HashData

目录

开发指南

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。

HashData 数据仓库系统概述

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:

- *HashData 数据仓库架构*

HashData 数据仓库架构

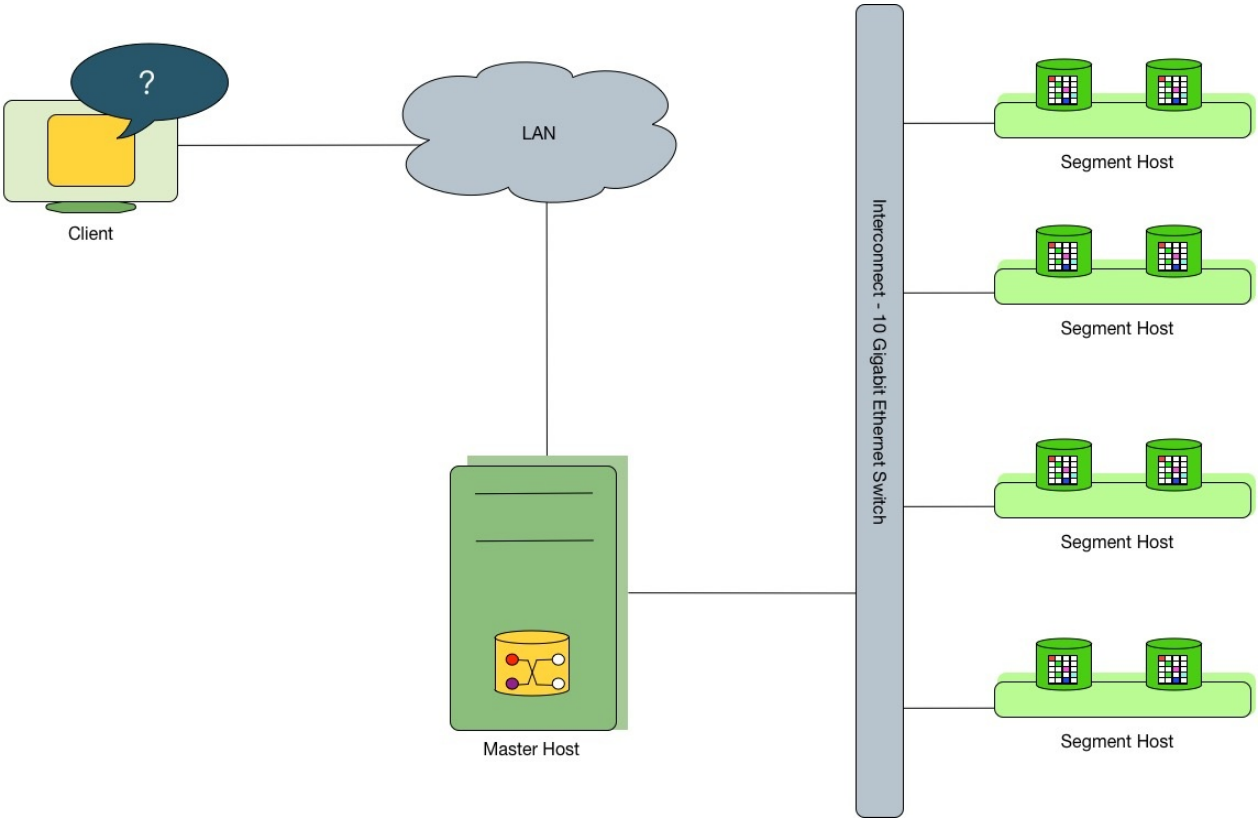
HashData 数据仓库是为了管理大容量分析型数据仓库和商业智能分析业务而设计的大规模并行处理（MPP）数据库服务系统。

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 和 postgres 开源数据库技术，通过对 postgres 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 master 节点是整个数据库系统的入口节点，用户通过客户端连接 master 节点来提交 SQL 查询语句。master 节点将会协调其它 segment 节点来存储和处理用户的数据。

图1. 系统架构



master 节点

HashData 数据仓库的 master 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 Segment 节点上执行。

HashData 数据仓库的用户可以像使用 postgres 一样，通过 master 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。

master 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）。master 节点没有存储用户数据的信息，所有的用户数据都存储在 Segment 节点，master 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 Segment 节点，协调各个 Segment 节点执行，并汇总执行结果，最后返回给客户端程序。

Segment 节点

HashData 数据仓库的 Segment 节点实际上同样是一个经过修改的 postgres 数据库，每个 Segment 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 master 节点，并且发送一个查询时，每个 Segment 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 Segment 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 Segment 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 HashData 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 postgres 的客户端程序来连接 HashData 数据仓库的 master 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 postgres 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 postgres 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 postgres 实现，因此可以直接使用 postgres 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 postgres 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。 postgres 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 postgres 驱动程序，并配置连接 HashData 数据仓库主节点的信息。

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。

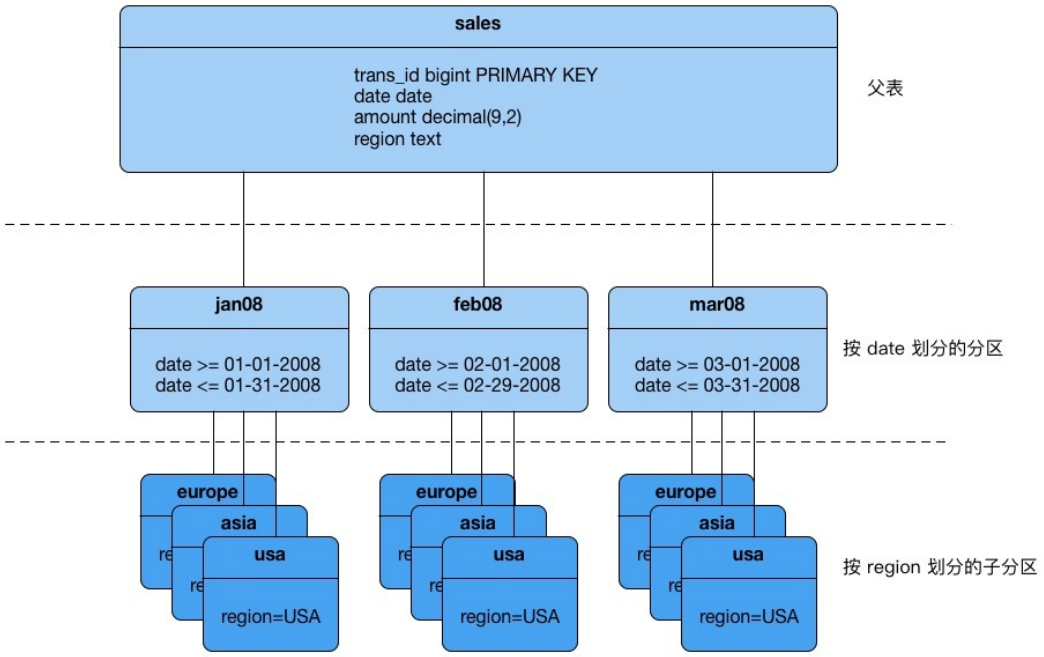


图**1.** 多层分区表结构

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。

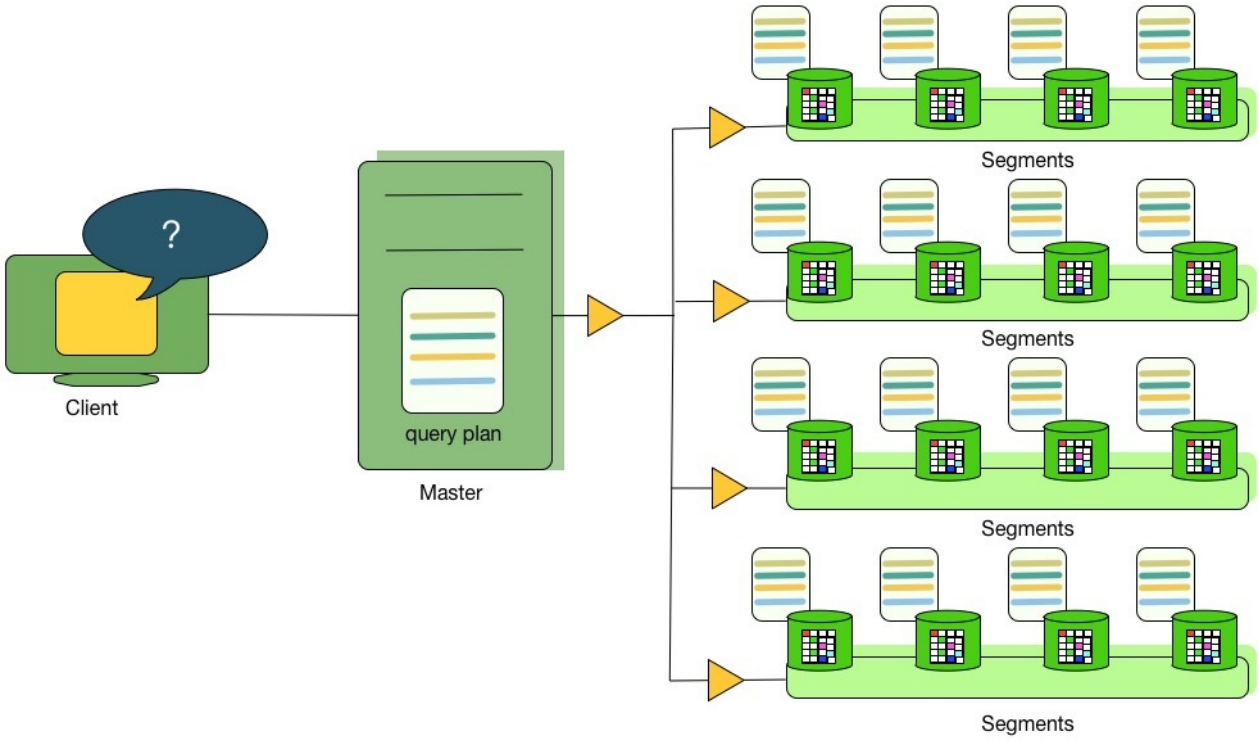


图**1**：分发并行查询计划

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。

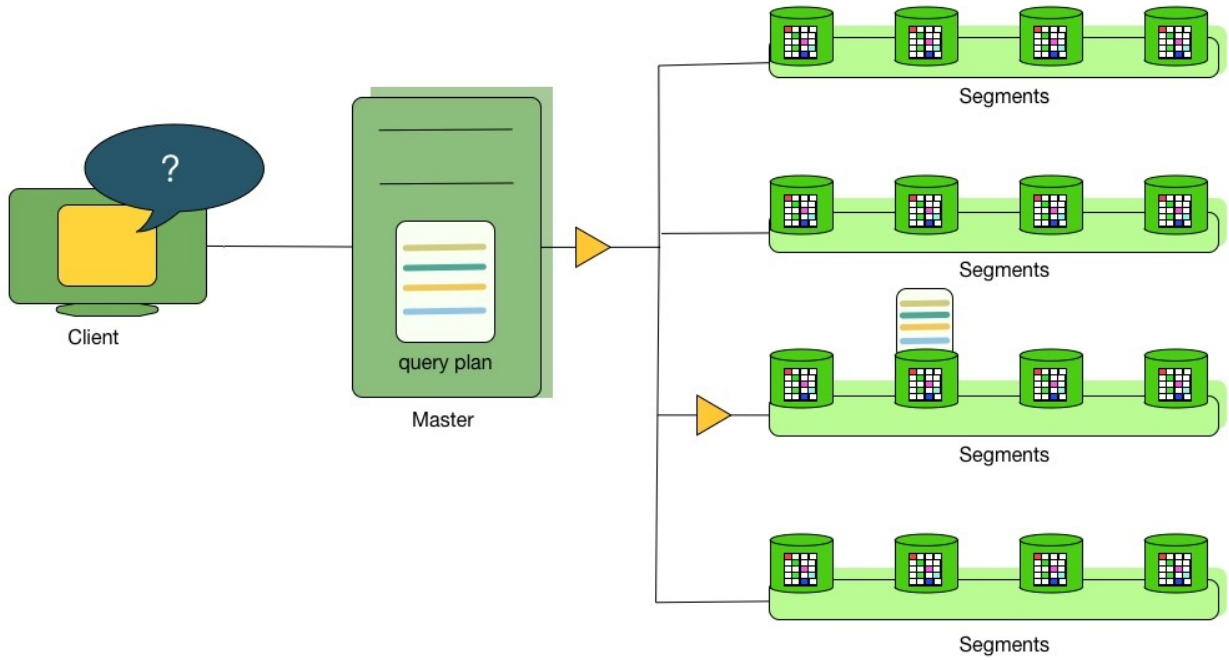


图**2**：分发特定节点查询计划

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。

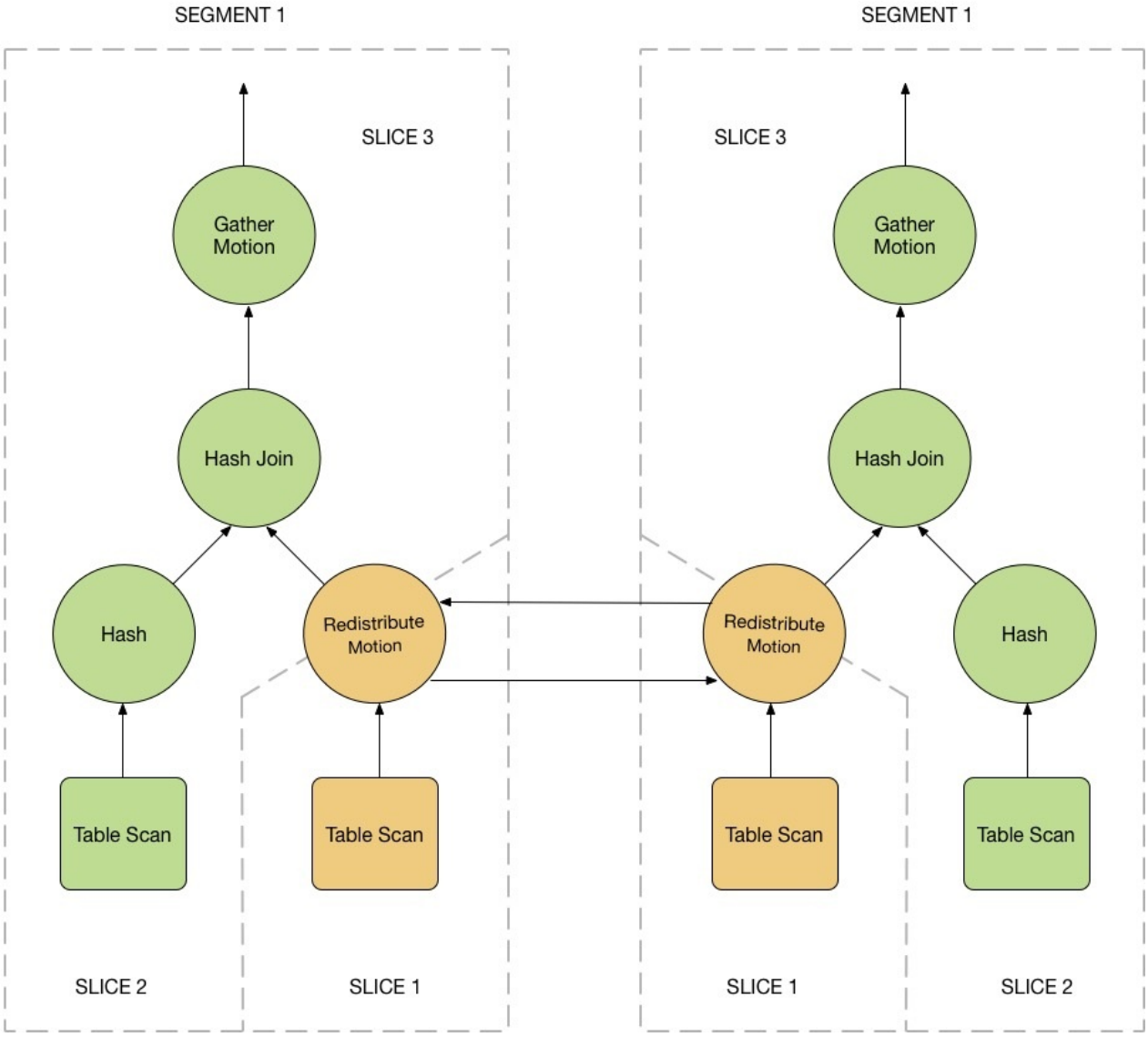


图**3**：查询计划切片

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。

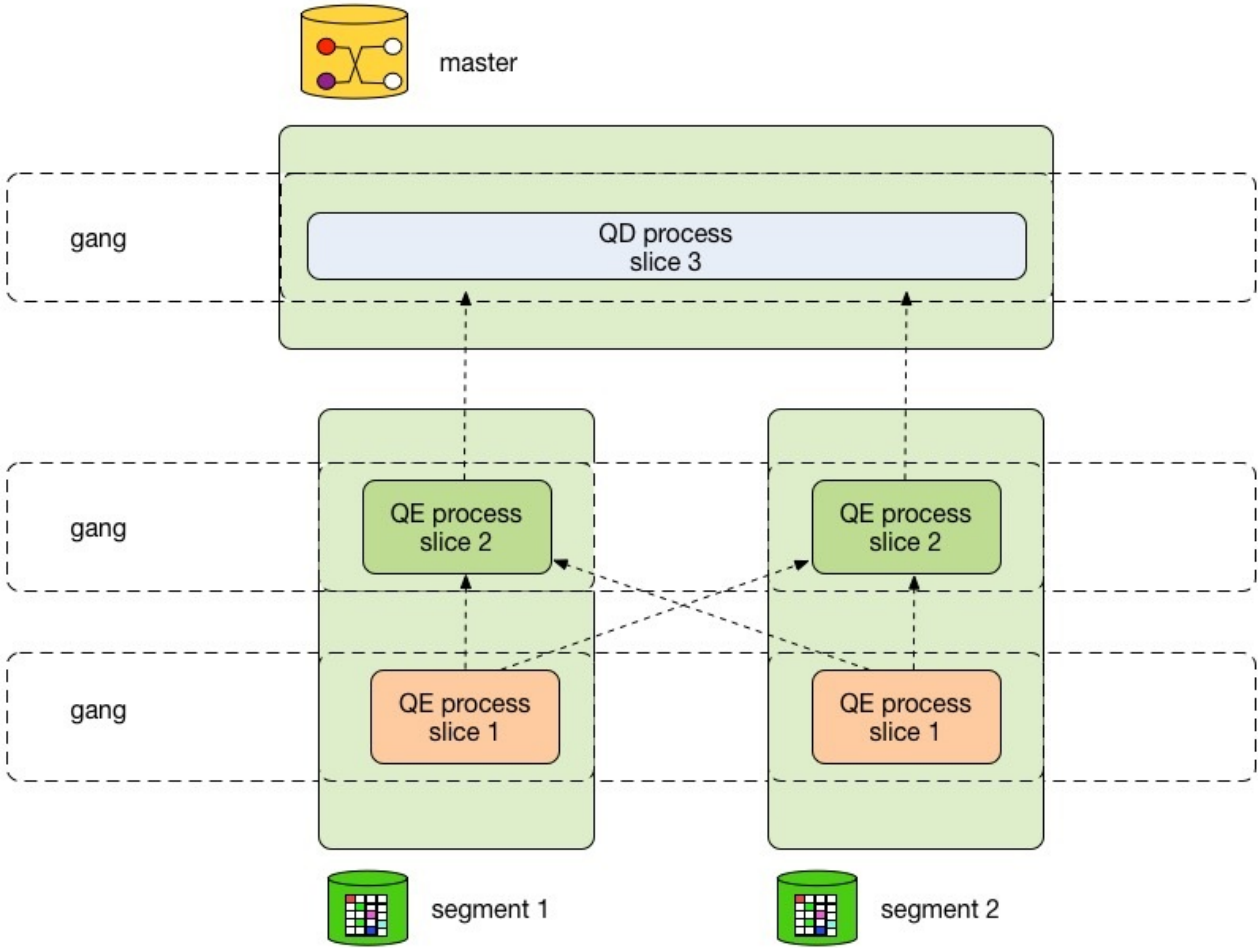


图**4**：查询执行器处理请求

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)
```