREATE USER 或 ALTER USER 中指定密码而没有写入 ENCRYPTED 或 UNENCRYPTED字段时，该参数决定是否对密码进行加密。

值范围	默认	设置分类
Boolean	on	master session reload

password_hash_algorithm

指定在存储加密的Greeenplum数据库用户密码时使用的加密散列算法。默认的算法是MD5。

有关设置密码散列算法以保护用户密码的信息，请参阅 HashData 数据库管理员指南中的“在 HashData 数据库中保护密码”。

值范围	默认	设置分类
MD5 SHA-256	MD5	master session reload superuser

pgstat_track_activity_query_size

设置存储在系统目录视图pg_stat_activity中的 current_query列中的查询文本的最大长度限制。最小长度为1024字节。

值范围	默认	设置分类
整数	1024	local system restart

pljava_classpath

冒号（：）分隔的jar文件或包含PL/Java函数所需的jar文件的目录的列表。必须指定jar文件或目录的完整路径，除非 `$GPHOME/lib/postgresql/java` 目录中的jar文件可以省略路径。jar文件必须安装在所有 HashData 主机上的相同位置，并可由 gpadmin 用户读取。

pljava_classpath 参数用于在每个用户会话开始时组合 PL/Java 类路径。会话启动后添加的jar文件不可用于该会话。

如果在 pljava_classpath 完整的jar文件的路径，则将其添加到PL/Java 类路径。当指定了一个目录，该目录包含任何jar文件都会被添加到PL/Java 类路径。搜索不会下降到指定目录的子目录中。如果 pljava_classpath 中包含的jar文件没有路径，则该jar文件必须在 `$GPHOME/lib/postgresql/java` 目录中。

> 注意： 如果有很多目录要搜索，或者有很多的 jar 文件，这将会影响性能。

如果 pljava_classpath_insecure 为 false，设置 pljava_classpath 参数需要超级用户权限。当代码由没有超级用户权限的用户执行时，在SQL代码中设置类路径将会失败。该 pljava_classpath 参数必须先前由超级用户或在 postgresql.conf 文件中设置。在 postgresql.conf 文件中改变类路径需要重载（gpstop -u）。

值范围	默认	设置分类
string		master session reload superuser

pljava_classpath_insecure

控制服务器配置参数 pljava_classpath 是否可以由用户设置，而无需Geenplum数据库的超级用户权限。当为真 true 时，pljava_classpath 可以由常规用户设置。否则，pljava_classpath 只能由数据库超级用户设置。 默认值为 false。

警告： 启用此参数给非管理员数据库用户运行未经授权的Java方法可能的风险。

值范围	默认	设置分类
Boolean	false	master session restart superuser

pljava_statement_cache_size

为准备语句设置 JRE MRU（最近常使用的）缓存的大小（KB）。

值范围	默认	设置分类
千字节数	10	master system restart superuser

pljava_release_lingering_savepoints

如果为真，PL/Java 函数中使用的保留点将在函数退出时释放。如果为 false，则保留点将会被回滚。

值范围	默认	设置分类
Boolean	true	master system restart superuser

pljava_vmoptions

定义Java VM的启动选项。默认值是空的字符串（“”）。

值范围	默认	设置分类
string		master system restart superuser

port

一个 HashData 数据库实例的监听端口。主站和每个段都有自己的端口。必须在 gp_segment_configuration 目录中更改 HashData 系统的端口号。用户必须在更改端口号之前关闭用户的 HashData 数据库系统。

值范围	默认	设置分类
any valid port number	5432	local system restart

random_page_cost

设置遗传查询优化器（计划程序）的非连续读取的磁盘页面的成本估计。这是以顺序页面提取的成本的倍数来衡量的。更高的值使得更可能使用顺序扫描，较低的值使得更可能使用索引扫描。

值范围	默认	设置分类
浮点	100	master session reload

readable_external_table_timeout

当SQL从外部表读取时候，参数值指定当数据停止从外部表返回时， HashData 数据库在取消查询之前等待查询的时间（以秒为单位）。

默认值为 0，指明没有时间限制， HashData 数据库不会取消查询。

如果使用gpfdist的查询运行了很长时间，然后返回错误“间歇性网络连接问题”，则用户可以为readable_external_table_timeout指定一个值。如果gpfdist在指定的时间内没有返回任何数据，则 HashData 将取消查询。

值范围	默认	设置分类
整数 >= 0	0	master system reload

repl_catchup_within_range

对 HashData 数据库主镜像，控制对活跃主机的更新。如果没有由 walsender 处理的WAL段文件的数量超过此值，则 HashData 数据库会更新活跃的主机。

如果段文件的数量没有超过此值，则 HashData 数据库将阻止更新以允许 walsender 处理文件。如果所有的WAL段都被处理了，则更新活跃的主机。

值范围	默认	设置分类
0 - 64	1	master system reload superuser

replication_timeout

对 HashData 数据库主机镜像，设置活跃主机上的 walsender 进程等待备用主机上的walreceiver 进程的状态消息的最大时间。如果没有收到信息，walsender 会记录一条错误信息。

wal_receiver_status_interval 控制 walreceiver 状态信息之间的间隔。

值范围	默认	设置分类
0 - INT_MAX	60000 ms (60 seconds)	master system reload superuser

regex_flavor

‘扩展’设置可能有助于与 PostgreSQL7.4 之前的版本精确的向后兼容。

值范围	默认	设置分类
advanced extended basic	advanced	master session reload

resource_cleanup_gangs_on_wait

如果通过资源队列提交语句，则在资源队列进行锁定之前，请清理任何空闲的查询执行工作进程。

值范围	默认	设置分类
Boolean	on	master system restart

resource_select_only

设置资源队列管理的查询类型。如果设置为on，则会对 SELECT，SELECT INTO，CREATE TABLE AS SELECT 和 DECLARE CURSOR 进行评估。如果设置为off，则也会对 INSERT，UPDATE 和 DELETE 命令也将被评估。

值范围	默认	设置分类
Boolean	off	master system restart

runaway_detector_activation_percent

设置触发终止查询的 HashData 数据库vmem内存的百分比。如果用 HashData 数据库段的vmem内存的百分比超过了指定的值，那么 HashData 会根据内存的使用情况终止查询，从消耗内存最大量的查询开始。直到被使用的vmem的百分比低于指定的百分比为止，查询才被终止。

使用服务器配置参数 gp_vmem_protect_limit指定活跃的 HashData 数据库段实例的最大vmem值。

例如，如果vmem 内存设置到10GB，并且 runaway_detector_activation_percent 为 90（90%），当使用的vmem内存量超过9GB时， HashData 数据库将禁用查询的自动终止。

值为0禁用基于使用vmem百分比的查询。

值范围	默认	设置分类
percentage (integer)	90	local system restart

search_path

当在没有模式组件的简单名称引用对象时，指定模式被搜索的顺序。当在不同的模式中存在同名的对象，将使用在搜索路径中首先找到的对象。系统目录模式， *pg_catalog* ，始终被搜索，无论是否在路径中提及。当创建对象没有指定目标模式时，它们会被放在搜索路径的第一个模式中。可以通过SQL函数 *current_schemas()* 检查搜索路径当前的有效值。*current_schemas()* 显示如何解决出现在 *search_path* 的请求。

值范围	默认	设置分类
a comma- separated list of schema names	`$user,public`	master session reload

seq_page_cost

对于遗传查询优化器（计划器），设置作为一系列顺序提取的一部分的磁盘页面提取的成本估计。

值范围	默认	设置分类
浮点	1	master session reload

server_encoding

报告数据库编码（字符集）。确定 HashData 数据库数组何时被初始化。通常，客户端只需关心 *client_encoding* 的值。

值范围	默认	设置分类
\<依赖于系统>	UTF8	read only

server_version

报告该 HashData 数据库发行版基于的 PostgreSQL 的版本。

值范围	默认	设置分类
string	8.3.23	read only

server_version_num

报告该 HashData 数据库发行版基于的PostgreSQL的整数版本。

值范围	默认	设置分类
整数	80323	read only

shared_buffers

设置 HashData 数据库段实例用户共享内存的内存量。此设置必须至少为128KB并且至少16KB倍的 max_connections数。

每个 HashData 数据库段实例根据段配置计算并尝试分配一定量的共享内存。 shared_buffers 是共享内存计算的重要部分，但不是全部。当设置 shared_buffers时，操作系统参数 SHMMAX 或 SHMALL 的值可能也需要调整。

操作系统参数 SHMMAX 指定了单个共享内存分配的最大大小。 SHMMAX 的值必须大于此值：

```
shared_buffers \+ other\_seg\_shmem
```

other_seg_shmem 的值是 HashData 数据库共享内存计算的部分，这不是由 shared_buffers 值所负责的。
other_seg_shmem 的值将根据段配置而有所不同。

使用默认的 HashData 数据库参数的值， other_seg_shmem 的值，对于 HashData 数据库段大约是111MB，而对于主机大约是79MB。

该操作系统参数 SHMALL 指定主机上共享内存的最大数量。SHMALL 必须大于此值：

```
(num\_instances\_per_host \* ( shared_buffers \+ other\_seg\_shmem )) \+ other\_app\_shared_mem
```

other_app_shared_mem 值是主机上其他应用程序和进程使用的共享内存量。

当共享内存分配出现错误，解决共享内存分配问题的可能方法是增加 SHMMAX 或 SHMALL或 减少 shared_buffers 或 max_connections。

有关参数 SHMMAX 和 SHMALL的 HashData 数据库值的信息，请参考“ HashData 数据库安装指南”。

值范围	默认	设置分类
整数 > 16K * *max_connections*	125MB	local system restart

shared_preload_libraries

在服务器启动时预先加载的共享库的逗号分隔列表。 PostgreSQL过程语言可以这样预先加载，通常通过使用语法 `$libdir/plXXX` ，其中 XXX 是 pgsql, perl, tcl, 或 python。通过预先加载库，库首次使用时候，可以避免库启动时间。如果未找到指定的库，则服务器将无法启动。如果未找到指定的库，则服务器将无法启动。

值范围	默认	设置分类
		local system restart

ssl

启用 SSL 连接。

值范围	默认	设置分类
Boolean	off	master system restart

ssl_ciphers

指定允许在安全连接上使用的SSL密码列表。有关支持的密码列表，请参阅openssl手册页。

值范围	默认	设置分类
string	ALL	master system restart

standard_conforming_strings

确定普通字符串文字（'...'）是否按照SQL标准中的规定字面处理反斜杠。默认值为on。关闭此参数以将字符串文字中的反斜杠视为转义字符而不是字面反斜杠。应用程序可以检查此参数以确定如何处理字符串文字。此参数的存在也可以作为支持转义字符串语法（E'...'）的指示。

值范围	默认	设置分类
Boolean	on	master session reload

statement_mem

为每个查询分配主机内存。使用此参数分配的内存量不能超过 max_statement_mem 或查询提交的资源队列上的内存限制。当 gp_resqueue_memory_policy =auto时，statement_mem 和资源队列内存限制了控制查询内存的使用。

如果查询需要额外的内存，则会使用磁盘上的临时溢出文件。

该计算可用于估计各种情况下的合理值。

```
( gp\_vmem\_protect\_limitGB * .9 ) / max\_expected\_concurrent\_queries
```

将 gp_vmem_protect_limit 设置为 8192MB (8GB) 并假设最大40个并发查询和10%的缓冲区。

```
(8GB * .9) / 40 = .18GB = 184MB
```

值范围	默认	设置分类
千字节数	128MB	master session reload

statement_timeout

终止任何占据了指定毫秒数量的语句。0为关闭该限制。

值范围	默认	设置分类
毫秒数	0	master session reload

stats_queue_level

收集数据库活动的资源队列的统计信息。

值范围	默认	设置分类
Boolean	off	master session reload

superuser_reserved_connections

决定为 HashData 数据库超级用户保留的连接节点的数量。

值范围	默认	设置分类
整数 < *max_connections*	3	local system restart

tcp_keepalives_count

在连接被认为是死亡之前，可能会丢失的Keepalive的数量。0值使用系统默认值。如果不支持 TCP_KEEPCNT，则该参数必须为0。

对于不再主段和镜像段之间的所有连接，请使用此参数。对于主段和镜像段之间的设置，请使用gp_filerep_tcp_keepalives_count。

值范围	默认	设置分类
number of lost keepalives	0	local system restart

tcp_keepalives_idle

在空闲连接上发送keepalive之间的秒数。值为0使用系统默认值。如果不支持TCP_KEEPIDLE ，则此参数必须为0。

对于不再主段和镜像段之间的所有连接，请使用此参数。对于主段和镜像段之间的设置，请使用gp_filerep_tcp_keepalives_idle。

值范围	默认	设置分类
秒数	0	local system restart

tcp_keepalives_interval

在重新传输之前等待keepalive响应的秒数。值为0使用系统默认值。如果不支持TCP_KEEPINTVL，该参数必须是0。

对于不在主段和镜像段之间的所有连接，请使用此参数。对于主段和镜像段之间的设置，请使用gp_filerep_tcp_keepalives_interval。

值范围	默认	设置分类
秒数	0	local system restart

temp_buffers

设置每个数据库会话使用的临时缓冲区的最大数量。这些是仅用于访问临时表的会话本地缓冲区。该设置可以在单个会话中进行更改，但只能在会话首次使用临时表前进行更改。在实际上不需要大量临时缓冲区的会话中设置大值的代价只是每个增量的缓冲区描述符，或大约64字节。但是，如果实际使用了缓冲区，则会消耗额外的8192字节。

值范围	默认	设置分类
整数	1024	master session reload

TimeZone

设置显示和解释时间戳的时区。默认值是使用系统环境指定的时区作为时区。请参阅PostgreSQL 文档的 Date/Time Keywords。

值范围	默认	设置分类
time zone abbreviation	local	restart

timezone_abbreviations

设置服务器接受日期时间输入的时区缩写集合。默认值为 Default，它是世界上绝大多数地区都应用的集合。Australia 和 India 和为特定安装定义的其他集合。可能的值是存储在 `$GPHOME/share/postgresql/timezonesets/` 中的配置文件的名称。

要将 HashData 数据库配置为使用自定义的时区集合，请将包含时区定义的文件复制到 HashData Database主节点和分段主机上的 `$GPHOME/share/postgresql/timezonesets/` 目录中。然后将服务器配置参数 timezone_abbreviations 设置到文件中。例如，使用包含默认时区和WIB（Waktu Indonesia Barat）时区的 custom 文件。

1. 拷贝文件 Default 从目录 $GPHOME/share/postgresql/timezonesets/ 到文件 custom中。将 WIB 时区信息从文件 Asia.txt 添加到 custom文件中。
2. 复制 custom 文件到 HashData 数据库主和段主机上的目录 $GPHOME/share/postgresql/timezonesets/ 中。
3. 将服务器配置参数 timezone_abbreviations 设置为 custom。
4. 重新加载服务器配置参数（gpstop -u）。

值范围	默认	设置分类
string	Default	master session reload

track_activities

启用对每个会话当前执行命令的统计信息的收集以及该命令开始执行的时间。启用后，所有用户都不会看到此信息，该信息只对超级用户和用户该会话的用户可见。该数据可以通过 *pg_stat_activity* 系统视图访问。

值范围	默认	设置分类
Boolean	on	master session reload

track_counts

启用数据库活动上的行和块级统计数据的收集。如果启用，则可以通过 *pg_stat* 和 *pg_statio* 系列视图来访问生成的数据。

值范围	默认	设置分类
Boolean	off	local system restart

transaction_isolation

设置当前的事务的隔离级别。

值范围	默认	设置分类
read committed serializable	read committed	master session reload

transaction_read_only

设置当前事务的只读状态。

值范围	默认	设置分类
Boolean	off	master session reload

transform_null_equals

当为on时，表单表达式 expr = NULL（或 NULL = expr）被视为expr IS NULL，也就是说如果expr 计算为控制，则返回为true，否则返回为false。expr = NULL 的正确的SQL规范兼容行为是始终返回null（未知）。

值范围	默认	设置分类
Boolean	off	master session reload

unix_socket_directory

指定服务器上侦听来自客户端应用程序的连接的UNIX-域的套接字的目录。

值范围	默认	设置分类
目录路径	unset	local system restart

unix_socket_group

设置UNIX-域套接字的所属组。默认情况下是一个空字符串，该默认值使用当前用户的默认组。

值范围	默认	设置分类
UNIX组名	unset	local system restart

unix_socket_permissions

设置UNIX-域套接字的访问权限。UNIX-域套接字使用通常的UNIX文件系统的权限集合。请注意，对于UNIX-域套接字，只有写入权限才是最重要的。

值范围	默认	设置分类
数字形式的UNIX文件权限模式（ *chmod* 或者 *umask* 命令接受的形式）	511	local system restart

update_process_title

每当服务器接收到新的SQL命令时，都可以更新进程的标题。进程标题通常由 ps 命令查看。

值范围	默认	设置分类
Boolean	on	local system restart

vacuum_cost_delay

当超过代价限制时候，进程休眠的时间长度。0禁用基于时间的清理延迟功能。

值范围	默认	设置分类
milliseconds < 0 (in multiples of 10)	0	local system restart

vacuum_cost_limit

积累的代价值，该代价会导致清理进程的休眠。

值范围	默认	设置分类
整数 > 0	200	local system restart

vacuum_cost_page_dirty

当修改一个之前是干净的块时所估计的代价。它表示将脏块再次刷新到磁盘所需的额外I/O。

值范围	默认	设置分类
整数 > 0	20	local system restart

vacuum_cost_page_hit

基于共享缓存的缓冲区清理的估计代价。它代表锁定缓冲池，查找共享哈希表并扫描页面的内存的代价。

值范围	默认	设置分类
整数 > 0	1	local system restart

vacuum_cost_page_miss

清理必须从磁盘读取的缓冲区的估计代价。该代表了锁定缓冲池，查找共享哈希表，从磁盘读取所需块并扫描其内容的代价。

值范围	默认	设置分类
整数 > 0	10	local system restart

vacuum_freeze_min_age

指定事务中的截止年龄，当扫描表的时候，VACUUM 应该使用该年龄来决定是否使用 *FrozenXID* 替代事务ID。

有关 VACUUM 和事务ID管理的信息，请参阅 HashData 数据库管理员指南 的“管理数据”和 PostgreSQL 文档。

值范围	默认	设置分类
整数 0-100000000000	100000000	local system restart

validate_previous_free_tid

启用验证空闲元组ID（TID）列表的测试。该列表由 HashData 数据库维护和使用。 HashData 数据库通过确保当前空闲元

组的先前空闲TID是有效的空闲元组来确定TID列表的有效性。默认值为 true，启用该测试。

如果 HashData 数据库监测到空闲TID列表中的损害，则重建免费TID列表，记录警告，并且检查失败的查询返回警告。HashData 数据库尝试执行查询。

值范围	默认	设置分类
Boolean	true	master session reload

vmem_process_interrupt

在数据库查询执行期间，可以在为查询预留vmem内存之前检查中断。在为查询预留更多的vmem之前，请检查查询的当前会话是否有待处理的查询取消或其他挂起的中断。这确保了更多的响应中断处理，包括查询取消请求。默认值是off。

值范围	默认	设置分类
Boolean	off	master session reload

wal_receiver_status_interval

对于 HashData 数据库主镜像，请设置发送到活跃主机的 walreceiver 进程状态信息之间的间隔。在重型负载下，该值可能会更长。

该 replication_timeout 值控制了 walsender 进程 walreceiver 等待接收信息的时间。

值范围	默认	设置分类
整数 0- INT_MAX/1000	10 sec	master system reload superuser

writable_external_table_bufsize

HashData 数据库用于网络通信的缓冲区大小（以KB为单位），例如 gpfdist 实用程序和外部web表（实用http）。HashData 数据库在数据库数据写出之前将数据存储在缓冲区中。有关 gpfdist的信息，请参阅 HashData 数据库使用指南。

值范围	默认	设置分类
整数 32 - 131072 (32KB - 128MB)	64	local system reload

xid_stop_limit

发生事务ID环绕的ID之前的事务ID数。达到此限制时， HashData 数九将停止创建新事务，以避免由于事务ID环绕而导致的数据丢失。

值范围	默认	设置分类
整数 10000000 - 2000000000	1000000000	local system restart

xid_warn_limit

在xid_stop_limit指定的限制之前的事务ID数量。当 HashData 数据库达到此限制时，他会发出警告以执行VACUUM操作，来避免由于事务ID环绕而导致的数据丢失。

值范围	默认	设置分类
整数 10000000 - 2000000000	500000000	local system restart

xmlbinary

指定二进制值如何在XML数据中编码。例如，当 bytea 值转化为XML值时。该二进制数据可以转换为base64编码或者十六进制编码。默认值是base64。

可以为数据库系统，单个数据库或会话设置该参数。

值范围	默认	设置分类
base64 hex	base64	master session reload

xmloption

指定是否将XML数据视为执行隐式解析和序列化的操作的XML文档（document）或者 XML 内容片段（content）。默认值为content。

此参数影响 xml_is_well_formed()执行的验证。如果值为 document，则该函数检查格式良好的XML文档。如果值为content，则该函数将检查格式良好的XML内容片段。

注意： 包含文档类型声明（DTD）的XML文档不被视为有效的XML内容片段。如果将 xmloption 设置为 content， 则包含DTD的XML不被视为有效的XML。

。要将包含DTD的字符换转化为 xml 数据类型，请将 xmlparse 函数和 document 关键字一起使用，或将 xmloption 值改为document。

可以为数据库系统，单个数据库或会话设置参数。设置此选项的SQL命令也可以在 HashData 数据库中使用。

```
SET XML OPTION { DOCUMENT | CONTENT }
```

值范围	默认	设置分类
document content	content	master session reload

内置函数摘要

HashData 数据库支持内置函数和操作符，包括可用于窗口表达式的分析函数和窗口函数。有关使用内置 HashData 数据库函数的信息，参阅 HashData 数据库管理员指南中的 “使用函数和操作符”。

- HashData 数据库函数类型
- 内建函数和操作符
- JSON 函数和操作符
- 窗口函数
- 高级聚集函数

上级主题： HashData 数据库参考指南

HashData 数据库函数类型

HashData 数据库评估 SQL 表达式中使用的函数和操作符。一些函数和操作符只允许在主机上执行，因为它们可能会导致 HashData 数据库段实例不一致。本表介绍了 HashData 数据库函数类型。

表 **1. HashData** 数据库中的函数

函数类型	**HashData** 是否支持	描述	注释
IMMUTABLE	是	仅依赖于其参数列表中的信息。给定相同的参数值，始终返回相同的结果。	
STABLE	在大多数情况下是的	在单个表扫描中，对相同的参数值返回相同的结果，但结果将通过 SQL 语句进行更改。	结果取决于数据库查找或参数值。current_timestamp 系列函数是 STABLE; 值在执行中不会改变。
VOLATILE	受限制的	函数值可以在单个表扫描中更改。 例如: random(), currval(), timeofday().	任何具有副作用的函数的都不稳定的，即使其结果是可预测的。 例如: setval().

在 HashData 数据库中，数据通过字段被区分——没一个字段是一个独立的 Postgres 数据库。为了防止不一致或意外的结果，如果它们包含 SQL 命令或以任何方式修改数据库， 则不要在该子段级别执行分类为 VOLATILE 的函数。例如， 不允许在 HashData 数据库中的分布式数据上执行 setval() 等函数，因为它们可能导致段实例之间的数据不一致。

为了确保数据的一致性，用户可以在主服务器上计算和运行的语句中安全地使用 VOLATILE 和 STABLE 函数。 例如，以下语句在 master 上运行(没有 FROM 子句的语句):

```
SELECT setval('myseq', 201);
SELECT foo();
```

如果语句具有包含分布式表的 FROM 子句，并且 FROM 子句中的函数返回一组行，则语句可以在字段上运行：

```
SELECT * from foo();
```

HashData 数据库不支持返回表引用函数 (rangeFuncs) 或使用 refCursor 数据类型的函数。

内建函数和操作符

下表列出了 PostgreSQL 支持的内置函数和操作符的类别。除了 STABLE 和 VOLATILE 函数外, postgreSQL 中的所有函数和操作符在 HashData 数据库中都被支持，但是他们 受到 HashData 数据库函数类型 中所述的限制。 有关这些内置函数和操作符的更多信息，请参阅 PostgreSQL 文档的 Functions and Operators 部分。

表 2. 内建函数和操作符

操作符/函数类别	VOLATILE 函数	STABLE 函数	限制
逻辑操作符			
比较操作符			
数学函数和操作符	random setseed		
字符串函数和操作符	所有内建转换函数	convert pg_client_encoding	
二进制字符串函数和操作符			
位串函数和操作符			
模式匹配			
数据类型格式化功能		to_char to_timestamp	
日期/时间函数和操作符	timeofday	age current_date current_time current_timestamp localtime localtimestamp now	
枚举支持功能			
几何函数和操作符			
网络地址功能和操作符			
序列操作功能	currval lastval nextval setval		

条件表达式			
数组函数和操作符		所有数组函数	
聚合函数			
子查询表达式			
行和数组比较			
设置返回功能	generate_series		
系统信息功能		所有会话信息函数 所有访问特权查询函数 所有方案可见性查询函数 所有系统目录信息函数 所有注释信息函数 所有事务ID和快照	
系统管理功能	set_config pg_cancel_backend pg_reload_conf pg_rotate_logfile pg_start_backup pg_stop_backup pg_size_pretty pg_ls_dir pg_read_file pg_stat_file	current_setting 所有数据库对象尺寸函数	注意：函数 pg_column_size 显示存储值所需的字节，可能使用 TOAST 压缩。
XML函数和类似函数的表达式		xmlagg(xml) xmlexists(text, xml) xml_is_well_formed(text) xml_is_well_formed_document(text) xml_is_well_formed_content(text) xpath(text, xml) xpath(text, xml, text[]) xpath_exists(text, xml) xpath_exists(text, xml, text[]) xml(text) text(xml) xmlcomment(xml) xmlconcat2(xml, xml)	

JSON函数和操作符

用于创建和操作 JSON 数据的内置函数和操作符。

- JSON 操作符
- JSON 创建函数
- JSON 处理功能

> 注解： 对于 json 的值， 对象包含重复的键/值对，也保留所有键/值对。处理功能将最后一个值视为可操作的。

JSON 操作符

该表描述了可用于的操作符 json 的数据类型。

表 3. json 操作符

操作符	右操作数类型	描述	示例	示例结果
->	int	获取 JSON 数组元素（从零索引）。	'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2	{"c":"baz"}
->	text	通过键获取JSON对象字段。	'{"a": {"b":"foo"}}'::json->'a'	{"b":"foo"}
->>	int	获取JSON数组元素作为文本。	'[1,2,3]'::json->>2	3
->>	text	获取JSON对象字段作为文本。	'{"a":1,"b":2}'::json->>'b'	2
#>	text[]	在指定的路径获取JSON对象。	'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}'	{"c": "foo"}
#>>	text[]	以指定路径获取JSON对象作为文本。	'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'	3

JSON 创建函数

此表描述了创建的函数的 json 值。

表 4. JSON 创建函数

函数	描述	示例	示例结果
to_json(anyelement)	将值作为有效的 JSON 对象返回。数组和复合类型被递归处理并转换为数组和对象。 如果输入包含类型为 json 的造型, 则使用造型函数执行转换; 否则，将生成 JSON 标量值。对于除数字，布尔值或空值之外的任何标量类型，将使用文本表示，正确引用和转义，以使其为有效的 JSON 字符串。	`to_json('Fred said "Hi."'::text)`	`"Fred said "Hi.""`
array_to_json(anyarray [, pretty_bool])	将数组作为JSON数组返回。HashData 数据库的多维数组成为数组的JSON数组。如果 pretty_bool 是 true，则在第一个元素之间添加换行符。	array_to_json('{{1,5},{99,100}}'::int[])	`[[1,5],[99,100]]`
`row_to_json(record [,pretty_bool])`	将该行作为 JSON 对象返回。如果 elements if pretty_bool 是 true 则在第 1 级元素之间添加换行符。	`row_to_json(row(1,'foo'))`	`{"f1":1,"f2":"foo"}`

> 注解：除了提供精细打印选项外，array_to_json 和 row_to_json 具有与 to_json 相同的行为。 对于 to_json 描述的行为也适用于由其他 JSON 创建函数转换的单个值。

JSON 处理函数

此表描述处理 json 值的函数。

表 5. JSON 处理函数

函数	返回类型	描述	示例	示
json_each(json)	set of key text, value json set of key text, value jsonb	将最外层的JSON对象扩展为一组键/值对。	`select * from json_each('{"a":"foo", "b":"bar"}')`	a
json_each_text(json)	setof key text, value text	将最外层的JSON对象扩展为一组键/值对。返回的值是类型 text。	`select * from json_each_text('{"a":"foo", "b":"bar"}')`	a
json_extract_path(from_json json, VARIADIC path_elems text[])	json	返回指定为的JSON值path_elems。相当于 #> 操作符。	`json_extract_path('{"f2": {"f3":1},"f4": {"f5":99,"f6":"foo"}}','f4')`	{
json_extract_path_text(from_json json, VARIADIC path_elems text[])	text	返回指定为的JSON值path_elems作为文本。 相当于 #>> 操作符。	`json_extract_path_text('{"f2": {"f3":1},"f4": {"f5":99,"f6":"foo"}}','f4', 'f6')`	foo
json_object_keys(json)	setof text	返回最外层JSON对象中的键集。	`json_object_keys('{"f1":"abc","f2": {"f3":"a", "f4":"b"}}')`	j f2
json_populate_record(base anyelement, from_json json)	anyelement	扩展对象from_json 其列与由 base 定义的记录类型匹配。	`select * from json_populate_record(null::myrowtype, '{"a":1,"b":2}')`	a
json_populate_recordset(base anyelement, from_json json)	setof anyelement	扩展最外层的对象数组from_json 到一组行，其列与定义的记录类型 base 相匹配 。	`select * from json_populate_recordset(null::myrowtype, '[{"a":1,"b":2},{"a":3,"b":4}]')`	a
json_array_elements(json)	setof json	将JSON数组扩展为一组JSON值。	`select * from json_array_elements('[1,true, [2,false]]')`	v [2,

> 注意： 许多这些函数和操作符将 Unicode字符串中的 Unicode 转义转换为常规字符。这些函数为不能在数据库编码中表示的字符引发错误。

对于 json_populate_record 和 json_populate_recordset，从 JSON 类型强制是尽力而为，可能不会导致某些类型的期望值。JSON 键与目标行类型中的相同列名匹配。输出中不会出现未显示在目标行类型中的 JSON 字段， 并且不匹配任何 JSON 字段的目标列返回 NULL。

窗口函数

以下内置窗口函数是 PostgreSQL 数据库的 HashData 扩展。所有窗口函数都是不可变的。有关窗口函数的更多信息，请参阅 HashData 数据库管理员指南。

表 **6.** 窗口函数

函数	返回类型	完整语法	描述
cume_dist()	double precision	CUME_DIST() OVER ([PARTITION BY expr] ORDER BY expr)	计算一组值中的值的累积分布。具有相等值的行总是计算相同的累积分布值。
dense_rank()	bigint	DENSE_RANK () OVER ([PARTITION BY expr] ORDER BY expr)	计算有序的一组行中的行的排名，而不跳过排名。具有相等值的行具有相同的等级值。
first_value(expr)	same as input expr type	FIRST_VALUE(expr) OVER ([PARTITION BY expr] ORDER BY expr [ROWS RANGE frame_expr])	返回有序值集中的第一个值。
lag(expr [,offset] [,default])	same as input expr type	LAG(expr [, offset] [, default]) OVER ([PARTITION BY expr] ORDER BY expr)	提供访问同一个表的多行，而不进行自我连接。 给定从查询返回的一系列行和光标的位置， LAG在该位置之前提供对给定物理偏移量的行的访问。默认 offset 是1. 默认设置如果偏移量超出窗口范围则返回的值。 如果默认没被指定，默认值为null。
last_value(expr)	same as input expr type	LAST_VALUE(expr) OVER ([PARTITION BY expr] ORDER BY expr [ROWS RANGE frameexpr])	返回有序值集中的最后一个值。
lead(expr [,offset] [,default])	same as input expr type	LEAD(expr [,offset] [,expr default]) OVER ([PARTITION BY expr] ORDER BY expr)	提供访问同一个表的多行，而不进行自我连接。给定从查询返回的一系列行和光标的位置， lead 在该位置之后提供对给定物理偏移量的行的访问。 如果未指定偏移量，则默认偏移量为 1。默认设置如果偏移超出窗口范围则返回的值。 If 默认如果没被指定，默认值为 null.
ntile(expr)	bigint	NTILE(expr) OVER ([PARTITION BY expr] ORDER BY expr)	将有序数据集划分为多个桶 (由 expr定义) 并为每一行分配一个桶号。
percent_rank()	double precision	PERCENT_RANK () OVER ([PARTITION BY expr] ORDER BY expr)	计算假设行的排名 R 减1， 除了被评估的行数（在窗口分区内）除以1。
rank()	bigint	RANK () OVER ([PARTITION BY expr] ORDER BY expr)	计算有序的一组值中的一行的排名。排名标准相等值的行获得相同的排名。绑定行的数量被添加到等级号以计算下一个等级值。在这种情况下，排名可能不是连续的数字。
row_number()	bigint	ROW_NUMBER () OVER ([PARTITION BY expr] ORDER BY expr)	为其应用的每一行分配唯一的编号（窗口分区中的每一行或查询的每一行）。

高级聚集函数

以下内置的高级分析功能是 PostgreSQL 数据库的 HashData 扩展。分析功能是不可变的。

> 注意：用于分析的 HashData MADlib 扩展提供了其他高级功能，可以使用 HashData 数据库数据执行统计分析和机器学习。请参阅 用于分析的 HashData MADlib 扩展。

表 7. 高级聚集函数

函数	返回类型	完整语法	描述	
MEDIAN (expr)	timestamp, timestamptz, interval, float	MEDIAN (expression)\	Example:SELECT department_id, MEDIAN(salary) FROM employees GROUP BY department_id;	可以采用二维数组作为输入。将这样的数组视为矩阵。
PERCENTILE_CONT (expr) WITHIN GROUP (ORDER BY expr [DESC/ASC])	timestamp, timestamptz, interval, float	PERCENTILE_CONT(percentage) WITHIN GROUP (ORDER BY expression) Example:SELECT department_id,PERCENTILE_CONT (0.5) WITHIN GROUP (ORDER BY salary DESC) "Median_cont"; FROM employees GROUP BY department_id;	执行假定连续分布模型的逆分布函数。它需要百分位数值和排序规范，并返回与参数的数字数据类型相同的数据类型。该返回值是执行线性插值后的计算结果。在此计算中将忽略空值。	
PERCENTILE_DISC (expr) WITHIN GROUP (ORDER BY expr [DESC/ASC])	timestamp, timestamptz, interval, float	ERCENTILE_DISC(percentage) WITHIN GROUP (ORDER BY expression) Example:SELECT department_id, PERCENTILE_DISC (0.5) WITHIN GROUP (ORDER BY salary DESC) "Median_desc"; FROM employees GROUP BY department_id;	执行一个假设离散分布模型的逆分布函数。它需要一个百分位数值和一个排序规格。这个返回值是集合中的一个元素。在此计算中将忽略空值。	
sum(array[])	smallint[],int[], bigint[], float[]	sum(array[[1,2],[3,4]]) Example:CREATE TABLE mymatrix (myvalue int[]);INSERT INTO mymatrix VALUES (array[[1,2],[3,4]]); INSERT INTO mymatrix VALUES (array[[0,1],[1,0]]); SELECT sum(myvalue) FROM mymatrix;sum : {{1,3},{4,4}}	执行矩阵求和。可以将二维数组作为输入，作为矩阵处理。	
pivot_sum (label[], label, expr)	int[], bigint[], float[]	pivot_sum(array['A1','A2'], attr, value)	使用 sum 来解析重复条目的枢轴聚合。	
unnest (array[])	set of anyelement	unnest(array['one', 'row', 'per', 'item'])`	将一维数组转换成行。返回一组 anyelement (一种多态的 PostgreSQL 中的伪类型。	

PL/Python 语言扩展

本节包含 HashData 数据库的 PL/Python 语言的概述。

- 关于 HashData 的 PL/Python
- 启用和删除 PL/Python 的支持
- 使用 PL/Python 开发函数
- 安装 Python 模块
- 示例
- 参考

上一个话题: HashData 数据库参考指南

关于 HashData 的 PL/Python

PL/Python 是一种可加载的程序语言。使用 HashData 数据库的 PL/Python 扩展，用户可以在 Python 中编写一个 HashData 数据库用户定义的函数，利用 Python 功能和模块的优势可以快速构建强大的数据库应用程序。

用户可以将 PL/Python 代码块作为匿名代码块运行。见 HashData 数据库参考指南中的 DO 命令。

HashData 数据库的 PL/Python 扩展默认安装在 HashData 数据库中。HashData 数据库安装了一个 Python 和 PL/Python 版本。这是 HashData 数据库使用的 Python 的安装位置。

```
$GPHOME/ext/Python/
```

HashData 数据库的 PL/Python 的限制

- HashData 数据库不支持 PL/Python 触发器。
- PL/Python 仅作为 HashData 数据库不可信的语言使用。
- 不支持可更新的游标 (UPDATE...WHERE CURRENT OF and DELETE...WHERE CURRENT OF)。

启用和删除 PL/Python 支持

PL/Python 语言与 HashData 数据库一起安装。为了在数据库中创建和运行 PL/Python 用户定义的函数（ UDF ）,用户必须使用数据库注册 PL/Python 语言。

启用 PL/Python 支持

对于每个需要使用的数据库，请使用 SQL 命令 CREATE LANGUAGE 或 HashData 数据库功能 createlang 注册 PL/Python 语言。因为 PL/Python 是不可信的语言，只有超级用户可以使用数据库注册 PL/Python。例如，作为 gpadmin 系统用户使用名为 testdb 的数据库注册 PL/Python 时运行以下命令:

```
$ createlang pl Python u -d testdb
```

PL/Python 被注册了成一种不可信语言。

删除 PL/Python 支持

对于一个不再需要 PL/Python 语言支持的数据库来说, 可以使用 SQL 命令 DROP LANGUAGE 或者 HashData 数据库 droplang 功能删除对 PL/Python 的支持。 因为 PL/Python 是一种不可信语言, 只有超级用户能够从数据库中删除对 PL/Python 语言的支持。例如，作为 gpadmin 系统用户使用名为testdb的数据库删除对 PL/Python 语言的支持时运行以下命令:

```
$ droplang plPython u -d testdb
```

当用户删除对 PL/Python 语言的支持时, 用户在数据库中创建的 PL/Python 用户自定义函数将不再有效。

使用 PL/Python 开发函数

PL/Python 用户定义函数的主体是 Python 脚本。 当函数被调用时，其参数被作为数组 args[] 的元素传递。命名参数也作为普通变量传递给 Python 脚本。 结果是通过 PL/Python 函数中的 return 语句返回的, 或者是通过 yield 语句返回结果集。

数组和列表

用户将 SQL 列表的值传递给具有 Python 列表的 PL/Python 函数。同样的, PL/Python 函数将 SQL 数组值作为 Python 列表返回。 在典型的 PL/Python 使用模式中, 用户将使用 [] 来指定数组。

以下的列子创建了一个 PL/Python 函数并返回整数数组:

```
CREATE FUNCTION return_py_int_array()
  RETURNS int[]
AS $$
  return [1, 11, 21, 31]
$$ LANGUAGE pl Python u;

SELECT return_py_int_array();
 return_py_int_array
---------------------
 {1,11,21,31}
(1 row)
```

PL/Python 将多维数组视为列表的列表。用户使用嵌套的 Python 列表将多维数组传递给 PL/Python 函数。当 PL/Python 函数返回多维数组时, 每个级别的内部列表必须具有相同的大小。

接下来的示例创建了一个 PL/Python 函数，该函数以多维整数数组作为输入。该函数显示所提供参数的类型，并返回该多维数组:

```
CREATE FUNCTION return_multidim_py_array(x int4[])
  RETURNS int4[]
AS $$
  plpy.info(x, type(x))
  return x
$$ LANGUAGE pl Python u;

SELECT * FROM return_multidim_py_array(ARRAY[[1,2,3], [4,5,6]]);
INFO:  ([[1, 2, 3], [4, 5, 6]], <type 'list'>)
CONTEXT:  PL/Python  function "return_multidim_py_type"
 return_multidim_py_array
--------------------------
 {{1,2,3},{4,5,6}}
(1 row)
```

PL/Python 还接受其他 Python 序列,列如元组,作为与不支持多维数组的 HashData 版本向后兼容的函数参数。在这种情况下， Python 序列总是被作为一维数组对待，因为它们与复合类型不一致。

复合类型

用户可以使用 Python 映射传送一个复合类型的参数给 PL/Python 函数。映射的元素名称是复合类型的属性名称。如果属性有空值，则其映射为 None。

用户可以将复合类型结果作为序列类型（列表或者元组）返回。在多维数组的使用中时，用户必须将复合类型指定为元组，而不是列表。用户不能将复合类型数组作为列表返回，因为确定列表是否表示复合类型或另一个数组维度是不明确的。在典型的使用模式中，用户将使用 () 指定复合类型元组。

在接下来的列子中, 用户可以创建一个复合类型和一个返回复合类型数组的 PL/Python 函数:

```
CREATE TYPE type_record AS (
  first text,
  second int4
);

CREATE FUNCTION composite_type_as_list()
  RETURNS type_record[]
AS $$
  return [[('first', 1), ('second', 1)], [('first', 2), ('second', 2)], [('first', 3), ('second', 3)]];
$$ LANGUAGE pl Python u;

SELECT * FROM composite_type_as_list();
                              composite_type_as_list
-------------------------------------------------------------------------------
 {{"(first,1)","(second,1)"},{"(first,2)","(second,2)"},{"(first,3)","(second,3)"}}
(1 row)
```

参考 PostgreSQL 中的 Arrays, Lists 文档寻求关于 PL/Python 处理数组和复合类型的附加信息。

执行和准别 SQL 查询

PL/Python 中的plpy 模块提供了两个 Python 函数去为一个查询语句执行 SQL 查询和准备执行计划, plpy.execute and plpy.prepare。 准备查询的执行计划是有用的，如果用户从多个 Python 函数中运行查询。

plpy.execute

使用查询字符串和可选的限制参数调用 plpy.execute 会导致运行查询，并且在 Python 结果对象中返回结果。结果对象模拟列表或字典对象。可以通过行号和列名访问结果对象中返回的行。结果集行号从 0 开始。结果对象可以修改。结果对象有这些附加的方法：

- nrows 返回查询返回的函数
- status 是 SPI_execute() 返回的值

例如，这个 PL/Python 用户自定义函数中的 Python 语句执行了一个查询：

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

这个 plpy.execute 函数从 my_table 中返回最多 5 行。 结果集储存在 the rv 对象中。 如果 my_table 有一列 my_column，那它可以通过如下访问：

```
my_col_data = rv[i]["my_column"]
```

由于这个函数做多返回 5 行, 下标 i 是 0-4 之间的整数。

plpy.prepare

函数 plpy.prepare 为查询准备了执行计划。 如果查询中有参数引用，则用查询字符串和参数类型列表调用该函数。例如，该语句可以在 PL/Python 用户自定义的函数中：

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE
  first_name = $1", [ "text" ])
```

字符串 text 是变量 `$1` 传递的数据类型。 在准备好语句之后, 用户可以使用函数 plpy.execute 去运行它:

```
rv = plpy.execute(plan, [ "Fred" ], 5)
```

第三个参数就是可选的返回结果的行数限制。

当用户使用使用 PL/Python 模块准备一个执行计划的时候，该计划是自动保存的。更多执行计划的信息请参见 Postgres 服务器编程接口 (SPI) 文档 https://www.postgresql.org/docs/8.3/static/spi.html。

为了在函数调用中有效的使用保存的计划，用户可以使用一个 Python 持久存储字典 SD 或 GD。

全局字典 SD 可用于在函数调用之间存储数据。该变量是私有静态变量。全局字典 GD 是公共数据，可用于会话内的所有 Python 函数。小心使用 GD。

每个函数在 Python 解释器中都有自己的执行环境，所以 myfunc 的全局数据和函数参数对 myfunc2 不可见。 除非该数据在 GD 字典中, 正如前所述。

使用 SD 字典的例子:

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
  if SD.has_key("plan"):
    plan = SD["plan"]
  else:
    plan = plpy.prepare("SELECT 1")
    SD["plan"] = plan

  # rest of function

$$ LANGUAGE pl Python u;
```

处理 **Python** 错误和消息

Python 模块 plpy 实现了这个函数来管理错误和消息：

- plpy.debug
- plpy.log
- plpy.info
- plpy.notice
- plpy.warning
- plpy.error
- plpy.fatal
- plpy.debug

消息函数 plpy.error and plpy.fatal 抛出一个 Python 异常，该异常如果没有被捕获, 则会被传播到调用查询, 导致当前事务和子事务终止。 该函数 raise plpy.ERROR(msg) 和 raise plpy.FATAL(msg) 分别等效于调用 plpy.error 和 plpy.fatal。 其他消息函数仅仅生成不同优先级的消息。

是否向客户端报告特定优先级的消息，写入服务器日志或两者都由 HashData 数据库服务器配置参数log_min_messages 和 client_min_messages 控制。有关参数的信息，请参阅 HashData 数据库参考指南。

使用字典 **GD** 提升 **PL/Python** 性能

在性能方面，导入 Python 模块操作代价高昂，而且影响性能。如果用户频繁导入相同的模块，用户可以使用 Python 全局变量在第一次调用的时候加载该模块而不需要在子调用中导入该模块。接下来的 PL/Python 函数使用 GD 持久存储字典从而避免导入已经导入过的且在 GD 中的模块。

```
psql=#
   CREATE FUNCTION pytest() returns text as $$
      if 'mymodule' not in GD:
        import mymodule
        GD['mymodule'] = mymodule
    return GD['mymodule'].sumd([1,2,3])
$$;
```

安装 **Python** 模块

当用户在 HashData 数据库中安装 Python 模块的时候， HashData 数据库 Python 环境必须将该模块添加到集群中所有的段主机和镜像主机上。当扩展 HashData 数据库时，用户必须添加该 Python 模块到新的段主机中。用户可以使用 HashData 数据库实用程序 gpssh 和 gpscp 在 HashData 数据库主机上运行命令并将文件复制到主机。 更多关于实用程序的信息，请

参阅 HashData 数据库实用程序指南 。

作为 HashData 数据库安装的一个部分， gpadmin 用户环境配置为使用与 HashData 数据库一起的安装的 Python 。

为了检查 Python 环境，用户可以使用 which 命令：

```
which  Python
```

该命令返回 Python 的安装路径。和 HashData 数据库一起安装的 Python 实在 HashData 数据库中 ext/ Python 目录下。

```
/path_to_ HashData -db/ext/ Python /bin/ Python
```

如果用户正在构建 Python 模块，则必须确保构建创建正确的可执行文件。例如在 Linux 系统上，构建应创建一个 64 位的可执行文件。

在构建一个安装的模块之前，请确保安装并正确配置相应的构建模块的软件。构建环境仅在构建模块的主机上需要。

安装和测试 Python 模块的列子：

- 简单 Python 模块安装例子 (setuptools)
- 复杂 Python 安装例子 (NumPy)
- 测试安装的 Python 模块

简单 Python 模块安装例子 (setuptools)

此例从 Python 包索引储存库手动安装 Python setuptools 模块。该模块可让用户轻松下载，构建，安装，升级和卸载 Python 包。

该示例首先从一个程序包生成模块，并将该模块安装到在一单个主机上，然后该模块将被构建并安装在段主机上。

1. 从 Python 包索引站点获取模块包，例如，以 gpadmin 用户身份在 HashData 数据库主机上运行 wget 命令，来获得 tar 文件

   ```
   wget --no-check-certificate https://pypi.Python.org/packages/source/s/setuptools/setuptools-18.4.tar.gz
   ```

2. 从 tar 文件中提取文件。

   ```
   tar -xzvf distribute-0.6.21.tar.gz
   ```

3. 转到包含包文件的目录，并运行 Python 脚本来构建和安装 Python 包

   ```
   cd setuptools-18.4
    Python  setup.py build &&  Python  setup.py install
   ```

4. 如果模块已经安装好，则接下来的 Python 命令将不返回错误。

   ```
    Python  -c "import setuptools"
   ```

5. 使用 gpscp 工具将包复制到 HashData 数据库主机。例如，这个命令从当前主机复制 tar 文件到文件 remote-hosts 中所列的主机中。

   ```
   gpscp -f remote-hosts setuptools-18.4.tar.gz =:/home/gpadmin
   ```

6. 使用 gpssh 在 remote-hosts 文件列出的主机中运行命令构建, 安装和测试该软件包。该文件 remote-hosts 列出了所有 HashData 数据库的段主机：

```
gpssh -f remote_hosts
>>> tar -xzvf distribute-0.6.21.tar.gz
>>> cd setuptools-18.4
>>>  Python  setup.py build &&  Python  setup.py install
>>>  Python  -c "import setuptools"
>>> exit
```

setuptools 包安装了 easy_install 使用程序，该程序可以使用从 Python 包索引存储库安装 Python 包。例如，这个命令从 Python 索引站点安装 Python PIP 实用程序。

```
easy_install pip
```

用户可以使用 gpssh 实用程序在所有 HashData 数据库段主机中运行 easy-install 命令。

复杂的 Python 安装例子（NumPy）

这个例子构建和安装了 Python 模块 NumPy。 NumPy 是 Python 的一个科学计算模块。更多关于 NumPy 的信息，请参阅 http://www.NumPy.org/。

构建 NumPy 包所需的软件：

- OpenBLAS 库,一个开源的 BLAS 实现(基础线性代数子程序)。
- gcc 编译器: gcc, gcc-gfortran, and gcc-c++。这些编译器需要来构建 OpenBLAS 库。

这个例子过程假定 yum 安装在 HashData 数据库段主机上，而且 gpadmin 用户是主机上具有root权限的 sudoers 成员。

下载 OpenBLAS 和 NumPy 源文件。例如，这些 wget 命令下载 tar 文件到目录 packages 中：

```
wget --directory-prefix=packages http://github.com/xianyi/OpenBLAS/tarball/v0.2.8
wget --directory-prefix=packages http://sourceforge.net/projects/ NumPy /files/ NumPy /1.8.0/ NumPy -1.8.0.tar.gz/download
```

将软件分发到 HashData 数据库主机。例如，如果用户将软件下载到/home/gpadmin/packages 这些命令在主机中创建文件夹并且将软件复制到 gpdb_remotes 文件所列的主机中。

```
gpssh -f gpdb_remotes mkdir packages
gpscp -f gpdb_remotes packages/* =:/home/gpadmin/packages
```

OpenBLAS 先决条件

如果需要的话，使用 yum 从系统仓库安装 gcc 编译器。所有用户编译 OpenBLAS 的主机都需要该编译器：

```
sudo yum -y install gcc gcc-gfortran gcc-c++
```

> 注意： 如果用户不能使用 yum 安装正确的编译器版本，用户可以从源文件下载 gcc 编译器（包括gfortran），并进行安装。这两个命令下载并安装该编译器：

```
wget http://gfortran.com/download/x86_64/snapshots/gcc-4.4.tar.xz
tar xf gcc-4.4.tar.xz -C /usr/local/
```

如果用户是从 tar 文件手动安装的 gcc 编译器，添加新的 gcc 库到 PATH 和 LD_LIBRARY_PATH中：

```
export PATH=$PATH:/usr/local/gcc-4.4/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/gcc-4.4/lib
```

创建一个符号链接到 g++ 并称其为 gxx

```
sudo ln -s /usr/bin/g++ /usr/bin/gxx
```

用户可能也需要创建一个符号链接到具有不同版本的库例如 libppl_c.so.4 到 libppl_c.so.2.

如果需要的话，用户可以使用 gpscp 实用工具来复制文件到 HashData 数据库的主机中，并使用 gpssh 实用工具在主机上运行命令。

构建和安装 **OpenBLAS** 库

在构建和安装 NumPy 模块之前，用户应安装 OpenBLAS 库。本节介绍如何在单个主机上构建和安装该库。

1. 这些命令从 OpenBLAS tar文件中提取文件，并简化了包含 OpenBLAS 文件的目录名。

```
tar -xzf packages/v0.2.8 -C /home/gpadmin/packages
mv /home/gpadmin/packages/xianyi-OpenBLAS-9c51cdf /home/gpadmin/packages/OpenBLAS
```

2. 编译 OpenBLAS。这些命令设置 LIBRARY_PATH 环境变量并运行 make 命令构建 OpenBLAS 库。

```
cd /home/gpadmin/packages/OpenBLAS
export LIBRARY_PATH=$LD_LIBRARY_PATH
make FC=gfortran USE_THREAD=0
```

3. 这些命令作为 root 用户在 /usr/local 目录下安装 OpenBLAS 库并且将文件的所有者改为 gpadmin 用户。

```
cd /home/gpadmin/packages/OpenBLAS/
sudo make PREFIX=/usr/local install
sudo ldconfig
sudo chown -R gpadmin /usr/local/lib
```

这些是安装的库和创建的符号链接：

```
libopenblas.a -> libopenblas_sandybridge-r0.2.8.a
libopenblas_sandybridge-r0.2.8.a
libopenblas_sandybridge-r0.2.8.so
libopenblas.so -> libopenblas_sandybridge-r0.2.8.so
libopenblas.so.0 -> libopenblas_sandybridge-r0.2.8.so
```

用户可以使用 gpssh 工具来在多主机中构建和安装 OpenBLAS 库。

所有的 HashData 数据库主机 (主机和段主机) 都有相同的配置。用户可以从构建它们的系统中复制 OpenBLAS 库，而不是在所有主机上构建 OpenBlas 库。例如，这些 gpssh 和 gpscp 命令在 gpdb_remotes 文件列出的主机中复制并且安装了 OpenBLAS 库。

```
gpssh -f gpdb_remotes -e 'sudo yum -y install gcc gcc-gfortran gcc-c++'
gpssh -f gpdb_remotes -e 'ln -s /usr/bin/g++ /usr/bin/gxx'
gpssh -f gpdb_remotes -e sudo chown gpadmin /usr/local/lib
gpscp -f gpdb_remotes /usr/local/lib/libopen*sandy* =:/usr/local/lib

gpssh -f gpdb_remotes
>>> cd /usr/local/lib
>>> ln -s libopenblas_sandybridge-r0.2.8.a libopenblas.a
>>> ln -s libopenblas_sandybridge-r0.2.8.so libopenblas.so
>>> ln -s libopenblas_sandybridge-r0.2.8.so libopenblas.so.0
>>> sudo ldconfig
```

构建和安装 **NumPy**

在用户安装完 OpenBLAS 库之后，用户可以构建和安装 NumPy 模块。 这些步骤可以在单个主机中安装 NumPy 模块。用户可以使用 gpssh 工具在多主机中构建和安装 NumPy 模块。

1. 转到 packages 子目录并且获取 NumPy 模块源码并解压该文件。

```
cd /home/gpadmin/packages
tar -xzf  NumPy -1.8.0.tar.gz
```

2. 为构建和安装 NumPy 设置环境。

```
export BLAS=/usr/local/lib/libopenblas.a
export LAPACK=/usr/local/lib/libopenblas.a
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib/
export LIBRARY_PATH=$LD_LIBRARY_PATH
```

3. 转到 NumPy 目录并构建和安装 NumPy 。构建 NumPy 包可能需要一些时间。

```
cd  NumPy -1.8.0
 Python  setup.py build
 Python  setup.py install
```

> 注意： 如果 NumPy 模块没有成功构建，那 NumPy 构建过程可能需要一个指定OpenBLAS库位置的 site.cfg 。
> 创建文件 site.cfg 在 NumPy 包目录中：

```
cd ~/packages/ NumPy -1.8.0
touch site.cfg
```

添加以下脚本到 site.cfg 文件中并再次运行 NumPy 构建命令：

```
[default]
library_dirs = /usr/local/lib

[atlas]s
atlas_libs = openblas
library_dirs = /usr/local/lib

[lapack]
lapack_libs = openblas
library_dirs = /usr/local/lib

# added for scikit-learn
[openblas]
libraries = openblas
library_dirs = /usr/local/lib
include_dirs = /usr/local/include
```

4. 以下的 Python 命令确保该模块可用于在主机系统上由 Python 导入。

```
 Python  -c "import  NumPy "
```

如同在简单的模块安装中一样，用户可以使用 gpssh 工具来构建，安装和测试 HashData 数据库中段主机中的模块。

在构建 NumPy 时所需的环境变量，在 gpadmin 用户环境中运行 Python NumPy 函数时同样需要。用户可以使用 gpssh 工具 和 echo 命令将环境变量添加到 .bashrc 文件中。例如，这些 echo 命令添加环境变量到用户家（user home）目录下的 .bashrc 文件中。

```
echo -e 'n#Needed for  NumPy ' >> ~/.bashrc
echo -e 'export BLAS=/usr/local/lib/libopenblas.a' >> ~/.bashrc
echo -e 'export LAPACK=/usr/local/lib/libopenblas.a' >> ~/.bashrc
echo -e 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib' >> ~/.bashrc
echo -e 'export LIBRARY_PATH=$LD_LIBRARY_PATH' >> ~/.bashrc
```

测试安装的 **Python** 模块

用户可以创建一个简单的 PL/Python 用户自定义函数 (UDF) 来验证 HashData 数据库中的 Python 模块可用。这个例子测试的是 NumPy 模块。

这个 PL/Python UDF 导入 NumPy 模块。如果模块被正确导入，则该函数返回 SUCCESS ，否则返回 FAILURE 。

```
CREATE OR REPLACE FUNCTION plpy_test(x int)
returns text
as $$
  try:
      from  NumPy  import *
      return 'SUCCESS'
  except ImportError, e:
      return 'FAILURE'
$$ language pl Python u;
```

创建一个包含每个 HashData 数据库段实例上的数据的表。根据用户的 HashData 数据库安装的大小，用户可能需要生成更多数据，以确保数据分发到所有段实例。

```
CREATE TABLE DIST AS (SELECT x FROM generate_series(1,50) x ) DISTRIBUTED RANDOMLY ;
```

该 SELECT 命令在数据存储在主段实例中的段主机上运行 UDF。

```
SELECT gp_segment_id, plpy_test(x) AS status
  FROM dist
  GROUP BY gp_segment_id, status
  ORDER BY gp_segment_id, status;
```

如果在 HashData 数据库段实例中UDF成功导入 Python 模块，该 SELECT 命令返回 SUCCESS 。如果该 SELECT 命令返回 FAILURE，用户可以找到段实例主机的段主机。 HashData 数据库系统表 gp_segment_configuration 包含镜像和段配置信息。该命令返回段ID的主机名。

```
SELECT hostname, content AS seg_ID FROM gp_segment_configuration
  WHERE content = seg_id ;
```

如果返回 FAILURE，有以下几种可能：

- 访问所需库时遇到问题。对于 NumPy 例子， HashData 数据库可能在访问段主机上的OpenBLAS库或者 Python 库时遇到问题。

  请确保用户以 gpadmin 用户在端主机上运行命令时没有返回错误。该 gpssh 命令测试在段主机 mdw1上导入 NumPy 模块。

  ```
  gpssh -h mdw1  Python  -c "import  NumPy "
  ```

- 如果 Python import 命令没有返回错误， 则环境变量可能没有在 HashData 数据库环境中配置。例如，变量不在 .bashrc 文件中，或者 HashData 数据库在向 .bashrc 文件中添加环境变量之后没有重启。

  确保环境变量被正确设置，然后重新启动 HashData 数据库。对于 NumPy 例子，确保被列在 构建和安装 NumPy 末尾部分的环境变量被定义在主机和段主机上 gpadmin 用户的.bashrc文件中。

  > 注意: 在 HashData 数据库主机和段主机上，gpadmin 用户的 .bashrc 文件必须来源于文件 `$GPHOME/HashData\_path.sh` 。

例子

该 PL/Python UDF（用户自定义函数）返回两个整数的较大者：

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
      return None
  if a > b:
     return a
  return b
$$ LANGUAGE pl Python u;
```

用户可以使用 STRICT 属性来执行空处理而不是使用两个条件语句。

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer AS $$
return max(a,b)
$$ LANGUAGE pl Python u STRICT ;
```

用户可以运行用户自定义函数 pymax 通过用 SELECT 命令。该例子运行用户自定义函数并且显示了输出。

```
SELECT ( pymax(123, 43));
column1
---------
     123
(1 row)
```

该例子从针对表运行的SQL查询返回数据。这两个命令创建了一个简单的表，并向表中添加了数据。

```
CREATE TABLE sales (id int, year int, qtr int, day int, region text)
  DISTRIBUTED BY (id) ;

INSERT INTO sales VALUES
 (1, 2014, 1,1, 'usa'),
 (2, 2002, 2,2, 'europe'),
 (3, 2014, 3,3, 'asia'),
 (4, 2014, 4,4, 'usa'),
 (5, 2014, 1,5, 'europe'),
 (6, 2014, 2,6, 'asia'),
 (7, 2002, 3,7, 'usa') ;
```

该 PL/Python UDF 执行了一个从表中返回 5 行的 SELECT 命令。 Python 函数从输入值指定的行返回 REGION 值。在 Python 函数中，行标从 0 开始。该函数有效的输入是0到4之间的整数。

```
CREATE OR REPLACE FUNCTION mypytest(a integer)
  RETURNS text
AS $$
  rv = plpy.execute("SELECT * FROM sales ORDER BY id", 5)
  region = rv[a]["region"]
  return region
$$ language pl Python u;
```

运行该 SELECT 语句从结果集的第三行返回 REGION 列值。

```
SELECT mypytest(2) ;
```

该从数据库中删除了 UDF（用户自定义函数）。

```
DROP FUNCTION mypytest(integer) ;
```

该例子在前面的示例中使用 DO 命令作为匿名块执行 PL/Python 函数。在该例子中，匿名块从临时表中检索输入值。

```
CREATE TEMP TABLE mytemp AS VALUES (2) DISTRIBUTED RANDOMLY;

DO $$
  temprow = plpy.execute("SELECT * FROM mytemp", 1)
  myval = temprow[0]["column1"]
  rv = plpy.execute("SELECT * FROM sales ORDER BY id", 5)
  region = rv[myval]["region"]
  plpy.notice("region is %s" % region)
$$ language pl Python u;
```

参考

技术参考

更多关于 Python 语言的信息，请参阅 https://www.Python.org/。

更多关于 PL/Python 的信息，请参阅 PostgreSQL 文档。

更多关于 Python 包索引（PyPI）的信息，请参阅 https://pypi.Python.org/pypi。

这些是一些可以下载的 Python 模块：

- SciPy 库提供用户友好和高效的数值例程，例如数值积分和优化的例程 http://www.scipy.org/scipylib/index.html。该 wget 命令可以下载 SciPy 包的tar文件。

  ```
  wget http://sourceforge.net/projects/scipy/files/scipy/0.10.1/scipy-0.10.1.tar.gz/download
  ```

- 自然语言工具包 (nltk) 是一个构建 Python 程序以处理人数语言数据的平台。http://www.nltk.org/。该 wget 命令下载 nltk 包的tar文件。

  ```
  wget http://pypi.Python.org/packages/source/n/nltk/nltk-2.0.2.tar.gz#md5=6e714ff74c3398e88be084748df4e657
  ```

  > 注意： 对 nltk 来说， Python Distribute 包 https://pypi.Python.org/pypi/distribute是必须的。该 Distribute 模块应该安装ntlk包。该 wget 命令可以下载 Distribute 包的 tar 文件。

  ```
  wget http://pypi.Python.org/packages/source/d/distribute/distribute-0.6.21.tar.gz
  ```

用于分析的 **MADlib** 扩展

本章节包含了以下信息：

- 关于 MADlib
- 示例
- 参考

上级主题： HashData 数据库参考指南

关于 **MADlib**

MADlib 是一个可扩展数据库分析的开源库。通过 HashData 的 MADlib 扩展，用户可以在 HashData 数据库中使用 MADlib 功能。

MADlib 为结构化数据以及非结构化数据提供了数学、统计学以及机器学习方法的数据并行的实现。它提供了一整套基于 SQL 的机器学习、数据挖掘以及统计学算法，只需要运行在数据库引擎中，而不需要在 HashData 和其它工具之间进行数据的传递。

MADlib 可以与 PivotalR 一同使用，一个 PivotalR 包允许用户使用 R 客户端同 HashData 的数据进行交互。见 关于 MADlib、R、PivotalR。

> 注意：当使用 MADlib，设置配置参数 optimizer_control 为 on（默认值）。如果该参数被设置为 off，那么这些 MADlib 函数将不会工作：决策树、随机森林、LDA、决策树的 PMML、随机森林的 PMML。参数 optimizer_control 控制是配置参数 optimizer 能够被修改。参数 optimizer 控制在执行 SQL 查询时 GPORCA 优化器是否被打开。一些 MADlib 安装检查和函数改变 optimizer 的值来提高性能。如果将 optimizer_control 设置为 off，那么 optimizer 的值不能修改，同时函数会失败。

示例

下面是使用 HashData MADlib 扩展的示例：

- 线性回归
- 关联规则
- 朴素贝叶斯分类

见 MADlib 文档获取更多的示例。

线性回归

该示例是在表 regr_example 执行一个线性回归。因变量数据在 y 列中，独立变量数据在 x1 和 x2 列中。

下面的语句创建 regr_example 表同时加载样本数据：

```
DROP TABLE IF EXISTS regr_example;
CREATE TABLE regr_example (
   id int,
   y int,
   x1 int,
   x2 int
);
INSERT INTO regr_example VALUES
   (1,  5, 2, 3),
   (2, 10, 7, 2),
   (3,  6, 4, 1),
   (4,  8, 3, 4);
```

MADlib 的 linregr_train() 函数产生一个根据一个输入表包含的训练数据产生一个回归模型。下面的SELECT 语句在表

regr_example 执行一个简单的多元回归同时保存模型在表 reg_example_model中。

```
SELECT madlib.linregr_train (
   'regr_example',         -- source table
   'regr_example_model',   -- output model table
   'y',                    -- dependent variable
   'ARRAY[1, x1, x2]'      -- independent variables
);
```

madlib.linregr_train() 函数能够添加参数来设置分组的列以及计算模型的异方差性。

> 注释:截距通过将一个独立变量设置为常数1来计算，如前一个例子所示。

在表 regr_example 上执行该查询并创建带有一行数据的 regr_example_model 表：

```
SELECT * FROM regr_example_model;
-[ RECORD 1 ]-----------+----------------------
coef                    | {0.111111111111127,1.14814814814815,1.01851851851852}
r2                      | 0.968612680477111
std_err                 | {1.49587911309236,0.207043331249903,0.346449758034495}
t_stats                 | {0.0742781352708591,5.54544858420156,2.93987366103776}
p_values                | {0.952799748147436,0.113579771006374,0.208730790695278}
condition_no            | 22.650203241881
num_rows_processed      | 4
num_missing_rows_skipped | 0
variance_covariance     | {{2.23765432098598,-0.257201646090342,-0.437242798353582},
                          {-0.257201646090342,0.042866941015057,0.0342935528120456},
                          {-0.437242798353582,0.0342935528120457,0.12002743484216}}
```

被保存到 regr_example_model 表中的模型能够同 MADlib 线性回归预测函数使用， madlib.linregr_predict() 来查看残差：

```
SELECT regr_example.*,
       madlib.linregr_predict ( ARRAY[1, x1, x2], m.coef ) as predict,
       y - madlib.linregr_predict ( ARRAY[1, x1, x2], m.coef ) as residual
FROM regr_example, regr_example_model m;
 id | y  | x1 | x2 |     predict      |      residual
----+----+----+----+------------------+--------------------
  1 |  5 |  2 |  3 | 5.46296296296297 | -0.462962962962971
  3 |  6 |  4 |  1 | 5.72222222222224 |  0.277777777777762
  2 | 10 |  7 |  2 | 10.1851851851852 | -0.185185185185201
  4 |  8 |  3 |  4 | 7.62962962962964 |  0.370370370370364
(4 rows)
```

关联规则

这个例子说明了关联规则的数据挖掘技术在交易数据集。关联规则挖掘是发现大数据集中的变量之间关系的技术。这个例子将考虑那些在商店中通常一起购买的物品。除了购物篮分析，关联规则也应用在生物信息学中，网络分析，和其他领域。

这个例子分析利用 MADlib 函数 MADlib.assoc_rules 分析存储在表中的关于七个交易的购买信息。函数假定数据存储在两列中，每行有一个物品和交易 ID。多个物品的交易，包括多个行，每行一个物品。

这些命令创建表。

```
DROP TABLE IF EXISTS test_data;
CREATE TABLE test_data (
   trans_id INT,
   product text
);
```

该 INSERT 命令向表中添加数据。

```
INSERT INTO test_data VALUES
   (1, 'beer'),
   (1, 'diapers'),
   (1, 'chips'),
   (2, 'beer'),
   (2, 'diapers'),
   (3, 'beer'),
   (3, 'diapers'),
   (4, 'beer'),
   (4, 'chips'),
   (5, 'beer'),
   (6, 'beer'),
   (6, 'diapers'),
   (6, 'chips'),
   (7, 'beer'),
   (7, 'diapers');
```

MADlib 函数 madlib.assoc_rules() 分析数据同时确定具有以下特征的关联规则。

- 一个值至少为 .40 的支持率。支持率表示包含 X 的交易与所有交易的比。
- 一个值至少为 .75 的置信率。置信率表示包含 X 的交易与包含Y的交易的比。可以将该度量看做给定 Y 下 X 的条件概率。

该 SELECT 命令确定关联规则，创建表 assoc_rules 同时天啊件统计信息到表中。

```
SELECT * FROM madlib.assoc_rules (
   .40,          -- support
   .75,          -- confidence
   'trans_id',   -- transaction column
   'product',    -- product purchased column
   'test_data',  -- table name
   'public',     -- schema name
   false);       -- display processing details
```

这是 SELECT 命令的输出。这有两条符合特征的规则。

```
 output_schema | output_table | total_rules | total_time
---------------------------------------------------------------
public        | assoc_rules  |           2 | 00:00:01.153283
(1 row)
```

为了查看关联规则，用户可以使用该 SELECT 命令。

```
SELECT pre, post, support FROM assoc_rules
   ORDER BY support DESC;
```

这是输出。pre 和 post 列分别是关联规则左右两边的项集。

```
    pre    |  post  |      support
-----------+--------+-------------------
 {diapers} | {beer} | 0.714285714285714
 {chips}   | {beer} | 0.428571428571429
(2 rows)
```

基于数据，啤酒和尿布经常一起购买。为了增加销售额，用户可以考虑将啤酒和尿布放在一个架子上。

朴素贝叶斯分类

朴素贝叶斯分析，基于一个或多个独立变量或属性，预测一类变量或类结果的可能性。类变量是非数值类型变量，一个变量可以有一个数量有限的值或类别。类变量表示的整数，每个整数表示一个类别。例如，如果类别可以是一个“真”、“假”，或“未知”的值，那么变量可以表示为整数 1, 2 或 3。

属性可以是数值类型、非数值类型以及类类型。训练函数有两个签名 - 一个用于所有属性为数值 另外一个用于混合数值和类类型的情况。后者的附加参数标识那些应该被当做数字值处理的属性。属性以数组的形式提交给训练函数。

MADlib 朴素贝叶斯训练函数产生一个特征概率表和一个类的先验表，该表可以同预测函数使用为属性集提供一个类别的概率。

朴素贝叶斯示例 **1 -** 简单的所有都是数值属性

在第一个示例中，类别变量取值为 1 或者 2，同时这里有三个整型属性。

1. 下面的命令创建输入表以及加载样本数据。

```
DROP TABLE IF EXISTS class_example CASCADE;
CREATE TABLE class_example (
   id int, class int, attributes int[]);
INSERT INTO class_example VALUES
   (1, 1, '{1, 2, 3}'),
   (2, 1, '{1, 4, 3}'),
   (3, 2, '{0, 2, 2}'),
   (4, 1, '{1, 2, 1}'),
   (5, 2, '{1, 2, 2}'),
   (6, 2, '{0, 1, 3}');
```

   在生产环境中的实际数据比该示例中的数据量更大，也能获得更好的结果。更大的训练数据集能够显著地提高分类的精确度。

2. 使用 create_nb_prepared_data_tables() 函数训练模型。

```
SELECT * FROM madlib.create_nb_prepared_data_tables (
   'class_example',         -- name of the training table
   'class',                 -- name of the class (dependent) column
   'attributes',            -- name of the attributes column
   3,                       -- the number of attributes
   'example_feature_probs', -- name for the feature probabilities output table
   'example_priors'         -- name for the class priors output table
    );
```

3. 为了使用模型进行分类，创建带有数据的表。

```
DROP TABLE IF EXISTS class_example_topredict;
CREATE TABLE class_example_topredict (
   id int, attributes int[]);
INSERT INTO class_example_topredict VALUES
   (1, '{1, 3, 2}'),
   (2, '{4, 2, 2}'),
   (3, '{2, 1, 1}');
```

4. 用特征概率、类的先验和 class_example_topredict 表创建一个分类视图。

```
SELECT madlib.create_nb_probs_view (
   'example_feature_probs',    -- 特征概率输出表
   'example_priors',           -- 先验输出表
   'class_example_topredict',  -- 带有要分类数据的表
   'id',                       -- 关键字的列名
   'attributes',               -- 属性列的名称
    3,                         -- 属性的数目
    'example_classified'       -- 要创建的视图的名称
    );
```

5. 显示分类结果。

```
SELECT * FROM example_classified;
 key | class | nb_prob
-----+-------+---------
   1 |     1 |     0.4
   1 |     2 |     0.6
   3 |     1 |     0.5
   3 |     2 |     0.5
   2 |     1 |    0.25
   2 |     2 |    0.75
(6 rows)
```

朴素贝叶斯示例 **2 –** 天气和户外运动

该示例计算在给定的天气条件下，用户要进行户外运动，例如高尔夫、网球等的概率。

表 weather_example 包含了样本值。

表的标识列是 day，整型类型。

play 列包含了因变量以及两个类别：

- 0 - No
- 1 - Yes

有四个属性：outlook、temperature、humidity、以及 wind。他们是类变量。 MADlib create_nb_classify_view() 函数希望属性提供的是 INTEGER、NUMERIC、 或者 FLOAT8 值类型的数组，所以该示例的属性均用为整型进行编码：

- *outlook* 可能取值为 sunny (1), overcast (2), or rain (3)。
- *temperature* 可能取值为 hot (1), mild (2), or cool (3).
- *humidity* 可能取值为 high (1) or normal (2).
- *wind* 可能取值为 strong (1) or weak (2).

下面的表显示了编码钱的训练数据。

```
  day | play | outlook  | temperature | humidity | wind
-----+------+----------+-------------+----------+--------
  2  | No   | Sunny    | Hot         | High     | Strong
  4  | Yes  | Rain     | Mild        | High     | Weak
  6  | No   | Rain     | Cool        | Normal   | Strong
  8  | No   | Sunny    | Mild        | High     | Weak
 10  | Yes  | Rain     | Mild        | Normal   | Weak
 12  | Yes  | Overcast | Mild        | High     | Strong
 14  | No   | Rain     | Mild        | High     | Strong
  1  | No   | Sunny    | Hot         | High     | Weak
  3  | Yes  | Overcast | Hot         | High     | Weak
  5  | Yes  | Rain     | Cool        | Normal   | Weak
  7  | Yes  | Overcast | Cool        | Normal   | Strong
  9  | Yes  | Sunny    | Cool        | Normal   | Weak
 11  | Yes  | Sunny    | Mild        | Normal   | Strong
 13  | Yes  | Overcast | Hot         | Normal   | Weak
(14 rows)
```

1. 创建一个训练表。

```
DROP TABLE IF EXISTS weather_example;
CREATE TABLE weather_example (
   day int,
   play int,
   attrs int[]
);
INSERT INTO weather_example VALUES
   ( 2, 0, '{1,1,1,1}'), -- sunny, hot, high, strong
   ( 4, 1, '{3,2,1,2}'), -- rain, mild, high, weak
   ( 6, 0, '{3,3,2,1}'), -- rain, cool, normal, strong
   ( 8, 0, '{1,2,1,2}'), -- sunny, mild, high, weak
   (10, 1, '{3,2,2,2}'), -- rain, mild, normal, weak
   (12, 1, '{2,2,1,1}'), -- 等
   (14, 0, '{3,2,1,1}'),
   ( 1, 0, '{1,1,1,2}'),
   ( 3, 1, '{2,1,1,2}'),
   ( 5, 1, '{3,3,2,2}'),
   ( 7, 1, '{2,3,2,1}'),
   ( 9, 1, '{1,3,2,2}'),
   (11, 1, '{1,2,2,1}'),
   (13, 1, '{2,1,2,2}');
```

2. 根据训练表创建模型。

```
SELECT madlib.create_nb_prepared_data\_tables (
   'weather_example',  -- 训练源数据
   'play',             -- 因变类别列
   'attrs',            -- 属性列
   4,                  -- 属性数目
   'weather_probs',    -- 特征概率输出表
   'weather_priors'    -- 类别先验
   );
```

3. 查看特征概率：

```
SELECT * FROM weather_probs;
 class | attr | value | cnt | attr_cnt
-------+------+-------+-----+----------
     1 |    3 |     2 |   6 |        2
     1 |    1 |     2 |   4 |        3
     0 |    1 |     1 |   3 |        3
     0 |    1 |     3 |   2 |        3
     0 |    3 |     1 |   4 |        2
     1 |    4 |     1 |   3 |        2
     1 |    2 |     3 |   3 |        3
     1 |    2 |     1 |   2 |        3
     0 |    2 |     2 |   2 |        3
     0 |    4 |     2 |   2 |        2
     0 |    3 |     2 |   1 |        2
     0 |    1 |     2 |   0 |        3
     1 |    1 |     1 |   2 |        3
     1 |    1 |     3 |   3 |        3
     1 |    3 |     1 |   3 |        2
     0 |    4 |     1 |   3 |        2
     0 |    2 |     3 |   1 |        3
     0 |    2 |     1 |   2 |        3
     1 |    2 |     2 |   4 |        3
     1 |    4 |     2 |   6 |        2
(20 rows)
```

4. 用模型分类一组记录，首先装载数据到一个表中。在该示例中，表 t1 有四个行将要分类。

```
DROP TABLE IF EXISTS t1;
CREATE TABLE t1 (
   id integer,
   attributes integer[]);
insert into t1 values
   (1, '{1, 2, 1, 1}'),
   (2, '{3, 3, 2, 1}'),
   (3, '{2, 1, 2, 2}'),
   (4, '{3, 1, 1, 2}');
```

5. 使用 create_nb_classify_view() 函数对表中的行进行分类。

```
SELECT madlib.create_nb_classify_view (
   'weather_probs',      -- 特征概率表
   'weather_priors',     -- 类先验名
   't1',                 -- 包含要分类值的表
   'id',                 -- 主键列
   'attributes',         -- 属性列
   4,                    -- 属性数目
   't1_out'              -- 输出表的名称
);
```

结果有四行，每行对应表 t1 中的一条记录。

```
SELECT * FROM t1_out ORDER BY key;
 key | nb_classification
-----+-------------------
 1 | {0}
 2 | {1}
 3 | {1}
 4 | {0}
 (4 rows)
```

参考

MADlib 网站在 http://madlib.incubator.apache.org/.

MADlib 文档在 http://madlib.incubator.apache.org/documentation.html.

PivotalR 是第一类能够让用户使用R客户端对 HashData 驻留的数据和 MADLib 进行交互的R包。

关于 MADlib、R、PivotalR

R 语言是一门用于统计计算的开源编程语言。PivotalR 是一个能够让用户通过 R 客户端与常驻 HashData 数据库的数据进行交互的R语言包。使用 PivotalR 要求 MADlib 已经安装在了 HashData 数据库中。

PivotalR 允许 R 用户不用离开R命令行就能利用数据库内分析的可扩展性和性能。计算工作在数据库内执行，而终端用户受益于熟悉的R语言接口。与相应的原生 R 函数相比，在可扩展性上得到提同时在执行时间上有降低。此外，PivotalR 消除了对于非常大的数据集需要花费几个小时完成的数据移动。

PivotalR 包的关键特征：

- 以R语法的方式探索和操作数据库内的数据。SQL 翻译由 PivotalR 来执行。
- 使用熟悉的 R 语法的预测分析算法，例如线性和逻辑回归。 PivotalR 访问 MADlib 数据库内分析函数调用。
- 对于广泛关于以标准 R 格式的示例文档包能够通过 R 客户端来访问。
- PivotalR 包也支持 MADlib 功能的访问。

更多关于 PivotalR 的信息包括支持的 MADlib 功能的信息，见 https://cwiki.apache.org/confluence/display/MADLIB/PivotalR。

PivotalR 的 R 语言包可以在https://cran.r-project.org/web/packages/PivotalR/index.html找到。

PostGIS 扩展

本章包含以下信息：

- 关于 PostGIS
- HashData PostGIS 扩展
- 启用 PostGIS 支持
- 用法
- PostGIS 扩展支持和限制

父主题：HashData 数据库参考指南

关于 PostGIS

PostGIS 是 PostgreSQL 的空间数据库扩展，允许 GIS（地理信息系统）对象存储在数据库中。HashData 数据库 PostGIS 扩展包括对基于 GiST 的 R-Tree 空间索引和用于分析和处理 GIS 对象的功能的支持。

有关 PostGIS 的更多信息，请转到 http://postgis.refractions.net/。

有关 HashData 数据库 PostGIS 扩展支持的信息，请参阅 PostGIS 扩展支持和限制。

HashData PostGIS 扩展

HashData 数据库 PostGIS 扩展可从 Pivotal Network 获得。您可以使用 HashData Package Manager 进行安装（gppkg）。

- HashData 数据库支持 PostGIS 扩展包 2.0 版（PostGIS 2.0.3）。要在 HashData 数据库中使用扩展包，必须安装和使用 HashData 数据库扩展包（ gppkg 文件和 contrib 模块）。

查看 PostGIS 文档以获取更改列表：http://postgis.net/docs/manual-2.0/release_notes.html

> 警告：PostGIS 2.0 删除了许多弃用的功能，但在 PostGIS 1.4 中可用。使用 PostGIS 1.4 中弃用的函数编写的函数和应用程序可能需要重写。请参阅 PostGIS 文档以获取新功能，增强功能或更改功能的列表：http://postgis.net/docs/manual-2.0/PostGIS_Special_Functions_Index.html#NewFunctions

HashData 数据库 PostGIS 限制

HashData 数据库 PostGIS 扩展不支持以下功能：

- 拓扑
- 光栅
- 少量用户定义的函数和聚合
- PostGIS 长事务支持
- 几何和地理类型修饰符

有关 HashData 数据库 PostGIS 支持的信息，请参阅 PostGIS 扩展支持和限制。

启用 PostGIS 支持

安装 PostGIS 扩展包后，为需要使用的每个数据库启用 PostGIS 支持。要启用支持，请在目标数据库中运行随 PostGIS 软件包一起提供的启用 SQL 脚本。

对于 PostGIS ，您运行两个 SQL 脚本 postgis.sql 和 spatial_ref_sys.sql 在您的目标数据库中。

例如：

```
psql -d mydatabase -f
  $GPHOME/share/postgresql/contrib/postgis-2.0/postgis.sql
psql -d mydatabase -f
  $GPHOME/share/postgresql/contrib/postgis-2.0/spatial_ref_sys.sql
```

> 注意：spatial_ref_sys.sql 填充 spatial_ref_sys 带有 EPSG 坐标系定义标识符的表。如果您已覆盖标准条目并想要使用这些覆盖，请不要在创建新数据库时加载 spatial_ref_sys.sql 文件。

您的数据库现在在空间上启用了。

用法

以下示例 SQL 语句创建非 OpenGIS 表和几何。

```
CREATE TABLE geom_test ( gid int4, geom geometry,
  name varchar(25) );
INSERT INTO geom_test ( gid, geom, name )
  VALUES ( 1, 'POLYGON((0 0 0,0 5 0,5 5 0,5 0 0,0 0 0))', '3D Square');
INSERT INTO geom_test ( gid, geom, name )
  VALUES ( 2, 'LINESTRING(1 1 1,5 5 5,7 7 5)', '3D Line' );
INSERT INTO geom_test ( gid, geom, name )
  VALUES ( 3, 'MULTIPOINT(3 4,8 9)', '2D Aggregate Point' );
SELECT * from geom_test WHERE geom &&
  Box3D(ST_GeomFromEWKT('LINESTRING(2 2 0, 3 3 0)'));
```

以下示例 SQL 语句将创建一个表，并使用引用 SPATIAL_REF_SYS 表中的条目的 SRID 整数值向表中添加几何列。该 INSERT 语句添加到表中的地址。

```
CREATE TABLE geotest (id INT4, name VARCHAR(32) );
SELECT AddGeometryColumn('geotest','geopoint', 4326,'POINT',2);
INSERT INTO geotest (id, name, geopoint)
  VALUES (1, 'Olympia', ST_GeometryFromText('POINT(-122.90 46.97)', 4326));
INSERT INTO geotest (id, name, geopoint)|
  VALUES (2, 'Renton', ST_GeometryFromText('POINT(-122.22 47.50)', 4326));
SELECT name,ST_AsText(geopoint) FROM geotest;
```

空间索引

PostgreSQL 提供对 GiST 空间索引的支持。即使在大型物体上，GiST 方案也提供索引。它使用有损索引系统，其中较小的对象充当索引中较大对象的代理。在 PostGIS 索引系统中，所有对象都使用边界框作为索引中的代理。

建立空间索引

您可以按如下所示构建 GiST 索引：

```
CREATE INDEX indexname
ON tablename
USING GIST ( geometryfield );
```

PostGIS 扩展支持和限制

本节介绍 HashData PostGIS 扩展功能支持和限制。

- 支持的 PostGIS 数据类型
- 支持的 PostGIS 索引
- PostGIS 扩展限制

HashData 数据库 PostGIS 扩展不支持以下功能：

- 拓扑

- 光栅

支持的 PostGIS 数据类型

HashData 数据库 PostGIS 扩展支持以下 PostGIS 数据类型：

- Box2D的
- box3d
- 几何
- 地理
- 球体

支持的 PostGIS 索引

HashData 数据库 PostGIS 扩展支持 GiST（广义搜索树）索引。

PostGIS 扩展限制

本节列出了用户定义函数（ UDF ），数据类型和聚合的 HashData 数据库 PostGIS 扩展限制。

- HashData 数据库不支持与 PostGIS 拓扑或栅格功能相关的数据类型和功能，例如: TopoGeometry 和 ST_AsRaster 。
- ST_Estimated_Extent 函数不受支持。该功能需要用于 HashData 数据库不可用的用户定义数据类型的表列统计信息。
- 不支持 ST_GeomFromGeoJSON 函数。该功能需要 JSON 支持。HashData 数据库支持 JSON。
- HashData 数据库不支持这些 PostGIS 聚合：
  - ST_MemCollect
  - ST_MakeLine

  在具有多个分段的 HashData 数据库中，如果聚合重复多次调用，聚合可能会返回不同的答案。
- HashData 数据库不支持 PostGIS 长事务。

  PostGIS 依赖触发器和 PostGIS 表 *public.authorization_table* 来支持长事务。当 PostGIS 尝试获取长事务的锁定时，HashData 数据库会报告错误，指出该函数无法访问关系 *authorization_table* 。
- HashData 数据库不支持用户定义类型的类型修饰符。解决方法是使用 AddGeometryColumn 函数用于 PostGIS 几何。例如，无法使用以下 SQL 命令创建具有 PostGIS 几何的表：

  ```
  CREATE TABLE geometry(id INTEGER,geom geometry(LINESTRING));
  ```

  使用 AddGeometryColumn 函数将 PostGIS 几何添加到表中。例如，以下 SQL 语句将创建一个表并将 PostGIS 几何添加到表中：

  ```
  CREATE TABLE geometry(id INTEGER);
  SELECT AddGeometryColumn('public','geometries','geom',0,'LINESTRING',2);
  ```